Third Edition

SQL Server Execution Plans

What goes on beneath the surface with your queries

By Grant Fritchey

Technical Review by Hugo Kornelis

SQL Server Execution Plans

Third Edition

For

SQL Server 2008 through to 2017

and Azure SQL Database

By Grant Fritchey

Published by Redgate Publishing 2018

First Edition 2008

ISBN 978-1-910035-22-1

Technical Reviewer: Hugo Kornelis

Editor: Tony Davis

Typeset by Gower Associates

Contents

About the Author

Grant Fritchey is a SQL Server MVP with over 30 years' experience in IT including time spent in support, development, and database administration.

Grant has worked with SQL Server since version 6.0, back in 1995. He has developed in VB, VB.Net, C#, and Java. Grant joined Redgate as a Product Evangelist in January 2011.

He writes articles for publication at SQL Server Central, Simple Talk, and other community sites, and has published multiple books including the one you're reading now and *SQL Server Query Performance Tuning*, 5th Edition (Apress, 2018). Grant also blogs on this topic and others at https://scarydba.com.

You can contact him through *grant@scarydba.com*.

About the Technical Reviewer

Hugo Kornelis has been working in IT for almost 35 years, the last 20 of which have been focused almost completely on SQL Server.

Within the SQL Server community, Hugo has answered thousands of questions on various online forums. He also blogs at https://sqlserverfast.com/blog/, has contributed articles to SQL Server Central and Simple Talk, and has authored a Pluralsight course on relational database design. He has been a speaker at many conferences in Europe, and a few in the rest of the world. In recognition of his community contributions, Microsoft has awarded Hugo SQL Server MVP and Data Platform MVP 11 times (2006–2016).

Hugo has started to document his impressive knowledge of execution plans on sqlserverfast.com, which is an excellent resource for anyone who has finished reading this book and wants to know even more about all the nitty-gritty detail in their execution plans. You'll find articles that expose interesting or uncommon patterns in execution plans, and describe exactly how each one works, as well as *The SQL Server Execution Plan Reference* (https://sqlserverfast.com/epr/), which, eventually, will list all operators, with their exact behavior and all their properties.

Introduction

Frequently, a T-SQL query you wrote behaves in ways you don't expect, and causes slow response times for the application users, and resource contention on the server. Sometimes, you didn't write the offending query; it came from a third-party application, or was code generated by an improperly-used Object Relational Mapping layer. In any of these situations, and a thousand others, query tuning becomes quite difficult.

Often, it's very hard to tell, just by looking at the T-SQL code, why a query is running slowly. SQL is a declarative language, and a T-SQL query describes only the set of data that we want SQL Server to return. It does not tell SQL Server *how to execute* the query, to retrieve that data.

When we submit a query to SQL Server, several server processes kick into action whose collective job is to manage the querying or modification of the data. Specifically, a component of the relational database engine called the **Query Optimizer** has the job of examining the submitted query text and defining a strategy for executing it. The strategy takes the form of an execution plan, which contains a series of **operators**, each describing an action to perform on the data.

So, if a query is performing poorly, and you can't understand why, then the execution plan will tell you, not only what data set is coming back, but also what SQL Server did, and in what order, to get that data. It will reveal how the data was retrieved, and from which tables and indexes, what types of joins were used, at what point filtering and sorting occurred, and a whole lot more. These details will often highlight the likely source of any problem.

What the Execution Plan Reveals

An execution plan is, literally, a set of instructions on how to execute a query. The optimizer passes each plan on to the execution engine, which executes the query according to those instructions. The optimizer also stores plans in an area of memory called the plan cache, so that it can reuse existing execution strategies where possible.

During development and testing, you can request the plan very easily, using a few buttons in SQL Server Management Studio. When investigating a query problem on a live production system, you can often retrieve the plan used for that query from the plan cache, or from the Query Store.

Armed with the execution plan, you have a unique window into what's going on behind the scenes in SQL Server, and a wealth of information on how SQL Server has decided to resolve the T-SQL that you passed to it. You can see things like:

- the order in which the optimizer chose to access the tables referenced in the query
- which indexes it used on each table, and how the data was pulled from them
- how many rows the optimizer thought an operator would return, based on its statistical understanding of the underlying data structures and data, and how many rows it found in reality
- how keys and referential constraints affect the optimizer's understanding of the data, and therefore the behavior of your queries
- how data is being joined between the tables in your query
- when filtering and sorting occurred, how any calculations and aggregation were performed, and more.

Execution plans are one of your primary tools for understanding how SQL Server does what it does. If you're a data professional of any kind there will be times when you need to wade into the guts of an execution plan, and so you'll need to know what it is that you're looking at, and how to proceed.

That is why I wrote this book. My goal was to gather into a single location as much useful information on execution plans as possible. I'll walk you through the process of reading them, and show you how to understand the information that they present to you. Specifically, I will cover:

- how to capture execution plans using manual and automatic methods
- a documented method for interpreting execution plans, so that you can make sense of them in your own environment
- how SQL Server represents and interprets the common SQL Server objects, such as indexes, views, stored procedures, derived tables, and so on, in execution plans
- how to control execution plans with hints and plan guides, and why this is a double-edged sword
- how the Query Store works with, and collects data on, execution plans and how you can take control of them using the Query Store.

These topics and a slew of others, all related to execution plans and their behavior, are covered throughout this book. I focus always on the details of the execution plans, and how the behaviors of SQL Server are manifest in the execution plans.

As we work through each topic, I'll explain all the individual elements of the execution plan, how each operator works, how they interact, and the conditions in which each operator works most efficiently. With this knowledge, you'll have everything you need to allow you to tackle every execution plan, regardless of complexity, and understand what it does.

Fixing Query Problems Using Execution Plans

Execution plans provide all the information you need, to understand how SQL Server executed your queries. Paradoxically though, given that most people look at an execution plan hoping to improve the performance of a query, this book isn't, and couldn't be, a book about query performance tuning. The two topics are linked, but separate. If you are specifically looking for information on how to optimize T-SQL, or build efficient indexes, then you need a book dedicated to those topics.

Neither is the execution plan the first place to look, if you need to tune performance on a production system. You'll check for misconfigurations of servers or database settings, you'll look for obvious points of resource contention on the server, which may be causing severe locking and blocking problems, and so on. At this point, if performance is still slow, you'll likely have narrowed the cause down to a few "hot" tables and one or two queries on those tables. *Then*, you can examine the plans and look for possible causes of the problem.

However, execution plans are not necessarily designed to help the occasional user find the cause of a query problem quickly, in the heat of firefighting poor SQL Server performance. You need first to have invested time in learning the "language" of the plan and how to read it, and what led SQL Server to choose that plan, and those operators, to execute your query.

And this book is that investment.

As you work through it, you will start to recognize each of the different operators SQL Server might use to access the data in a table, or to join two tables, or to group and aggregate data. As you learn how these operators work, and how they process the data they receive, you will begin to recognize why some operators are designed for handling small numbers of rows, and why others are better for larger data sets. You will start to understand the "properties" of the data (such as uniqueness, and logical ordering) that will allow certain operators to work more efficiently.

As you make connections between all of this and the behavior and performance of your queries, you will suddenly find that you have an *expectation* of what a plan will reveal before you even look at it, based on your understanding of the query logic, and of the data. Therefore, any unexpected operators in the plan will catch your attention, and you'll know where to look for possible issues, and what to do about them.

You are now at the stage where you can use plans to solve problems. Usually the optimizer makes good choices of plan. Occasionally, it errs. The possible causes are many. Perhaps, it is missing critical information about the database, because of a lack of keys or constraints. Adding them might improve the query performance. Sometimes, its statistical understanding of the data is inaccurate, or out of date. It may simply have no *efficient* means to retrieve the initial data set, and you need to add an index or modify an existing one. Sometimes our query logic simply defeats efficient optimization, and the best course is a rewrite, although that's not always possible when troubleshooting a production system.

This book's job is to teach you how to read the plan, so that you can understand what is causing the bad performance. It is then your job to work out how best to fix it, armed with the understanding of execution plans that will give a much better chance of success.

This knowledge is also hugely valuable when writing new queries, or updating existing code. Once you've verified that the code returns the correct results, you can test its performance. Does it fall within expectations? If not, before you rip up the query and try again, look at the plan, because you may just have made a simple mistake that means SQL Server isn't executing it as efficiently as it could.

If you can test the query under different data loads, you'll be able to gauge whether query performance will scale smoothly once the query hits a full production-size database. As the data volume grows, and the data changes, the optimizer will often devise a different plan. Is it still an efficient plan? If not, perhaps you can then try to rewrite the query, or modify the data structures, to prevent performance issues before the code ever reaches production!

Before deploying T-SQL code, every database developer and DBA should get into the habit of looking at the execution plan for any query that is beyond a certain level of complexity, if it is intended to be run on a large-scale production database.

Changes in this Third Edition

The way I think about how to use execution plans, and how to read them, has changed a lot over the years. I've now rearranged the book to reflect that. After the early chapters have established an understanding of the basics of the optimizer and how to capture execution plans, the later chapters focus more on the methods of reading plans, not just on what is in the operators and their properties. And, of course, Microsoft has continued to make changes to SQL Server, so there are new operators and mechanisms that must be covered.

Some of the new topics include:

- automate capturing execution plans using Extended Events
- new warnings and operators
- batch mode processing
- adaptive query processing
- additional functionality added to SQL Server 2014, 2016, and 2017, as well as Azure SQL Database.

There are lots more changes because, with the help of my tech editor, Hugo Kornelis, and long-time (and long-suffering) editor, Tony Davis, we've basically rewritten this book from the ground up.

With the occasional hiatus, this book took over three years to rewrite and, during that time, three versions of SQL Server were released, and who knows how many changes in Azure were introduced. Microsoft has also divorced SQL Server Management System (SSMS) releases from the main product, so that more and more new functionality has been introduced, faster. I've done my level best to keep up, and the text should be up to date for May 2018. Any changes that came out after that, won't be in this edition of the book.

Code Examples

Throughout this book, I'll be supplying T-SQL code that you're encouraged to run for yourself, to generate execution plans. From the following URL, you can obtain all the code you need to try out the examples in this book:
https://scarydba.com/resources/ExecutionPlansV3.zip.

Most of the code will run on all editions and versions of SQL Server, starting from SQL Server 2012. Most, although not all, of the code will work on Azure SQL Database. Unless noted otherwise, all examples were written for, and tested on, the SQL Server sample database, **AdventureWorks2014**, and you can get a copy of it from GitHub: https://bit.ly/2yyW1kh.

If you test the code on a different version of **AdventureWorks**, or if Microsoft updates **AdventureWorks2014**, then statistics can change, and you may see a different execution plan than the one I display in the book. If you are working with procedures and scripts other than those supplied, please remember that encrypted stored procedures will not display an execution plan.

The initial execution plans will be simple and easy to read from the samples presented in the text. As the queries and plans become more complicated, the book will describe the situation but, to see the graphical execution plans or the complete set of XML, it will be necessary for you to generate the plans. So, please, read this book next to your machine, if possible, so that you can try running each query yourself!

Chapter 1: Introducing the Execution Plan

An execution plan is a set of instructions for executing a query. Devised by the SQL Server Query Optimizer, an execution plan describes the set of operations that the execution engine needs to perform to return the data required by a query.

The execution plan is your window into the SQL Server Query Optimizer and query execution engine. It will reveal which tables and indexes a query accessed, in which order, how they were accessed, what types of joins were used, how much data was retrieved initially, and at what point filtering and sorting occurred. It will show how aggregations were performed, how calculated columns were derived, how and where foreign keys were accessed, and more.

Any problems created by the query will frequently be apparent within the execution plan, making it an excellent tool for troubleshooting poorly-performing queries. Rather than guess at why a query is sending your I/O through the roof, you can examine its execution plan to identify the exact operation, and associated section of T-SQL code, that is causing the problem. For example, the plan may reveal that a query is reading every row in a table or index, even though only a small percentage of those rows are being used in the query. By modifying the code within the `WHERE` clause, SQL Server may be able to devise a new plan that uses an index to find directly (or *seek*) only the required rows.

This chapter will introduce execution plans. We'll explore the basics of obtaining an execution plan and start the process of learning how to read them, covering the following topics:

- **A brief background on the query optimizer** – execution plans are a result of the optimizer's operations, so it's useful to know at least a little bit about what the optimizer does, and how it works.
- **The plan cache and plan reuse** – execution plans are usually stored in an area of memory called the plan cache and may be reused. We'll discuss why plan reuse is important.
- **Actual and estimated execution plans** – clearing up the confusion over estimated versus actual execution plans and how they differ.
- **Capturing an execution plan** – we'll capture a plan for a simple query and introduce some of the basic elements of a plan, and the information they contain.

What Happens When a Query is Submitted?

Every time we submit a query to SQL Server, several server processes kick into action; their job collectively is to manage the querying or modification of that data. Within the *relational engine*, the query is parsed by the parser, bound by the algebrizer and then finally optimized by the query optimizer, where the most important part of the work occurs. Collectively, we refer to these processes as **query compilation**. The SQL Server relational engine takes the input, which is the SQL text of the submitted query, and compiles it into a plan to execute that query. In other words, the process generates an **execution plan**, effectively a series of instructions for processing the query.

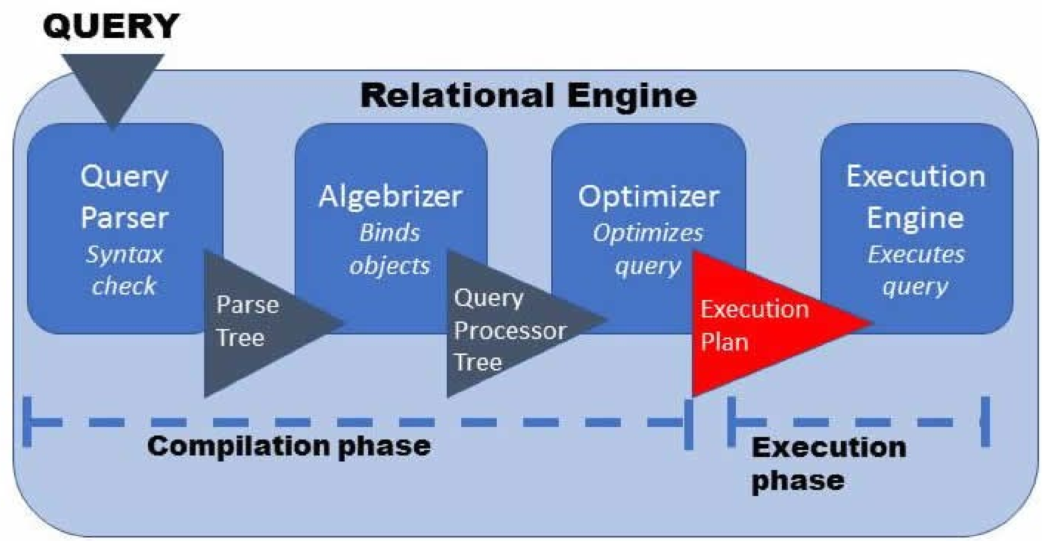


Figure 1-1: Query compilation and execution.

The plan generated is stored in an area of memory called the **plan cache**. The next time the optimizer sees the same query text, it will check to see if a plan for that SQL text exists in the plan cache. If it does, it will pass the cached plan on to the *query execution engine*, bypassing the full optimization process.

The query execution engine will execute the query, according to the instructions laid out in the execution plan. It will generate calls to the *storage engine*, the process that manages access to disk and memory within SQL Server, to retrieve and manipulate data as required by the plan.

Query compilation phase

Since **execution plans** are created and managed from within the relational engine, that's where we'll focus our attention in this book. The following sections review briefly what happens during query compilation, covering the parsing, binding, and particularly the optimization phase, of query processing.

Query parsing

When a request to execute a T-SQL query reaches SQL Server, either an ad hoc query from a command line or application program, or a query in a stored procedure, user-defined function, or trigger, the query compilation and execution process can begin, and the action starts in the relational engine.

As the T-SQL arrives in the relational engine, it passes through a process that checks that the T-SQL is written correctly, that it's well formed. This process is query *parsing*. If a query fails to parse correctly, for example, if you type `SELETC` instead of `SELECT`, then parsing stops and SQL Server returns an error to the query source. The output of the **Parser** process is a parse tree, or query tree (or it's even called a sequence tree). The parse tree represents the logical steps necessary to execute the requested query.

Query binding

If the T-SQL string has parsed correctly, the parse tree passes to the **algebrizer**, which performs a process called query binding. The algebrizer resolves all the names of the various objects, tables, and columns referred to within the query string. It identifies, at the individual column level, all the data types `(varchar(50)` versus `datetime` and so on) for the objects being accessed. It also determines the location of aggregates, such as `SUM` and `MAX`, within the query, a process called *aggregate binding*.

This algebrizer process is important because the query may have aliases or synonyms, names that don't exist in the database, that need to be resolved, or the query may refer to objects not in the database. When objects don't exist in the database, SQL Server returns an error from this step, defining the invalid object name (except in the case of *deferred name resolution*). As an example, the algebrizer would quickly find the table `Person.Person` in the `AdventureWorks` database. However, the `Product.Person` table, which doesn't exist, would cause an error and the whole compilation process would stop.

Stored procedure and deferred name resolution

On creating a stored procedure, its statement text is parsed and stored in `sys.sql_modules` catalog view. However, the tables referenced by the text do not have to exist in the database at this point. This gives more flexibility because, for example, the text can reference a temporary table that is not created by the stored procedure, and does not yet exist, but that we know will exist at execution time. At execution time, the query processor finds the names of the objects referenced, in `sys.sql_modules`, and makes sure they exist.

The algebrizer outputs a binary called the **query processor tree**, which is then passed on to the **query optimizer**. The output also includes a hash, a coded value representing the query. The optimizer uses the hash to determine whether there is already a plan for this query stored in the plan cache, and whether the plan is still valid. A plan is no longer considered valid after some changes to the table (such as adding or dropping indexes), or when the statistics used in the optimization were refreshed since the plan was created and stored. If there is a valid cached plan, then the process stops here and the cached plan is reused.

Query optimization

The query optimizer is a piece of software that considers many alternate ways to achieve the requested query result, as defined by the query processor tree passed to it by the algebrizer. The optimizer estimates a "cost" for each possible alternative way of achieving the same result, and attempts to find a plan that is cheap enough, within as little time as is reasonable.

Most queries submitted to SQL Server will be subject to a **full cost-based optimization** process, resulting in a **cost-based plan**. Some very simple queries can take a "fast track" and receive what is known as a **trivial plan**.

Full cost-based optimization

The full cost-based optimization process takes three inputs:

- The **Query processor tree** – gives the optimizer knowledge of the logical query structure and of the underlying tables and indexes.
- **Statistics** – index and column statistics give the optimizer an understanding of volume and distribution of data in the underlying data structures.
- **Constraints** – the primary keys, enforced and trusted referential constraints, and any other types of constraints in place on the tables and columns that make up the query, tell the optimizer the limits on possible data stored within the tables referenced.

Using these inputs, the optimizer applies its model, essentially a set of rules, to transform the logical query tree into a plan containing a set of **operators** that, collectively, will physically execute the query. Each operator performs a dedicated task. The optimizer uses various operators for accessing indexes, performing joins, aggregations, sorts, calculations, and so on. For example, the optimizer has a set of operators for implementing logical join conditions in the submitted query. It has one specialized operator for a **Nested Loops** implementation, one for a **Hash Match**, one for a **Merge**, and one for an **Adaptive Join**.

The optimizer will generate and evaluate many possible plans, for each candidate testing different methods of accessing data, attempting different types of join, rearranging the join order, trying different indexes, and so on. Generally, the optimizer will choose the plan that its calculations suggest will have the lowest total cost, in terms of the sum of the estimated CPU and I/O processing costs.

During these calculations, the optimizer assigns a number to each of the steps within the plan, representing its estimation of the combined amount of CPU and disk I/O time it thinks each step will take. This number is the **estimated cost** for that step. The accumulation of costs for each step is the estimated cost for the execution plan itself. We'll shortly cover the estimated costs, and why they are estimates, in more detail.

Plan evaluation is a *heuristic* process. The optimizer is not attempting to find the best possible plan but rather the lowest-cost plan in the fewest possible iterations, meaning the shortest amount of time. The only way for the optimizer to arrive at a perfect plan would be to be able to take an infinite amount of time. No one wants to wait that long on their queries.

Having selected the lowest-cost plan it could find within the allotted number of iterations, the query execution component will use this plan to execute the query and return the required data. As noted earlier, the optimizer will also store the plan in the plan cache. If we submit a subsequent request with identical SQL text, it will bypass the entire compilation process and simply submit the cached plan for execution. A parameterized query will be parsed, and if a plan with a matching query hash is found in the cache, the remainder of the process is short-circuited.

Trivial plans

For very simple queries, the optimizer may simply decide to apply a **trivial plan**, rather than go through the full cost-based optimization process. The optimizer's rules for deciding when it can simply use a trivial plan are unclear, and probably complex. However, for example, a very simple query, such as a SELECT statement against a single table with no aggregates or calculations, as shown in Listing 1-1, would receive a trivial plan.

```
SELECT d.Name
FROM HumanResources.Department AS d
WHERE d.DepartmentID = 42;
```

Listing 1-1

Adding even one more table, with a JOIN, would make the plan non-trivial. Also, if additional indexes exist on the table, or if the possibility of parallelism exists (discussed more in Chapter 11), then you will get further optimization of the plan.

It's also worth noting here that this query falls within the rules covered by auto-parameterization, so the hard-coded value of "42" will be replaced with a parameter when the plan is stored in cache, to enable plan reuse. We'll cover that in more detail in Chapter 9.

All data manipulation language (DML) statements are optimized to some extent, even if they receive only a trivial plan. However, some types of Data Definition Language (DDL) statement may not be optimized at all. For example, if a CREATE TABLE statement parses correctly, then there is only one "right way" for the SQL Server system to create a table. Other DDL statements, such as using ALTER TABLE to add a constraint, will go through the optimization process.

Query execution phase

The query execution engine executes the query per the instructions set out in the execution plan. At runtime, the execution engine cannot change the optimizer's plan. However, it can under certain circumstances force a plan to be recompiled. For example, if we submit to the query processor a batch or a stored procedure containing multiple statements, the whole batch will be compiled at once, with plans produced for every statement. Even if we have IF…THEN or CASE flow control in our queries, all statements within the batch will be compiled. At runtime, each plan is checked to ensure it's still valid. As for plans taken in the plan cache, if the plan's associated statement references tables that have changed or had statistics updated since the plan was compiled, then the plan is no longer considered valid. If that occurs, then the execution is temporarily halted, the compilation process is invoked, and the optimizer will produce a new plan, only for the affected statement in the batch or procedure.

Introduced in SQL Server 2017, there is also the possibility of **interleaved execution** when the object being referenced in the query is a **multi-statement table valued user-defined function**. During an interleaved execution, the optimizer generates a plan for the query, in the usual fashion, then the optimization phase pauses, the pertinent subtree of a given plan is executed to get the actual row counts, and the optimizer then uses the actual row counts to optimize the remainder of the query. We'll cover interleaved execution and multi-statement table valued user-defined functions in more detail in Chapter 8.

Working with the Optimizer

Most application developers, when writing application code, are used to exerting close control, not just over the required result of a piece of code, but also over how, step by step, that outcome should be achieved. Most compiled languages work in this manner. SQL Server and T-SQL behave in a different fashion.

The query optimizer, not the database developer, decides how a query should be executed. We focus solely on designing a T-SQL query to describe logically the required set of data. We do not, and should not, attempt to dictate to SQL Server how to execute it.

What this means in practice is the need to write efficient SQL, which generally means using a set-based approach that describes as succinctly as possible, in as few statements as possible, just the required data set. This is the topic for a whole other book, and one that's already been written by Itzik Ben-Gan, *Inside SQL Server T-SQL Querying*.

However, beyond that, there are some practical ways that the database developer or DBA can help the optimizer generate efficient plans, and avoid unnecessary plan generation:

- maintaining accurate, up-to-date statistics
- promoting plan reuse.

The importance of statistics

As we've discussed, the optimizer will choose the lowest-cost plan, based on estimated cost. The principal driver of these estimates is the statistics on your indexes and data. Ultimately, this means that the quality of the plan choice is limited by the quality of the statistics the optimizer has available for the target tables and indexes.

We don't want the optimizer to read all the data in all the tables referenced in a query each time it tries to generate a plan. Instead, the optimizer relies on statistics, aggregated information based on a sample of the data, that provides the information used by the optimizer to represent the entire collection of data.

The estimated cost of an execution plan depends largely on its *cardinality estimations*, in other words, its knowledge of how many rows are in a table, and its estimations of how many of those rows satisfy the various search and join conditions, and so on.

New cardinality estimator in SQL Server 2014

In SQL Server 2014, the cardinality estimator within SQL Server was updated for the first time since SQL Server 7.0. It's very likely that you may see a difference in plans generated in SQL Server 2014 compared to previous versions, just because of the update to the cardinality estimator, let alone any updates to other processes within the optimizer.

These cardinality estimations rely on statistics collected on columns and indexes within the database that describe the data distribution, i.e. the number of different values present, and how many occurrences of each value. This in turn determines the **selectivity** of the data. If a column is unique, then it will have the highest possible selectivity, and the selectivity degrades as the level of uniqueness decreases. A column such as "gender," for example, will likely have a low selectivity.

If statistics exist for a relevant column or index, then the optimizer will use them in its calculations. If statistics don't exist then, by default, they'll be created immediately, in order for the optimizer to consume them.

The information that makes up statistics is divided into three subsections:

- **the header** – general data about a given set of statistics
- **the density graph** – the selectivity, uniqueness, of the data, and, most importantly
- **a histogram** – a tabulation of counts of the occurrence of a particular value, taken from up to 200 data points that are chosen to best represent the complete data in the table.

It's this "data about the data" that provides the information necessary for the optimizer to make its calculations. The key measure is selectivity, i.e. the percentage of rows that pass the selection criteria. The worst possible selectivity is 1.0 (or 100%) meaning that every row will pass. The cardinality for a given operator in the plan is then simply the selectivity of that operator multiplied by the number of input rows.

The reliance the optimizer has on statistics means that your statistics need to be as accurate as possible, or the optimizer could make poor choices for the execution plans it creates. Statistics, by default, are created and updated automatically within the system for all indexes or for any column used as a Predicate, as part of a `WHERE` clause or `JOIN` criteria.

The automatic update of statistics that occurs, assuming it's on, only samples a subset of the data in order to reduce the cost of the operation. This means that, over time, the statistics can become a less-and-less-accurate reflection of the actual data. All of this can lead to SQL Server making poor choices of execution plans.

There are other statistical considerations too, around the objects types we choose to use in our SQL code. For example, table variables do not ever have statistics generated on them, so the optimizer makes assumptions about them, regardless of their actual size. Prior to SQL Server 2014, that assumption was for one row. SQL Server 2014 and SQL Server 2016 now assume one hundred rows in multi-statement user-defined functions, but remain with the one row for all other objects. SQL Server 2017 can, in some instances, use interleaved execution to arrive at more accurate row counts for these functions.

Temporary tables do have statistics generated on them and their statistics are stored in the same type of histogram as permanent tables, and the optimizer can make use of these statistics. In places where statistics are needed, say, for example, when doing a `JOIN` to a temporary table, you may see advantages in using a temporary table over a table variable. However, further discussion of such topics is beyond the scope of this book.

As you can see from all the discussion about statistics, their creation and maintenance have a large impact on your systems. More importantly, statistics have a large impact on your execution plans. For more information on this topic, check out Erin Stellato's article *Managing SQL Server Statistics* in Simple Talk (http://preview.tinyurl.com/yaae37gj).

The plan cache and plan reuse

All the processes described previously, which are required to generate execution plans, have an associated CPU cost. For simple queries, SQL Server generates an execution plan in less than a millisecond, but for very complex queries, it can take seconds or even minutes to create an execution plan.

Therefore, SQL Server will store plans in a section of memory called the plan cache, and reuse those plans wherever possible, to reduce that overhead. Ideally, if the optimizer encounters a query it has seen before, it can bypass the full optimization process and just select the plan from the cache.

However, there are a few reasons why the plan for a previously executed query may no longer be in the cache. It may have been aged out of the cache to make way for new plans, or forced out due to memory pressure, or someone manually clearing the cache. In addition, certain changes to the underlying database schema, or statistics associated with these objects, can cause plans to be recompiled (i.e. recreated from scratch).

Plan aging

Each plan has an associated "age" value that is the estimated CPU cost of compiling the plan multiplied by the number of times it has been used. So, for example, a plan with an estimated compilation cost of 10 that has been referenced 5 times has an "age" value of 50. The idea is that frequently-referenced plans that are expensive to compile will remain in the cache for as long as possible. Plans undergo a natural aging process. The **lazywriter** process, an internal process that works to free all types of cache (including the plan cache), periodically scans the objects in the cache and decreases this value by one each time.

Plans will remain in the cache unless there is a specific reason they need to be moved out. For example, if the system is under memory pressure, plans may be aged, and cleared out, more aggressively. Also, plans with the lowest age value can be forced out of the cache if the cache is full and memory is required to store newer plans. This can become a problem if the optimizer is being forced to produce a very high volume of plans, many of which are only ever used one time by one query, constantly forcing older plans to be flushed from the cache. This a problem known as *cache churn*, which we'll discuss again shortly.

Manually clearing the plan cache

Sometimes, during testing, you may want to flush all plans from the cache, to see how long a plan takes to compile, or to investigate how minor query adjustments might lead to slightly different plans. The command DBCC FREEPROCCACHE will clear the cache for all databases on the server. In a production environment, that can result in a significant and sustained performance hit because then each subsequent query is a "new" query and must go through the optimization process. We can flush only specific queries or plans by supplying a plan_handle or sql_handle. You can retrieve these values from either the plan cache itself using Dynamic Management Views (DMVs) such as sys.dm_exec_query_stats, or the Query Store (see Chapter 16). Once you have the value, simply run DBCC FREEPROCCACHE(<plan_handle>) to remove a specific plan from the plan cache.

Similarly, we can use DBCC FLUSHPROCINDB(db_id) to remove all plans for a specific database, but the command is not officially documented. SQL Server 2016 introduced a new, fully-documented method to remove all plans for a single database, which is to run the following command within the target database:

```
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE
Criteria for plan reuse
```

When we submit a query to the server, the algebrizer process creates a hash value for the query. The optimizer stores the hash value in the QueryHash property of the associated execution plan (covered in more detail in Chapter 2). The job of the QueryHash is to identify queries with the same, or very similar logic (there are rare cases where logically different queries end up with the same hash value, known as hash collisions).

For each submitted query, the optimizer looks for a matching QueryHash value among the plans in the plan cache. If found, it performs a detailed comparison of the SQL text of the submitted query and SQL text associated with the cached plan. If they match exactly (including spaces and carriage returns) this returns the plan_handle, a value that uniquely identifies the plan in memory. This plan may be reused, if the following are also true:

- **the plan was created using the same SET options** (see Chapter 2) – otherwise there will be multiple plans created even if the SQL texts are identical
- **the database IDs match** – identical queries against different databases will have separate plans.

Note that it's also possible that lack of schema-qualification for the referenced objects in the query will lead to separate plans for different users.

Generally, however, a plan will be reused if all four of the above match (QueryHash, SQL text, SET options, database ID). If so, the entire cost of the optimization process is skipped and the execution plan in the plan cache is reused.

Avoiding cache churn: query parameterization

It is an important best practice to write queries in such a way that SQL Server can reuse the plans in cache. If we submit ad hoc queries to SQL Server and use hard-coded literal values then, for most of those queries, SQL Server will be forced to complete the full optimization process and compile a new plan each time. On a busy server, this can quickly lead to cache bloat, and to older plans being forced relatively quickly from the cache.

For example, let's say we submit the query in Listing 1-2.

```
SELECT  p.ProductID ,
        p.Name AS ProductName ,
        pi.Shelf ,
        l.Name AS LocationName
FROM    Production.Product p
        INNER JOIN Production.ProductInventory AS pi
             ON pi.ProductID = p.ProductID
        INNER JOIN Production.Location AS l
             ON l.LocationID = pi.LocationID
WHERE   l.Name = 'Paint';
GO
```

Listing 1-2

We then submit the same query again, but for a different location name (say, 'Tool Cribs' instead of 'Paint'). This will result in two separate plans stored in cache, even though the two queries are essentially the same (they will have the same QueryHash values, assuming no other changes are made).

To ensure plan reuse, it's best to use either stored procedures or parameterized queries, where the variables within the query are identified with parameters, rather than hard-coded literals, and we simply pass in the required parameter values at runtime. This way, the SQL text the optimizer sees will be "set in stone," maximizing the possibility of plan reuse.

These are also called "prepared queries" and are built from the application code. For an example of using prepared statements, see this article in Technet (http://preview.tinyurl.com/ybvc2vcs). You can also parameterize a query by using sp_executesql from within your T-SQL code.

Another way to mitigate the churn from ad hoc queries is to use a server setting called **Optimize For Ad Hoc Workloads**. Turning this on will cause the optimizer to create what is known as a "plan stub" in the plan cache, instead of putting the entire plan there the first time a plan is created. This means that single-use plans will take up radically less memory in your plan cache.

Plan recompilation

Certain events and actions, such as changes to an index used by a query, can cause a plan to be **recompiled**, which simply means that the existing plan will be marked for recompilation, and a new plan generated the next time the query is called. It is important to remember this, because recompiling execution plans can be a very expensive operation. This only becomes a problem if our actions as programmers force SQL Server to perform excessive recompilations.

We'll discuss recompiles in more detail in Chapter 9, but the following actions can lead to recompilation of an execution plan (see http://preview.tinyurl.com/y947r969 for a full list):

- changing the structure of a table, view or function referenced by the query
- changing, or dropping, an index used by the query
- updating the statistics used by the query
- calling the function `sp_recompile`
- mixing DDL and DML within a single batch
- changing certain `SET` options within the T-SQL of the batch
- changes to cursor options within the query
- deferred compiles
- changes to a remote rowset if you're using a function like `OPENQUERY`.

Getting Started with Execution Plans

Execution plans assist us in writing efficient T-SQL code, troubleshooting existing T-SQL behavior or monitoring and reporting on our systems. How we use them and view them is up to us, but first we need to understand what information is contained within the plans, and how to interpret that information. One of the best ways to learn about execution plans is to see them in action, so let's get started.

Permissions required to view execution plans

In order to view execution plans for queries you must have the correct permissions within the database. If you are `sysadmin`, `dbcreator` or `db_owner`, you won't need any other permission. If you are granting this permission to developers who will not be in one of those privileged roles, they'll need to be granted the `ShowPlan` permission within the database being tested. Run the statement in Listing 1-3.

```
GRANT SHOWPLAN TO [username];
```

Listing 1-3

Substituting the username will enable the user to view execution plans for that database. Additionally, in order to run the queries against the Dynamic Management Objects (DMO), either `VIEW SERVER STATE` or `VIEW DATABASE STATE`, depending on the DMO in question, will be required. We'll explore DMOs more in Chapter 15.

Execution plan formats

SQL Server can output the execution plan in three different ways:

- as an XML plan
- as a text plan
- as a graphical plan.

The one you choose will depend on the level of detail you want to see, and on the methods used to capture or view that plan.

In each format, we can retrieve the execution plan without executing the query, (so without runtime information), which is known as the **estimated plan**, or we can retrieve the plan with added runtime information, which of course requires executing the query, and is known as the **actual plan**. While, strictly speaking, the terms *actual* and *estimated* are exclusive to graphical plans, it is common to see them applied to all execution plan formats and, for simplicity, we'll use those terms for each format here.

XML plans

XML plans present a complete set of data available on a plan, all on display in the structured XML format. The XML format is great for transmitting to other data professionals if you want help on an execution plan or need to share with coworkers. Using XQuery, we can also query the XML data directly (see Chapter 13).

We can use one of the following two commands to retrieve the plan in XML format:

- **`SET SHOWPLAN_XML ON`** – generates the estimated plan (i.e. the query is not executed).
- **`SET STATISTICS_XML ON`** – generates the actual execution plan (i.e. with runtime information).

XML plans are extremely useful, but mainly for querying, not for standard-style reading of plans, since the XML is not human readable. Useful though these types of plan are, you're more likely to use graphical plans for simply browsing the execution plan.

Every graphical execution plan is actually XML under the covers. Within SSMS, simply right-click on the plan itself. From the context menu select **Show Execution Plan XML…** to open a window with the XML of the execution plan.

Text plans

These can be quite difficult to read, but detailed information is immediately available. Their text format means that they we can copy or export them into text manipulation software such as NotePad or Word, and then run searches against them. While the detail they provide is immediately available, there is less detail overall from the execution plan output in these types of plan, so they can be less useful than the other plan types.

Text plans are on the deprecation list from Microsoft. They will not be available in a future version of SQL Server. I don't recommend using them.

Nevertheless, here are the possible commands we can use to retrieve the plan in text format:

- **`SET SHOWPLAN_ALL ON`** – retrieves the estimated execution plan for the query.
- **`SET STATISTICS PROFILE ON`** – retrieves the actual execution plan for the query.
- **`SET SHOWPLAN_TEXT ON`** – retrieves the estimated plan but with a very limited set of data, for use with tools like **osql.exe.**

Graphical plans

Graphical plans are the most commonly viewed format of execution plan. They are quick and easy to read. We can view both estimated and actual execution plans in graphical format and the graphical structure makes understanding most plans very easy. However, the detailed data for the plan is hidden behind **Tooltips** and **Property** sheets, making it somewhat more difficult to get to, other than in a one-operator-at-a-time approach.

Retrieving cached plans

There is some confusion regarding the different types of plan and what they really mean. I've heard developers talk about estimated and actual plans as if they were two completely different plans. Hopefully this section will clear things up. The salient point is that the query optimizer produces the plan, and there is only one valid execution plan for a query, at any given time.

When troubleshooting a long-running query retrospectively, we'll often need to retrieve the **cached plan** for that query from the plan cache. As discussed earlier, once the optimizer selects a new plan for a query, it places it in the plan cache, and passes it on to the query execution engine for execution. Of course, the optimizer never executes any queries, it merely formulates the plan based on its knowledge of the underlying data structures and statistical knowledge of the data. Cached plans don't contain any runtime information, except for the row counts in interleaved plans.

We can retrieve this cached plan manually, via the Dynamic Management Objects, or using a tool such as Extended Events. We'll cover techniques to automate capture of the cached plan later in the book (Chapter 15).

Plans for ad hoc queries: estimated and actual plans

Most of the time in this book, however, we'll retrieve the execution plan simply by executing ad hoc queries within SSMS. At the point we submit the query, we have the option to request either the **estimated plan** or the **actual plan**.

If we request the **estimated** plan, we do not execute the query; we merely submit the query for inspection by the optimizer, in order to see the associated plan. If there exists in the plan cache a plan that exactly matches the submitted query text, then the optimizer simply returns that cached plan. If there is no match, the optimizer performs the optimization process and returns the new plan. However, because there is no intent to execute the query, the next two steps are skipped (i.e. placing the plan in the cache, if it's a new plan, and sending it for execution). Since estimated plans never access data, they are very useful during development for testing large, complex queries that could take a long time to run.

If, when we submit the query, we request a plan *with* runtime information, (what SSMS refers to as an **actual** plan), then all three steps in the process are performed.

If there is a cached plan that exactly matches the submitted query text, then the optimizer simply passes the cached plan to the execution engine, which executes it, and adds the requested runtime values to the displayed plan. If there is no cached plan, the optimizer produces a new plan, places it in the cache and passes it on for execution and, again, we see the plan with runtime information. For example, we'll see runtime values for the number of rows returned and the number of executions of each operator, alongside the optimizer's estimated values. Note that SQL Server does not store anywhere a second copy of the plan with the runtime information. These values are simply injected into the copy of the plan, whether displayed in SSMS, or output through other means.

Will the estimated and actual plans ever be different?

Essentially, the answer to this is "No." As emphasized previously, there is only one valid execution plan for a query at any given time, and the estimated and actual plans will not be different.

You may see differences in parallelization between the runtime plan and the estimated plan, but this doesn't mean the execution engine "changed" the plan. At compile time, if the optimizer calculates that the cost of the plan might exceed the cost threshold for parallelism, then it produces a parallel version of the plan (see Chapter 11). However, the engine gets the final say on whether the query is executed in parallel, based on current server activity and available resources. If resources are too scarce, it will simply strip out the parallelism and run a serial version of the plan.

Sometimes, you might generate an estimated plan and then, later, an actual plan for the same query, and see that the plans are different. In fact, what will have happened here is that, in the time between the two requests, something happened to invalidate the existing plan in the cache, forcing the optimizer to perform a full optimization and generate a new plan. For example, changes in the data or data structures might have caused SQL Server to recompile the plan. Alternatively, processes or objects within the query, such as interleaving Data Definition Language (DDL) and data manipulation language (DML), result in a recompilation of the execution plan.

If you request an actual plan and then retrieve from the cache the plan for the query you just executed (we'll see how to do that in Chapter 9), you'll see that the cached plan is the same as your actual plan, except that the actual plan has runtime information.

One case where the estimated and actual plans will be genuinely different is when the estimated plan won't work at all. For example, try generating an estimated plan for the simple bit of code in Listing 1-4.

```
CREATE TABLE TempTable
    (
      Id INT IDENTITY(1, 1) ,
      Dsc NVARCHAR(50)
    );
INSERT  INTO TempTable
        ( Dsc
        )
        SELECT  [Name]
        FROM    [Sales].[Store];
SELECT  *
FROM    TempTable;
DROP TABLE TempTable;
```

Listing 1-4

You will get this error:

```
Msg 208, Level 16, State 1, Line 7
Invalid object name 'TempTable'.
```

The optimizer runs the statements through the algebrizer, the process outlined earlier that is responsible for verifying the names of database objects but, since SQL Server has not yet executed the query, the temporary table does not yet exist.

The plan will get marked for deferred name resolution. In other words, while the batch is parsed, bound, and compiled, the `SELECT` query is excluded from compilation because the algebrizer has marked it as deferred. Capturing the estimated plan doesn't execute the query, and so doesn't create the temporary table, and this is the cause of the error. At runtime, the query will be compiled and now a plan does exist. If you execute Listing 1-4 and request the actual execution plan, it will work perfectly.

A second case where the estimated and actual plans will be different, new in SQL Server 2017, is when the optimizer uses interleaved execution. If we request an estimated plan for a query that contains a multi-statement table valued function (MSTVF), then the optimizer will use a fixed cardinality estimation of 100 rows for the MSTVF. However, if we request an actual plan, the optimizer will first generate the plan using this fixed estimate, and then run the subtree containing the MSTVF to get the actual row counts returned, and recompile the plan based on these real row counts. Of course, this plan will be stored in the plan cache, so subsequent requests for either an estimated or an actual plan will return the same plan.

Capturing graphical plans in SSMS

In SSMS, we can capture both the estimated and the actual plans for a query, and there are several ways to do it, in each case. Perhaps the most common, or at least the route I usually take, is to use the icons in the toolbar. Figure 1-2 shows the **Display Estimated Execution Plan** icon.

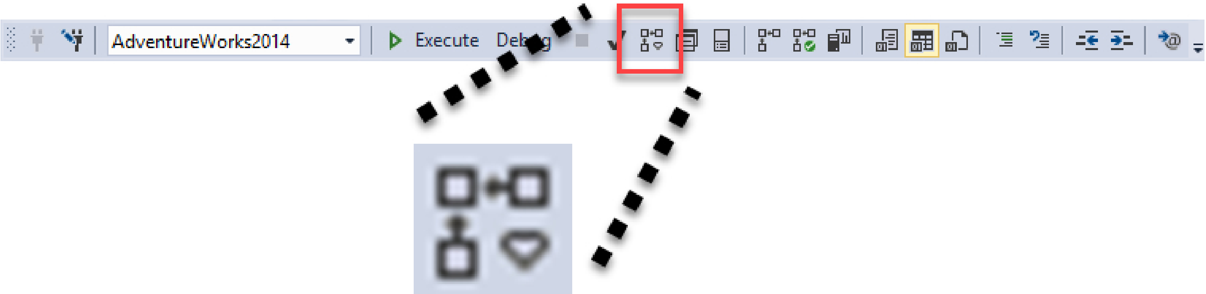


Figure 1-2: Capturing the estimated plan.

A few icons to the right, we have the **Include Actual Execution Plan** icon, as shown in Figure 1-3.

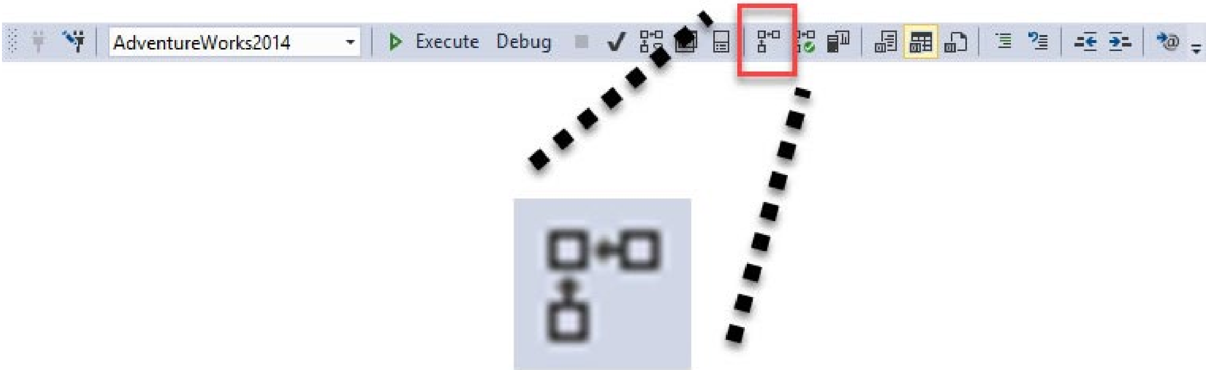


Figure 1-3: Capturing the actual plan.

Alternatively, for either type of plan, you could:

- right-click in the query window and select the same option from the context menu
- click on the **Query** option in the menu bar and select the same choice
- use the keyboard shortcut (**CTRL+L** for estimated; **CTRL+M** for actual within SSMS or **CTRL+ALT+L** and **CTRL+ALT+M** for the same within Visual Studio).

For estimated plans, we have to click the icon, or use one of the alternative methods, each time we want to capture that type of plan for a query. For the actual plan, each of these methods acts as an "on/off" switch for the query window. When the actual plan is switched on, at each execution, SQL Server will then capture an actual execution plan for all queries run from that window, until you turn it off again for each query window within SSMS.

Finally, there is one additional way to view a graphical execution plan, a live execution plan. The view of the plan is based on a DMV, `sys.dm_exec_query_statistics_xml`, introduced in SQL Server 2014. This DMV returns live statistics for the operations within an execution plan. The graphical view of this DMV was introduced in SQL Server 2016. You toggle it on or off similarly to what you do with an actual execution plan. Figure 1-4 shows the button.

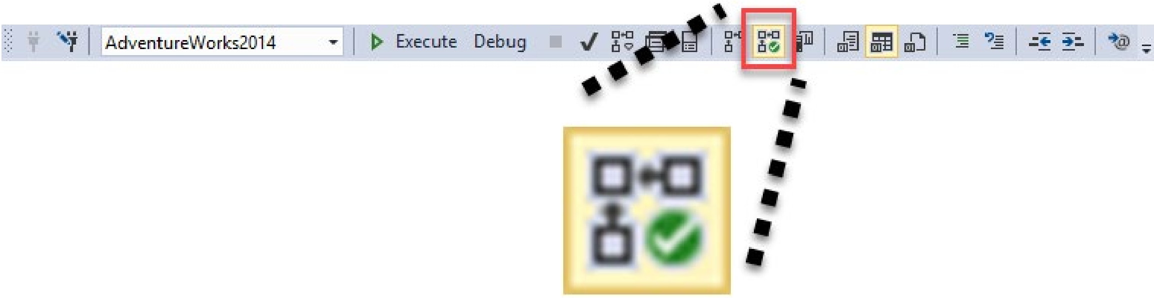


Figure 1-4: Enabling the live execution plan.

We'll explore this completely in Chapter 17.

Capturing our first plan

It's time to capture our first execution plan. We'll start off with a relatively simple query that nevertheless provides a fairly complete view into everything you're going to do when reading execution plans.

As noted in the introduction to this book, we strongly encourage you to follow along with the examples, by executing the relevant script and viewing the plans. Occasionally, especially as we reach more complex examples later in the book, you may see a plan that differs from the one presented in the book. This might be because we are using different versions of SQL Server (different service pack levels and cumulative updates), different editions, or we are using slightly different versions of the **AdventureWorks** sample database. We use `AdventureWorks2016` in this book; other versions are slightly different, and even if you use the same version, its schema or statistics may have been altered over time. So, while most of the plans you get should be very similar, if not identical, to what we display here, don't be too surprised if you try the code and see something different.

Open a new query tab in SSMS and run the query shown in Listing 1-5.

```
USE AdventureWorks2014;
GO
SELECT p.LastName + ', ' + p.FirstName,
    p.Title,
    pp.PhoneNumber
FROM Person.Person AS p
    INNER JOIN Person.PersonPhone AS pp
```

```
        ON pp.BusinessEntityID = p.BusinessEntityID
    INNER JOIN Person.PhoneNumberType AS pnt
        ON pnt.PhoneNumberTypeID = pp.PhoneNumberTypeID
WHERE pnt.Name = 'Cell'
      AND p.LastName = 'Dempsey';
GO
```

Listing 1-5

Click the **Display Estimated Execution Plan** icon and in the execution plan tab you will see the estimated **execution plan**, as shown in Figure 1-5.

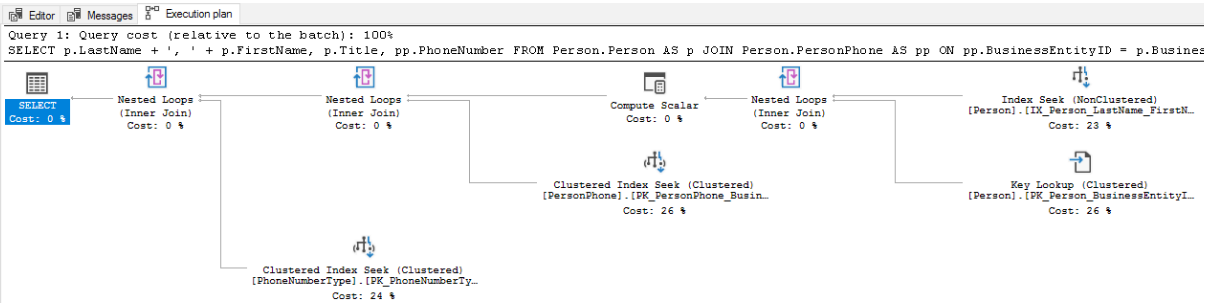


Figure 1-5: Estimated execution plan.

Notice that there is no **Results** tab, because we have not actually executed the query. Now, highlight the **Include Actual Execution Plan** icon and execute the query. This time you'll see the result set retuned (a single row) and the **Execution plan** tab will display the actual execution plan, which should also look as shown in Figure 1-5.

The components of a graphical execution plan

We're now going to explore each section of the plan from Figure 1-5 in more detail, but still at a high level. We won't start exploring the details of individual operators until Chapter 3. You'll notice that it's rather difficult to read the details on the plan in Figure 1-5. Here, and throughout the book we'll be following a method where I show the whole plan, and then drill down into sections of the plan to discuss individual parts or segments of the plan.

Most people start on the right-hand side, when reading plans, where you will find the operators that read data out of the base tables and indexes. From there we follow the data flow, as indicated by the arrows, from right to left until it reaches the **SELECT** operator, where the rows are passed back to the client. However, it's equally valid to read the plan from left to right, which is the order in which the operators are called – essentially data is pulled from right to left as each operator in turn calls the child operator on its right, but we'll discuss this in more detail in Chapter 3.

Operators

Operators, represented as icons in the plan, are the workhorses of the plan. Each operator implements a specific algorithm designed to perform a specialized task. The operators in a plan tell us exactly how SQL Server chose to execute a query, such as how it chose to access the data in a certain table, how it chose to join that data to rows in a second table, how and where it chose to perform any aggregations, sorting, calculations, and so on.

In this example, let's start on the right-hand side of the plan, with the operators shown in Figure 1-6.

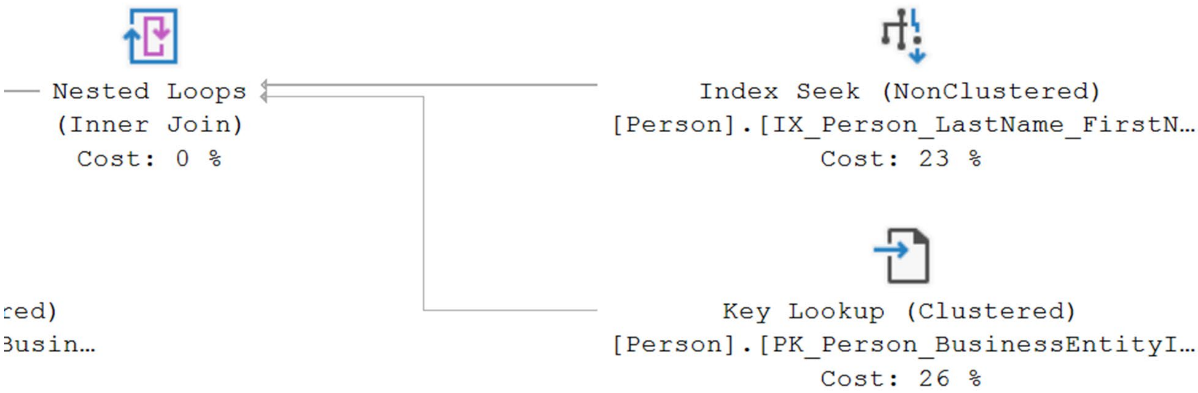


Figure 1-6: Two data access operators and a join operator.

Here we see two data access operators passing data to a join operator. The first operator is an **Index Seek**, which is pulling data from the `Person` table using a nonclustered index, `Person.IX_Person_LastName_FirstName_MiddleName`. Each qualifying row (rows where the last name is Dempsey) passes to a **Nested Loops** operator, which is going to pull additional data, not held in the nonclustered index, from the **Key Lookup** operator.

Each operator has both a physical and a logical element. For example, in Figure 1-6, **Nested Loops** is the physical operator, and **Inner Join** is the logical operation it performs.

So the logical component describes what the operator actually does (an INNER JOIN operation) and the physical part is how the optimizer chose to implement it (using a Nested Loops algorithm).

From the first **Nested Loops** operator, the data flows to a **Compute Scalar** operator. For each row, it performs its required task (in this case, concatenating the first and last names with a comma) and then passes it on to the operator on its left. This data is joined with matching rows in the PersonPhone table, and then in turn with matching rows in the PhoneNumberType table. Finally, the data flows to the **SELECT** operator.

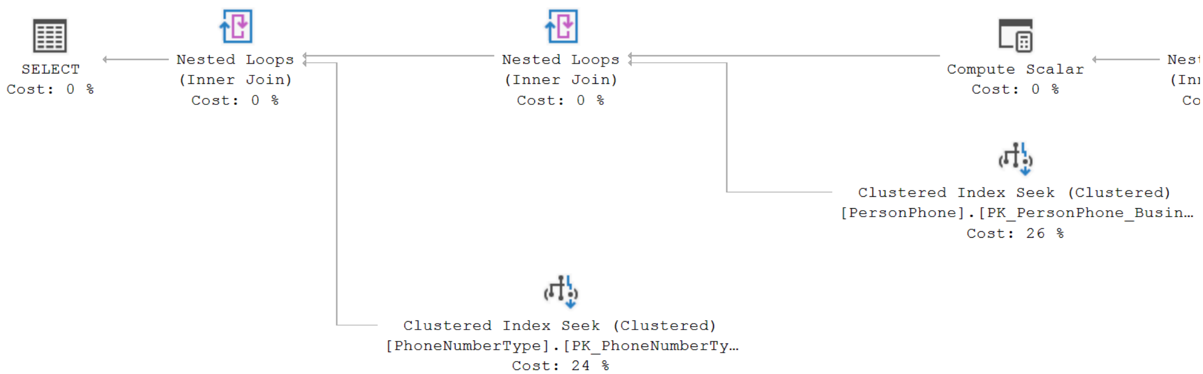


Figure 1-7: Broader section of the plan showing more operators.

The **SELECT** icon is one that you're going to frequently reference for the important data it contains. Of course, every operator contains important data (see the *Operator properties* section, a little later), but what sets the **SELECT** operator apart is that it contains data about the plan as a whole, whereas other icons only expose information about the operator itself.

Data flow arrows

The arrows represent the direction of data flow between the operators, and the thickness of the arrow reflects the amount of data passed, a thicker arrow meaning more rows. Arrow thickness is another visual clue as to where performance issues may lie. For example, you might see a big thick arrow emerging from a data access operator, on the right side of the plan, but very thin arrows on the left, since your query ultimately returns only two rows. This is a sign that a lot of data was processed to produce those two output rows. That may be unavoidable for the functional requirements of the query, but equally it might be something you can avoid.

You can hover with the mouse pointer over these arrows and it will show the number of rows that it represents in a tooltip that you can see in Figure 1-8. In an execution plan that contains runtime statistics (the actual plan), the thickness is determined by the actual, rather than the estimated, number of rows.

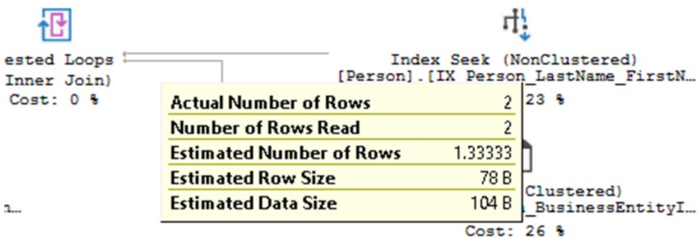


Figure 1-8: Tooltip for the data flow arrow.

Estimated operator costs

Below each individual icon in a plan is displayed a number as a percentage. This number represents the *estimated* cost for that operator relative to the *estimated* cost of the plan as a whole. These numbers are best thought of as "cost units," based on the mathematical calculations of anticipated CPU and I/O within the optimizer. The estimated costs are useful as measures, but these costs don't represent real-world measures of actual CPU and I/O. There is generally a correlation between high estimated cost within the plan, and higher actual performance costs, but these are still just estimated values.

The origin of the estimated cost values

The story goes that the developer tasked with creating execution plans in SQL Server 7 used his workstation as the basis for these numbers, and they have never been updated. See Danny Ravid's blog at: http://preview.tinyurl.com/yawet2l3.

All operators will have an associated cost, and even an operator displaying 0% will actually have a small associated cost, which you can see in the operator's properties (which we'll discuss shortly).

If you compare the operator- and plan-costs side by side for the estimated and actual plan of the same query, you'll see that they are identical. **Only the optimizer generates these cost values**, which means that **all costs in all plans are estimates**, based on the optimizer's statistical knowledge of the data.

Estimated total query cost relative to batch

At the top of every execution plan is displayed as much of the query string as will fit into the window, and a "cost (relative to the batch)" of 100%.

```
Query 1: Query cost (relative to the batch): 100%
SELECT p.LastName + ', ' + p.FirstName , p.Title , pp.PhoneNumber FROM Pers
```

Figure 1-9: Query and the estimated query cost at the top of the execution plan.

Just as each query can have multiple operators, and each of those operators will have a cost relative to the query, you can also run multiple queries within a batch and get execution plans for them. Each plan will then have different costs. The estimated cost of the total query is divided by the estimated cost of all queries in a batch. Each operator within a plan displays its estimated costs relative to the plan it's a part of, not to the batch as a whole.

Never lose sight of the fact that the costs you see, even in actual plans, are an estimated cost, not real, measured, performance metrics. If you focus your tuning efforts exclusively on the queries or operators with high estimated costs, and it turns out the cost estimations are incorrect, then you may be looking in the wrong area for the cause of performance issues.

Operator properties

Right-click any icon within a graphical execution plan and select the **Properties** menu item to get a detailed list of information about that operation. Each operator performs a distinct task and therefore each operator will have a distinct set of property data. The vast majority of useful information to help you read and understand execution plans is contained in the **Properties** window for each operator. It's a good habit to get into when reading an execution plan to just leave the **Properties** window open and pinned to your SSMS window at all times. Sadly, due to the vagaries of the SSMS GUI, you may sometimes have to click two places to get the properties you want to properly display.

Figure 1-10 compares the **Properties** window for the same **Index Seek** operator at the top right of Figure 1-5, which performs a seek operation on a nonclustered index on the `Person` table. The left-hand pane is from the estimated plan, and the right-hand pane is for the actual plan.

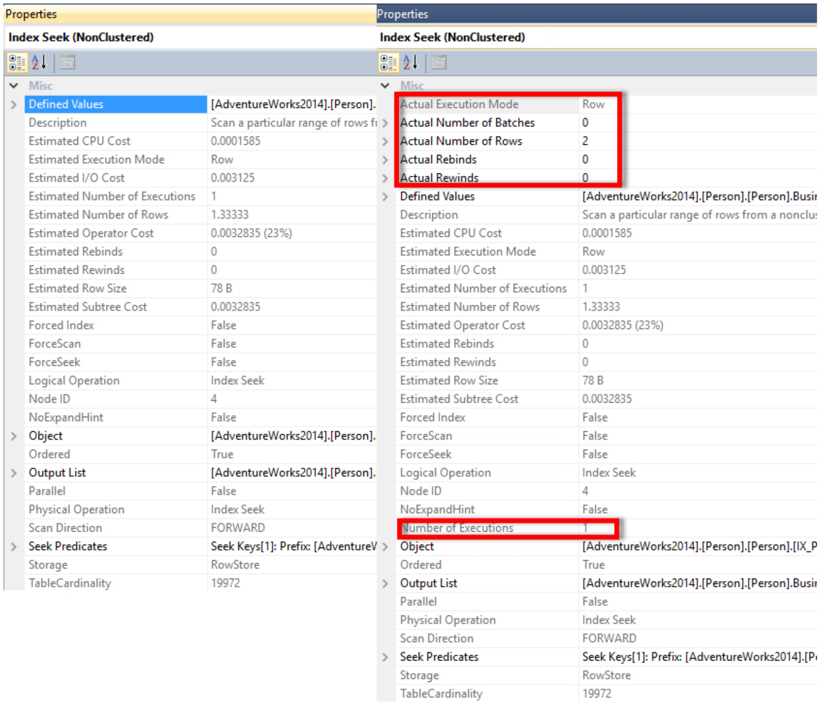


Figure 1-10: Comparing properties of the Index Seek operator for the estimated and actual plans.

As you can see, in the actual plan we see the actual, as well as the estimated, number of rows that passed through that operator, as well as the actual number of times the operator was executed. Here we see that the optimizer estimated 1.3333 rows and 2 were actually returned.

When comparing the properties of an operator, for the estimated and actual plans, look out for very big differences between the estimated and the actual number of rows returned, such as an estimated row count of 100 and an actual row count of 100,000 (or vice versa). If a query that returns hundreds of thousands of rows uses a plan the optimizer devised for returning 10 rows, it is likely to be very inefficient, and you will need to investigate the possible cause. It might be that the row count has changed significantly since the plan was generated but statistics have not yet auto-updated, or it might be caused by problems with parameter sniffing, or by other issues. We'll return to this topic in detail in Chapter 9.

I'm not going into detail here on all the properties and their meanings, but I'll mention briefly a few that you'll refer to quite frequently:

- **Actual Number of Rows** – the true number of rows returned according to runtime statistics. The availability of this value in actual plans is the biggest difference between these and cached plans (or estimated plans). Look out for big differences between this value and the estimated value.
- **Defined Values –** values introduced by this operator, such as the columns returned, or computed expressions from the query, or internal values introduced by the query processor.
- **Estimated Number of Rows** – calculated based on the statistics available to the optimizer for the table or index in question. These are useful for comparing to the **Actual Number of Rows**.
- **Estimated Operator Cost –** the estimated operator cost as a figure (as well as a percentage). This is an estimated cost even in actual plans.
- **Object** – the object accessed, such as the index being accessed by a scan or a seek operation.
- **Output List** – columns returned.
- **Predicate –** a "pushed down" search Predicate.
- **Table Cardinality** – number of rows in the table.

You'll note that some of the properties, such as **Object**, have a triangle icon on their left, indicating that they can be expanded. Some of the longer property descriptions have an ellipsis at the end, which allows us to open a new window, making the longer text easier to read. Almost all properties, when you click on them, display a description at the bottom of the **Property** pane.

All these details are available to help us understand what's happening within the query in question. We can walk through the various operators, observing how the subtree cost accumulates, how the number of rows changes, and so on. With these details, we can identify queries that are estimated to use excessive amounts of CPU or tables that need more indexes, or identify other performance issues.

Tooltips

Associated with each of the icons and the arrows is a pop-up window called a **tooltip**, which you can access by hovering your mouse pointer over the icon or arrow. I already used one of these in Figure 1-8. Essentially, the tooltip for an operator is a cut-down version of the full

Properties window. It's worth noting that the tooltip and the properties for given operators change as SQL Server itself changes. You may see differences in the tooltips between one version of SQL Server and the next. Most of the examples in this book are from SQL Server 2016.

Figure 1-11 shows the tooltip window for the **SELECT** operator for the estimated execution plan for the query in Listing 1-4.

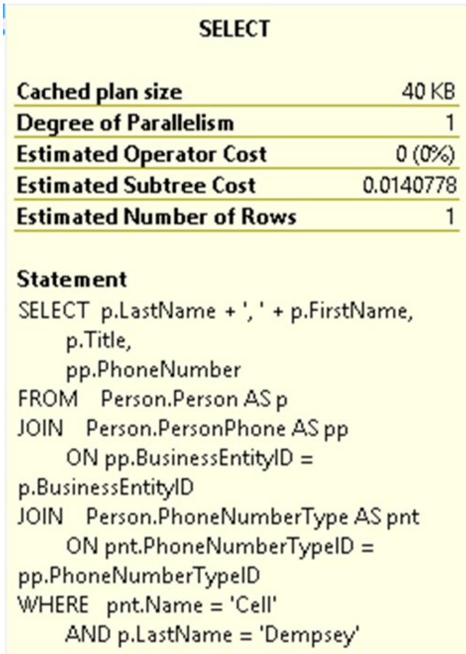


Figure 1-11: Tooltip for the **SELECT** operator.

The properties of the **SELECT** operator are often particularly interesting, since this provides information relating to the plan as a whole. For example, we see the following two property values (among others, several of which we'll review in detail in Chapter 2):

- **Cached plan size** – how much memory the plan generated by this query will take up in the plan cache. This is a useful number when investigating cache performance issues because you'll be able to see which plans are taking up more memory.
- **Degree of Parallelism** – whether this plan was designed to use (or did use) multiple processors. This plan uses a single processor as shown by the value of 1. (See Chapter 11.)

In Figure 1-11, we also see the **statement** that represents the entire query that SQL Server is processing. You may not see the **statement** if it's too long to fit into the tooltip window. The same thing applies to other properties in other operators. This is yet another reason to focus on using the **Properties** window when working with execution plans.

The information available in the tooltips can be extremely limited. But, it's fairly quick to see the information available in them since all you have to do is hover your mouse to get the tips. To get a more consistent and more detailed view of information about the operations within an execution plan, you should use the full **Properties** window.

Saving execution plans

We can save an execution plan from the graphical execution plan interface by right-clicking within the execution plan and selecting **Save Execution Plan As**. Way back in SQL Server 2005, we then had to change the filter to "*.*" and, when typing the name of the file we wanted to save, add **.sqlplan** as the extension. Thankfully, SQL Server 2008, and later, automatically selects the **.sqlplan** file type.

What we are saving is simply an XML file. One of the benefits of extracting an XML plan and saving it as a separate file is that we can share it with others. For example, we can send the XML plan of a slow-running query to a DBA friend and ask them their opinion on how to rewrite the query. Once the friend receives the XML plan, he or she can open it up in Management Studio and review it as a graphical execution plan.

You can look at the underlying XML of a plan as well by right-clicking on the plan and selecting **Show Execution Plan XML** from the context menu. That will open the raw XML in another window where you can browse the XML manually if you like. Alternatively, you can open the **.sqlplan** file in Notepad. We'll explore the XML within execution plans in detail in Chapter 13.

Summary

In this chapter, we've described briefly the role of the query optimizer in producing the execution plan for a query, and how it selects the lowest-cost plan, based on its knowledge of the data structures and statistical knowledge of the data distribution. We also covered the plan cache, the importance of plan reuse, and how to promote this.

We explored the different execution plan formats, and then focused on graphical execution plans, how to read these plans, and the various components of these plans. We are going to spend a lot of time within the graphical plans when interpreting individual execution plans, so understanding the information available within the plans is important.

I also tried to clear up any confusion regarding what the terms "estimated plan" and "actual plan" really mean. I've even heard people talk about "estimated and actual plans" as if they were two completely different plans, or that the estimated plan might be somehow "inaccurate." Hopefully this chapter dispelled those misunderstandings.

Chapter 2: Getting Started Reading Plans

The aim of this chapter is to show you how to start reading graphical execution plans. We're still going to stay relatively high level, using a few simple queries and basic filters to explain the mechanics of reading a plan, and what to look for in a plan. In subsequent chapters, we'll start drilling down into the details of the various individual operators and their properties.

Specifically, we'll cover:

- **a brief review of most common execution plan operators** – categorized per their basic function.
- **the basics of how to read a graphical plan** – do we read a plan right to left, or left to right? Both!
- **what to look for in a plan** – a few key warning signs and operator properties that can often help rapidly identify potential issues.
- **the SELECT operator** – contains a lot of useful information about the plan as a whole.

The Language of Execution Plans

In some ways, learning how to read execution plans is like learning a new language, except that this language is based on a series of operators, each of which is represented as an icon in a graphical plan. Fortunately, unlike a language, the number of words (operators) we must learn is minimal. There are approximately 85 available operators and most queries use only a small subset of them.

Common operators

Books Online (http://preview.tinyurl.com/y97wndcf) lists all the operators in (sort of) alphabetical order. This is fine as a reference, but it isn't the easiest way to learn them, so we will forgo being "alphabetically correct" here.

A graphical execution plan displays three distinct types of operator:

- **Physical Operators (and their associated logical operations)** appear as blue-based icons and represent query execution. They include DML and parallelism operators. These are the only type of operator you'll see in an actual execution plan.
- **Cursor Operators** have yellow icons and represent T-SQL cursor operations.
- **Language Elements** are green icons and represent T-SQL language elements, such as `ASSIGN`, `DECLARE`, `IF`, `WHILE`, and so on.

The focus of this chapter, and of the book, is on the physical operators and their corresponding logical operations. However, we will also cover cursor operators in Chapter 14, and there will be a few dives into some of the special information available in the language element operators.

A physical operator represents the physical algorithm chosen by the optimizer to implement the required logical operation. Every physical operator is associated with one or more logical operations. Generally, the name of the physical operator will be followed in brackets by the name of the associated logical operation (although Microsoft isn't entirely consistent about this). For example, **Nested Loops (Inner Join)**, where **Nested Loops** is the physical implementation of the logical operation, **Inner Join**.

The optimizer has at its disposal sets of operators for reading data, combining data, ordering and grouping data, modifying data, and so on. Each operator performs a single, specialized task. The following table lists some of the more common physical operators, categorized according to their basic purpose.

Reading data	**Combining data**	**Grouping and ordering data**
Table/Index Scan	Nested Loops	Sort
Index Seek	Merge Join	Stream Aggregate
Lookup	Hash Match	Hash Match (Aggregate)
Constant Scan	Adaptive Join	Window Aggregate
	Sequence	Segment
	Concatenation	Window Spool
	Switch	

Manipulating data	Modifying data	Performance
Compute Scalar	Table/Index Insert	Bitmap
Filter	Table/Index Update	Spools
Top	Table/Index Delete	Parallelism
Sequence Project	Table/Index Merge	
	Assert	
	Split	
	Collapse	

Which plan operators you see most frequently as a developer or DBA depends a lot on the nature of the workload. For an OLTP workload you will hope to see a lot of **Index Seek** and **Nested Loops** operators, characteristic of frequent queries that return relatively small amounts of data. For a BI system, you are likely to see more **Index Scans**, since these are often more efficient when reading a large proportion of data in a table, and **Merge Join** or **Hash Match** joins, which are join algorithms that become more efficient when joining larger data streams.

Understanding all the internal mechanisms of a given operator is only possible if you run a debugger on SQL Server. I absolutely do not recommend that you do this, but if you're looking for deep knowledge of operator internals, then I recommend Paul White's blog (http://preview.tinyurl.com/y75n6f5z).

Generally, however, we can learn a lot about what an operator is doing by observing how they function and relate to one another within execution plans. The key is to start by trying to understand the basic mechanics of the plan as a whole, and then drill down into the "interesting" operators. These might be the operators with the highest estimated cost, such as a high-cost Index Scan or seek, or it might be a "blocking" operator such as a **Sort** (more on blocking versus streaming operators shortly). Having chosen a starting point, look at the properties of these operators, where all the details about the operator are available. Each operator has a different set of characteristics. For example, they manage memory in different ways. Some operators, primarily **Sort**, **Hash Match**, and **Adaptive Join**, require a variable amount of memory in order to execute. As such, a query with one of these operators may have to wait for available memory prior to execution, possibly adversely affecting performance.

Reading a plan: right to left, or left to right?

Should we read an execution plan from right to left, or from left to right? The answer, as we discussed briefly in Chapter 1, is that we generally read execution plans from right to left, following the data flow arrows, but that it is equally valid, and frequently helpful, to read from left to right.

Let's take a look at a very simple example. Listing 2-1 shows a simple query against the `AdventureWorks2014` database, retrieving details from the `Person.Person` table, within a certain date range.

```
SELECT TOP ( 5 )
        BusinessEntityID ,
        PersonType ,
        NameStyle ,
        Title ,
        FirstName ,
        LastName ,
        ModifiedDate
FROM    Person.Person
WHERE   ModifiedDate >= '20130601'
        AND ModifiedDate <= CURRENT_TIMESTAMP ;
```

Listing 2-1

Figure 2-1 shows the resulting execution plan.

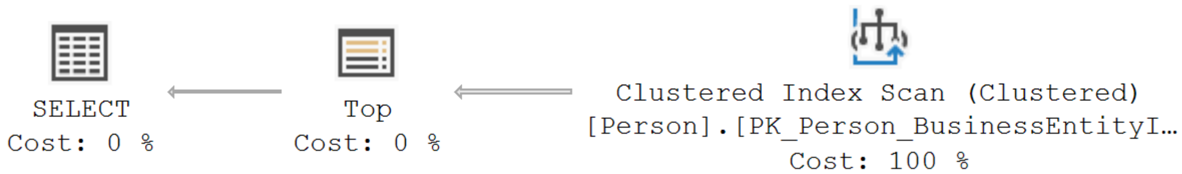


Figure 2-1: Simple execution plan, read right to left.

If we read the plan from right to left, following the data flow direction, the first action in the plan is to read the data from the `Person` table, via a **Clustered Index Scan**. The data passes to the **Top** operator, which in turn passes the first five rows back to the `SELECT`. This is a perfectly valid way to read the plan, and is the way most people read one. However, this data flow order could imply that, first, the **Clustered Index Scan** reads the data in the `Person` table and passes on the rows that match the search condition in the `WHERE` clause (there are over 13 K qualifying rows), and then the **Top** operator only sends on the first five.

Of course, this would be highly inefficient, and is not what happens, as you can tell from the thin arrow between the **Clustered Index Scan** and **Top** operators. The **Clustered Index Scan** only reads 5 rows from the `Person` table.

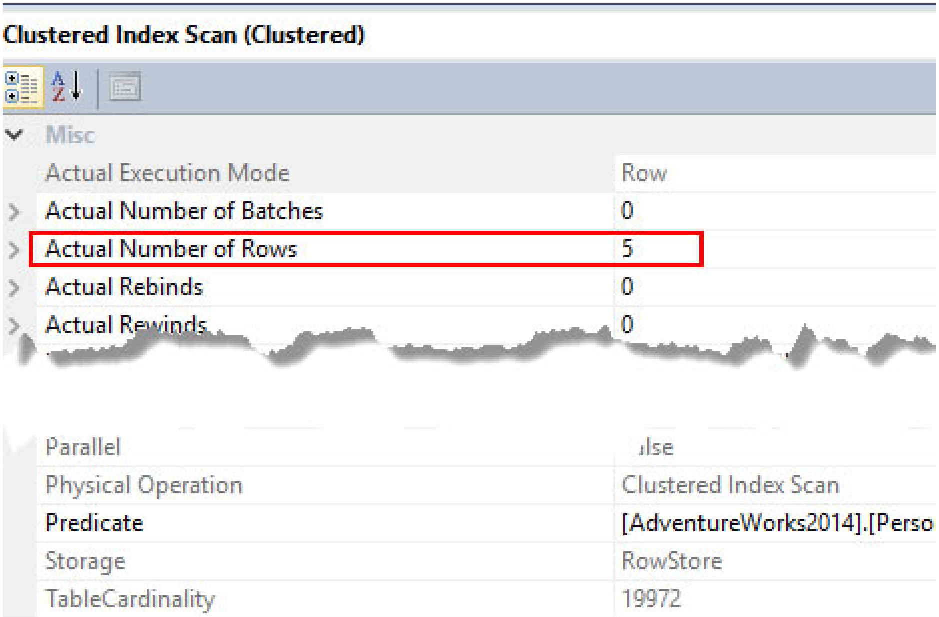


Figure 2-2: Actual number of rows processed.

In fact, this example illustrates clearly that, during plan execution, the operators are called from left to right, so if we follow the order in which the operators are called, we must read the plan left to right.

Each operator supports a **GetNext** method ("*Give me the next row*") and the first action in this case is a **GetNext** call from the **Top** operator to the **Clustered Index Scan**, which passes the first qualifying row, filtered according to the `WHERE` clause, back to **Top** and then the cycle repeats for each row, steadily streaming rows back to the client. Once the **Top** operator has all the rows it needs, five rows in this case, execution stops, so the rest of the table is never read.

Streaming versus blocking operators

Many of the operators you see in plans will be non-blocking, a.k.a. streaming, operators. A streaming operator creates output data at the same time as it receives the input. In other words, it will pass on rows to the next operator as soon as it has performed its task on that row.

Some operators, however, are blocking operators and must gather the entire set of input data and then perform their work on the entire data set, before passing on any rows. Add `ORDER BY ModifiedDate` to Listing 2-1, and re-execute the query, requesting the actual execution plan, as shown in Figure 2-3.

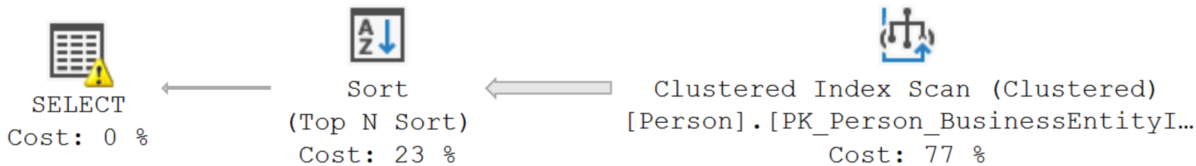


Figure 2-3: Execution plan showing blocking operators.

The **Clustered Index Scan** (discussed in detail later in Chapter 3) is a streaming operator, and passes on rows as they are read from the index. A scan indicates that it will read all rows in the table, or index, until all rows are processed (unless a different operator, such as **Top** in the previous example, ends execution early). When it finds a row that falls in the required date range, it passes that row on to the next operator, in this case, a **Sort**.

The **Sort** operator reorders data, representing here the `ORDER BY` statement in the query. The **Sort** operator is a blocking operator. This logical operation is a **Top N Sort** because of the `TOP` operation in the query. It must read every row from its child operator, in this case over 13K qualifying rows, sort them according to the specified criteria, `ModifiedDate`, and then pass on the top five rows. In this sort of situation, especially for a very large input, such blocking operators could slow down performance.

Some operators are only semi-blocking, and must complete only part of their work before releasing the first row. For example, the join operator **Hash Match** first processes all rows from its first input, but then processes and returns rows from the second input as it reads them.

Microsoft maintains no definitive listing of blocking and non-blocking operators. Instead, you can infer their behavior by the definitions and relationships within the plan. Again, the key to understanding execution plans is to start to learn how to understand what the operators do and how this affects your query.

The warning shown in the plan in Figure 2-3, the little exclamation point, will be discussed in the next section.

What to Look for in an Execution Plan

As queries grow complex, so their executions plans can quickly become rather unwieldy and harder to understand, regardless of whether we read the plan right to left or left to right. Rather than trawling through every operator, we can often identify potential issues by looking out for a few key warning signs, and by examining the properties behind certain important operators.

The following recommendations don't preclude the need to understand the plan as a whole, and its operators, but they can help you read through a plan a little faster than trying to trace all the data paths and all the behaviors one at a time.

We'll discuss why each of these are important "pointers" to sources of possible problems, but we won't drill into specific examples. Throughout the rest of the book, we'll expand on these recommendations, with specific examples.

First operator

The first operator, on the left-hand side of the execution plan, is the `SELECT/INSERT/UPDATE/DELETE` (and sometimes others, such as `MERGE`) operator, and the first time you look at an execution plan it's always worth examining its properties.

Whereas the **Properties** window for other operators reveals information specific to the action of that operator, the first operator offers a lot of information about the plan itself and its generation. It includes information such as the time, CPU and memory required to compile the plan, the ANSI connection settings, whether the optimizer completed optimization or terminated the optimization process early because a good enough plan was found or it didn't find what it considered an optimal plan (this is referred to as a "timeout").

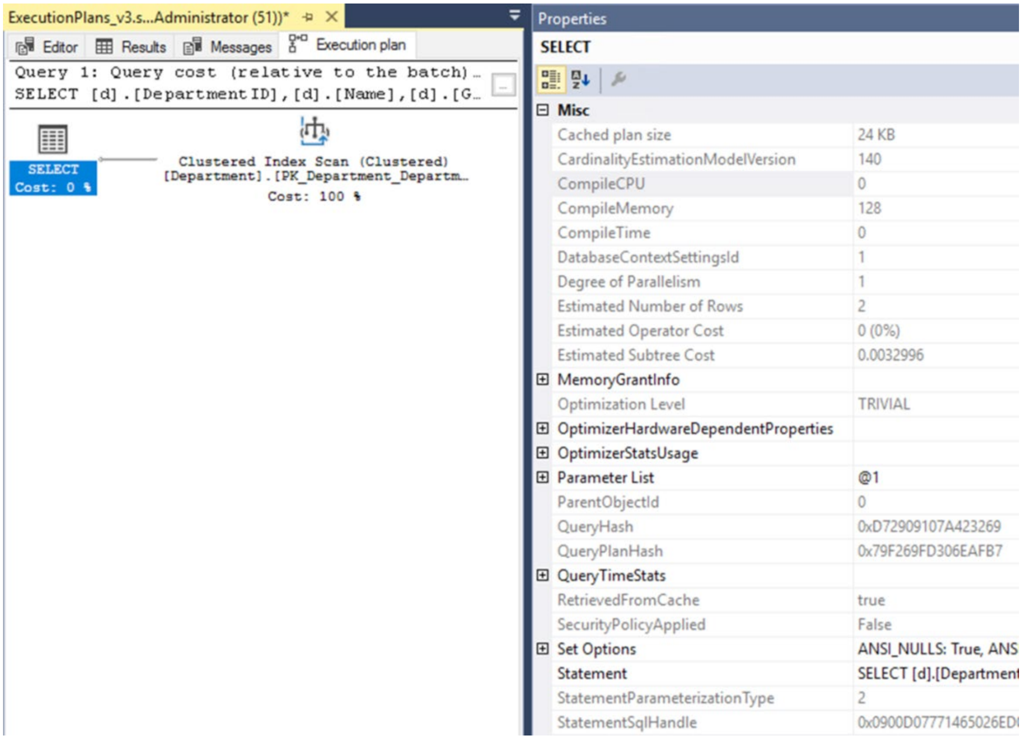


Figure 2-4: First operator properties.

We'll review some of the details of this operator later in this chapter, and continue, throughout the book, to explore the interesting pieces of information it provides.

When capturing plans using Extended Events (see Chapter 15), you may not see the first operator and all the great information it provides, which is a pity. However, most of the important information is still available in the plans captured through Extended Events, within the XML that defines the plan.

Warnings

Within an execution plan, you may see (on SQL Server 2012 and later) small icons appear on an operator, specifically a yellow or red exclamation mark. These are warnings. Not every warning indicates a grave problem, but whenever you see one, check the properties for that icon, which will contain a description of the warning.

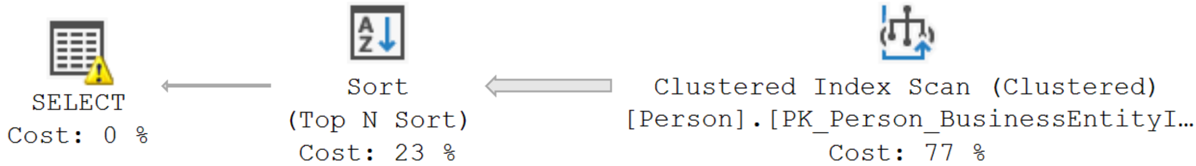


Figure 2-5: Execution plan with a warning.

Figure 2-5 shows a warning on the **SELECT** (in this case caused by a memory allocation mismatch), but there are other types of warning, such as a warning on a **Sort** operator that spilled to disk, and we'll go over several of them as we encounter them in execution plans throughout the rest of the book.

Estimated versus actual number of rows

It is very important to remember that all costs you will ever see in a plan are based on cardinality estimations, never on actual row and execution counts. Therefore, these costs are only as accurate as the optimizer's cardinality estimations.

One of the first things to check in a plan before digging deeper, and certainly before looking at the costs associated with individual operators, is to compare estimated and actual row counts and make sure they are within reasonable margins, to confirm the accuracy of the cardinality estimates associated with the estimated costs. Sometimes, you'll see an operator with a very high estimated cost, because the optimizer estimated it would need to process many rows, when in fact it had to process very few rows (or vice versa, for low estimated costs).

If estimated and actual rows counts differ significantly, you need to work out the cause and fix that first. Only then can you look at estimated cost of operators.

Operator cost

Having verified that cardinality estimates were accurate, we can look for the costliest operators as a means of determining where to focus our initial efforts. It's often useful to compare the cost of one operator to another within the plan. However, we can't compare operator cost within one plan to operator cost within a second plan because the cost estimates are mathematical constructs and don't really lend themselves directly to that type of comparison.

Also, some operators, and we'll discuss them as we go, don't have costs associated with them, or they're "fixed" costs based on assumptions within the optimizer, which may or may not be accurate. For example, a **Compute Scalar** operator always has a very low fixed cost (zero-point-lots-of-zeros-one), which is often fine but occasionally misleading, as we'll see in Chapter 4.

So, while cost estimates are important and we will use them, just remember that they can't be blindly trusted as an accurate measure of actual cost within the plan.

"Missing Index" suggestions

Often, you'll see a message at the top of a plan saying that there is a missing index that will "reduce the cost" of an operator by some impressive percentage. Treat them as suggestions only, rather than going ahead and creating each index that's suggested. Remember, an index that may help a single query, which is all that a given execution plan represents, may be detrimental to the performance of your workload as a whole. Also, there may be more than one index suggested. You'll only see one at the top of the plan. Check the first operator to see if there are additional suggestions.

Data flow

As discussed previously, the data flow within an execution plan is defined by the arrows connecting one operator to the next. These arrows, because they represent the flow of data, are frequently referred to as pipes. The thickness of the pipe is based on actual row count when available (actual execution plan), and on estimates otherwise (cached or estimated plan). A thicker pipe indicates more data being processed; a thinner pipe indicates less data. In some cases, some of the operators in an actual plan do not report an actual row count, in which case the estimated row count is used to set the pipe size.

Look out not only for "fat pipes," but also for abrupt transitions in pipe thickness as you read through the execution plan. For example, a very fat pipe at the beginning of a plan narrowing to a very thin pipe on the left-hand side of the plan suggests that filtering is happening late. Small pipes that get bigger and bigger suggest that your query is somehow multiplying data.

Extra operators

There is not really any such thing as an extra operator; every operator in a plan performs a specific function. The idea of an "extra" operator is one that I've made up as a good way to help people get started reading execution plans. Here's how it works.

Every time you're reading a plan and you see an operator you've never seen, or an operator that you've seen and understand, but can't determine why it's in the spot it's in within the plan, then that is an "extra" operator. It's an operator that you don't know, or you don't understand why it's affecting the plan.

Your response is simple: understand what the operator is and what it's doing and then it is no longer an "extra" operator.

Read operators

We'll detail the various read operators in the next chapter. The ones we'll focus on here are the scan and the seek. A scan operator (an **Index Scan** or **Table Scan**) is just an indicator of one type of data access that reads across the pages in an index or a table. However, it's a type of data access that indicates, frequently, that a lot of rows are being accessed.

A seek operator is an indicator of another type of data access that uses the structure of an index to identify a starting point, and possibly an ending point, for a targeted scan through the pages of an index. A seek indicates, most of the time, that only a small number of rows are being accessed.

Most people when reading plans have a "scans bad, seeks good" mentality. In fact, neither of these operations is good or bad, by definition. What you want to look out for in a plan are high-cost scans that retrieve limited data sets (sometimes indicating a missing or poorly structured index), or seeks that retrieve extremely large data sets.

The Information Behind the First Operator

Many people in the habit of reading plans right to left immediately focus their attention on the data access operators over on the right-hand side. They forget to look at the properties of the first operator, which is a pity because they are missing a lot of valuable information about the plan, as a whole. Hopefully, this section will put that right. As you will see, there is a lot of information available in the first operator about the process that the optimizer went through to arrive at this plan.

That's why the first operator in a plan, reading left to right, makes a good starting point for exploring the execution plan of any query. Microsoft defines these operations as "Language Elements." They represent the process that the query is performing. The official name of the first operator is the **Result Showplan** operator, but all the labels within plans and the tooltip refer to it by a different name: **SELECT** in a `SELECT` query, **UPDATE** in an `UPDATE` query, and various other names are possible. Rather than confusing things, we'll use its actual name, such as SELECT, rather than refer to it as the **Result Showplan**.

Let's start with a simple query against the `HumanResources.Department` table in the `AdventureWorks2014` database.

```
SELECT d.DepartmentID,
       d.Name,
       d.GroupName
FROM HumanResources.Department AS d
WHERE d.GroupName = 'Manufacturing';
```

Listing 2-2

Execute the query in SSMS and capture the execution plan for this query, as shown in Figure 2-6.

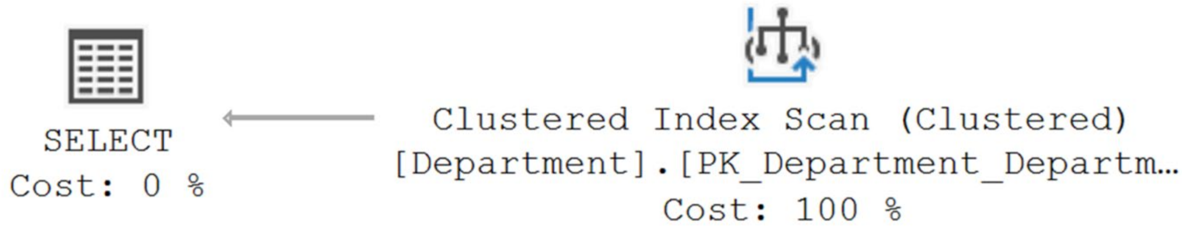
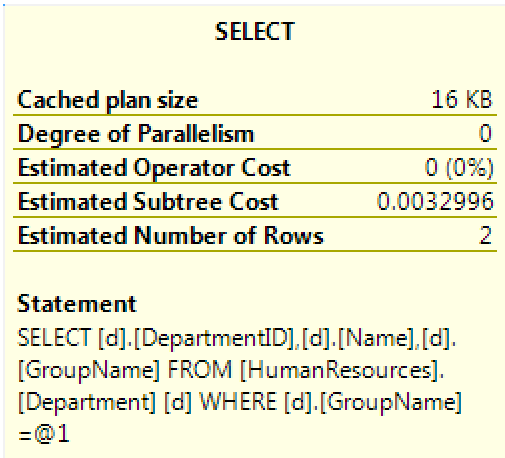


Figure 2-6: Simple plan.

The plan has only two operators, a **Clustered Index Scan**, which we'll discuss in Chapter 3, and the **SELECT**. When exploring the information provided by the **SELECT** operator, use the full **Properties** window, because the tooltip, shown in Figure 2-7, provides only a subset of the available information and almost none of the most important ones.



SELECT

Cached plan size	16 KB
Degree of Parallelism	0
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0032996
Estimated Number of Rows	2

Statement

SELECT [d].[DepartmentID],[d].[Name],[d].[GroupName] FROM [HumanResources].[Department] [d] WHERE [d].[GroupName] =@1

Figure 2-7: Tooltips often don't display important properties.

To bring up the full **Properties** window, as shown in Figure 2-8, simply right-click on the **SELECT** operator and select **Properties** from the context menu. Throughout the rest of the book, we'll be using only the **Properties** window, so it makes sense to pin this window to your SSMS desktop. This will preclude the need to right-click on each operator and you can simply select the operator from that point forward.

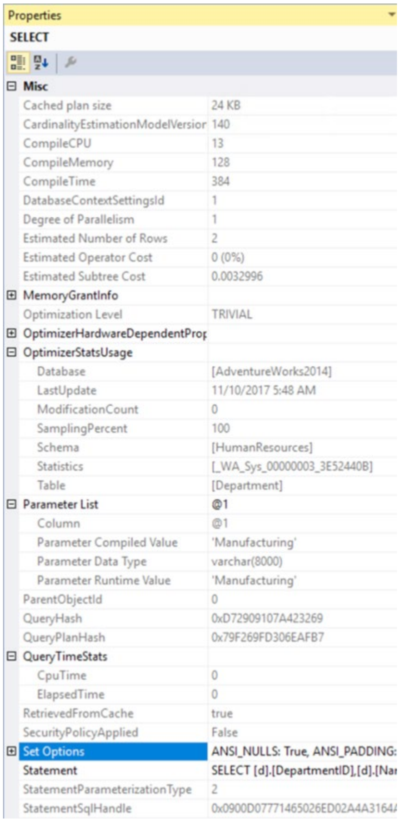


Figure 2-8: Full property page for SELECT operator.

All of the property values are stored with the plan and are visible in the XML as well as in the graphical plan. I'm not going to explain every property here, but I will start by listing out a few that are occasionally useful and then describe, in a bit more detail, some of the ones that you will use on a regular basis:

- **Cached plan size** – This property is important because it indicates just how much memory this plan will take up within the plan cache of SQL Server.
- **CardinalityEstimationModelVersion** – Starting with SQL Server 2014, a new cardinality estimator can be used by the optimizer. You can tell if the plan in question is using the new or the old. The value in Figure 2-8 is 140, signifying the new estimator. If it was 70, it would be the old version from SQL Server 7.
- **CompileCPU**, **CompileMemory**, **CompileTime** – The resources used to produce the plan. The time is in milliseconds. The memory is in kilobytes.

- **RetrievedFromCache** – This is something of a misnomer. Instead of telling you that this plan was pulled from cache, it basically says that this plan was stored in the cache. You'll only see a value of "False" here if the plan in question is not stored in cache.
- **QueryTimeStats** – Introduced in SQL Server 2016, this property shows the execution time for the query, when you're capturing an actual query.

Optimization level

This shows the level of optimization required to produce the plan. Generally, you'll see either "Trivial" or "Full." A trivial plan, such as this one, can only be resolved one way by the optimizer, as described in Chapter 1. Exactly what makes a plan trivial is the lack of choices possible to the optimizer. For example, a `SELECT *` statement against a single table without a `WHERE` clause can only be resolved one way. Another example is an `INSERT` statement against a table using `VALUES`. This can only be resolved a single way by the optimizer, making the plan trivial.

Full optimization just means it's not a trivial plan, but doesn't actually tell you the extent of work that the optimizer put into the optimization of this particular plan. To see the optimization level in action, we'll add a `JOIN` to the query as you can see in Listing 2-3.

```
SELECT d.DepartmentID,
       d.Name,
       d.GroupName,
       edh.StartDate
FROM HumanResources.Department AS d
INNER JOIN HumanResources.EmployeeDepartmentHistory AS edh
   ON edh.DepartmentID = d.DepartmentID
WHERE d.GroupName = 'Manufacturing';
```

Listing 2-3

Figure 2-9 shows the actual execution plan.

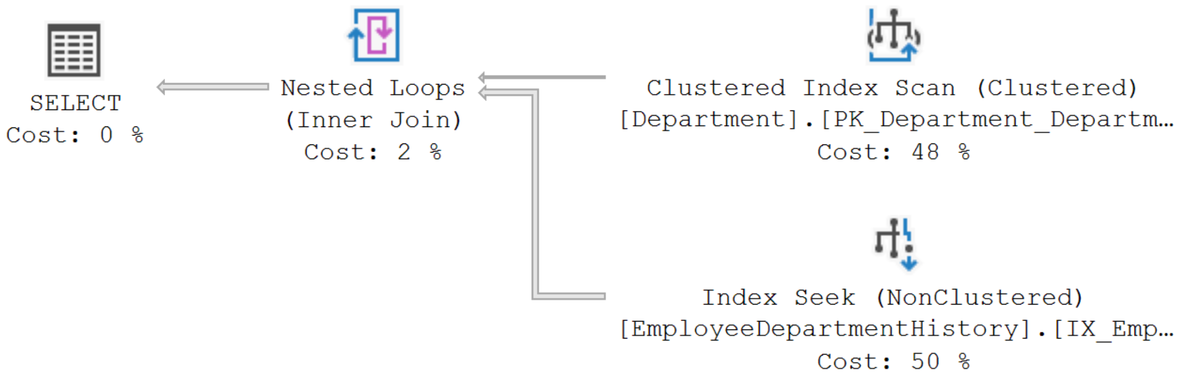


Figure 2-9: Execution plan illustrating FULL optimization.

We won't examine the whole plan now as it contains operators we won't discuss till later in the book. However, if we look at the properties for the **SELECT** operator, we see **FULL** optimization level, as shown in Figure 2-10.

Optimization Level	FULL
OptimizerHardwareDependentProperties	
Physical Operation	
QueryHash	0x88B004192F0536D
QueryPlanHash	0x88EE4F51C38A0CC4
Reason For Early Termination Of Statement (	Good Enough Plan Found

Figure 2-10: Subset of SELECT operator properties.

We also see a value for a related property called **Reason For Early Termination Of Statement Optimization**.

If a plan is produced via the **FULL** optimization process, then there will be a reason for the optimizer to stop processing and present its selected plan. For simple queries, the reason you'll commonly see here is **Good Enough Plan Found**. This means that after at least one of the optimization phases, the estimated cost of the cheapest plan was below the threshold for entering the next phase, and therefore the optimizer selected that plan as good enough.

For more complex queries, if this property value is not reported, it indicates that the plan was simply the one selected by the full optimization process after completing all possible optimizations in whatever phase the optimizer chose to put the plan through.

You'll see two other values in this property, **Timeout** and **Memory Limit Exceeded**. A value of **Timeout** indicates that the optimizer attempted to go through its full optimization process, but didn't succeed. Instead, it ran through as many optimization attempts as it thought necessary for the query, but it didn't find what it considered to be a mathematically good enough plan. So, it returned the least-cost plan that it had found so far.

A value of **Memory Limit Exceeded** means an extremely large and complex query against very complex structures. The plan generated is probably not optimal for the query if you have a **Timeout** or **Memory Limit Exceeded**. However, without simplifying your query or your structure, you're unlikely to get a better plan.

Parameter List

In our query in Listing 2-2, the single-table query, we hard-coded the value supplied for `GroupName`, in the `WHERE` clause. In other words, we did not use parameters or local variables. However, the **Properties** window displays a **Parameter List**, the expanded view of which is shown in Figure 2-11, where we see a parameter named `@1` and its corresponding compile time and runtime values.

Parameter List	@1
Column	@1
Parameter Compiled Value	'Manufacturing'
Parameter Runtime Value	'Manufacturing'

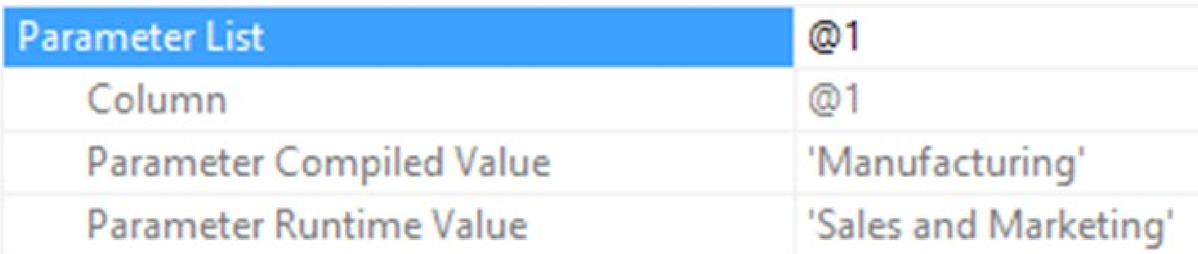
Figure 2-11: SELECT properties showing the Parameter List.

Since this is a very simple query, the optimizer has been able to perform a process called **simple parameterization**. This is a process where the optimizer recognizes that, if you were using a parameter instead of the hard-coded value supplied, it would be able to create an execution plan it can reuse. So, it substitutes a parameter of its own. In this case, the optimizer parameterized our search argument so that the `WHERE` clause of our query is now

`WHERE d.GroupName = @1`. As a result, we can see this parameter in the **SELECT** operator of our queries. When you see this sort of parameterization, it is also important to inspect the query (in the **SELECT** operator) to check which of the hard-coded values in the original query is replaced by which parameter.

Without simple parameterization, if we were to execute the query in Listing 2-2 again, but with a different value in the search condition, such as `WHERE d.GroupName = 'Sales and Marketing'`, then the query text has changed, no plan will match, and the optimizer will generate a new plan, even though we've executed what is essentially the same query.

However, with our newly parameterized query, the query text remains static from one execution to the next, and SQL Server swaps in the required value for the `@1` parameter on each subsequent execution. Assuming no `SET` options change, the optimizer will reuse the existing plan. Figure 2-12 shows the **Parameter List** for a second execution of the query, with a different value supplied in the search condition.



Parameter List	@1
Column	@1
Parameter Compiled Value	'Manufacturing'
Parameter Runtime Value	'Sales and Marketing'

Figure 2-12: SELECT properties with varying Compiled and Runtime values.

However, you will note that we don't see a **Parameter List** in the **SELECT** properties for the two-table query in Listing 2-3. The optimizer can only perform simple parameterization for simple, one-table queries. The best way to promote plan reuse is to actively parameterize your queries, using stored procedures.

Whenever a parameter is used, the value passed to that parameter is used to compare to the statistics of the column or index being used. This is known as "parameter sniffing" (or "variable sniffing"). The use of the specific value leads the optimizer to make better choices based on your statistics. So, you can look to the **SELECT** operator to get the compile and runtime values for parameters to understand how parameter sniffing was resolved on any given query. We'll discuss parameter sniffing, and the occasional problems it causes, in more detail when we get to stored procedures.

QueryHash and QueryPlanHash

The **QueryHash** is a hash value of the query, which is stored with the plan and used by the optimizer to identify plans with the same or very similar logic. As discussed in Chapter 1, if the value of a submitted query matches the **QueryHash** for a plan in the cache, the optimizer analyzes the SQL text and, if it's identical, can reuse the plan, assuming no difference in `SET` options, or database ID. The **QueryHash** can be very useful in situations where you're dealing with ad hoc or dynamic T-SQL and need to identify if there are multiple, similar queries in the system for which separate plans are being created.

The **QueryPlanHash** is like the **QueryHash** value but for the plan itself. It identifies plans that are the same in terms of the operations they perform, and the order they perform them.

Leaving aside cases where the optimizer performs "auto-parameterization," we can have cases such as the following:

- If we make a change only to literal values, and it doesn't affect the plan, we can see multiple plans in the cache, each with the same **QueryHash** and the same **QueryPlanHash**.
- If we change only the literals but it results in a different plan, then we'll see multiple plans, each with the same **QueryHash** but different values for **QueryPlanHash**.
- If we make a logical change to the query that does not affect the execution plan, then we might see multiple plans in the cache, each with a different **QueryHash** but the same **QueryPlanHash**.

SET options

Figure 2-13 shows the ANSI connection settings and other `SET` options that were used when the plan was created. These are very handy values because, as mentioned above, changing these settings can result in multiple plans in the cache for what are, in all other respects, identical queries.

Set Options	ANSI_N
ANSI_NULLS	True
ANSI_PADDING	True
ANSI_WARNINGS	True
ARITHABORT	True
CONCAT_NULL_YIELDS_NULL	True
NUMERIC_ROUNDABORT	False
QUOTED_IDENTIFIER	True

Figure 2-13: ANSI settings within SELECT properties.

Other Useful Tools and Techniques when Reading Plans

One of the primary (but not the only) uses of execution plans is in understanding how a query is being executed, in order to understand why it is performing poorly.

As such, it's often very useful to collect performance metrics alongside your execution plans, especially when you're attempting to tune a query in your development environment. There are multiple ways to gather query metrics:

- `SET STATISTICS IO/TIME`
- Include Client Statistics
- SQL Trace (Profiler)
- Extended Events
- Query Store (covered in Chapter 16)

There are actually a few other ways, but these are the most used and the most useful. I'm going to recommend that you use Extended Events for detailed metrics, and Query Store, where possible, for aggregated metrics. There are several reasons for this, but let's start with using `STATISTICS IO/TIME`.

I/O and timing statistics using SET commands

People often use `STATISTICS IO/TIME` to capture individual query performance when tuning a query. All we do is surround the query with the `SET` commands, as shown in Listing 2-4.

```
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
  SELECT d.DepartmentID,
      d.Name,
      d.GroupName
  FROM HumanResources.Department AS d
  WHERE d.GroupName = 'Manufacturing';
SET STATISTICS IO OFF;
SET STATISTICS TIME OFF;
```

Listing 2-4

Look at the complete output of these values for the execution of a single query as shown in Listing 2-5.

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
(2 row(s) affected)
Table 'Department'. Scan count 1, logical reads 2, physical reads
0, read-ahead reads 0, lob logical reads 0, lob physical reads 0,
lob read-ahead reads 0.
(1 row(s) affected)
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 6 ms.
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 0 ms.
```

Listing 2-5

Without someone explaining to you exactly what to look for, can you tell the number of reads and exactly how long the query took to execute? Once it's explained, sure, but the output here is quite unclear. The one advantage is that the I/O is broken down by table, which can be handy at times; because of this, depending on the situation, I will use `STATISTICS IO`, but with the following caveat: capturing `STATISTICS IO` can negatively impact execution time because of the additional overhead of transferring the I/O information to the client after it's captured. If you're attempting to tune a query and you want to see if it's running faster or slower, as well as capture the number of reads, you need your measures to be accurate and they simply won't be with `STATISTICS IO`.

Also, it also doesn't always reveal all the work done. For example, if you have code that makes a lot of calls to a user-defined function, it won't count that I/O, whereas Extended Events does.

Include Client Statistics

If you are investigating queries that run fast but often, then the overhead of showing the results in grid or text is often significant enough to invalidate the performance measurements.

A useful technique in such cases is to change the query options to discard the results after execution, then add a high number after `GO` commands so that the query runs lots of times (e.g. `GO 100` to run a query 100 times), and use SSMS's **Include Client Statistics** option to look at the elapsed time.

SQL Trace and Profiler

The Profiler GUI uses a different buffering mechanism than Trace Events which can directly affect your server in such a way that gathering metrics can negatively impact the server or even take it down. I don't recommend ever running Profiler on your production server, and running it on a development server can invalidate the gathering of metrics. Trace Events can't be filtered at the point of capture. Instead, all Trace Events are captured and then filtered afterwards, radically increasing their overhead on your system. Further, Trace and Profiler are on the list for deprecation. This means that in an upcoming edition of SQL Server they will no longer be available. It's time to stop using them.

Extended Events

My recommendation is to capture your I/O and timing metrics using Extended Events. They're in active support from Microsoft. They offer better and more effective filtering than Trace. They operate lower within the call stack within SQL Server so they have a much lower impact on performance. Their measure of performance and reads is clear and easy to understand. When working in SQL Server 2012 or greater, there's a fully-functional graphical interface for looking at the metrics gathered.

Because of all these reasons, I strongly advise you to use Extended Events. Listing 2-6 offers a basic mechanism for capturing stored procedures and batches.

```
CREATE EVENT SESSION QueryPerformance ON SERVER
ADD EVENT sqlserver.rpc_completed (
    WHERE (sqlserver.database_name = N'AdventureWorks2014')),
ADD EVENT sqlserver.sql_batch_completed (
    WHERE (sqlserver.database_name = N'AdventureWorks2014'))
ADD TARGET package0.event_file (SET filename = N'QueryPerformance')
WITH (MAX_MEMORY = 4096 KB,
      EVENT_RETENTION_MODE = ALLOW_SINGLE_EVENT_LOSS,
      MAX_DISPATCH_LATENCY = 3 SECONDS,
      MAX_EVENT_SIZE = 0 KB,
      MEMORY_PARTITION_MODE = NONE,
      TRACK_CAUSALITY = OFF,
      STARTUP_STATE = OFF);
```

Listing 2-6

Summary

This chapter introduced the basics of reading execution plans, starting with defining the "language" used by the plans themselves. We also introduced a basic set of things to look for within execution plans. This can act as a guide to reading all execution plans, no matter how large. Just remember that the details of the plan are very important and the information presented here is only a guide. We covered the often-neglected information behind the first operator. We rounded off with some useful tools and techniques that are often used side by side with execution plans to gather useful execution statistics.

Chapter 3: Data Reading Operators

In this chapter, we're going to examine the data reading operators, which represent the optimizer's different mechanisms for reading data. They can also act as a filtering mechanism, to pass on the qualifying rows to the next operator.

We'll cover the following operators in detail:

- **Clustered Index Scan**
- **Index Scan (nonclustered)**
- **Clustered Index Seek**
- **Index Seek (nonclustered)**
- **Key Lookup (clustered)**
- **Table Scan**
- **RID Lookup (heap)**.

As we progress, you'll learn how the operators work, and start to deepen your knowledge of execution plans generally, the various operators that they use, and how to read the plan and to understand the optimizer's choices on how the query should be executed.

Reading an Index

Traditional SQL Server indexes, which excludes memory-optimized, columnstore, full-text indexes, and others, consist of 8 K pages connected in a b+tree structure. These are frequently referred to as balanced-tree, bushy-tree or even Bayer-tree, after the lead researcher who developed them.

The overriding majority of tables in a SQL Server database should have a **clustered index**. The leaf-level pages of a clustered index store the data rows, ordered according to all the columns of the clustered index key. A clustered index is not a "copy" of the table. It is the table, with a b+tree structure built on top of it, so that the data is organized by the clustering key. This explains why we can only create one clustered index per table.

In addition to a clustered index, most tables have one or more **nonclustered indexes**, designed to improve the performance of critical, frequent, and expensive queries. A nonclustered index has the same b+tree structure, but the leaf-level pages do not contain the data rows, just the data for the index key columns, plus the clustered index key columns (assuming the table is not a heap), plus any columns that we optionally add to the index using the `INCLUDE` clause.

There are essentially three classes of operator that SQL Server can use to access data in an index: scan, seek, or lookup.

Index Scans

In a scan operation, SQL Server navigates down to the first or last leaf-level page of the index and then scans forward or backward through the leaf pages. A scan often reads all the pages in the leaf level of the index, but may read only a portion of the index in some cases.

A scan often occurs when all rows need to be read to satisfy the definition of the query. You can also see a scan when so many rows need to be read that scanning them all would take less time than navigating the index structure to find them (a.k.a. "seeking," discussed shortly). Sometimes, the optimizer chooses a scan because there is no usable index for the Predicate columns, or because the query is written in such a way that performing a seek against the index is not possible (for example, a function against a column will lead to scans).

If a scan occurs on a clustered index, we'll see the **Clustered Index Scan** operator, and if it's on a nonclustered index, we'll see an **Index Scan (nonclustered)** operator. It's the same operation in either case. In the case of a heap table, a table without a clustered index, you'll see a **Table Scan**, which is effectively the same operation, just done against a different structure, the heap as opposed to an index. This will be discussed further later in the chapter.

Clustered Index Scan

Listing 3-1 shows a simple query on the `Employee` table, looking for people with birthdays over 50 years ago.

```
SELECT  e.LoginID,
        e.JobTitle,
        e.BirthDate
FROM    HumanResources.Employee AS e
WHERE   e.BirthDate < DATEADD(YEAR, -50, GETUTCDATE());
```

Listing 3-1

Figure 3-1 shows the actual execution plan.

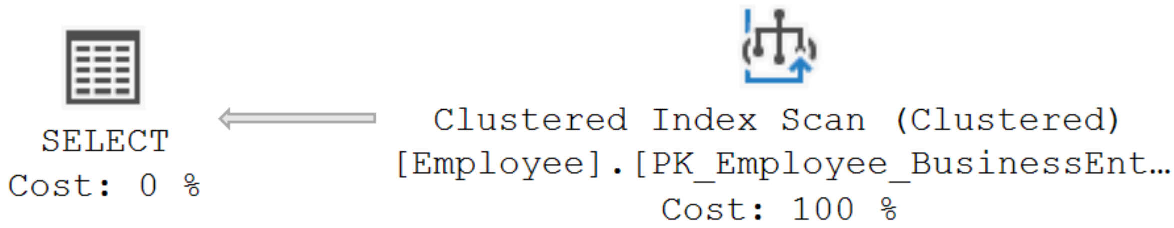


Figure 3-1: Execution plan with a Clustered Index Scan.

The optimizer chose a **Clustered Index Scan** operator to retrieve the required data. If your Property window is already up, click on the **Clustered Index Scan** to load it with information from that operator. Otherwise, right-click on the icon and select **Properties** from the context menu.

You're going to notice a lot of properties that repeat from one operator to the next. Some of these properties can be useful in understanding how the operator works and what it is doing, but some properties are reported for many operators, but are only interesting in the context of specific operators. For example, **Rebinds** and **Rewinds** (estimated and actual) are only important when dealing with the **Nested Loops** operator, but there are no joins of that type in this plan so, in this case, those values are useless to you.

Actual Execution Mode	Row
Actual Number of Batches	0
Actual Number of Rows	26
Actual Rebinds	0
Actual Rewinds	0
Defined Values	[AdventureWorks2014].[Human
Description	Scanning a clustered index, enti
Estimated CPU Cost	0.000476
Estimated Execution Mode	Row
Estimated I/O Cost	0.0075694
Estimated Number of Executions	1
Estimated Number of Rows	26
Estimated Operator Cost	0.0080454 (100%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	322 B
Estimated Subtree Cost	0.0080454
Forced Index	False
ForceScan	False
Logical Operation	Clustered Index Scan
Node ID	0
NoExpandHint	False
Number of Executions	1
Number of Rows Read	290
Object	[AdventureWorks2014].[Human
Ordered	False
Output List	[AdventureWorks2014].[Human
Parallel	False
Physical Operation	Clustered Index Scan
Predicate	[AdventureWorks2014].[Human
Storage	RowStore
TableCardinality	290

Figure 3-2: Properties of the Clustered Index Scan operator.

Some of the properties are self-explanatory. Looking at Figure 3-2, near the bottom of the Properties, you find the **Object** property. This indicates which object this operator references. In this case, the clustered index used was `HumanResources.Employee.PK_Employee_BusinessEntityID`.

Other interesting properties could include the **Output List**. These are the columns that are output from the operation. Near the top, though, you'll also see **Defined Values**. These are the values added to the process by this operator. In this case, the **Output List** and the **Defined Values** are the same, but in other cases, such as when a calculation is done in a **Compute Scalar** operator (discussed in the next chapter), or in any other operator, you'll see additional information in **Defined Values**.

As discussed in detail in previous chapters, all the properties that start with "Estimated," such as **Estimated I/O Cost** and **Estimated CPU Cost** are measures assigned by the optimizer, but do not represent actual I/O and CPU measures. Even in an actual plan, these values represent the estimates from the optimizer based on statistics. Each operator's estimated cost contributes to the overall estimated cost of the plan.

Since we captured an actual execution plan, we see both the **Estimated Number of Rows** and the **Actual Number of Rows**, which is the estimated and actual number of rows output by the operator. In this case, the operator outputs 26 rows (the number of rows with a `BirthDate` more than 50 years in the past). You can also see the number of rows that were accessed via the **Number of Rows Read** property. In this case it's 290, or the entire clustered index.

The **Ordered** property is **False**, indicating that the optimizer did not require the data to be retrieved in index key order. If we were to add an `ORDER BY e.BusinessEntityID` clause to Listing 3-1, then this property value would change to **True**, because it could use the clustered key order to perform that operation. The optimizer can choose to use the order of the index for its scans. This can be very useful if one of the next operators in line needed ordered data, because then no extra sorting operation is needed, possibly making this execution plan more efficient, depending on the needs of the query.

The **Predicate** property is important, and shows the Predicate applied by this operator (click on the ellipsis to see the full text):

```
[AdventureWorks2014].[HumanResources].[Employee].[BirthDate] as [e].
[BirthDate]<dateadd(year,(-50),getutcdate())
```

The operator is a scan, and it reads all the pages in the leaf level of the index. In other words, it reads all the rows in the table, 290 in this case (see the **Table Cardinality** property value). While a scan generally reads all rows, it does not always return them all. Here, it evaluates the Predicate for each of the 290 rows it reads, and outputs only the 26 rows that match the condition. This is an important difference between a **Predicate**, and a **Seek Predicate** (which we'll see shortly, when we discuss Index Seek operations). Although the filtering looks similar in each case, the latter reads only the rows that match the condition.

So why do we see a scan in this case? Simply because the optimizer does not have an index available that matches our Predicate column. The clustered index key is on `BusinessEntityID` so the data in the leaf level is organized by that column. The scan operator has to scan all the leaf pages to find the matching rows. Reading one page is one logical read, so the number of logical reads required to return the data will depend on the number of pages in the leaf level of the index.

Index Scan

An **Index Scan** is the same as a **Clustered Index Scan**. It's just against a different type of object. Let's examine the query in Listing 3-2.

```
SELECT  e.LoginID,
        e.BusinessEntityID
FROM    HumanResources.Employee AS e;
```

Listing 3-2

This small query is only retrieving two values, LoginID and BusinessEntityID. There happens to be an index on the HumanResources.Employee table, AK_Employee_LoginID. Figure 3-3 shows the execution plan.

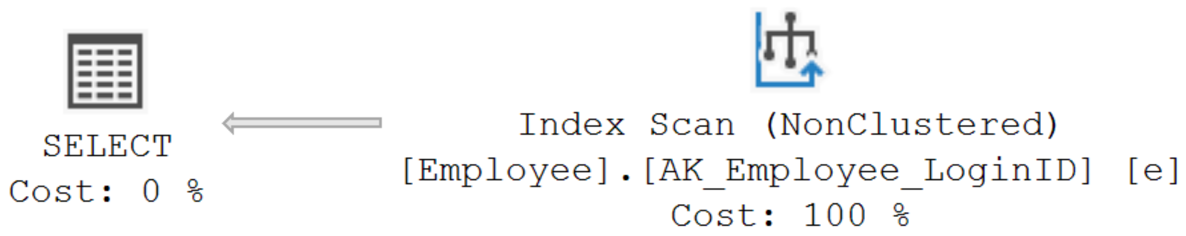


Figure 3-3: Execution plan with an Index Scan.

Since the query in question doesn't have a WHERE clause, there's little the optimizer can do to pick and choose how it's going to retrieve the information. It has to do a scan. However, based on the columns selected, it has a choice where it does that scan. Our index, AK_Employee_LoginID is keyed on the LoginID column. Since the clustered index key for this table is on BusinessEntityID, that key is included with the nonclustered index. This means that the optimizer can choose this index to satisfy the query. Further, since the size of this index, measured in the number of pages, is smaller than the primary key index, scans of this index will be faster and use fewer resources.

Other than the reasons for the choice of this index, the process of the scan is the same. It's retrieving the data from the leaf level of the index.

Are scans "bad?"

Scans are not a "bad" thing. If we want all, or most, of the data from a modestly-sized table, they can be a very efficient operation. In our **Clustered Index Scan** example, the fact that the operator processes 290 rows to output only 21 won't have a significant impact on performance in most systems. However, what if the optimizer opted to use a scan to output 21 rows from a table containing not 300 but 3 million rows? At that point, we are performing a lot of unnecessary logical reads, and we may need to consider either tuning the query to make better use of our existing index, or adding an index that will allow the optimizer to choose a plan where the SQL Server engine will only need to read the pages containing the 21 rows that we need to return.

As discussed earlier, there are other reasons we may see a scan operation. Sometimes, our query logic causes the optimizer to choose a scan when an index exists that it could, notionally, seek. One example of this would be when you have a query that embeds the indexed column in an expression. This prevents the optimizer from being able to determine which of the values stored in that column may match, because it has to evaluate the expression for each row, and so it has to scan the entire index.

It's also possible for the statistics on an index to become stale over time. In these cases, the optimizer can overestimate the number of rows that are likely to be returned, choosing to scan when a seek could have been more efficient.

Sometimes, our query may simply require all, or most, of the rows, so a scan is the most efficient way to do it. In the example in Listing 3-2, the lack of a `WHERE` clause forced the optimizer to request to return every row in the table.

An obvious question to ask, if you see an Index Scan in your execution plan, is whether you are processing more rows than is necessary. The business case, or the application, may ask for all the rows from a table, but then filter those down on the client or within the application. It's not unreasonable to push back on such requests. You could also see an unexpected number of rows where you know that you are filtering on a well-structured index with up-to-date statistics and you still see a scan. In this case, you should question why and how a scan is being used.

Processing unnecessary rows wastes SQL Server resources and hurts overall performance. That's why a scan can be an indicator of a potential issue, but a scan is not, by definition, a bad thing.

Index seeks

In a seek operation, SQL Server navigates directly to the page(s) containing the qualifying rows, or to the start/end of a range of rows, and processes only the rows that it needs to output.

Just as a scan is not necessarily "bad," a seek is not always "good." A seek is an efficient way to retrieve a small number of rows from a relatively large table. However, a seek operator can sometimes become highly inefficient, for example if inaccurate statistics have caused the optimizer to underestimate massively the number of rows it will need the operator to process.

A seek occurs when:

- an index exists that matches a Predicate column used in the query, and the index covers the query (can provide all the columns the query needs)
- an index matches the Predicate column used in the query, does not cover the query, but the Predicate is highly selective (returns only a small percentage of the rows).

If a seek occurs on a clustered index, we'll see the **Clustered Index Seek** operator, and if it's on a nonclustered index, we'll see an **Index Seek (nonclustered)** operator. It's the same operation in either case.

Clustered Index Seek

Let's examine a new query.

```
SELECT  e.BusinessEntityID,
        e.NationalIDNumber,
        e.LoginID,
        e.VacationHours,
        e.SickLeaveHours
FROM    HumanResources.Employee AS e
WHERE   e.BusinessEntityID = 226;
```

Listing 3-3

Execute this query and capture the actual plan and you will see the **Clustered Index Seek** operator, chosen by the optimizer to read the clustered index on the `Employee` table.

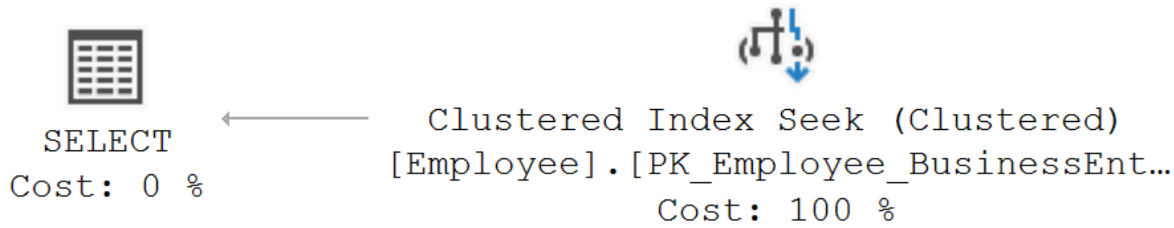


Figure 3-4: A simple plan showing a Clustered Index Seek operator.

Now that our query contains a search Predicate (`BusinessEntityID`) that matches the key of the clustered index, SQL Server's use of that index becomes analogous to looking up a word in the index of a book to get the exact pages that contain that word. The seek operator uses the key values to identify the row, or rows, of data needed and navigates through the b+tree structure directly to those pages.

This means that an **Index Seek** reads only those pages that contain data that is included in the filter. To return a single row while using an index, such as in the example, SQL Server performs only three logical reads to retrieve the data. This includes the pages it reads as it walks through the b+tree of the index to find the leaf-level page where the row is stored, plus the read of the leaf-level page.

As such, seeks can significantly reduce I/O compared to a scan, assuming the filter defines a small enough subset of the entire data set. Of course, the leaf-level pages of a clustered index store the actual data rows so no extra steps are required to return all the data required by the query.

Figure 3-5 shows a section of the properties for our **Clustered Index Seek**.

Object	[AdventureWorks2014].[HumanResources].[Emp
Ordered	True
Output List	[AdventureWorks2014].[HumanResources].[Emp
Parallel	False
Physical Operation	Clustered Index Seek
Scan Direction	FORWARD
Seek Predicates	Seek Keys[1]: Prefix: [AdventureWorks2014].[Hun

Figure 3-5: Properties of the Index Seek operator.

The index used, shown in the **Object** property, is the same as the example from Listing 3-1, specifically the `PK_Employee_BusinessEntityID`, which happens to be both the `PRIMARY KEY` constraint and the clustered index for this table. In this case, the index was created automatically to enforce the constraint; they are different objects but with the same name.

A seek operator has a property called **Seek Predicates**, which displays each of the predicates used to define the rows that need to be read:

```
Seek Keys[1]: Prefix: [AdventureWorks2014].[HumanResources].[Employee].
BusinessEntityID = Scalar Operator(CONVERT_IMPLICIT(int,[@1],0))
```

Once again, we can see the effects of simple parameterization. This time we also see a `CONVERT_IMPLICIT` function applied to the `@1` parameter value, for `BusinessEntityID`, since the value we supplied (226) is inferred to be a `smallint`, and needs to be converted to an `int` to enable a seek. The optimizer chooses the data type for simple parameterization based on the size of the value passed to it. If we passed a larger value, it would create the parameter as an `int` and it would create a second execution plan. However, as you can see, this didn't affect the choice of an **Index Seek** operation; some type conversions are harmful and lead to a scan when a seek should have been possible, others do not.

Index Seek (nonclustered)

Let's execute a simple query against the `Person.Person` table.

```sql
SELECT  p.BusinessEntityID,
        p.LastName,
        p.FirstName
FROM    Person.Person AS p
WHERE   p.LastName LIKE 'Jaf%';
```

Listing 3-4

This query takes advantage of a nonclustered index (`IX_Person_LastName_FirstName_MiddleName`) on the table as you can see from the execution plan in Figure 3-6.

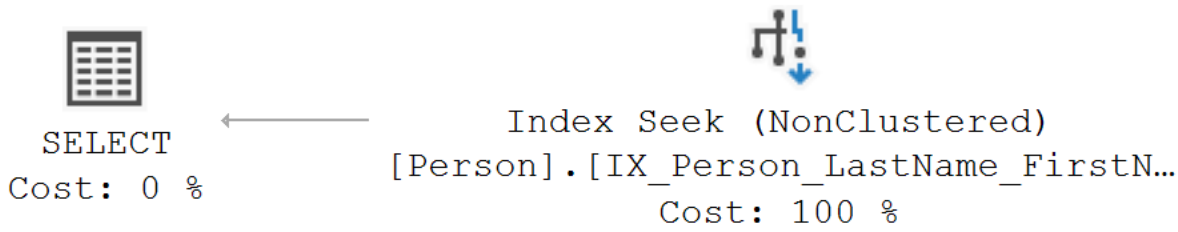


Figure 3-6: Plan showing Index Seek operator on nonclustered index.

A seek operator on a nonclustered index works in the same way as a seek operator on a clustered index. As such, there are no new properties to see for this operator compared to the **Clustered Index Seek**. However, it's worth noting that for this **Index Seek (nonclustered)** operator, we see both **Predicate** and **Seek Predicates** properties.

The Predicate looks like this, and essentially matches our `WHERE` clause:

```
[AdventureWorks2014].[Person].[Person].[LastName] as [p].[LastName] like
N'Jaf%'
```

The **Seek Predicates** property shows the following:

```
Seek Keys[1]:
  Start: [AdventureWorks2014].[Person].[Person].LastName >= Scalar
Operator(N'Jaf'),
  End: [AdventureWorks2014].[Person].[Person].LastName < Scalar
Operator(N'JaG')
```

Instead of a `LIKE 'Jaf%'`, as was passed in the query, the optimizer has modified the logic it uses so that an additional filter has been added as follows (minus a bit of formatting):

`Person.LastName >= 'Jaf'` AND `Person.LastName < 'JaG'`.

This is a good example of the sort of work performed by the optimizer, as outlined in Chapter 1. In this case, the optimizer optimized the `WHERE` clause Predicate, rewriting it from a `LIKE` condition to an interval defined by an `AND` condition. This is based on the fact that all values matching the `LIKE` condition logically have to be in the specified interval. Depending on collation, the interval might also contain values not matching the `LIKE` condition. Therefore, the latter is not removed but repeated in the **Predicate** property.

There's nothing new for us to see in the **SELECT** operator in the plan, except to note that this statement, unlike many of the simple statements we've been using as examples, did not go through simple parameterization. This is because a `LIKE` Predicate can be handled in different ways, depending whether the text-matching pattern starts with a wildcard, and so the optimizer can't do the parameterization.

As noted earlier, for a nonclustered index the leaf-level pages contain only the indexed columns, plus columns from the clustered index (`BusinessEntityID`, in this example), plus any columns we included using the `INCLUDE` clause. In this example, all the columns required by the query are contained in the leaf level of the nonclustered index. In other words, this is a covering index for this query.

Key lookups

A **Key Lookup (Clustered)** operator occurs in addition to an **Index Seek** (or sometimes an **Index Scan**), when the index used does not cover the query. The optimizer uses a **Key Lookup** to the clustered index, which will retrieve values for columns not available in the nonclustered index.

Let's take the same query from Listing 3-4 and modify it just slightly so that we also return the `NameStyle` column, as shown in Listing 3-5.

```
SELECT  p.BusinessEntityID,
        p.LastName,
        p.FirstName,
        p.NameStyle
FROM    Person.Person AS p
WHERE   p.LastName LIKE 'Jaf%';
```

Listing 3-5

If we run this query and capture the plan, it should look something like Figure 3-7.

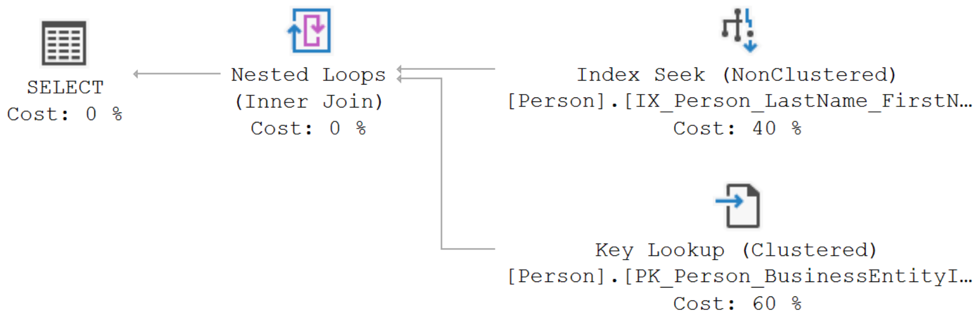


Figure 3-7: A plan with a Key Lookup operator.

The optimizer has still chosen an **Index Seek (nonclustered)** operator on the same nonclustered index as we saw previously, `IX_Person_LastName_FirstName_MiddleName`. However, in terms of the columns required by the query, the leaf level of the index stores only `LastName`, `FirstName` (since these are part of the index key), and `BusinessEntityID` (the clustered index key). It does not contain the `NameStyle` column, and so we see the additional **Key Lookup** operator, which uses the clustered index key values to retrieve the corresponding value for the `NameStyle` column from the leaf level of the clustered index.

A **Nested Loops** operator, which combines the results of these two operations, always accompanies a **Key Lookup**. We won't examine that operator until the next chapter.

Let's review some of the properties for this **Key Lookup** operator:

Object	[AdventureWorks2014].[Pers
Ordered	True
Output List	[AdventureWorks2014].[Pers
Alias	[p]
Column	NameStyle
Database	[AdventureWorks2014]
Schema	[Person]
Table	[Person]
Parallel	False
Physical Operation	Key Lookup
Scan Direction	FORWARD
Seek Predicates	Seek Keys[1]: Prefix: [Advent
Storage	RowStore
TableCardinality	19972

Figure 3-8: Properties showing the Output List of columns.

The **Object** property shows `PK_Person_BusinessEntityID`, which is the clustered index on this table, and the target of the **Key Lookup**. The expanded **Output List**, confirms that the output from this operator is the `NameStyle` column.

The **Seek Predicates** property shows the following:

```
Seek Keys[1]: Prefix: [AdventureWorks2014].[Person].[Person].
BusinessEntityID = Scalar Operator([AdventureWorks2014].[Person].[Person].
[BusinessEntityID] as [p].[BusinessEntityID])
```

If we look at the values for **Estimated** and **Actual Number** of rows, we see that it is 1 row, in each case, so the **Key Lookup** operator was only executed one time.

Actual Execution Mode	Row
Actual Number of Batches	0
Actual Number of Rows	1
Actual Rebinds	0
Actual Rewinds	0
Defined Values	[AdventureWorks20
Description	Uses a supplied clus
Estimated CPU Cost	0.0001581
Estimated Execution Mode	Row
Estimated I/O Cost	0.003125
Estimated Number of Executions	1.973518
Estimated Number of Rows	1
Estimated Operator Cost	0.0049499 (60%)
Estimated Rebinds	0.973518
Estimated Rewinds	0
Estimated Row Size	9 B
Estimated Subtree Cost	0.0049499
Forced Index	False
ForceScan	False
ForceSeek	False
Logical Operation	Key Lookup
Lookup	True
Node ID	3
NoExpandHint	False
Number of Executions	1

Figure 3-9: Properties comparing Estimated Number of Rows and Number of Executions.

A **Key Lookup**, depending on the number of rows being returned, could be an indication that query performance might benefit from a covering index, although it's never a good idea to create a covering index for every single query that uses a lookup, because that would result in a wild growth of little-used indexes. A **Key Lookup** becomes expensive only when it is executed a lot of times, because each lookup is a **Clustered Index Seek** that will cause several logical reads (usually three), to traverse the b+tree structure to the page containing the data.

If a **Key Lookup** seems problematic, it's a good habit to verify that all the columns being returned are needed by the consuming application. If they are, then try to cover the query by extending an existing index, rather than creating a new one.

A covering index is created by either having all the columns necessary as part of the key of the index, or by using the `INCLUDE` operation to store extra columns at the leaf level of the index so that they're available for use with the index.

Reading a Heap

A heap is a table without a clustered index and therefore the rows are not stored in any order (beyond "order of arrival"). We can add nonclustered indexes to a heap. In this case, the nonclustered index has the location, the row identifier, where the row is stored within the heap rather than the clustered key value.

There are only two ways SQL Server can read data from a heap: via a scan or via a lookup.

Table Scan

Table Scans only occur against heap tables, so let's experiment now with a couple of queries against tables without a clustered index.

```
SELECT   dl.DatabaseUser,
         dl.PostTime,
         dl.Event,
         dl.DatabaseLogID
FROM     dbo.DatabaseLog AS dl;
```

Listing 3-6

This query results in the execution plan on display in Figure 3-10.

Figure 3-10: Execution plan with a Table Scan operator.

There's nothing new in the **SELECT** operator, so we can go straight to the other operator in this plan, **Table Scan**. When reading an index, the equivalent operator is a **Clustered Index Scan**.

A **Table Scan** can occur for several reasons, but it's often because there are no useful nonclustered indexes on the table, and the query optimizer has to search through every row in order to identify the rows to return. Another common cause of a **Table Scan** is a query that requests all the rows of a table, as is the case in this example.

When all, or the majority, of the rows of a table are returned then, whether an index exists or not, it is often faster to scan through each row and return them than look up each row in an index. Last, sometimes, especially for a table with few rows, scanning the table is faster even when there could be a selective index.

If the number of rows in a table is relatively small, **Table Scans** are generally not a problem. On the other hand, if the table is large and many more rows are processed than you need for the query, then you might want to investigate ways to rewrite the query to read fewer rows, or add an appropriate index to speed performance.

RID Lookup

We can put filter criteria into a query that could result in a **RID Lookup** as in Listing 3-7.

```
SELECT  dl.DatabaseUser,
        dl.PostTime,
        dl.Event,
        dl.DatabaseLogID
FROM    dbo.DatabaseLog AS dl
WHERE   dl.DatabaseLogID = 1;
```

Listing 3-7

This query results in a different execution plan than before.

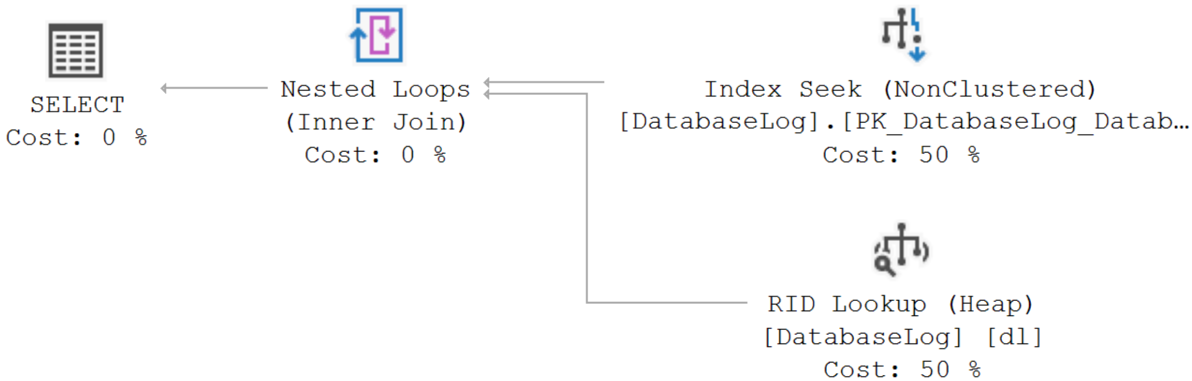


Figure 3-11: Execution plan showing a RID Lookup operator.

We have an **Index Seek** operator and a **RID Lookup (Heap)** operator, and a **Nested Loops** operator combining the two streams.

RID Lookup is the heap equivalent of the **Key Lookup** operation. As was mentioned before, nonclustered indexes don't always have all the data needed to satisfy a query. When they do not, an additional operation is required to get that data. When there is a clustered index on the table, it uses a **Key Lookup** operator as described above. When there is no clustered index, the table is a heap and must look up data using an internal identifier known as the **Row ID** or **RID**.

To return the results for this query, the query optimizer first performs an **Index Seek** on the primary key. While this index is useful in identifying the rows that meet the `WHERE` clause criteria, all the required data columns are not present in the index. How do we know this? In the Properties for the **Index Seek**, we see the value `Bmk1000` in the **Output List**.

Output List	Bmk1000, [AdventureWorks2014].[dbo].[DatabaseLog].DatabaseLogID
▷ [1]	Bmk1000
▷ [2]	[AdventureWorks2014].[dbo].[DatabaseLog].DatabaseLogID

Figure 3-12: Output list in the properties of the Index Seek.

This "`Bmk1000`" is an additional column, not referenced in the query. It's the RID, i.e. the location of the row in the heap, and it will be used in the **Nested Loops** operator to join with data from the **RID Lookup** operation. The `Bmk` prefix is a throwback from when these types of lookup operations were called "Bookmark Lookups."

If we look at the **Seek Predicates** of the **RID Lookup** operator as shown in Figure 3-13, you can see that the `Bmk1000` value is used again:

Seek Predicates	Seek Keys[1]: Prefix: Bmk1000 = Scalar Operator([Bmk1000])

Figure 3-13: Seek Predicates defined in the properties of the Index Seek operator.

`Bmk1000` is the key value, which is a row identifier or RID, from the nonclustered index. In this case, SQL Server had to look up only one row, which isn't a big deal from a performance perspective. If a **RID Lookup** returns many rows, however, you may need to consider taking a close look at the query to see how you can make it perform better by using less disk I/O – perhaps by rewriting the query, by adding a clustered index, or by using a covering index.

Summary

This chapter explained all the various mechanisms involved in reading data into execution plans using scans, seeks, and lookups against indexes, and scans and **RID Lookup**s against heap tables. A scan operator in a plan is not necessarily a bad thing, nor is a seek necessarily ideal. You need to read through the properties of the operators within execution plans to understand what each operator is doing, how many rows it processed, how many rows it returned, how the filtering mechanism worked, and so on. This will be a common theme throughout the rest of the book.

Chapter 4: Joining Data

In the previous chapter, we kept things simple, and stuck to single-table queries. However, in any real database, most of the execution plans you ever look at will have at least one join operator. After all, what's a relational database without the joins between tables? SQL Server is a relational database engine, which means that part of the designed operation is to combine data from different tables into single data sets. The execution plan exposes the operators that the optimizer uses to combine data.

This chapter is concerned mainly with various logical join operations in T-SQL. When implementing the join, SQL Server will take the two data inputs, one from each table generally, and combine the data according to the join criteria. The optimizer might choose to implement the join using one of four physical join operators:

- **Nested Loops** – For each row in the top data set, perform one search of the other data set for matching values.
- **Hash Match** – Using each row in the top data set, create a hash table, which will then be probed using the rows from the second data set to find any matching value.
- **Merge Join** – Read data from both inputs simultaneously and merge the two inputs, joining each matching row value. This requires both inputs to be sorted on the join column(s).
- **Adaptive Join** – Introduced in SQL Server 2017, this operator implements both the Nested Loops and the Hash Match algorithms, and chooses the option with the lowest cost at runtime, when the actual number of rows in the top input is known.

As we'll discuss, the physical join operator chosen by the optimizer will depend both on the size of the two input data streams and on how they are ordered.

Having covered these, we'll consider briefly other tasks that the optimizer can fulfill using `JOIN` operators, as well as other ways of combining data, such as via the `UNION` T-SQL command, and how SQL Server implements such operations.

Logical Join Operations

The join operators above implement eight logical join operations and two operations that combine data in a way that is not actually considered a join, as follows:

- Inner Join
- Outer Join (Left, Right, or Full)
- Semi Join (Left or Right)
- Anti Semi Join (Left or Right)
- Concatenation and Union.

The first two can be specified directly in T-SQL, whereas the **Semi Joins** are the logical operation associated with `EXISTS` (or `NOT EXISTS`) and `IN`, and Concatenation and Union are associated with `UNION ALL` and `UNION`.

The optimizer will choose what is deems to be the lowest-cost physical operator (**Nested Loops**, **Hash Match**, or **Merge Join**) to implement the logical join conditions described in the T-SQL statement.

Fulfilling JOIN Commands

This section is concerned explicitly with how the optimizer uses join operators to fulfill T-SQL `JOIN` commands.

Let's start off with the query in Listing 4-1, which retrieves `Employee` information from the `AdventureWorks2014` database, concatenating the `FirstName` and `LastName` columns in order to return the information in a more pleasing manner.

```
SELECT e.JobTitle, a.City,
  p.LastName + ', ' + p.FirstName AS EmployeeName
  FROM HumanResources.Employee AS e
    INNER JOIN Person.BusinessEntityAddress AS bea
      ON e.BusinessEntityID = bea.BusinessEntityID
    INNER JOIN Person.Address AS a
      ON bea.AddressID = a.AddressID
    INNER JOIN Person.Person AS p
      ON e.BusinessEntityID = p.BusinessEntityID;
```

Listing 4-1

Figure 4-1 shows the full, actual execution plan for this query.

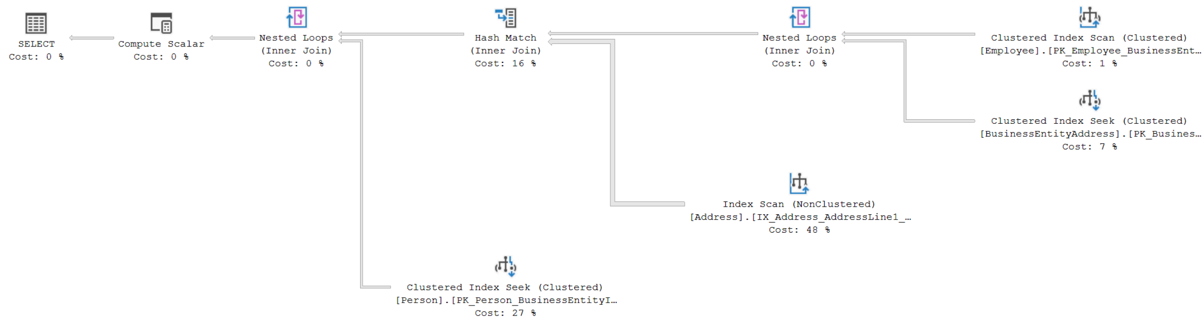


Figure 4-1: Execution plan showing two Nested Loops joins.

This plan has more operators than any we've seen so far but, as with every plan, we can read it either by starting at the top right and following the data arrows left, or read from left to right, following the order in which the operators are called.

If we were trying to tune this query, we might be tempted to simply jump in and look at those operators with the highest estimated cost, namely the **Clustered Index Seek** against the `Person.Person` table (27%), or the **Index Scan** on the `Person.Address` table (48%), or the **Hash Match** join operator (16%).

However, a better approach is first to take some time to understand broadly what the plan does. Reading right to left, it first joins matching rows in the `Employee` and `BusinessEntityAddress` tables using a **Nested Loops** operator, then uses a **Hash Match** operator to join rows in that data stream with rows in the `Address` table, based on matching `AddressID` values, and then uses another **Nested Loops** operator to join those rows with matching rows in the **Person** table (on `BusinessEntityID`). Finally, it adds a computed scalar value to each row and returns it.

We're going to focus on the role of each of the join operators, within the context of the plan as a whole, so we're just going to start in the top right of the plan, and take a more detailed look at the first **Nested Loops** join operator.

Nested Loops operator

A **Nested Loops** operator, often referred to as a **nested** iteration, takes a set of data, referred to as the "outer input," and compares it, one row at a time, to another set of data, called the "inner input" (on the graphical plan, these correspond to the two pipes feeding into the **Nested Loops** operator: the outer input on the top side, and the inner input on the bottom side). This sounds very like a cursor and, in effect, it is one. In fact, in this case, it's two cursors. The first cursor is the outer input data set. It will be processed one row at a time. The second cursor is the inner input, which will be processed one row at a time for each row from the outer input. As a result, the operator (or operators) in the inner input, the lower branch in the graphical plan, will each be executed multiple times, once for each row found in the outer input.

A **Nested Loops** operator can be highly efficient, as long as the outer input is small *and* it is cheap to search the inner input, which in the case of simple join operations is often achieved by indexing the "inner table" on the join column.

The execution plan in Figure 4-1 has two **Nested Loops** join operators. Let's start with an exploded view of the top right-hand corner of the plan, and take a look at one of them in more detail.

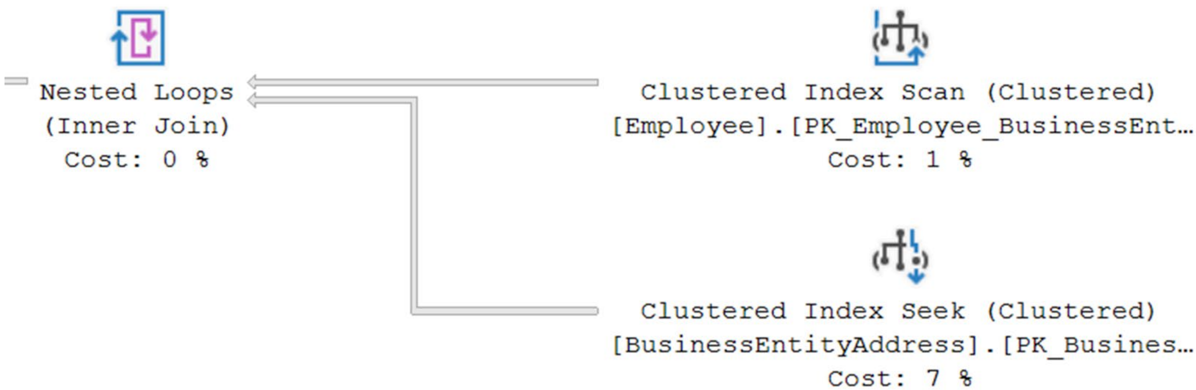


Figure 4-2: Nested Loops join within an inner and outer input.

In Figure 4-2, a **Nested Loops** iteration drives the joining of matching rows in the `Employee` and `BusinessEntityAddress` table. Notice that, in this example, an **Inner Join** is the logical operation associated with this physical operator.

The outer input for this **Nested Loops** operator is the data produced by a scan of the clustered index on the `Employee` table. It scans the entire index, retuning every row (290 rows, in this case). For each of these rows, the **Nested Loops** operator calls the operator in the inner input, searching for rows in the `BusinessEntityAddress` table with a matching `Busi-nessEntityID` value. In this case, this means that it executes 290 Index Seek operations on the clustered index. Figure 4-3 shows the properties of the **Nested Loops** operator.

Misc	
Actual Execution Mode	Row
Actual Number of Batches	0
Actual Number of Rows	290
Actual Rebinds	0
Actual Rewinds	0
Description	For each row in the top
Estimated CPU Cost	0.0012122
Estimated Execution Mode	Row
Estimated I/O Cost	0
Estimated Number of Executions	1
Estimated Number of Rows	275.573
Estimated Operator Cost	0.0012123 (0%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	69 B
Estimated Subtree Cost	0.0599363
Logical Operation	Inner Join
Node ID	4
Number of Executions	1
Optimized	False
Outer References	[AdventureWorks2014]
Output List	[AdventureWorks2014]
Parallel	False
Physical Operation	Nested Loops
WithUnorderedPrefetch	True

Figure 4-3: Property page of the Nested Loops operator.

As with most operators, there is a common set of properties on display, some of which don't apply and some of which are more useful than others. The following subsections review a few of the properties that are of interest in this case.

Estimated and Actual Number of Rows properties

Often, it's interesting to compare the **Actual Number of Rows**, 290, to the **Estimated Number of Rows**, 275.573 (proving this is a calculation, since you can't possibly return .573 rows).

A difference this small is not worth worrying about, but a larger discrepancy can be an indication that the optimizer has used inaccurate estimations of the number of rows that will need to be processed when selecting the plan, which could result in a suboptimal plan choice. There are many possible causes of this. For example, perhaps the optimizer had to generate a plan for a query containing a Predicate on a column with missing or stale statistics, or the optimizer may have reused a plan where the data volume or distribution in a column has changed significantly since the statistics were last created or updated. Alternatively, the data distribution in a column may be very non-uniform, making accurate cardinality estimations difficult, or the query may contain logic that defeats accurate estimations. Parameter sniffing may have occurred, resulting in a plan generated for an input parameter value with an estimated row count that is atypical of the row counts for subsequent input values. Chapter 8 discusses parameter sniffing in some detail.

There is another **Nested Loops** operator in Figure 4-1, which takes the 290 rows from the **Hash Match** join (discussed shortly) as the outer input, and so performs 290 separate seek operations of the clustered index on the inner `Person` table, joining matching rows in that table. Since the **Clustered Index Seek** on `Person` is estimated to be the costliest operation in the plan, it's worth peeking at its properties (see Figure 4-4).

Again, the first thing is to check that there is no wild disparity between estimated and actual number of rows processed. Initially, it seems like there might be, since the **Estimated Number of Rows** is just 1 but the **Actual Number of Rows** is 290. However, SSMS is inconsistent in how it reports these numbers; the estimated row count is per execution, and the optimizer estimated this **Clustered Index Seek** will be executed 275.573 times, for an estimated 275.573 rows returned. The actual rows count is simply the total number of rows processed, which is 290 (an average of 1 row returned per execution).

Actual Number of Rows	290
Actual Rebinds	0
Actual Rewinds	0
Defined Values	[AdventureWorks2014
Description	Scanning a particular
Estimated CPU Cost	0.0001581
Estimated Execution Mode	Row
Estimated I/O Cost	0.003125
Estimated Number of Executions	275.573
Estimated Number of Rows	1
Estimated Operator Cost	0.208827 (37%)
Estimated Rebinds	274.573
Estimated Rewinds	0
Estimated Row Size	113 B
Estimated Subtree Cost	0.208827
Forced Index	False
ForceScan	False
ForceSeek	False
Logical Operation	Clustered Index Seek
Node ID	10
NoExpandHint	False
Number of Executions	290

Figure 4-4: Nested Loops operator showing runtime statistics.

The fact that the optimizer estimates that it will execute this **Clustered Index Seek** on the `Persons` table about 257 times explains at least partly why it is the highest-cost operator in the plan. The **Clustered Index Seek** on the `BusinessEntityAddress` table is estimated to be executed even more often, 290 times, but because this table uses far fewer bytes per row it has one less level of index pages, reducing the amount of work per seek from three to two logical reads.

Taking the time to understand how the operations interact will permit you to understand why the costs are distributed the way they are.

Outer References property

There are two ways that the **Nested Loops** operator can resolve a join condition. One way is via the **Outer References** property. In this case, operators on the inner input of the join, the lower branch in a graphical plan, use values from the outer input to deliver the results. If ten values are pushed down from the outer input into the inner input, referred to as *Outer References*, then this implies that the inner input will be executed ten times, searching for matching rows. The inner input will only ever return matching rows, and so the **Nested Loops** operator does not have to do any work in terms of validating matching data.

You can see the **Outer References** property within the tooltip or the property page of the **Nested Loops** operator, as shown in Figure 4-5.

⊟ Outer References	[AdventureWorks2014].[HumanResources].[Employee].BusinessEntityID, Expr1008
⊟ [1]	[AdventureWorks2014].[HumanResources].[Employee].BusinessEntityID
Alias	[e]
Column	BusinessEntityID
Database	[AdventureWorks2014]
Schema	[HumanResources]
Table	[Employee]
⊟ [2]	Expr1008
Column	Expr1008

Figure 4-5: Outer References details of the Nested Loops join.

You can see that in this case values from the `BusinessEntityID` column are being pushed down to the inner input. The `BusinessEntityID` column is the leading column of a usable index on `BusinessEntityAddress`, so by pushing it into the inner input it facilitates a seek operation (see Figure 4-2).

Incidentally, the other value pushed down, `Expr1008`, has no other reference anywhere within the execution plan, even if you search the XML. Therefore, it's likely that it's an artifact of the process of comparison in the **Clustered Index Seek** operator.

The second way that the **Nested Loops** operator can resolve a join condition is via the **Predicate** property. This happens when the inner input has no pushed-down values, so it will always return the same results on every subsequent execution. Here, the **Nested Loops** operator applies the join **Predicate** to the rows returned from the inner input, and only passes on matching rows. We'll see an example of this in Chapter 5.

Rebind and Rewind properties

A **Rebind** and a **Rewind** both count of the number of times the **Init()** method is called by an operator, but do so under different circumstances. The **Init()** method initializes the operator and sets up any required data structures. In most cases, this happens once for an operator, in any plan. However, a **Nested Loops** operator executes its inner input once for every row in the outer input. This means that the **Init()** method on the operators in the inner input can be called more than once.

Every execution is either a **Rebind** or a **Rewind**. A **Rebind** occurs for the very first execution of the inner input, and then each time the values of the column pushed down from the outer input change (i.e. when the values marked by **Outer References** change).

A **Rewind** occurs when the values are unchanged, or when there are no Outer References (so the join condition is resolved using a Predicate, within the **Nested Loops** operator). In the latter case, you'll always see a single **Rebind** for the first execution, and then, from that point forward, a series of **Rewinds**.

For the **Nested Loops** operator depicted in Figure 4-5, the join is resolved using values in the `BusinessEntityID` column as the Outer References, and there are 290 distinct values for this column (it is the primary key). Notionally, this means that all 290 executions of the inner input are **Rebinds**.

However, Figure 4-6 shows the properties of the **Clustered Index Seek**, which is the inner input of the **Nested Loops** operator, and we can see that the **Rebinds** and **Rewinds** are zero in each case.

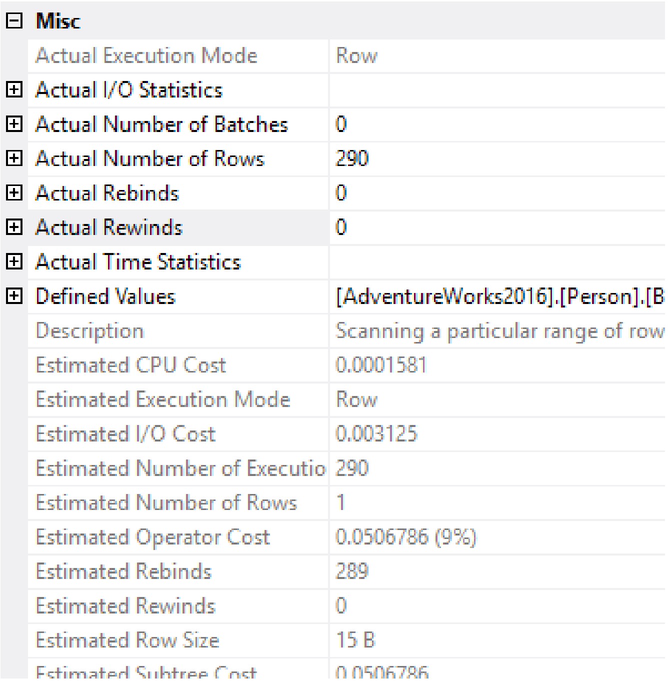


Figure 4-6: Properties of the Clustered Index Seek.

Of course, knowing whether the outer input value changed is only useful to the optimizer if the results of the previous execution of the inner input, for the same value, are stored somewhere. For example, **Spool** operators save their results in a worktable, a **Sort** saves them in memory and a **Table Valued Function** populates a table variable. When these operators are present, the optimizer can streamline the execution process because, if it knows it has the rows it needs stored somewhere then, when a **Rewind** occurs, there is no need to re-do all the work to produce them again.

Therefore, **Rebinds** and **Rewinds** are only relevant, and the property values only populated, when the **Nested Loops** operator interacts with one of the following operators, each of which can save the results from its previous execution:

- **Index Spool**
- **Remote Query**
- **Row Count Spool**
- **Sort**
- **Table Spool**
- **Table valued function**.

We won't describe any of the operators listed above until Chapter 5, so we won't walk through an example here. However, let's say the outer input of a **Nested Loops** join produces 14 rows, the join condition is resolved using Outer References and there are 10 distinct values in the Outer References column. The inner input is an **Index** Spool, the properties of which show that the 14 executions of this inner input comprise 10 **Rebinds** and 4 **Rewinds**.

⊞ Actual I/O Statistics	
⊞ Actual Number of Batches	0
⊞ Actual Number of Rows	14
⊞ Actual Rebinds	10
⊞ Actual Rewinds	4
⊞ Actual Time Statistics	
Description	Reformats the data from

Figure 4-7: Rebinds and Rewinds for an Index Spool.

For each **Rewind**, there is no need to execute any operators downstream (to the right) of the spool, as the matching values are already stored in the spool's worktable. This means that each of these operators execute only 10 times, once for each **Rebind** of the inner input.

Hash Match (join)

The optimizer can use a **Hash Match** operator to implement any of the logical `JOIN` operations, though it can only use it to implement a `UNION` in cases where the probe input is guaranteed to have no duplicates, and it is not used at all for Concatenation (`UNION ALL`), which is instead done by the **Concatenate** operator. A **Hash Match** can also aggregate data from a single data input, but we'll focus exclusively on join implementations here, covering aggregation in Chapter 5.

When used to implement logical join operations, the **Hash Match** operator makes a single pass over two data inputs. One data input (the "build") is stored in memory, in a so-called hash table, and then this structure is used to compare data by probing, or comparing, from the other data input, to arrive at the matching output set.

How Hash Match joins work

Figure 4-8 shows an exploded view of the section of the plan for Listing 4-1 that contains a **Hash Match**, in this case used to implement an inner join.

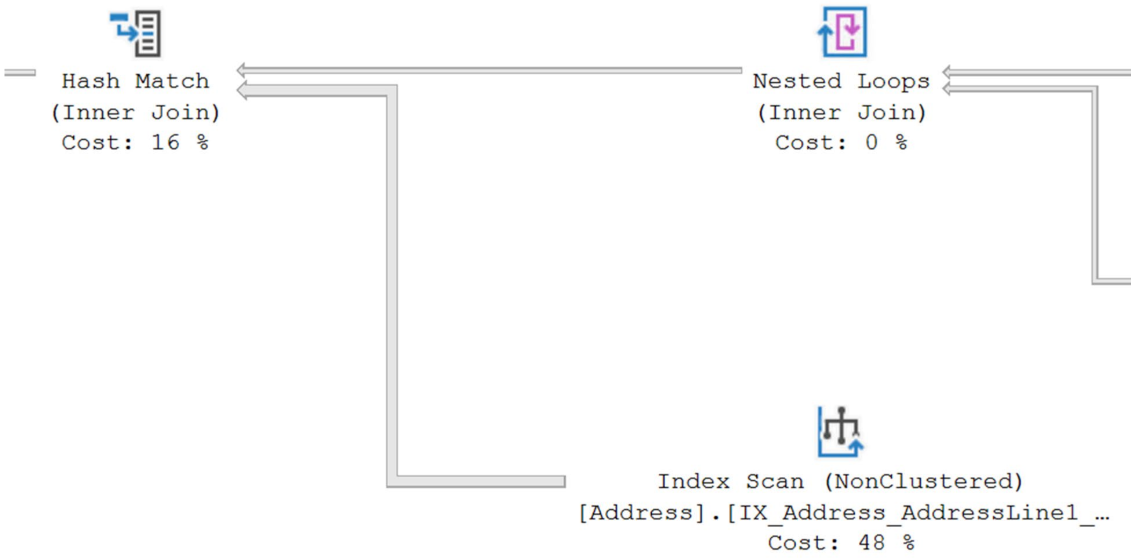


Figure 4-8: Hash Match join showing two inputs.

In a **Hash Match** join operator, the top input is called the **Build** input and the bottom input is called the **Probe** input. In this example, the **Build** input is the 290 rows produced by the first **Nested Loops** operator in the plan, discussed above. This is by far the smaller of the two inputs.

The **Hash Match** operator reads the **Build** input, *hashes* the join column (in this case `AddressID`), and stores the column values, and their hashes, in a *hash table*, in memory. It then reads the rows in the **Probe** input one row at a time, in this case the 19614 rows that result from a **Nonclustered Index Scan** on the `Address` table. For each row, it produces a hash value for the `AddressID` column that it can compare to the hashes in the hash table, looking for matching values.

Hashing and Hash Tables

Hashing is a programmatic technique where data is converted into a simple number to make searching for that data much more efficient. For example, SQL Server converts a row of data in a table into a value that is derived from the columns in that row that are designated as the input to the hash function.

A *hash table* is a data structure in which SQL Server attempts to divide all the elements into equal-sized categories, or buckets, to allow quick access to the elements. The hashing function determines into which bucket an element goes. For example, SQL Server can take a column from a table, hash it into a hash value, and then store the matching rows in memory, within the hash table, in the appropriate bucket.

Figure 4-9 shows the **Hash Keys Build** and **Hash Keys Probe** properties for the **Hash Match** join operator. These properties reveal which columns from each input are hashed by the operator, when building the hash table and comparing the rows from the **Probe** input.

⊟ Hash Keys Build	[AdventureWorks2014].[Person].[BusinessEntityAddress].AddressID
Alias	[bea]
Column	AddressID
Database	[AdventureWorks2014]
Schema	[Person]
Table	[BusinessEntityAddress]
⊟ Hash Keys Probe	[AdventureWorks2014].[Person].[Address].AddressID
Alias	[a]
Column	AddressID
Database	[AdventureWorks2014]
Schema	[Person]
Table	[Address]

Figure 4-9: Hash Keys Build and Hash Keys Probe values.

Performance considerations for Hash Match joins

A **Hash Match** join operator is blocking during the Build phase. It has to gather all the data in order to build a hash table prior to performing its join operations and producing output. The optimizer will only tend to choose **Hash Match** joins in cases where the inputs are not sorted according to the join column. **Hash Match** joins can be efficient in cases where there are no usable indexes or where significant portions of the index will be scanned. If the inputs are already sorted on the join column, or are small and cheap to sort, then the optimizer may often opt to use a **Merge Join** instead.

However, a **Hash Match** join is often the best choice when you have two unsorted inputs, both large, or one small and one large. The optimizer will always choose what it estimates to be the smaller of the data inputs to be the Build input, which provides the values in the hash table. The goal is many hash buckets with few rows per bucket (i.e. minimal hash collisions, as few duplicate hashed values as possible). This makes finding matching rows in the **Probe** input fast, even with two large inputs, because the optimizer only needs to search for matches in the basket with the same hash value, instead of scanning all the rows.

Performance problems with **Hash Match** only really occur when the Build input is much larger than the optimizer anticipated, so that it exceeds the memory grant, and subsequently spills to disk.

So, given that the section of our plan, in Figure 4-6, contains what the optimizer reckons are the second and third most expensive operators in the plan, in the **Index Scan** on the `Address` table, and the **Hash Match** join itself, should we attempt to "tune" these operations? Sometimes, you can. While a **Hash Match** join may represent the current, most efficient way for the query optimizer to join two tables, it's possible that we can tune our query to make available to the optimizer more-efficient join techniques, such as using **Nested Loops** or **Merge Join** operators. For example, seeing a **Hash Match** join in an execution plan sometimes indicates:

- a missing or unusable index
- a `WHERE` clause with a calculation or conversion that makes it non-SARGable (a commonly used term meaning that the search argument, "sarg," can't be used); this means it won't use an existing index.

However, it depends simply on what's happening in the query. Generally, you don't tune individual operators; you use them to understand the execution plan. Some expensive operators can be targeted, others are estimated to be expensive but aren't really, and some really are expensive but are still an essential element of the cheapest plan overall. A **Hash Match** join often falls in the latter category, as the alternatives are either **Nested Loops** with lots of executions of the inner input, or using sorts to enable a **Merge Join** (covered later). In this case, with no `WHERE` clause, the **Hash Match** is simply an efficient mechanism to put all the data together to satisfy the query in question.

Compute Scalar

As each row emerges from the second **Nested Loops** operator, in our plan in Figure 4-1, it passes into a **Compute Scalar** operator. This is not a type of join operation but since it appears in our plan, we'll cover it here.

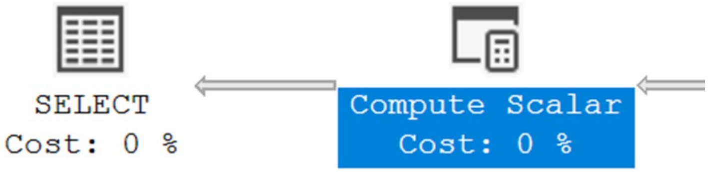
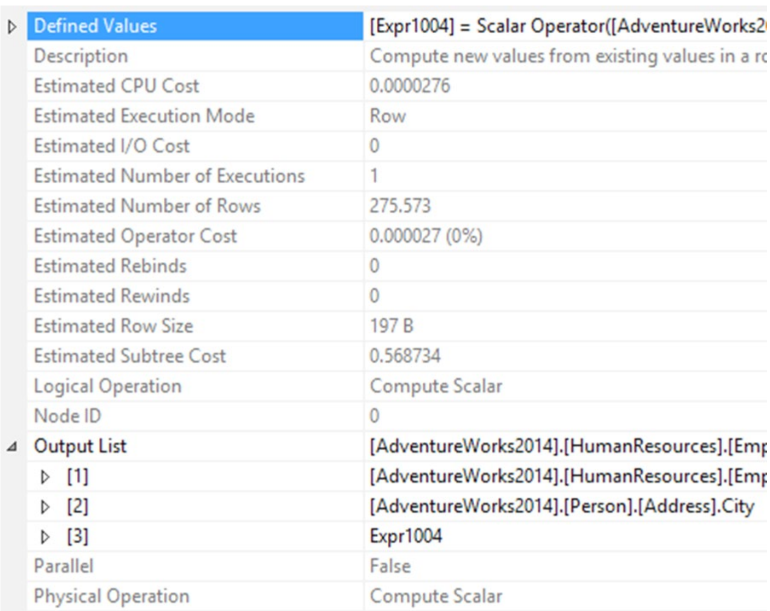


Figure 4-10: Compute Scalar operator.

Figure 4-11 shows the **Properties** window for this operator.



Defined Values	[Expr1004] = Scalar Operator([AdventureWorks2
Description	Compute new values from existing values in a ro
Estimated CPU Cost	0.0000276
Estimated Execution Mode	Row
Estimated I/O Cost	0
Estimated Number of Executions	1
Estimated Number of Rows	275.573
Estimated Operator Cost	0.000027 (0%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	197 B
Estimated Subtree Cost	0.568734
Logical Operation	Compute Scalar
Node ID	0
Output List	[AdventureWorks2014].[HumanResources].[Emp
[1]	[AdventureWorks2014].[HumanResources].[Emp
[2]	[AdventureWorks2014].[Person].[Address].City
[3]	Expr1004
Parallel	False
Physical Operation	Compute Scalar

Figure 4-11: Properties of the Compute Scalar operator.

This is simply a representation of an operation to produce one or more simple, scalar values, usually from a calculation – in this case, the alias `EmployeeName`, which combines the columns `Contact.LastName` and `Contact.FirstName` with a comma between them. While this was not a zero-cost operation, 0.000027, the cost estimate is so trivial in the context of the query as to be essentially free. You can see what this operation is doing by looking at the definition for the highlighted property, **Defined Values**, but to really see what the operation is doing, click on the ellipsis on the right side of the property page. This will open the expression definition as shown in Figure 4-12.

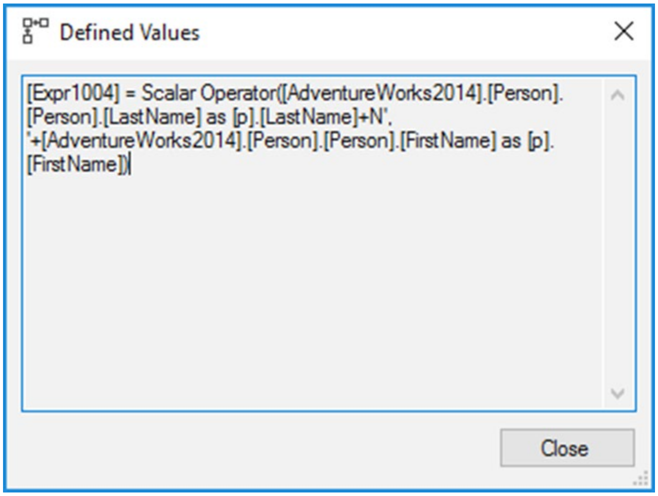


Figure 4-12: Defined values of the Compute Scalar operator.

While the **Compute Scalar** operator in this case is very straightforward and clear, this won't always be the case. These operations are not costed completely by the optimizer, so you may see situations where the estimates for the work involved are radically off. The value is calculated as 0.0000001 * (Estimated Number of Rows), regardless of the complexity or number of calculations being done. Also, the logical representation of where the **Compute Scalar** occurs within the plan is represented here; it's not necessarily where the physical process occurs within the plan. That's why you sometimes see no values for actual number of rows or actual executions on a **Compute Scalar** operator, in an actual execution plan; if all the computations are processed elsewhere, the operator does not run at all and can therefore not track these numbers.

Because of the lack of accurate estimated costs, you should understand exactly what a **Compute Scalar** operation represents within your execution plan because they can represent a hidden cost, especially when scalar user-defined functions (UDFs) are involved.

Merge Join

A **Merge Join** operator works from ordered data only. It takes the data from two inputs and uses the fact that the data in each input is ordered on the join column to simply merge the two inputs, joining rows based on the matching values, which it can do very easily because the order of the values will be identical. A **Merge Join** is a non-blocking operator; as it joins each row, with matching values on the join column, it passes it on to the next operator upstream.

If each data input is ordered by the join column, this can be one of the most efficient join operations. However, the data is frequently not ordered, and so sorting it for a **Merge Join** requires the addition of a **Sort** operator to ensure it works; the sorting requirement can make plans with a **Merge Join** operation less efficient, depending on how the sort is satisfied.

However, because a **Merge Join** ensures that the output from the join process itself is also ordered, it may sometimes be better to pay the cost of a single **Sort** operation to ensure ordered output for additional **Merge Join** operations in a plan.

How Merge Joins work

To demonstrate a **Merge Join** operator, we need a new query.

```
SELECT c.CustomerID
FROM Sales.SalesOrderDetail AS sod
    INNER JOIN Sales.SalesOrderHeader AS soh
        ON sod.SalesOrderID = soh.SalesOrderID
    INNER JOIN Sales.Customer AS c
        ON soh.CustomerID = c.CustomerID;
```

Listing 4-2

Figure 4-13 shows the execution plan for this query.

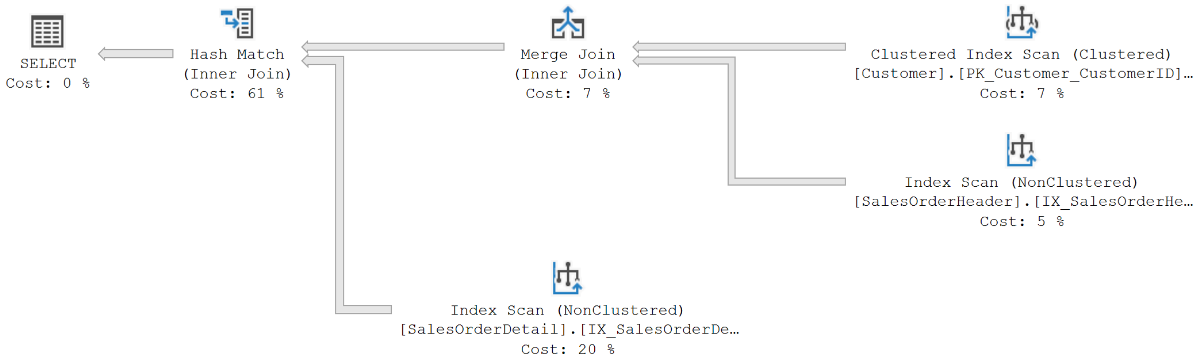


Figure 4-13: Execution plan showing a Merge Join.

Here, the optimizer has selected a **Merge Join** operator to perform the `INNER JOIN` between the `Customer` and `SalesOrderHeader` tables, based on matching values of `CustomerID`. Since the query did not specify a `WHERE` clause, a scan was performed on each table to return all the rows in each table. Also, you'll note that the order of the join operations is not the same as that specified by the query. The optimizer can choose to rearrange the order of tables within the plan as it sees fit, to arrive at the best possible plan. Here, the input with guaranteed unique values, the `Customer` table, is used as the top input, so we have a one-to-many join.

The data in the top input, the **Clustered Index Scan** on the `Customer` table, is ordered by `CustomerID`. The bottom input is the data from a **Nonclustered Index Scan** on the `SalesOrderHeader` table. Again, this nonclustered index is ordered by `CustomerID`. In other words, both data inputs are ordered on the join column, as confirmed, in the Properties of the **Merge Join** operator.

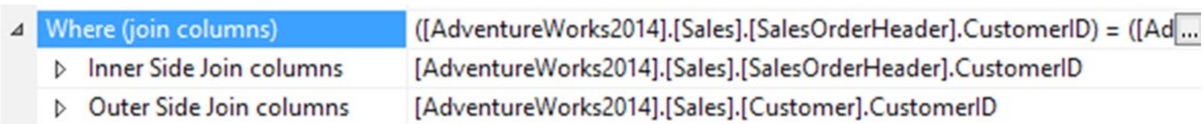


Figure 4-14: Properties of the Merge Join showing Where property values.

Once the **Merge Join** has joined two of the tables, the optimizer joins the third table to the first two using a **Hash Match** join, as discussed earlier. Finally, the joined rows are returned.

Performance considerations for Merge Joins

The key to the performance of a **Merge Join** is that the inputs are sorted by the join columns. We can see that the results from the scans are sorted if we consult the properties of those operators. Figure 4-13 shows the **Clustered Index Scan** operator, with an **Ordered** property value of **True**, meaning that the optimizer requires the input to be ordered.

Ordered	True
Output List	[AdventureWorks2014]
Parallel	False
Physical Operation	Clustered Index Scan
Scan Direction	FORWARD

Figure 4-15: Scan properties showing Ordered value.

If you see an **Ordered** property set to **False**, that doesn't mean that the data being retrieved is not, in fact, ordered; it merely means that the optimizer does not *require* the data to be ordered to satisfy the rest of the plan.

So, in this example, the output of the scans is ordered by the join columns, and no additional sorting is necessary. If one or more of the inputs is not ordered, and the query optimizer chooses to sort the data in a separate operation before it performs a **Merge Join**, it might indicate that you need to reconsider your indexing strategy, especially if the **Sort** operation is for a large data input. Could you, for example, modify an existing index so that the optimizer can avoid the need for the **Sort** operation?

The **Merge Join** in this example is for a one-to-many join, as we can see by inspecting the **Many to Many** property value for the operator, which is **False**.

Misc	
Actual Execution Mode	Row
Actual I/O Statistics	
Actual Number of Batches	0
Actual Number of Rows	31465
Actual Rebinds	0
Actual Rewinds	0
Actual Time Statistics	
Description	Match rows from
Estimated CPU Cost	0.116428
Estimated Execution Mode	Row
Estimated I/O Cost	0
Estimated Number of Executions	1
Estimated Number of Rows	31294.3
Estimated Operator Cost	0.1164305 (7%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	15 B
Estimated Subtree Cost	0.308297
Logical Operation	Inner Join
Many to Many	False
Node ID	1
Number of Executions	1
Output List	[AdventureWorks
Parallel	False
Physical Operation	Merge Join
Residual	[AdventureWorks
Where (join columns)	([AdventureWork

Figure 4-16: Merge Join properties showing Many to Many value.

However, a **Merge Join** for a many-to-many join condition can prove to be a lot more expensive, and the performance a lot worse. Consider the example in Listing 4-3.

```
SET STATISTICS IO ON;
SELECT      sod.ProductID,
            sod.SalesOrderID,
            pv.BusinessEntityID,
            pv.StandardPrice
FROM        Sales.SalesOrderDetail   AS sod
INNER JOIN  Purchasing.ProductVendor AS pv
      ON    pv.ProductID = sod.ProductID;
SET STATISTICS IO OFF;
```

Listing 4-3

Figure 4-17 shows the execution plan for this query.

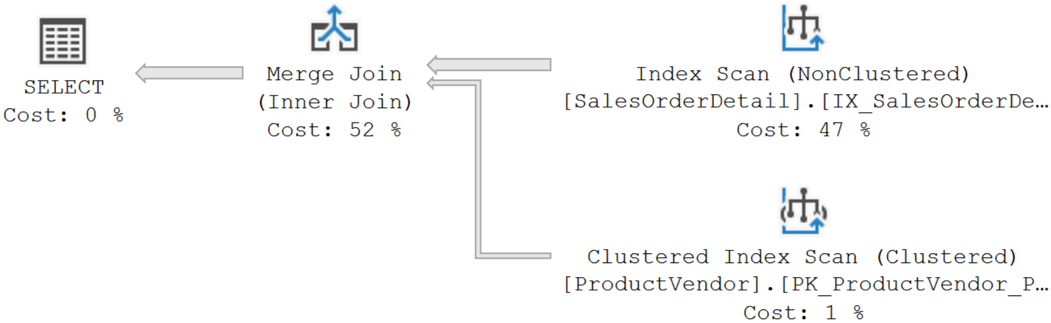


Figure 4-17: Execution plan with a Many to Many Merge Join.

The optimizer tries to infer uniqueness on the join columns of each input, by looking at unique indexes as well as at plan elements, such as a **Distinct** or **Aggregate** operator in a branch of the plan. If one of the inputs of the join would be guaranteed unique, it would be the top input, **Many to Many** would be False, and the join efficient. However, in this case, both inputs can, and do, have multiple rows with the same `ProductID` value. In the **Merge Join**, **Many to Many** is true, and the join becomes less efficient. This can be seen in the `SET STATISTICS IO` output:

```
(74523 rows affected)
Table 'Worktable'. Scan count 19, logical reads 18013, physical
reads 0, read-ahead reads 0, lob logical reads 0, lob physical
reads 0, lob read-ahead reads 0.
Table 'ProductVendor'. Scan count 1, logical reads 7, physical
reads 1, read-ahead reads 8, lob logical reads 0, lob physical
reads 0, lob read-ahead reads 0.
Table 'SalesOrderDetail'. Scan count 1, logical reads 250, physical
reads 0, read-ahead reads 326, lob logical reads 0, lob physical
reads 0, lob read-ahead reads 0.
```

The problem is that, for a **Many to Many** join, rows from the bottom input must be copied to a worktable in **tempdb**. If a new row from the top input has the same value in the join column as the previous, the temporary table is used to rewind to the start of the duplicates as needed in the comparison. If the data from the top input changes, the temporary table is cleared out and loaded with new matching rows from the bottom. The I/O stats demonstrate the impact of this extra activity in the temporary table: the number of logical reads is more than 98% of the total number of logical reads of the query as a whole.

In this case, there are duplicates for `ProductID` in both tables so there is little we can do to change this. However, it is not uncommon to see **Merge Join** operators with **Many to Many** set to True where it could have been False. This is often related to missing constraints in the tables, or to embedding columns in expressions (such as implicit or explicit data type conversions). The optimizer can only correctly infer uniqueness if there is a uniqueness constraint on a column that is not embedded in an expression.

Adaptive Join

Introduced in SQL Server 2017, and also available in Azure SQL Database and Azure SQL Data Warehouse, the **Adaptive Join** is a new join operation. Currently it only works with batch mode (see Chapter 12), but that may change as cumulative updates are released, or in updates to Azure.

The optimizer can choose an **Adaptive Join** operator to defer the exact choice of physical join algorithm, either a **Hash Match** or a **Nested Loops**, until runtime, when the actual number of rows in the top input is known rather than estimated.

To see the **Adaptive Join** in action, we need a batch mode plan, which requires a column-store index. Listing 4-4 creates a nonclustered columnstore index on the `Production.TransactionHistory` table.

Once you've finished testing the example in this section, please return to this listing and run the `DROP INDEX` batch to remove the columnstore index.

```
DROP INDEX IF EXISTS ix_csTest ON Production.TransactionHistory;
GO
CREATE NONCLUSTERED COLUMNSTORE INDEX IX_CSTest
ON Production.TransactionHistory
(
    TransactionID,
    ProductID,
    ActualCost
);
```

Listing 4-4

With this index in place, executing the simple query in Listing 4-5 (on SQL Server 2017 or Azure SQL Database, with database compatibility level set to at least 140 in either case) will result in an **Adaptive Join**.

```
SELECT p.Name AS ProductName,
       th.ActualCost
FROM Production.TransactionHistory AS th
    JOIN Production.Product AS p
        ON p.ProductID = th.ProductID
WHERE th.ActualCost > 0
      AND th.ActualCost < .21;
```

Listing 4-5

Figure 4-18 shows the actual execution plan.

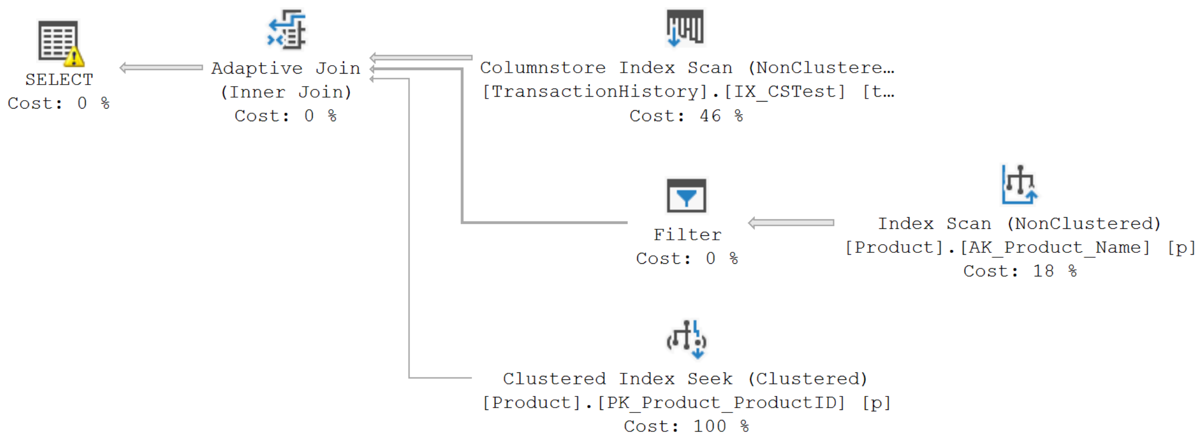


Figure 4-18: Execution plan showing an Adaptive Join.

The first thing I want to point out about this plan is the warning we have on the **SELECT** operator, which is an **Excessive Memory Grant** warning. We'll deal with that warning in Chapter 12.

The first thing you will likely notice about the **Adaptive Join** operator is that, unlike all the other join operators we've seen up to this point, it has three inputs. The top input is a scan of a nonclustered columnstore index (we won't cover the specifics of plans involving column-store indexes until Chapter 12). The lower inputs, an **Index Scan** plus **Filter** and a **Clustered Index Seek** are, respectively, the operators to support either a **Hash Match** join, or a **Nested Loops** join.

Since a columnstore index doesn't have statistics in the same way that a rowstore index does, there's not always an easy way for the optimizer to accurately estimate the number of rows returned.

All operations necessary for either join type are defined and stored with the execution plan at compilation time. If this plan were retrieved from the plan cache, or the Query Store, it would show both possible branches to support both possible join types. In short, you can't tell which path was taken without considering the properties of an actual plan. Any estimated plan will show both possible branches.

Just as for the **Hash Match** join operator, the **Adaptive Join** operator has a Build phase, which stores the rows for the top input in a hash table in memory, which is why there is a memory grant. The operator is blocking during this phase.

Once the top input is processed and stored in the hash table, the exact number of rows is known. This number is now used to decide whether to proceed as a **Hash Match** or **Nested Loops** join. That determination is made by comparing the number of values in the hash table to a threshold determined by the optimizer. For any given join operation, that value could vary depending on the data structures, the query, and the statistics on the indexes. You can check the value being used by looking to the properties of the **Adaptive Join** operator.

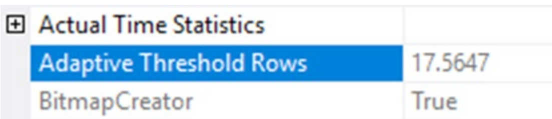


Figure 4-19: The Adaptive Threshold Rows property.

If the number of rows in the hash table is above this value, in this case 18 rows or greater, then a **Hash Match** join will be used. The hash table will use the upper branch of the two inputs to gather the necessary data and, from that point forward, acts just like a **Hash Match** join. In this case, that would mean an **Index Scan** against the `Product` table using the `AK_Product_Name` index. If the number of rows in the hash table falls below the threshold value, then the **Nested Loops** method is used, resulting in one **Clustered Index Seek** on the `Product` table, using a completely different index, `PK_Product_ProductID`, for each of the rows in the hash table.

There are three ways, within the execution plan, to tell which of the two choices was used during execution. Each method obviously requires you to capture an actual execution plan. The first method is to look to the properties of the **Adaptive Join** itself. Figure 4-20 shows that in this case the Actual Join Type is HashMatch.

Actual Join Type	HashMatch
Actual Number of Batches	2
Actual Number of Rows	323
Actual Rebinds	0
Actual Rewinds	0
Actual Time Statistics	
Adaptive Threshold Rows	17.5647
BitmapCreator	True
Defined Values	[[AdventureW
Description	Chooses dyna
Estimated CPU Cost	0.0000065
Estimated Execution Mode	Batch
Estimated I/O Cost	0
Estimated Join Type	HashMatch

Figure 4-20: Adaptive Join properties showing Actual and Estimated Join Type.

At the bottom is the **Estimated Join Type**, also HashMatch. So, the Estimated Number of Rows and the Actual Number of Rows were reasonably accurate. The row threshold was met, so the **Adaptive Join** used the hash table to complete the join process as a **Hash Match** join.

Another way to see what type of join was used is to look at the two inputs in the plans. Figure 4-21 shows the tooltip for each pipe feeding to the **Adaptive Join**. The top tooltip is for the **Hash Match** join input and the bottom is for the **Nested Loops** join input.

Actual Number of Rows	5
Estimated Number of Rows	50.4
Estimated Row Size	65 B
Estimated Data Size	3276 B

Actual Number of Rows	0
Estimated Number of Rows	1
Estimated Row Size	61 B
Estimated Data Size	61 B

Figure 4-21: Two tooltips showing Actual Number of Rows.

You can see that the top input had 5 actual rows and the bottom input has 0, indicating that, in this case, the **Adaptive Join** consumed the first of the two possible inputs.

Finally, you can look at the end of the branch, the data access point within the execution plan to count the number of executions. This can be the most reliable method since, even if zero rows were returned, at least one execution of one of the operators would still be recorded. Figure 4-22 illustrates the bottom branch which was not executed:

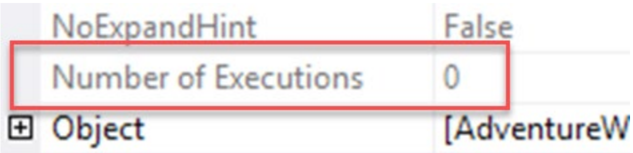


Figure 4-22: Properties showing no executions for an Index Seek.

You can also use Extended Events to capture **Adaptive Join** "misses," using the event **adaptive_join_skipped** to find out why an Adaptive Join couldn't be used by the optimizer, for a particular query.

To summarize, the **Adaptive Join** offers the optimizer the best of both worlds (almost). If the actual rowcounts are low, the **Nested Loops** branch of the plan will execute. This ends up costing slightly more than if the optimizer had just chosen a **Nested Loops** join during optimization, but if it had chosen the **Hash Match** join during optimization, for what turned out to be a low rowcount, it would have been a far less efficient plan. For high rowcounts, the **Nested Loops** branch of the adaptive plan will execute, which results in a very similar plan cost as for a standard **Hash Match** join.

Other Uses of Join Operators

The optimizer uses the physical join operators to fulfill tasks other than the T-SQL `JOIN` keyword. In Chapter 3, for example, we saw the optimizer use a **Nested Loops** operator to combine data from an **Index Seek** and its associated **Key Lookup** data.

In addition, sometimes the optimizer uses a join operator to implement a non-join request in a query, such as `APPLY` or `EXISTS`. We'll save coverage of `APPLY` until Chapter 7, but let's take a brief look here at how the optimizer implements `EXISTS` operations. These are sometimes called **Semi Joins**, because even though the sources need to be combined, returned data is still from a single source only. Listing 4-6 shows a simple example.

```
SELECT bom.ProductAssemblyID,
       bom.PerAssemblyQty
FROM Production.BillOfMaterials AS bom
WHERE EXISTS (  SELECT *
                FROM Production.BillOfMaterials AS bom2
                WHERE bom.BillOfMaterialsID = bom2.ComponentID
                      AND bom2.EndDate IS NOT NULL
             );
```

Listing 4-6

When we run this query, the execution plan is a little different than the straightforward join operations listed earlier.

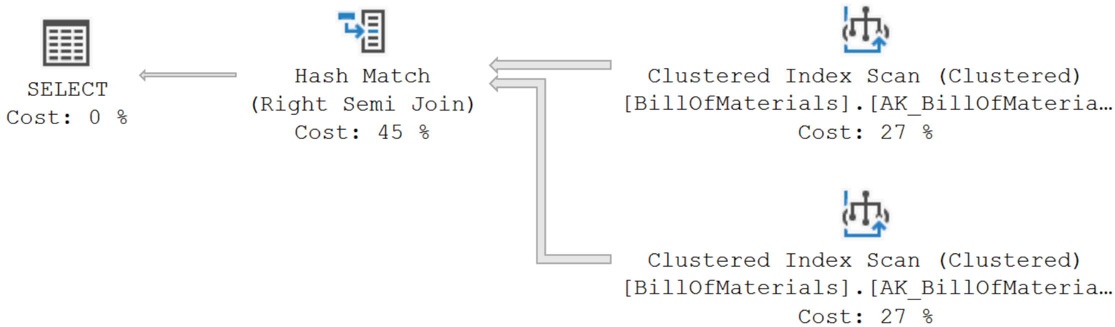


Figure 4-23: Execution plan showing Right Semi Join.

The optimizer selected a plan that performs a scan of the clustered index two times to satisfy the query and then the results are put together using a **Hash Match** join operation. However, this **Hash Match** is designated as a **Right Semi Join**, unlike the earlier ones which were all **Inner Joins**.

Unlike an **Outer Join**, which will return all valid combinations of rows from the two inputs plus a single copy of each unmatched row from the top input, a **Semi Join** returns a single copy of each row from one input that has at least one matching row in the other input. It does not add rows from the other input to the data; it is only used for the existence of a matching row.

The optimizer uses, in this case, a **Hash Match** operator to perform the **Semi Join** logical processing. A hash table of values from the first data set is created and then probes from the second data set are used to find matching values. If any value matches, the row from the second data set is returned and no other comparisons are made.

There are both **Right** and **Left Semi Joins**. The optimizer determines which direction it's going to perform the functions depending on the rest of the necessary operations to satisfy the query in question.

You may also see **Anti Semi Join** logical join types used in an execution plan. As suggested by the name, these are the reverse of the **Semi Join** operations: they return a single copy of each row from one input that does *not* have a match in the other input (similar to `NOT EXISTS`).

Concatenating Data

Finally, as well as joining data together, it is possible to concatenate data. The most common type of data concatenation is through the `UNION ALL` keyword. However, you may also see concatenation operations occur within an execution plan from other types of queries. For example, using variables in an `IN` clause may result in a concatenation operation within an execution plan. A **Concatenation** operator will always have two or more inputs, and it simply processes each of the inputs in order, from top to bottom, and concatenates them.

Let's look at a simple example of concatenation.

```
SELECT p.LastName,
       p.BusinessEntityID
FROM Person.Person AS p
UNION ALL
SELECT p.Name,
       p.ProductID
FROM Production.Product AS p;
```

Listing 4-7

This query combines a list of the `Person.LastName` column with the `Product.Name` column. The execution plan looks like Figure 4-24.

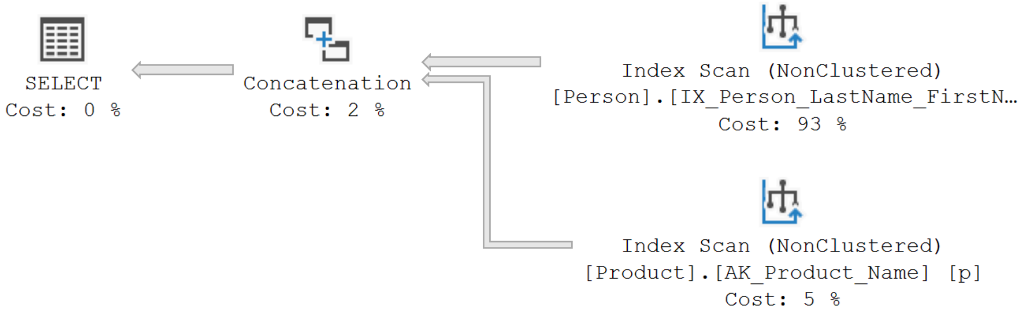


Figure 4-24: Execution plan showing Concatenation operator.

This execution plan is very straightforward. The **Concatenation** operator first calls the top input, passing rows retrieved to its parent, until it has received all rows. After that it moves on to the second input, repeating the same process. Each of the data access operators is simply retrieving all the data from the referenced indexes. In this case, there are only the two data sets, but **Concatenation** can have as many inputs as necessary. If we look at the properties for the operator, shown in Figure 4-25, you can see how the information is resolved.

⊟ **Misc**	
Actual Execution Mode	Row
⊞ Actual Number of Batches	0
⊞ Actual Number of Rows	20476
⊞ Actual Rebinds	0
⊞ Actual Rewinds	0
⊞ Actual Time Statistics	
⊟ Defined Values	[Union1002] = ([AdventureWorks2014].[Person].[Person].Last
⊟ Union1002	[AdventureWorks2014].[Person].[Person].LastName, [Adventu
⊞ [1]	[AdventureWorks2014].[Person].[Person].LastName
⊞ [2]	[AdventureWorks2014].[Production].[Product].Name
⊟ Union1003	[AdventureWorks2014].[Person].[Person].BusinessEntityID, [A
⊞ [1]	[AdventureWorks2014].[Person].[Person].BusinessEntityID
⊞ [2]	[AdventureWorks2014].[Production].[Product].ProductID
Description	Append multiple input tables to form the output table.
Estimated CPU Cost	0.0020476
Estimated Execution Mode	Row
Estimated I/O Cost	0
Estimated Number of Executions	1
Estimated Number of Rows	20476
Estimated Operator Cost	0.0020474 (2%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	65 B
Estimated Subtree Cost	0.111876
Logical Operation	Concatenation
Node ID	0
Number of Executions	1
⊟ Output List	Union1002, Union1003
⊟ [1]	Union1002
Column	Union1002
⊟ [2]	Union1003
Column	Union1003
Parallel	False
Physical Operation	Concatenation

Figure 4-25: Properties of the Concatenation operator.

The **Defined Values** have been expanded out so that you can see the combined output, defined as Union1002, consists of the `LastName` and `Name` columns from the respective tables.

Summary

This chapter represents a major step in learning how to read graphical execution plans. However, as we discussed at the beginning of the chapter, we only focused on join operators and we only looked at simple queries.

So, if you decide to analyze a 2000-line query and get a graphical execution plan that is just about as long, don't expect to be able to analyze it immediately. Learning how to read and analyze execution plans takes time and effort. However, having gained some experience, you will find that it becomes easier and easier to read and analyze, even for the most complex of execution plans. You already have enough knowledge to get started. Just remember to follow the key points to look for in a plan. They will act as guide-posts as you step through the operations of the plan.

Chapter 5: Sorting and Aggregating Data

In this chapter, we explore the execution plans for queries that sort, aggregate, and manipulate data. In some cases, we'll see that the plans can quickly get radically more complicated, but the mechanisms for reading and understanding these plans really don't change.

Specifically, we will cover:

- **Sorting data** – queries with `ORDER BY` and the operators the optimizer can use to perform the data ordering.
- **Aggregating data** – queries that use `GROUP BY`, or that perform aggregations, covering:
  - **Standard aggregation functions**, such as `SUM`, `COUNT`, and so on
  - **Filtering aggregations** using `HAVING`
  - **Window functions** – how the optimizer executes these queries.

Queries with ORDER BY

When retrieving data from a table, there is no defined order in which that data will be returned. If we want to guarantee the order in which the data is returned, we need to use the `ORDER BY` clause to establish that order. If the optimizer can retrieve the data from an index in which the data is already in the required order, and all the operators within the plan preserve that order, then no additional operations are necessary. If not, a **Sort** operator will be necessary in the plan. As we discussed in Chapter 2, **Sort** is a blocking operator; it must gather all the rows that it needs before passing on the first row to the calling operator.

We'll cover the following varieties of sort operation:

- **Sort**
- **Top N Sort**
- **Distinct Sort**

We'll also see what can cause **Sort warnings** to appear in the plan, and what this means.

Sort operations

Let's start with a very simple `SELECT` statement, returning data from the `ProductInventory` table, ordered according to shelf location.

```
SELECT   pi.Shelf
FROM     Production.ProductInventory AS pi
ORDER BY pi.Shelf;
```

Listing 5-1

Figure 5-1 shows the execution plan.

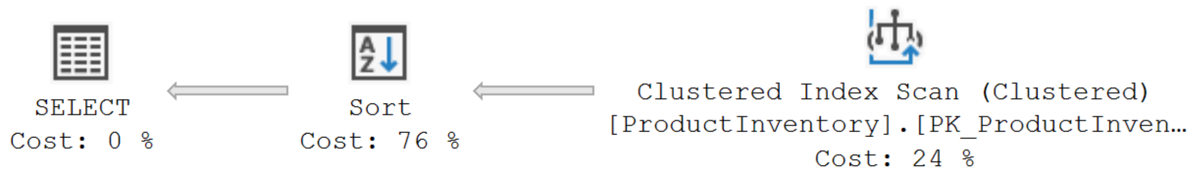


Figure 5-1: Execution plan showing a Sort operator.

Following the data flow from right to left, we see a **Clustered Index Scan** on the `Production.ProductInventory` table. The optimizer had no choice but to scan all the rows, since our query provided no `WHERE` clause filtering. The **Clustered Index Scan** passes 1069 rows to the **Sort** operator; we can see this by hovering over the arrow leading to the **Sort** operator, to bring up the tooltip window, or by looking at the **Actual Number of Rows** in the Properties pane for the scan.

The **Clustered Index Scan** passes on the rows in the order they are read from the index, in this case *probably* ordered by `ProductID`. Any order is not guaranteed, and we know this because the **Ordered** property is set to **False**, which means that the optimizer does not need the rows returned from the index to be in any order (more on the **Ordered** property shortly).

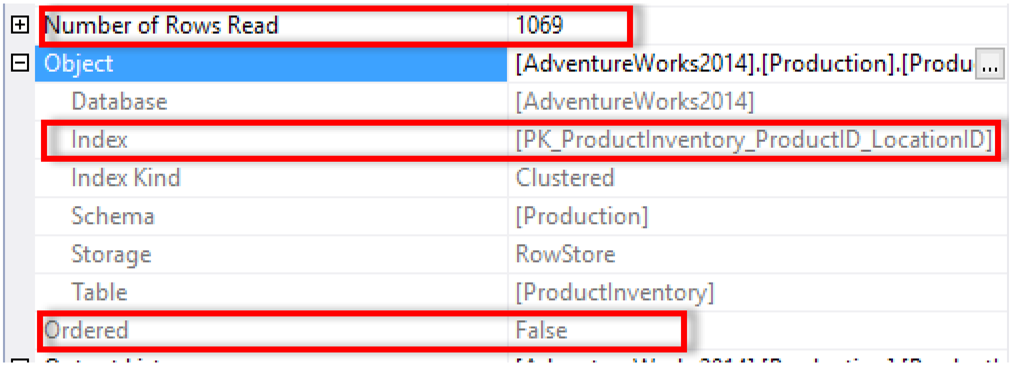


Figure 5-2: Properties of the Clustered Index Scan showing an unordered scan.

Since there is no index on the `Shelf` column, the optimizer must use a **Sort** operator within the query execution to achieve the required ordering. Once the **Sort** has all 1069 rows, it orders the data by `Shelf` and the rows pass back to the calling `SELECT`, and back to the client.

If an `ORDER BY` clause does not specify order, the default order is ascending, as you will see from the properties for the **Sort** icon in Figure 5-3.

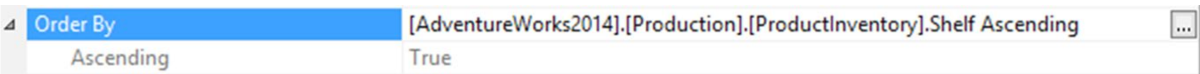


Figure 5-3: Order By property within the Sort operator.

Sort operations and the Ordered property of Index Scans

The execution engine can use the following retrieval methods to fulfill an Index Scan (clustered and nonclustered):

- **Ordered** – simply follow the index structure to the first leaf page, and then the page pointers until the end of the index, or until all the required data is collected. Data is returned in logical index order, but if data must come from disk then the access pattern is random.
- **IAM** – this is like a **Table Scan** and uses index allocation map pages to find pages allocated to index. Data is returned in "semi-random" order, but disk access is sequential, as long as the data page is not fragmented at the operating system level.

If the optimizer sets **Ordered** to **False**, it means that it doesn't care about order. In that case, at runtime the engine can choose either retrieval method, if it can guarantee to return the correct results (not always possible for IAM).

The optimizer sets **Ordered** to **True** if it needs the data to be in order. In that case the engine will always use the ordered retrieval method. For example, if instead of `ORDER BY Shelf`, this query used `ORDER BY ProductID`, then the query optimizer sets the **Ordered** property to **True**. Now that the data, as retrieved through the index, is already in the correct logical order, there is no need for a **Sort** operator in the execution plan.

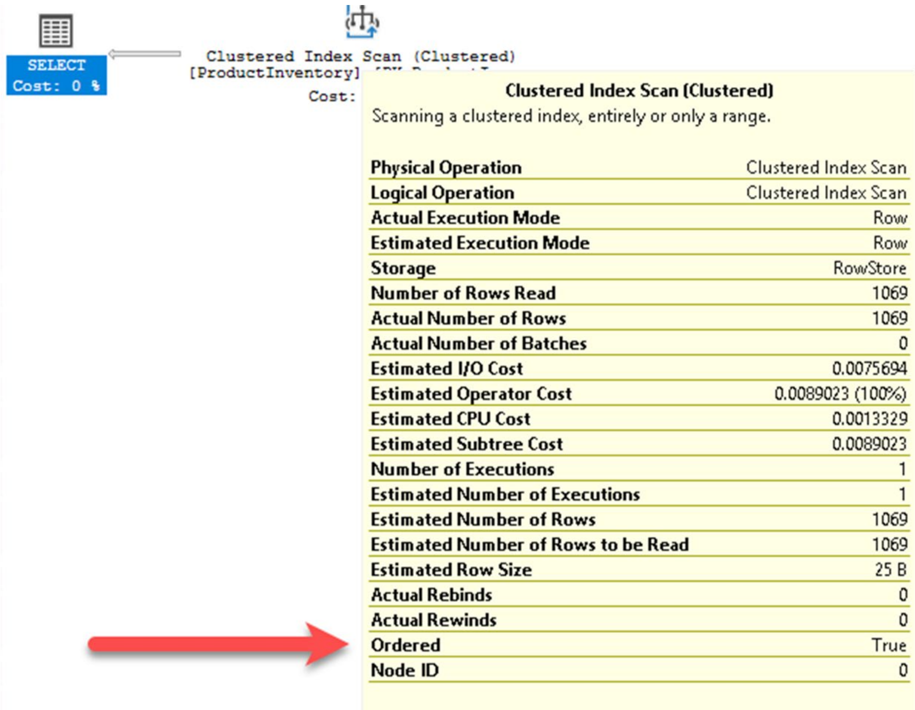


Figure 5-4: A Clustered Index Scan showing an Ordered scan in the tooltip.

Dealing with expensive Sorts

In Figure 5-1, the **Sort** operation is estimated to account for 76% of the cost of the query. This is no reason to panic. There are only two operators in this entire plan, and so 76% is quite reasonable as a percentage of all the work being done. If there were 5 or 10 operators and one of them was 76% of the estimated cost, then that would be much more concerning over all.

Nevertheless, if sorting takes a significant portion of a query's total estimated cost *and* the query is running slowly, or otherwise causing issues, then you may need to review it carefully and see if you can optimize it.

A **Sort** operation, like any other expensive operation, may not be problematic in and of itself. The first thing you need to do is establish why the operation is there; it may be there simply to fulfill an `ORDER BY` clause, but there are other reasons. You may also see the **Sort** operator added by the optimizer when the data must be ordered for a **Merge Join** operation, just as an example. In more complex plans, the purpose of a **Sort** may not be immediately obvious, since it could be necessary for other parts of the execution plan. Once you understand why the sort is there, then the next question to ask is, "Is the **Sort** really necessary?"

You may find cases where an `ORDER BY` clause has been added to a query when it wasn't needed. Developers often use an `ORDER BY` when developing and debugging a query because it's easier to verify results that way, and then to forget to take it out, even though it's not needed in the final production code.

Beyond that, SQL Server often performs the **Sort** operation within the query execution due to the lack of an appropriate index. With the appropriate index, in this case an index ordered by `Shelf`, the data may come presorted. It is not always possible, or desirable, to create a new index, but if it is, you might save sorting overhead. If it were decided that the rows did not have to be returned ordered by `Shelf`, then we might be in an easier situation.

If the data must be ordered by `Shelf`, and we're not able to create an index, then the alternatives are limited, unless we're allowed to alter the logic of the query. Notably, for example, this query has no `WHERE` clause. Is the query returning more rows than are strictly necessary? Even if a `WHERE` clause exists, you need to ensure that it limits the number of rows to only the required number of rows to be sorted, not rows that will never be used. Regardless, the **Sort** operation will still be expensive, just because sorting is not a cheap operation.

If an execution plan has multiple **Sort** operators, review the query to see if they are all necessary, or if you can rewrite the code so that fewer sorts will accomplish the goal of the query. Obviously, this is not always possible or even desirable. However, because the **Sort** operator is so expensive, it's worth ensuring that you need to order the data.

Top N Sort

A different kind of **Sort** operation can be performed when the number of rows to be returned are limited. Consider the query in Listing 5-2.

```sql
SELECT TOP (50)
    p.LastName,
    p.FirstName
FROM Person.Person AS p
ORDER BY p.FirstName DESC;
```

Listing 5-2

This query selects the last and first names of the 50 people that come last in the alphabet, when sorted by first name. Figure 5-5 shows how the optimizer resolves this query.

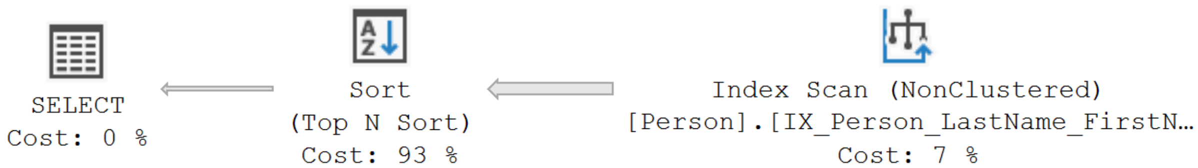


Figure 5-5: An execution plan displaying a Top N Sort operator.

There is no index that can satisfy the `ORDER BY` clause in the query. However, there is an index other than the clustered index on the table that holds the `FirstName` and `LastName` columns, `IX_Person_LastName_FirstName_MiddleName`. This index will only hold the key columns defined plus the clustered key column, so it will be a smaller index than the clustered index. Therefore, scanning it will be cheaper, which is why it was chosen by the optimizer. All 19,972 rows will be scanned and fed into the **Sort** operator.

The **Sort** operator in this case is a unique type, **Top N Sort**. Like the regular **Sort** operator, this is a blocking operator. It will retrieve all 19,972 rows and then sort the data, and then return the first 50 rows. This is defined right within the properties.

Figure 5-6: Properties of Top N Sort operator.

You can also see it in the data pipe leading away from the **Sort** operator in the execution plan shown in Figure 5-5. Below 100 rows, a sort mechanism that uses CPU more than memory is in play, to help with memory management. Above 100 rows, more memory intensive mechanisms are used, because the CPU cost would be far too high.

Distinct Sort

Sometimes, the optimizer may choose to use a **Sort** operation to satisfy a query that does not specify an `ORDER BY` clause. The intent of Listing 5-3 is to return a list of the unique combinations of the parts of a name, `LastName`, `FirstName`, `MiddleName`, `Suffix`.

```
SELECT DISTINCT
    p.LastName,
    p.FirstName,
    p.MiddleName,
    p.Suffix
FROM Person.Person AS p;
```

Listing 5-3

Figure 5-7 shows the resulting execution plan, a scan of the clustered index followed by a Sort operation.

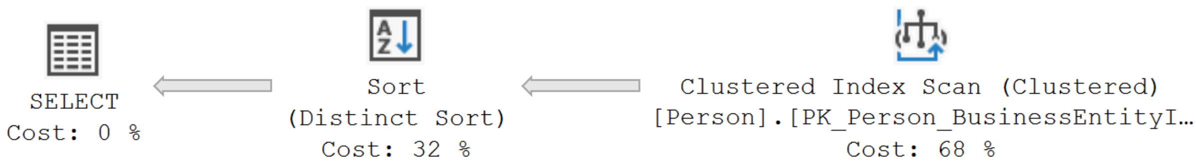


Figure 5-7: An execution plan with a Distinct Sort operator.

This time, we see a **Distinct Sort**. The optimizer is using the **Sort** operation, not only to order the data, but also to eliminate duplicates. You can see what's happening by expanding the Properties of the **Sort** operator to look at the **Order By** property, shown in Figure 5-8.

⊟ Order By	[AdventureWorks2014].[Person].[Person].LastName Ascending, [Adv
⊞ [1]	[AdventureWorks2014].[Person].[Person].LastName Ascending
⊞ [2]	[AdventureWorks2014].[Person].[Person].FirstName Ascending
⊞ [3]	[AdventureWorks2014].[Person].[Person].MiddleName Ascending
⊞ [4]	[AdventureWorks2014].[Person].[Person].Suffix Ascending

Figure 5-8: Properties of Sort (Distinct Sort) operator demonstrating sorting on all columns.

By sorting on all columns in the `SELECT` list, duplicate rows are immediately adjacent, and so can easily be skipped when the **Sort** operator returns the sorted data.

Sort warnings

The **Sort** operator is very dependent on the row estimates provided to the optimizer because it needs memory to perform the sort. When an inadequate amount of memory is allocated for a sort, data gets stored in **tempdb** through a process referred to as a *spill*. This is so problematic for performance that, in SQL Server 2012 and later, you get a warning in the execution plan itself (or in an Extended Event, starting in SQL Server 2008).

Listing 5-4 shows an apparently simple query that returns the data in descending order of the `ModifiedDate`.

```
SELECT sod.CarrierTrackingNumber,
       sod.LineTotal
FROM Sales.SalesOrderDetail AS sod
WHERE sod.UnitPrice = sod.LineTotal
ORDER BY sod.ModifiedDate DESC;
```

Listing 5-4

Figure 5-9 shows the actual execution plan that this query generates.

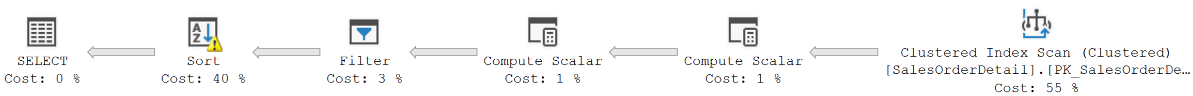


Figure 5-9: An execution plan that has generated a Sort warning.

There are several things worth exploring in this execution plan, but the one that should immediately pop out is the warning symbol on the **Sort** operator below.

Figure 5-10: The Sort warning, blown up for easier viewing.

If you hover over the operator, the tooltip will show a message about the warning, but the details are in the properties, so we'll go there first. The full message of the warning is shown in Figure 5-11.

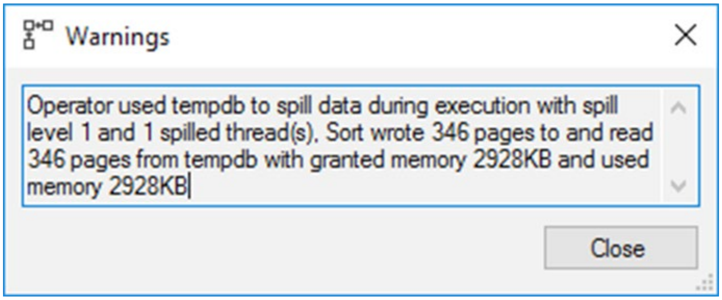


Figure 5-11: Full description of the Sort warning.

The warning lays out specifically what happened. An additional 346 pages were used in **tempdb** despite memory being allocated for 2,928 KB. Why did this happen? That information is also available in the properties. Figure 5-12 has the full property sheet with a few facts highlighted.

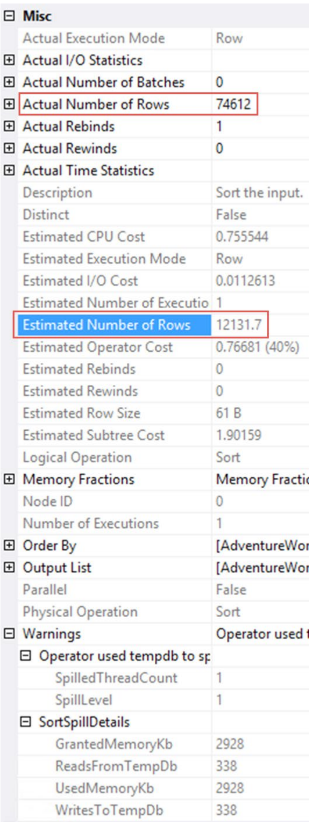


Figure 5-12: Difference between the Estimated and Actual Rows leading to a spill.

As you can see, the Estimated Number of Rows is 12,131.7. The actual number of rows was 74,612. That's nearly six times as many rows being processed as SQL Server expected. While the memory grant does include some margin of error, there was not enough memory allocated to deal with this much data. That's why the **Sort** operation was forced to spill to **tempdb**. Your investigation then has to determine where the estimates went wrong. The way to do that is to walk through the other operators in the execution plan.

The data being read from the disk is coming from the **Clustered Index Scan** of the `PK_SalesOrderDetail` index, at the far right of the plan in Figure 5-9. The Estimated Number of Rows is 121,317 and the actual number of rows is the same. This means that the initial operation went as expected.

The next two operators are **Compute Scalar**. The first has a pair of calculations shown in Figure 5-13.

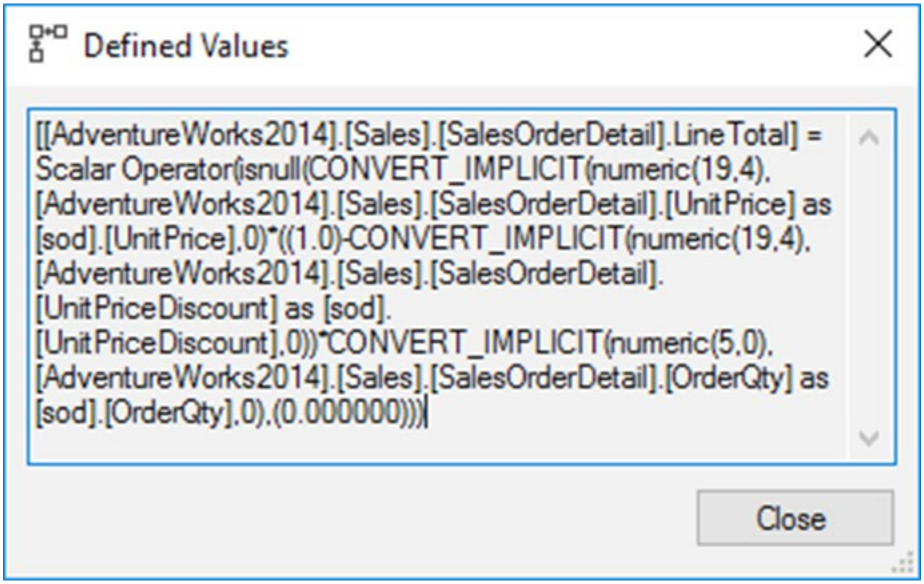


Figure 5-13: Details of the first Compute Scalar operator.

These two calculations are benign and directly related to the data we're working with in the query. The next **Compute Scalar** operator is simply aliasing the calculations from the preceding operator. `LineTotal` is a computed column in the table definition, and this is how you can see that within the execution plan.

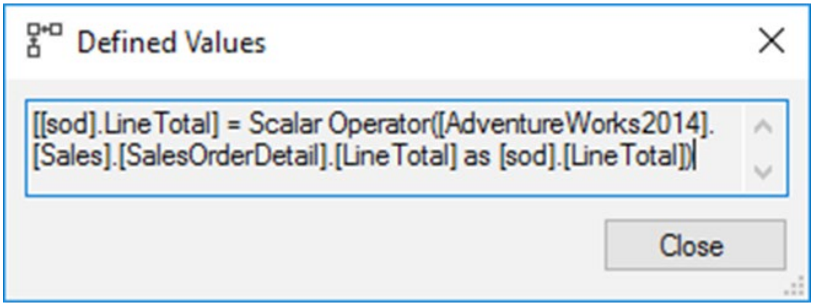


Figure 5-14: Calculation made by the second Compute Scalar operator.

None of these processes will affect the row estimates. The next operation is the **Filter** operator (covered in more detail later in the chapter). A Filter operator inspects the data in each row it receives with the goal of eliminating rows that are not required; only rows that meet the Predicate criteria are passed on to the calling operator.

Normally, this type of operation is done at the table or index level, through seeks and scans. However, because we're dealing with calculated values, the `LineTotal`, those calculations must be performed before the data set can be filtered. We can see the Predicate calculation in the properties of the operator. All the brackets and fully-qualified object names may make reading a little difficult. The core calculation is `sod.UnitPrice = sod.LineTotal`.

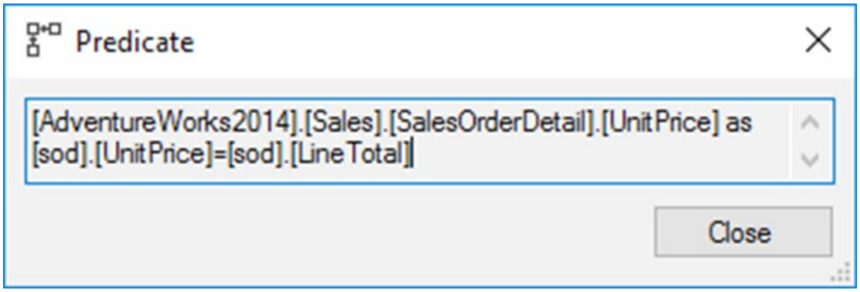


Figure 5-15: Details of the Predicate property of the Filter operator.

However, this calculation is itself not the issue. Instead, we need to look to the Estimated Number of Rows processed by the **Filter** operator, 12,131.7. In other words, of the 121,317 rows that were read from the clustered index, the optimizer assumed only 10% would match the Predicate condition. This is a fixed estimate, which the optimizer uses because it can't know for certain how many values will match, when comparing to a calculated value.

In fact, 74,612 were returned, and this is the cause of the inappropriate memory estimates for the **Sort** operator and the subsequent spill to **tempdb**.

Aggregating Data

One of the most common uses for data, after it has been collected and cleaned, is to apply some math to it to get the number of records (**COUNT**), the mean value of a column (**AVG**), the maximum value (**MAX**), and others. These calculations require that we combine the data in a process known as "aggregation."

Aggregation is a powerful feature within T-SQL that enables us, in many instances, to perform these types of calculations in a much more efficient manner because we can aggregate the data as we retrieve it. In short, if we get aggregation operations early in a plan, we're frequently working with less data in the rest of the plan, making that plan more efficient. We're also saving huge amounts of network traffic, if the alternative is to aggregate on the client.

This section will explore the mechanisms through which SQL Server aggregates information, based on the data, your data structures, and the T-SQL code you have written.

Stream Aggregate

The first aggregation operator we'll look at is the **Stream Aggregate**. This operator uses data that is sorted to build a set of aggregate values. We'll use the simple query in Listing 5-5 to create an aggregate count of the number of `TerritoryID` values within the `Sales.Customer` table.

```
SELECT c.TerritoryID,
       COUNT(*)
FROM Sales.Customer AS c
GROUP BY c.TerritoryID;
```

Listing 5-5

If we run this query and capture the execution plan, we'll see the **Stream Aggregate** operator in use.

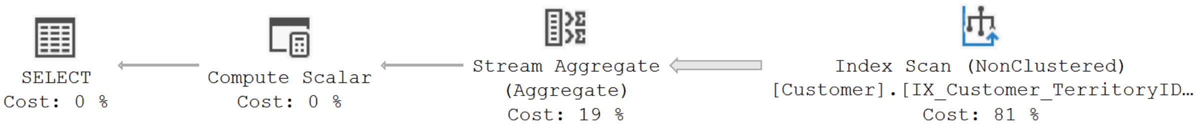


Figure 5-16: Execution plan with a Stream Aggregate operator.

Reading this plan in the order of data flow, we see that it uses the **IX_Customer_TerritoryID** nonclustered index to scan the data. This data flows into the **Stream Aggregate** operator, which aggregates the data, and then on to a **Compute Scalar** operator before returning as a result set.

The first requirement for the use of the **Stream Aggregate** operator is that the data be sorted by the columns being aggregated. If we check the properties of the **Index Scan** operator, we'll see that the **Ordered** property is set to **True**, meaning that the data will be accessed in the logical order in which it's stored in the index (by **TerritoryID**), and so no additional **Sort** operator is required. This helps explain why the optimizer has chosen to use this nonclustered index to retrieve the data.

Number of Rows Read	19820
Object	[AdventureW
Ordered	True
Output List	[AdventureW
Parallel	False
Physical Operation	Index Scan
Scan Direction	FORWARD
Storage	RowStore
TableCardinality	19820

Figure 5-17: Properties of the Index Scan showing an Ordered operation.

We can look the properties of the **Stream Aggregate** operator to see how the data is being processed. Figure 5-18 shows the properties for the `GROUP BY` clause of our query.

⊟ Group By	[AdventureWorks2014].[
Alias	[c]
Column	TerritoryID
Database	[AdventureWorks2014]
Schema	[Sales]
Table	[Customer]

Figure 5-18: Group By properties of the Stream Aggregate operator.

The **Defined Values** property discloses the calculations we're requesting from this aggregation.

⊟ Defined Values	[Expr1004] = Scalar Operato
⊟ Expr1004	Scalar Operator(Count(*))
⊟ Aggregate	
AggType	countstar
Distinct	False
ScalarString	Count(*)

Figure 5-19: Output of the aggregated values shown as Defined Values.

The aggregations occur within the **Stream Aggregate** operator, as it reads the ordered data. The **AggType** of **countstar** indicated that in this case it's performing an aggregate count for each `TerritoryID` value.

Why, then, is there a **Compute Scalar** operator within this plan? Figure 5-20 shows its properties.

⊟ Defined Values	[Expr1001] = Scalar Operator(CONVERT
⊟ Expr1001	Scalar Operator(CONVERT_IMPLICIT(ir
⊟ Convert	
DataType	int
Implicit	True
⊞ ScalarOperator	Scalar Operator
Style1	0
ScalarString	CONVERT_IMPLICIT(int,[Expr1004],0)

Figure 5-20: Compute Scalar operator showing data conversion.

The output data type of the **countstar** aggregation from the operator properties shown in Figure 5-19 is BIGINT. The optimizer added a **Compute Scalar** operator to perform an implicit conversion of that data to a type of INT, before returning it within the result set of the query because this query is asking for a COUNT (which outputs as INT). This is changing the data to an INT. If we used COUNT_BIG in the query, the **Compute Scalar** would be removed.

The **Stream Aggregate** operator is generally straightforward. It calculates the information as it retrieves it, in a stream, because the data is ordered. This can make for a very efficient operation. However, the requirement that the data be ordered implies that, depending on the data structures involved, a **Sort** operation may be a part of the plan. This could possibly lead to poor performance of the **Stream Aggregate**, suggesting the need for a new or different index to better support retrieving the data in an ordered fashion.

Hash Match (Aggregate)

Let's consider another simple aggregate query against a single table, where we want to know the average discount offered, for each unit price.

```
SELECT sod.UnitPrice,
       AVG(sod.UnitPriceDiscount)
FROM Sales.SalesOrderDetail AS sod
GROUP BY sod.UnitPrice;
```

Listing 5-6

Figure 5-21 shows the actual execution plan.

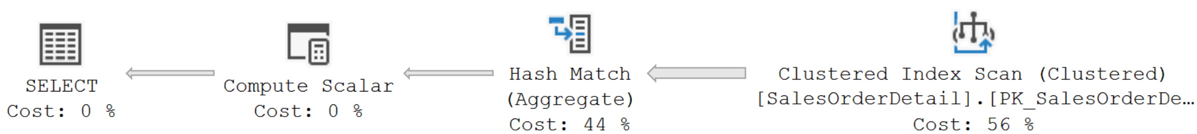


Figure 5-21: Execution plan generated with a Hash Match aggregation operator.

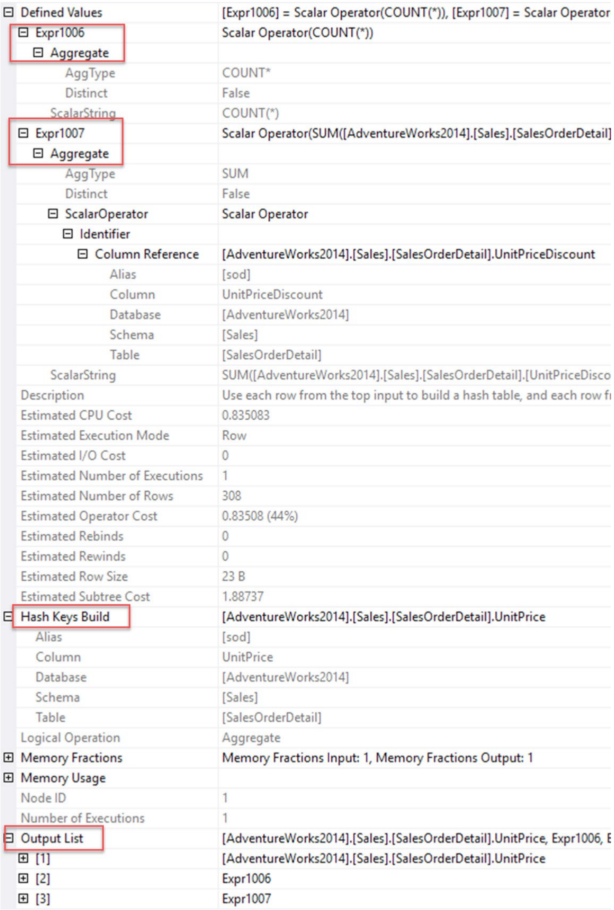
The data flow of the query execution begins with a **Clustered Index Scan**, because all rows are returned by the query; there is no `WHERE` clause to filter the rows. Next, the optimizer aggregates these rows, to start the process of the requested `AVG` aggregate calculation. To count the number of rows for each **UnitPrice**, the optimizer chooses to perform a **Hash Match (Aggregate)** operator.

In Chapter 4, we looked at the **Hash Match(Join)** operator for joins. This same **Hash Match** operator can also occur when we perform aggregations within a query, or because the optimizer decides to use aggregation for some other reason. As with a **Hash Match** with a join, a **Hash Match** with an aggregate causes SQL Server to create a temporary hash table in memory in which it stores the results of all aggregate computations; it can count rows, track minimum and maximum values, calculate a sum, and so on.

In this example, for each value in the `GROUP BY` column, which is **UnitPrice**, it stores a row with that **UnitPrice**, a tally of rows and a total discount. As it builds the hash table, it increases the tally and total discount whenever it processes a row with the same `UnitPrice`.

As a general rule, the memory used by a **Hash Match(Aggregate)** will usually be less than that used by a **Hash Match(Join)**, because the join operator must create a hash table for all the data, while for the aggregate operator, the hash table contains only the aggregation key and the computation results. Certainly, one can envision exceptions; for example, if we have a very small table consisting of two columns, but a query with a very large number of aggregate calculations, but generally the rule will hold true.

We can see how the aggregations are performed by looking at the properties of the **Hash Match(Aggregate)**, shown in Figure 5-22.



⊟ Defined Values	[Expr1006] = Scalar Operator(COUNT(*)), [Expr1007] = Scalar Operator
⊟ Expr1006	Scalar Operator(COUNT(*))
⊟ Aggregate	
AggType	COUNT*
Distinct	False
ScalarString	COUNT(*)
⊟ Expr1007	Scalar Operator(SUM([AdventureWorks2014].[Sales].[SalesOrderDetail]
⊟ Aggregate	
AggType	SUM
Distinct	False
⊟ ScalarOperator	Scalar Operator
⊟ Identifier	
⊟ Column Reference	[AdventureWorks2014].[Sales].[SalesOrderDetail].UnitPriceDiscount
Alias	[sod]
Column	UnitPriceDiscount
Database	[AdventureWorks2014]
Schema	[Sales]
Table	[SalesOrderDetail]
ScalarString	SUM([AdventureWorks2014].[Sales].[SalesOrderDetail].[UnitPriceDisco
Description	Use each row from the top input to build a hash table, and each row fi
Estimated CPU Cost	0.835083
Estimated Execution Mode	Row
Estimated I/O Cost	0
Estimated Number of Executions	1
Estimated Number of Rows	308
Estimated Operator Cost	0.83508 (44%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	23 B
Estimated Subtree Cost	1.88737
⊟ Hash Keys Build	[AdventureWorks2014].[Sales].[SalesOrderDetail].UnitPrice
Alias	[sod]
Column	UnitPrice
Database	[AdventureWorks2014]
Schema	[Sales]
Table	[SalesOrderDetail]
Logical Operation	Aggregate
⊞ Memory Fractions	Memory Fractions Input: 1, Memory Fractions Output: 1
⊞ Memory Usage	
Node ID	1
Number of Executions	1
⊟ Output List	[AdventureWorks2014].[Sales].[SalesOrderDetail].UnitPrice, Expr1006, E
⊞ [1]	[AdventureWorks2014].[Sales].[SalesOrderDetail].UnitPrice
⊞ [2]	Expr1006
⊞ [3]	Expr1007

Figure 5-22: Properties of the Hash Match aggregate operator detailing the function of the operator.

Highlighted at the top you can see there are two aggregates, and neither is the average. The first is a `COUNT *` calculation being executed to get a row count for each `UnitPrice`, returned as `Expr1006`. The second aggregation is a `SUM` of the `UnitPriceDiscount` column for each `UnitPrice`, returned as `Expr1007`. Further down you can see how the hash table is being created on the `UnitPrice` column.

As you can see from the **Output List**, the `UnitPrice`, `Expr1006` and `Expr1007` are passed on to a **Compute Scalar** operator, which performs the calculation below for each `UnitPrice` value.

```
 [Expr1001] = Scalar Operator(CASE WHEN [Expr1006]=(0) THEN NULL
ELSE [Expr1007]/CONVERT_IMPLICIT(money,[Expr1006],0) END)
```

If a given `UnitPrice` value, as expressed by `Expr1006`, has no rows, then this will return `NULL` for that `UnitPrice`. If there are rows for that `UnitPrice`, the average `UnitPriceDiscount` is calculated by dividing `Expr1007` by `Expr1006`, first having converted `Expr1006` to a `MONEY` data type, using the `CONVERT_IMPLICIT` command.

Quite often, aggregations within queries can be comparatively expensive operations, depending on the number of rows that need to be aggregated. However, it is almost always far more efficient to aggregate on the server, and push a limited number of rows over to the client, than to push all data and aggregate on the client. Also, in cases where the aggregated data is used in the rest of a larger query, or stored in a temporary table and then joined to other data, the savings get even bigger because all subsequent operators work on far fewer rows.

One tactic when attempting to tune an aggregation is to add a covering index, or to remove unneeded columns so that an existing index becomes covering, sorted on the `GROUP BY` columns. This will allow the optimizer to use the **Stream Aggregate** instead of **Hash Match Aggregate**.

You can also pre-aggregate data by using an indexed view, although that tactic incurs the overhead of maintaining the data in the view, as well as the table, when data is modified.

Filtering aggregations using HAVING

The optimizer uses the **Filter** operator to limit the output to the rows that meet the specified criteria. In Listing 5-7 we add a `HAVING` clause, to limit the result set to only those rows where the average unit price discount is greater than 0.2.

```
SELECT sod.UnitPrice,
       AVG(sod.UnitPriceDiscount)
FROM Sales.SalesOrderDetail AS sod
GROUP BY sod.UnitPrice
HAVING AVG(sod.UnitPriceDiscount) > .2
```

Listing 5-7

Figure 5-23 shows the execution plan, which now contains a **Filter** operator after the **Compute Scalar**.

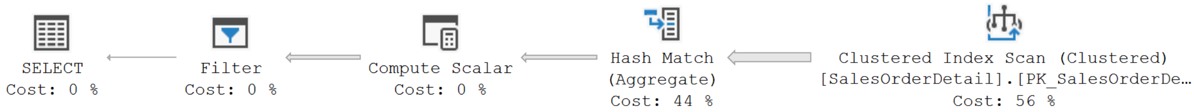


Figure 5-23: Execution plan uses a Filter operator to satisfy the HAVING clause.

The **Filter** operator limits the output to those values of the column, `UnitPriceDiscount`, that have an average value greater than .2, to satisfy the `HAVING` clause. This is accomplished by applying a Predicate against the output of the **Compute Scalar** operator, as we can see from the properties of the **Filter** operator.

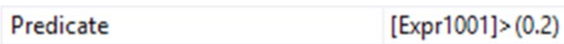


Figure 5-24: Properties of the Filter operating showing the filtering calculation.

In this case, the nature of the `HAVING` clause meant that the optimizer had no way to verify the Predicate without first doing the aggregation. The **Hash Match (Aggregate)** receives 121317 rows and passes on 287 (hover over the data flow arrows to see this), which is the number processed by the **Filter** operator.

However, if there is a way to filter before aggregation, the optimizer will usually find it. To offer a trivial example, if we were to change the `HAVING` clause to `sod.UnitPrice > 800`, the optimizer is sensible enough to, essentially, rewrite `HAVING` to `WHERE`, in which case the filtering is pushed down into the **Clustered Index Scan**, as you'll see by running the modified query and examining the **Predicate** property of this operator (rewriting the query to use `WHERE` rather than `HAVING` will have the same effect).

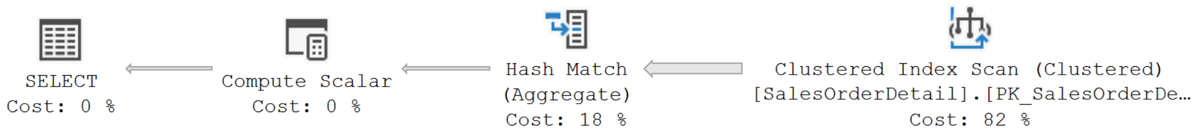


Figure 5-25: An execution plan that shows filtering occurring during aggregation.

When filtering on aggregated rows, the optimizer has no choice but to add a **Filter** operator to the plan, after the aggregation is complete. Notionally, this adds a minimal extra cost to the plan. However, this is more than compensated for by the need to return fewer rows to the client. Also, when the aggregated and filtered data is used elsewhere in a larger query, the savings are even greater. If the optimizer can find a way to apply the filtering earlier, it will do it.

Plans with aggregations and spools

A **Spool** operator uses a temporary worktable to store data that may need to be reused multiple times within an execution plan. This section will review a couple of examples where spools are used to store the results of aggregation calculations for plans that use **Nested Loops** joins. However, spools can appear in many other situations where, by storing the results in a worktable, the optimizer can reuse that data many times, instead of having to execute sets of operators multiple times.

There are several types of spool, represented by the following physical operators**: Index Spool**, **Rowcount Spool**, **Table Spool** and **Window Spool**. Here, we'll only consider the **Table Spool** and the **Index Spool**, as they appear in the context of queries that contain aggregations. SQL Server will always have a clustered index for storing the data for any spool; an **Index Spool** will have an additional nonclustered index to make it easier to retrieve the data.

There are two logical types of **Spool** operator, **Lazy Spool** and **Eager Spool**. A **Lazy Spool** is a streaming operator. It requests a row from its child operator, stores it, and then passes it to its parent, passing control back to that parent. An Eager Spool, on the other hand, is a blocking operator, that will call its child node until it has all the rows, and only then return the first row from its worktable. Generally the optimizer will avoid the Eager Spool, but it is ideal for certain situations such as Halloween protection (covered in Chapter 6).

Table Spool

Let's start with an aggregation example that uses a **Table Spool**. The query in Listing 5-8 uses a subquery to calculate the total tax amount paid by customers, according to sales region (TerritoryID).

```
SELECT sp.BusinessEntityID,
       sp.TerritoryID,
       (   SELECT SUM(TaxAmt)
           FROM Sales.SalesOrderHeader AS soh
           WHERE soh.TerritoryID = sp.TerritoryID)
FROM Sales.SalesPerson AS sp
WHERE sp.TerritoryID IS NOT NULL
ORDER BY sp.TerritoryID;
```

Listing 5-8

Figure 5-26 shows the execution plan.

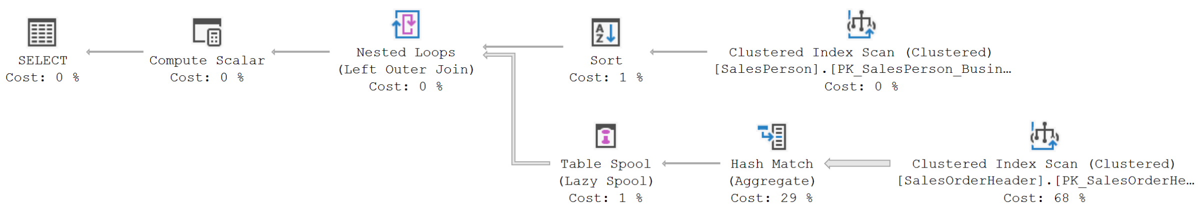


Figure 5-26: An execution plan using a Table Spool with aggregation.

The outer input of the **Nested Loops** join operator is a scan of the clustered index on the SalesPerson table, which returns 14 rows (sorted by TerritoryID). This means that the inner input, a **Table Spool**, will execute 14 times.

The first execution of the inner input is always a **Rebind**, so the **Table Spool** calls for a row from the **Hash Match**, which in turn calls for a row from the **Clustered Index Scan** on SalesOrderHeader. The **Hash Match** operator uses a temporary hash table to calculate the total tax amount collected for each distinct TerritoryID value in SalesOrder-Header. There are 10 distinct values of TerritoryID and, at some point, it will start returning each of these 10 rows to the **Table Spool**, which stores these in its worktable while passing them on (it's a **Lazy Spool**), until it has passed on all 10 rows.

If we examine the properties of the **Nested Loops** operator we see that it satisfies the join condition using a **Predicate** (see the *Nested Loops operator* section of Chapter 4 for a discussion of this topic). Essentially, the inner input is static, and will produce the same result for every value in the outer input.

For each of the other 13 rows returned from `SalesPerson` to the **Nested Loops** operator, the outer input has to rewind. This is where the **Table Spool** comes into play. Instead of calling the **Hash Match** again, 13 times, the worktable defined by the **Table Spool** is used.

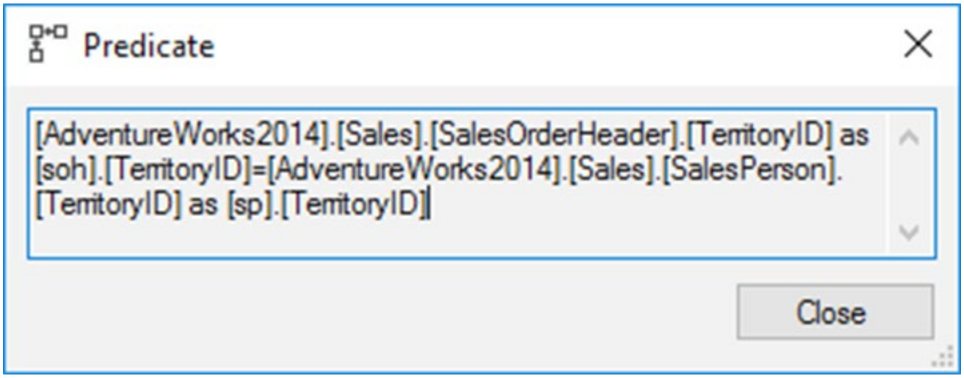


Figure 5-27: Properties showing how the Table Spool was used to filter data.

If you inspect the properties of the **Table Spool** you'll see 13 **Rewinds** and 1 **Rebind**. The **Hash Match** and **Clustered Index Scan** are only executed once each, for the initial **Rebind**, to load the data into the **Table Spool**.

This is a simple example of how the optimizer can use a **Table Spool** to make aggregation queries more efficient, where a single **Table Spool** reused its own information. However, very often, you'll encounter cases where a spool shares its information with other **Spool** operators in the same plan. If you check the properties of the **Table Spool**, you'll see that it has a **Node ID** value of 4. If a second spool were to reuse data from this first spool, then in the properties for the second spool you'd see both its own **Node ID** value, and a **Primary Node ID** value, which in this case would be 4. We'll see an example of this in Chapter 6.

Index Spool

To see an **Index Spool** operator, we just need to add a useful index that the optimizer can use to find the rows with matching `TerritoryID` values, in the `SalesOrderHeader` table.

```
CREATE INDEX IX_SalesOrderHeader_TerritoryID
ON Sales.SalesOrderHeader
(
    TerritoryID
)
INCLUDE
(
    TaxAmt
);
```

Listing 5-9

Now, re-execute the query in Listing 5-8. Figure 5-28 shows the execution plan, which is similar to the previous plan, except that now, for the inner input of the **Nested Loops** join, we see an **Index Seek** against the `SalesOrderHeader` table, a streaming aggregation instead of the blocking Hash Match aggregation, and then an **Index Spool** instead of a **Table Spool**.

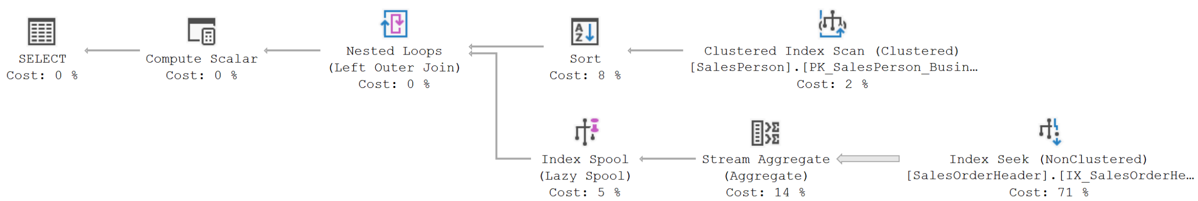


Figure 5-28: An execution plan using an Index Spool operator for aggregation.

The 14 rows returned by the scan of the `SalesPerson` table are ordered by the **Sort** operation on `TerritoryID`. Examine the properties of the **Nested Loops** operator and you'll see that it satisfies the join condition using the `TerritoryID` values as *Outer References*. That means that each of the values from the 14 rows is pushed down into the inner input, which returns only matching rows based on the **Index Seek** operation.

As before, the first execution of the inner input is a **Rebind**. The value of 1, the first `TerritoryID` row's value, is pushed down to the other operators. The **Index Spool** first initializes its child operators. The **Stream Aggregate** starts requesting rows from **Index Seek**, which uses the pushed-down value to find matching rows in the index. The spool passes the matching rows to the **Stream Aggregate**, which then returns a single row, the aggregation result for **TaxAmt**, to the **Index Spool**, which then stores it in an indexed worktable and returns it to **Nested Loops**.

The second and third rows coming into **Nested Loops** also have a TerritoryID value of 1, so the next two executions of **Index Spool** are **Rewinds**. **Index Spool** will not call **Stream Aggregate**, and instead immediately returns the previously stored results from the worktable.

For the fourth row, we have a TerritoryID value of 2, a new value. The data change forces the **Index Spool** to register a **Rebind** initializing the other operators again with the new pushed-down value. This will be the fourth execution of the **Index Spool**, but only the second execution of each of the child operators.

This pattern repeats until all 14 rows are processed. Look at the properties of the **Index Spool** and you'll see that there are 10 **Rebinds** and 4 **Rewinds**. Look at the properties of the **Stream Aggregate** or the **Index Seek** and you see only 10 executions, corresponding to the 10 distinct values for TerritoryID in the 14 rows.

Remember to drop the index created in Listing 5-9 before continuing.

```
DROP INDEX IX_SalesOrderHeader_TerritoryID ON Sales.
SalesOrderHeader;
```

Listing 5-10

Working with Window Functions

Introduced in SQL Server 2008, the OVER clause defines how to sort and partition the data, to which an aggregate function can be applied. A Window function is essentially one that operates on a window, or partition of data, as defined within the OVER clause. The ranking functions, ROW_NUMBER, RANK, DENSE_RANK and NTILE, are all Window functions. Aggregate functions, such as SUM or AVG, also support the OVER clause, but are not considered Window functions.

The query in Listing 5-10 partitions the data according to the CustomerID value, and within each partition orders the data by order date. To each partition, we apply the ROW_NUMBER ranking function, which simply numbers each row in each partition, so if a customer made 5 orders in that period, there would be 5 rows in their partition, numbered 1 to 5, with the earliest order having a RowNum of 1.

```
SELECT soh.CustomerID,
       soh.SubTotal,
       ROW_NUMBER() OVER (PARTITION BY soh.CustomerID
                          ORDER BY soh.OrderDate ASC) AS RowNum,
       soh.OrderDate
FROM Sales.SalesOrderHeader AS soh
WHERE soh.OrderDate BETWEEN '20130101'
                    AND     '20130701'
```

Listing 5-11

Figure 5-29 shows the resulting execution plan.

Figure 5-29: Execution plan to satisfy a Windowing function using a Segment and a Sequence operator.

Since there is no index that supports the `WHERE` clause, the optimizer chooses to scan the clustered index. It returns the orders that fall within the required period. These rows are then sorted by `CustomerID`, and secondarily by `OrderDate` in preparation for splitting the data into partitions.

Next, we encounter two new operators that we have not yet explored, **Segment** and **Sequence Project (Compute Scalar)**. Whenever you see an operator with which you're unfamiliar, or familiar operators whose role is not immediately clear to you, this is usually a good place to start.

A **Segment** operator splits the data into a series of partitions, or segments, based on the partition column or columns, defined within the query. In this case, we have chosen to partition the data by `CustomerID`. If we examine the **Group By** property of this operator, we see that the data is being grouped on the `CustomerID` column. We can also see that an output column is created, **Segment1002**, which marks the start of each new segment.

⊟ Group By	[AdventureWorks2014].[Sa
Alias	[soh]
Column	CustomerID
Database	[AdventureWorks2014]
Schema	[Sales]
Table	[SalesOrderHeader]
Logical Operation	Segment
Node ID	2
Number of Executions	1
⊟ Output List	[AdventureWorks2014].[Sa
⊞ [1]	[AdventureWorks2014].[Sa
⊞ [2]	[AdventureWorks2014].[Sa
⊞ [3]	[AdventureWorks2014].[Sa
⊞ [4]	Segment1002

Figure 5-30: Properties of the Segment operator showing the segmentation of the data.

All this data passes to the **Sequence Project (Compute Scalar)** operator, which is used exclusively by ranking functions, and works off an ordered set of data, with segment marks added by the **Segment** operator.

In Figure 5-31, we can see that in this case the **Sequence Project** operator simply counts the number of rows in each segment, and assigns a sequential number to them, rather like having an `IDENTITY` column assigned to each partition.

⊟ Defined Values	[Expr1001] = Scalar Operator(row_number)
⊟ Expr1001	Scalar Operator(row_number)
ScalarString	row_number
⊟ Sequence	
FunctionName	row_number

Figure 5-31: Properties of the Sequence operator showing the function of the operation.

That example is fine, but it doesn't show off the aggregations that are possible when you begin to use windowing functions. Listing 5-11 adds an additional column to the query, the average value of the `SubTotal`, across a given `CustomerID`, for the data range in question.

```
SELECT soh.CustomerID,
       soh.SubTotal,
       AVG  (soh.SubTotal) OVER (PARTITION BY soh.CustomerID) AS
AverageSubTotal,
       ROW_NUMBER() OVER (PARTITION BY soh.CustomerID ORDER BY soh.
OrderDate ASC) AS RowNum
FROM Sales.SalesOrderHeader AS soh
WHERE soh.OrderDate
BETWEEN '20130101' AND '20130701';
```

Listing 5-12

If we examine the execution plan for this query we'll see, in Figure 5-32, one that is much more complex than others have been so far in the book.

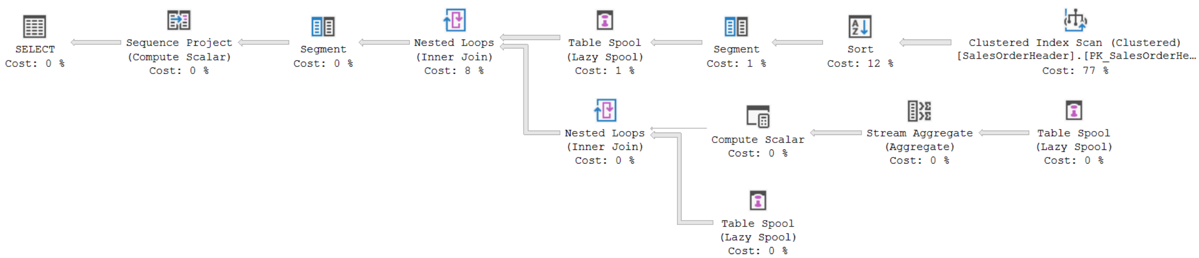


Figure 5-32: A more complex plan showing additional window functions.

That is hard to read, so we'll drill down on parts of the execution plan. Figure 5-33 shows the primary section relating to the data retrieval.

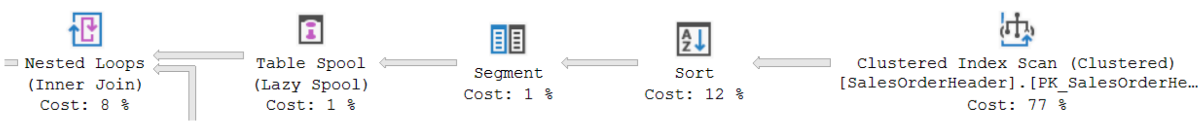


Figure 5-33: Details of the plan from Figure 5-32.

Just as before, there is no index that supports our `WHERE` clause, so we see a **Clustered Index Scan**. The data is ordered again through a **Sort** operation and then it is passed to the now familiar **Segment** operator. From there it passes to a **Table Spool (Lazy Spool)**, which is the outer input of a **Nested Loops** join operator. The inner input is another **Nested Loops** join, for which Figure 5-34 shows the outer and inner inputs.

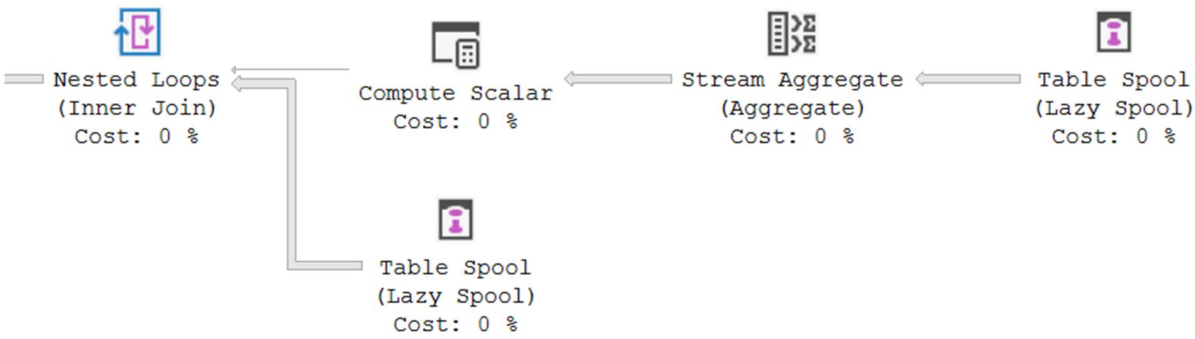


Figure 5-34: Additional details of the plan from Figure 5-32.

In Figure 5-34, you can see where we are reusing the data stored in the **Table Spool** operator. This operator deals with segmented data by slightly changing its behavior. In normal operation, a **Lazy Spool** reads a row, stores it and passes it on straight away. However, in this case, the **Table Spool** reads all rows for a segment of data, and then sends on the row for that segment to the following operations.

The data from the **Table Spool** is passed to a **Stream Aggregate** operator. The **Stream Aggregate** operation can be used because the data is ordered based on the **Sort** operator we see in Figure 5-33. If we look at the **Stream Aggregate** properties, we can then understand what it's doing within this execution plan.

⊟ Defined Values	[Expr1004] = Scalar Operato
⊟ Expr1004	Scalar Operator(Count(*))
⊟ Aggregate	
AggType	countstar
Distinct	False
ScalarString	Count(*)
⊟ Expr1005	Scalar Operator(SUM([Adve
⊟ Aggregate	
AggType	SUM
Distinct	False
⊞ ScalarOperator	Scalar Operator
ScalarString	SUM([AdventureWorks2014

Figure 5-35: Properties of the Stream Aggregate operator.

There are two new values being created, a count of the values within the aggregate of the `CustomerID` and a sum of the `SubTotal` column across that same aggregate. All of this is then passed to a **Compute Scalar** operator which performs another calculation.

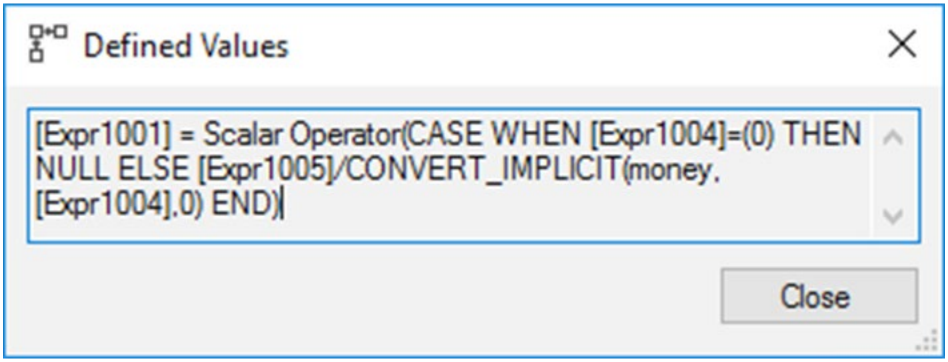


Figure 5-36: Calculation within the Compute Scalar operator.

This is creating a new value, **Expr1001**, which will either be null, or an average calculation of the values created in the **Stream Aggregate**. In short, this part of the process is satisfying the `AVG` function called for in the query in Listing 5-11. The output from the **Scalar Operator** is then run through another **Nested Loops** operator, which refers to our temporary storage in the **Table Spool**. Why?

This is where things get fun. We must aggregate our data in order to arrive at an average, so the number of rows being returned is going to change. You can see this if you look at the actual rows output from the **Stream Aggregate** operator and compare it to the number of rows output from the second instance of the **Table Spool** operator in Figure 5-34. The aggregate output is 2,464 and the temporary storage output is 2,784. The **Nested Loops** is necessary to put together the output of the aggregation operation with the information being stored temporarily in the **Table Spool**. All this is passed to the other **Nested Loops** operator (originally shown in Figure 5-33) to be combined with the output of the **Table Spool** for final processing of the query as shown in Figure 5-37.

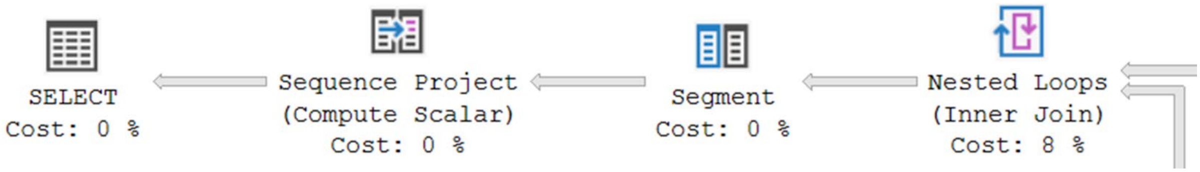


Figure 5-37: Details of the plan from Figure 5-32 showing Segment and Sequence operators.

This final section of the execution plan is where we see the functions necessary to support the `ROW_NUMBER()` function, from the original query in Listing 5-10. There is no final **Sort** operation because I dropped the `ORDER BY` clause in the query in Listing 5-11, just to simplify things a little bit.

Through all this now, you can see how the Window functions can be used for aggregations, how these functions and methods are satisfied within the execution plan, and how you read through an execution plan to understand what functions are being performed where. Reading through the plan is possible because you can see the creation of values such as **Expr1004** and **Expr1005** within the **Stream Aggregate** to be followed by their use to create an average represented by **Expr1001** created in the **Compute Scalar** operator. You can also see how each of the **Table Spool** operators is used to move the data through the necessary processing to arrive at the requested output.

Summary

This chapter focused primarily on the ordering and aggregation of data. You've seen several examples of execution plans that showed how to follow properties and values as they move between operators within an execution plan. This is one of the fundamentals to reading your own execution plan and you'll see it again and again throughout the rest of the book. During all this discussion we brought up the cost of certain operations. Just remember that no operation is inherently problematic. Each just represents the optimizer's best attempts at resolving the query in question. Don't focus on eliminating or changing any given operator; focus instead on the query in question.

Chapter 6: Execution Plans for Data Modifications

All the previous execution plans in the book have been for SELECT queries. However, the optimizer also generates execution plans for all data modification queries issued for the database, to instruct the execution engine how best to undertake the requested data change. This chapter will examine the characteristics of execution plans for INSERT, UPDATE, DELETE, and MERGE queries. You're going to find the execution plans for data modification queries very handy. You'll see how IDENTITY columns get resolved during INSERTs, and how referential constraints are managed during DELETEs, just to name a couple of the processes exposed within the execution plan. You'll also be able to use these plans in tuning your data modification queries, just like you would a SELECT query.

Plans for INSERTs

INSERT queries are always executed against a single table. This would lead you to believe that their execution plans will be simple. However, to account for IDENTITY columns, computed columns, referential integrity checks, and other table structures, execution plans for insert queries can be quite complicated.

Listing 6-1 shows a very simple INSERT query.

```
INSERT INTO Person.Address
(
    AddressLine1,
    AddressLine2,
    City,
    StateProvinceID,
    PostalCode,
    rowguid,
    ModifiedDate
)
VALUES
(   N'1313 Mockingbird Lane', -- AddressLine1 - nvarchar(60)
    N'Basement',              -- AddressLine2 - nvarchar(60)
    N'Springfield',           -- City - nvarchar(30)
```

```
    79,                          -- StateProvinceID - int
    N'02134',                    -- PostalCode - nvarchar(15)
    NEWID(),                     -- rowguid - uniqueidentifier
    GETDATE()                    -- ModifiedDate - datetime
    );
```

Listing 6-1

Just as for any other query, we can capture either the estimated or the actual execution plan. As discussed in Chapter 1, if we request the estimated plan, we don't execute the query and so don't insert any data; we simply submit the query for inspection by the optimizer, in order to see the plan.

If we want to see runtime information, we execute the query, requesting the actual plan. If we want to see the actual plan without modifying the data, we could wrap the query in a transaction and roll back that transaction after capturing the plan.

In this case, we'll just capture the estimated plan, as shown in Figure 6-1.

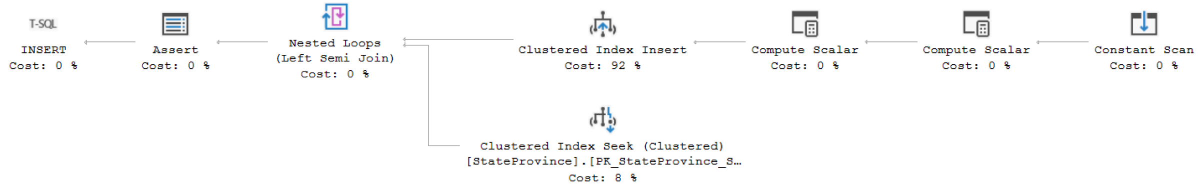


Figure 6-1: Estimated plan showing an INSERT.

The physical structure of the table that the `INSERT` query accesses can affect the resulting execution plan. This table has an `IDENTITY` column and a `FOREIGN KEY` constraint.

Just as with the `SELECT` queries we've examined, we can read this plan from right to left (data flow order) or from left to right (operator call order). However, before we attempt to follow the various steps in the plan, we'll start by looking behind the "first operator" because, as we discovered in Chapter 2, it contains a lot of useful information about the plan.

INSERT operator

Figure 6-2 shows the properties for the `INSERT` operator for this plan.

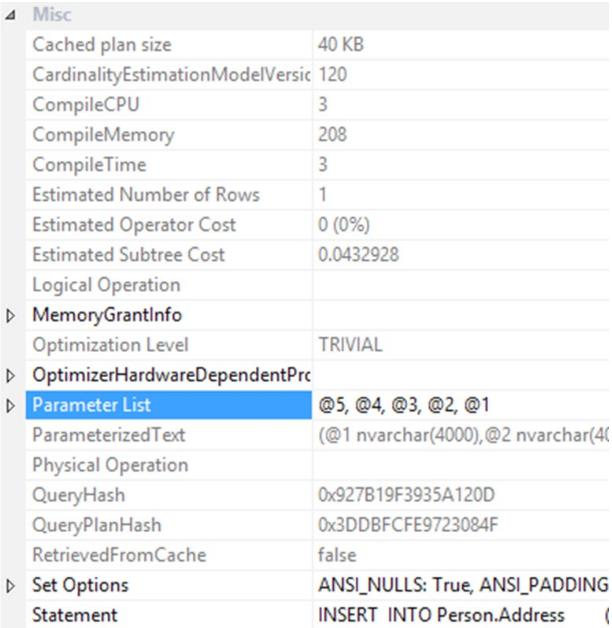


Figure 6-2: Properties for the INSERT operator.

Despite the larger number of operators in this plan, the optimizer still classified it as a trivial plan. Also note that the optimizer has performed simple parameterization on this query, swapping the hard-coded values supplied in the `VALUES` clause in Listing 6-1, with parameters, in order to promote plan reuse.

We can see how the parameters were resolved by looking at the **ParameterizedText** property value, shown in Listing 6-2 (after copying and pasting, and applying formatting to make it readable).

```
(@1 nvarchar(4000),@2 nvarchar(4000),@3 nvarchar(4000),@4 int,@5
nvarchar(4000))
INSERT  INTO [Person].[Address]
        ([AddressLine1],
         [AddressLine2],
         [City],
         [StateProvinceID],
         [PostalCode],
         [rowguid],
         [ModifiedDate]
        )
VALUES  (@1,
         @2,
         @3,
         @4,
         @5,
         newid(),
         getdate()
        )
```

Listing 6-2

Let's now step through the plan, reading from right to left, following the data flow. We started with an operator that is new to us: **Constant Scan**.

Constant Scan operator

The **Constant Scan** operator introduces into the results one or more rows, originating from a "scan of an internal table of constants." In other words, the rows come from the properties of the operator itself, specifically the **Values** properties, rather than from any external data source.

A **Constant Scan** generates one or more rows, consisting of one or more columns, and it has many possible roles within an execution plan. To understand its role in any specific execution plan, you need to look at what values it produces, and where in the plan these values are used. To do this, we need to look at the detailed properties of the operators.

You can see what columns it returned from the **Output List** property, and the row values from the **Values** property. Figure 6-3 shows the properties of the **Constant Scan** for a trivial query (`SELECT * FROM (VALUES (1,2),(3,4),(5,6)) AS x(a,b);`), showing that the operator generates two columns (**Union1006**, **Union1007**) and three rows.

⊞ Output List	Union1006, Union1007
Parallel	False
Physical Operation	Constant Scan
⊟ Values	(Scalar Operator((1)), Scalar Operator((2)
⊞ [1]	Scalar Operator((1)), Scalar Operator((2))
⊞ [2]	Scalar Operator((3)), Scalar Operator((4))
⊞ [3]	Scalar Operator((5)), Scalar Operator((6))

Figure 6-3: The defined values of the Constant Scan operator.

In less trivial cases, it's useful to follow the column names given in the **Output List** throughout the plan to see where else they are used, and why they are required.

For the **Constant Scan** in Figure 6-1, the **Output List** is blank, and the **Values** property absent, indicating that the operator generates a single, empty row. We can also see the row is empty by hovering over the data output pipe from **Constant Scan**. Notice that the **Row Size** is **9 B** (which indicates column header only).

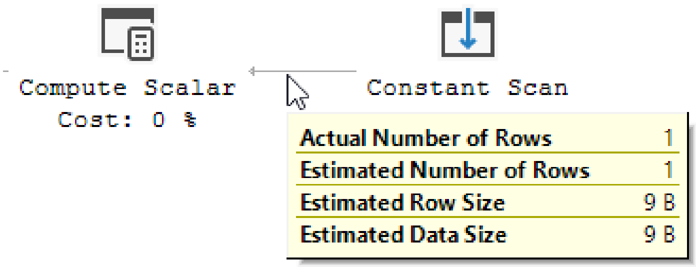


Figure 6-4: Tooltip showing an empty row returned a Constant Scan.

Sometimes, in a plan, you will see that a **Constant Scan** returns an empty row, essentially a place holder for information that will be added by other operators within the plan, such as a **Compute Scalar**. In Figure 6-1, the **Constant Scan** is followed by not one, but two of them.

The first **Compute Scalar** operator reads each of the rows from the **Constant Scan** (in this case, one row only) and for each row calls a function called `getidentity`, as you can see from the **Defined Values** property of this operator.

```
[Expr1002] = Scalar Operator(getidentity((373576369),(11),NULL))
```

This is where SQL Server generates an identity value, for the `AddressID` column, which is the Primary Key and is an `IDENTITY` column. The first two values being passed are the `object_id` and the `database_id`. I don't know what the third parameter represents, but here it's a `NULL` value.

The fact that this operation precedes the `INSERT`, and any integrity checks, within the plan, helps explain why, when an `INSERT` fails, you still get a gap in the `IDENTITY` values for a table. The input for this operator was a single empty row, and so its output, after adding **Expr1002**, is just a single row with one column holding the `IDENTITY` value.

The second **Compute Scalar** operator reads the row from the previous operator, and adds to it a series of columns for most of the parameterized values in the query, plus the new `uniqueidentifier` (`guid`) value, and the date and time from the `GETDATE` function.

The **Defined Values** property, in Figure 6-5 illustrates all this.

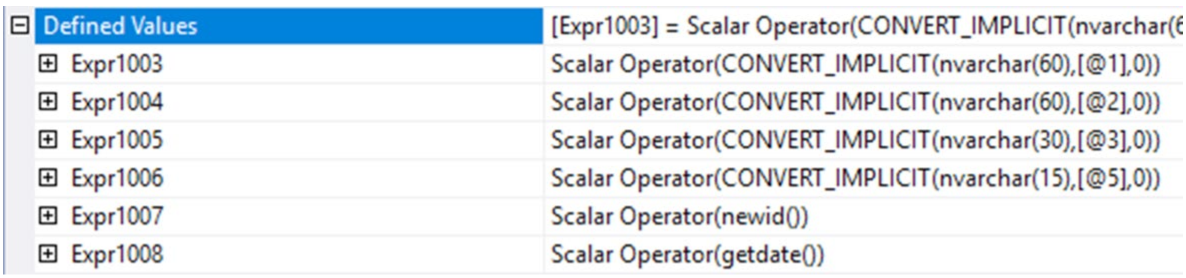


Figure 6-5: Defined Values of the Compute Scalar operator.

The hard-coded strings in the query were converted to variables with a data type of `nvarchar(4000)`. The expression for each column value converts them from their inferred data type to the data type of the corresponding column in the table.

The output from this second **Compute Scalar**, as confirmed by its **Output List** property, is a single row with columns containing the `IDENTITY` value (**Expr1002**) defined earlier, the parameter values (**Expr1003 – 1006**), the `guid` value (**Expr1007**) and the `getdate` value (**Expr1008**).

The reason we have 7 column values to insert (not including the identity), and only 6 defined values is that the inferred data type for the `StateProvinceID` variable is an `INT`, so this doesn't need conversion.

The **Clustered Index Insert** operator receives this single row, containing all of these values.

Clustered Index Insert operator

The **Clustered Index Insert** operator represents the insert of our data into the clustered index. In the execution plan in Figure 6-1, this operation represents most of the estimated cost of this plan (92%). Probably the most important property on this operator, for this example, is the **Object** property, shown in Figure 6-6.

⊟ Object	[AdventureWorks2014].[Person].[Address].[PK_Address_AddressID], [Adver
⊞ [1]	[AdventureWorks2014].[Person].[Address].[PK_Address_AddressID]
⊞ [2]	[AdventureWorks2014].[Person].[Address].[AK_Address_rowguid]
⊞ [3]	[AdventureWorks2014].[Person].[Address].[IX_Address_AddressLine1_Addr
⊞ [4]	[AdventureWorks2014].[Person].[Address].[IX_Address_StateProvinceID]

Figure 6-6: Multiple indexes on display in Clustered Index Insert operator.

You see that the insert affects four different indexes, one being the clustered index into which we insert the new row, and the other three being three nonclustered indexes on this table, to which data also needs to be added. In this case, these additional nonclustered indexes are modified by adding them to the object list of the clustered index modification operator. The alternative is that they can be modified from within their own operators (a per-index plan; we'll see a per-index `DELETE` plan later).

Filtered indexes and indexed, or materialized, views are always modified from within their own operators.

You can see the parameters that have been created and formatted in the **ScalarOperator** property that is inside the **Predicate** property.

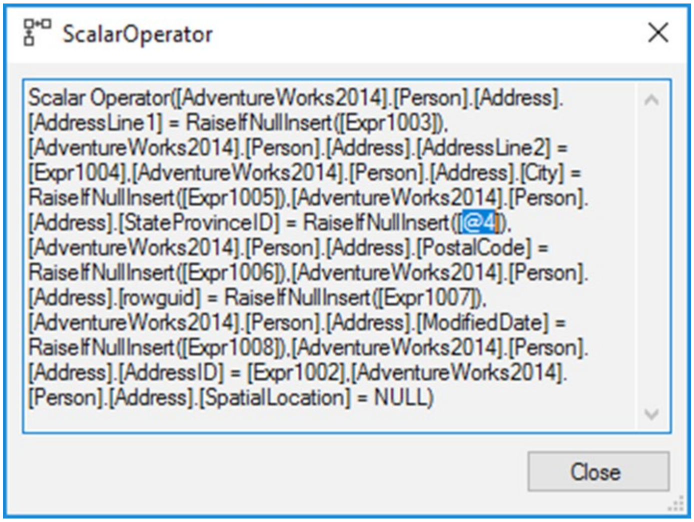


Figure 6-7: Parameters evaluated in the ScalarOperator.

This data is broken down within the properties of the operator, but they're broken down individually, so it doesn't make them any easier to read. I've highlighted the `@4` value of the `StateProvinceID`, mentioned earlier, highlighting the fact that it reads this variable directly, whereas all the other columns are set using the expressions, **Expr1003**, and so on, generated earlier in the **Compute Scalar** operator.

The next item of interest is the value of the **Output List** property, the `Person.Address.StateProvinceId` as shown in Figure 6-8. Since this column is a `FOREIGN KEY`, SQL Server needs to check for referential integrity.

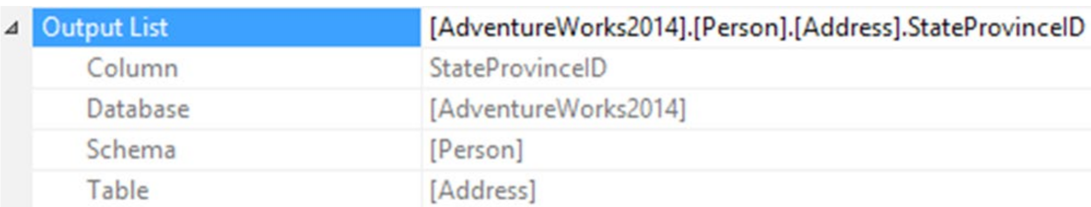


Figure 6-8: Output List property of a Clustered Index Insert.

We now come to the familiar **Nested Loops** join operator (the final part of the plan is reproduced in Figure 6-9).

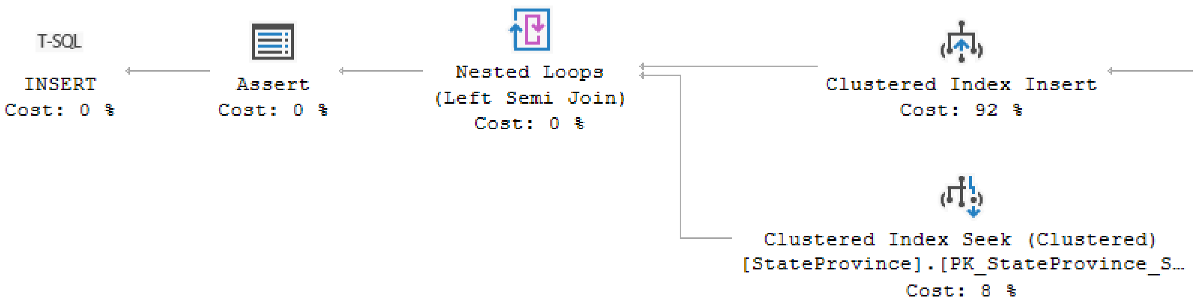


Figure 6-9: Section of the execution plan with the Nested Loops operator.

The **Nested Loops** receives the row with the `StateProvinceID` that has already been inserted, and then calls the **Clustered Index Seek**, which reads the `PRIMARY KEY` column of the parent table to check that the value we're inserting exists in that column. You'll note that the **Nested Loops** operator is marked as a **Left Semi Join**. This means that it's only looking for a single match rather than finding all matches. The output from the **Nested Loops** join is a new expression, which is tested by the next operator, **Assert**.

Assert operator

An **Assert** operator verifies that a certain condition, or conditions, can be met, all of which it lists in the **Predicate** property, which returns `NULL` if they are all met. Each non-`NULL` value results in a rollback; the exact error message is determined by the actual value.

In this example, the **Assert** operator checks that the value of **Expr1012** is not `NULL`. Or, in other words, that the data inserted into the `Person.Address.StateProvinceId` field matched a piece of data in the `Person.StateProvince` table; this was the referential check. You can see this in the **Predicate** property in Figure 6-10.

Physical Operation	Assert
Predicate	CASE WHEN [Expr1012] IS NULL THEN (0) ELSE NULL END
Startup Expression	False

Figure 6-10: The Predicate property of the Assert operator.

Plans for UPDATEs

`UPDATE` queries also work against one table at a time. Depending on the structure of the table, and the columns to be updated, the effect on the execution plan could be as significant as that shown above for the `INSERT` query. Consider the `UPDATE` query in Listing 6-3.

```
UPDATE Person.Address
SET City = 'Munro',
    ModifiedDate = GETDATE()
WHERE City = 'Monroe';
```

Listing 6-3

Figure 6-11 shows the estimated execution plan (not included is a **Missing Index** hint suggesting a possible index on the `City` column, to help the performance of the query).

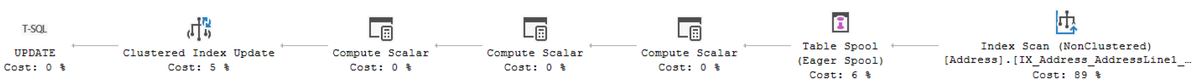


Figure 6-11: Execution plan showing an UPDATE.

Once again, we can start reading this plan by checking the **UPDATE** operator to see what's there. However, in this case, nothing new is introduced. This plan has gone through **FULL** optimization and a "Good Enough Plan Found" was the reason for early termination.

Stepping through the plan, reading from right to left, the first operator is an **Index Scan** on the table, which scans all the rows in this index, and will return only those rows `WHERE [City] = 'Monroe'` (see the **Predicate** property of the **Index Scan**).

The optimizer estimates that it will return only 4.6 rows, which helps explain why an index on `City` was suggested by the optimizer. As always, whether you create it depends entirely on the importance of the query within your workload, or its frequency of execution.

The **Index Scan** operator is called by the next operator along, a **Table Spool** (**Eager Spool**).

Table Spool (Eager Spool) operator

As we discussed in Chapter 5, the Table Spool operator provides a mechanism for storing the incoming data in a worktable, so that it may be reused, perhaps several times, within an execution plan. However, this is the first time we've encountered an **Eager Spool**, which keeps requesting rows from its child operator until it has all of them, and only then will pass on the first row. This means that it is a blocking operator, which the optimizer will generally try to avoid. However, in this case, it's exactly the behavior that is required; it is there to prevent the Halloween Problem (see: http://en.wikipedia.org/wiki/Halloween_Problem).

The spool reads all of the rows to be updated and stores them in its worktable, and this data is referenced throughout the rest of the processing of the query. By using only that worktable to drive the rest of the query, we are guaranteed to not see already updated data again.

The next three operators are all **Compute Scalar** operators, which we have seen before. In this case, they are used to evaluate expressions and to produce a computed scalar value, such as the `GETDATE()` function used in the query.

After these simple and clear computations, there are also computations creating the **Expr1012** value, derived from the **Expr1006** value, which are less easy to explain. Potentially, they play some role in ensuring that the data being updated is updated correctly and safely, but equally they could be an artifact of how the execution plan is generated. A **Compute Scalar** operator is very low cost, to the point where the optimizer sometimes does not even bother to remove computations that are no longer needed.

Clustered Index Update operator

Now we get to the core of the `UPDATE` query, the **Clustered Index Update** operator. This operator reads its input data, uses it to identify the rows to be updated, and updates them. If you examine the **Object** property you'll find that two objects are getting updated: the clustered index itself, and a nonclustered index that happens to have the `City` column as one of its keys.

In this example, the **Clustered Index Update** operator is updating rows passed in from an Index Scan, but in certain cases it can find the rows to update by itself, based on a Predicate. Listing 6-4 creates a very simple table, loads a row into it, and runs an `UPDATE` on that row.

```
CREATE TABLE dbo.Mytable (id INT IDENTITY(1, 1) PRIMARY KEY
CLUSTERED,
                          val VARCHAR(50));
INSERT dbo.Mytable (val)
VALUES ('whoop' -- val
    );
UPDATE dbo.Mytable
SET val = 'WHOOP'
WHERE id = 1;
```

Listing 6-4

The execution plan for the `UPDATE` is very simple because all the work is performed directly within the **Clustered Index Update** operator. The rows are filtered and updated in place. You can see the details by looking at the properties of the operator, the **Seek Predicate** property, in particular.

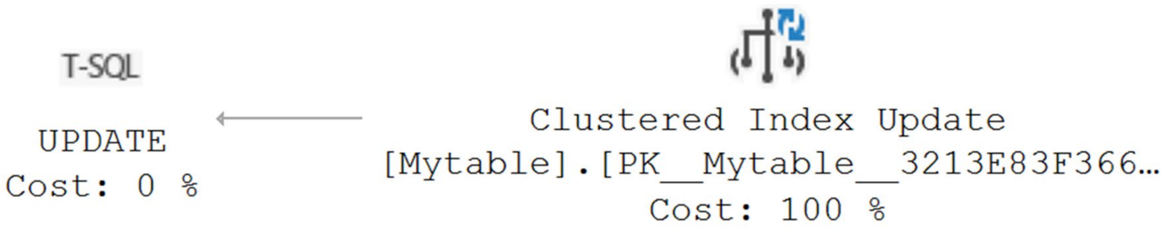


Figure 6-12: A simple execution plan for an UPDATE.

Plans for DELETEs

What kind of execution plan is created for a `DELETE` query? Let's find out!

A simple DELETE plan

Run the code in Listing 6-5 and capture the actual execution plan.

```
BEGIN TRAN;
DELETE FROM Person.EmailAddress
WHERE BusinessEntityID = 42;
GO
ROLLBACK TRAN;
```

Listing 6-5

Figure 6-13 shows the actual execution plan.

Figure 6-13: Simple execution plan for a DELETE.

Not all execution plans are complicated and hard to understand. In this case, the **Clustered Index Delete** operator defines the rows in the clustered index that need to be deleted, and deletes them. Not all `DELETE` plans will look this simple if the optimizer needs to validate referential integrity for the `DELETE` operation but, in this case, it didn't.

The `DELETE` operator shows a **TRIVIAL** plan and simple parameterization to help promote plan reuse. Figure 6-14 shows the properties of the **Clustered Index Delete**.

Object	[AdventureWorks2014].[Person].[EmailAddress].[PK_EmailAddress_
⊞ [1]	[AdventureWorks2014].[Person].[EmailAddress].[PK_EmailAddress_
⊞ [2]	[AdventureWorks2014].[Person].[EmailAddress].[IX_EmailAddress_I
Output List	
Parallel	False
Physical Operation	Clustered Index Delete
Seek Predicate	Seek Keys[1]: Prefix: [AdventureWorks2014].[Person].[EmailAddres:
⊟ [1]	Prefix: [AdventureWorks2014].[Person].[EmailAddress].BusinessEnt
⊟ Prefix	[AdventureWorks2014].[Person].[EmailAddress].BusinessEntityID =
⊞ Range Column:	[AdventureWorks2014].[Person].[EmailAddress].BusinessEntityID
⊟ Range Expressi	Scalar Operator(CONVERT_IMPLICIT(int,[@1],0))
⊟ Identifier	
⊞ Column R	ConstExpr1004
ScalarString	CONVERT_IMPLICIT(int,[@1],0)
Scan Type	EQ

Figure 6-14: Clustered Index DELETE operator properties.

As we have seen previously in this chapter, the **Object** property shows that more than just the clustered index has been modified. Even with this very simple execution plan, you can see that the nonclustered index modification is covered within this one operator. Also, you can see how the row or rows that will be deleted are found through the **Seek Predicate** operator. Finally, within the expression, you see that simple parameterization has occurred because we're not comparing the actual value of 42 that was supplied, but rather **@1**, a parameter.

A per-index DELETE plan

In the examples so far, all nonclustered indexes were modified in the same operator that modifies the clustered index. You'll see this referred to, occasionally, as a "narrow" plan. Another way that the optimizer can choose to modify all the required nonclustered indexes on a table is to process the modification of each one separately, referred to as a "wide" or "per-index" plan.

To see an example of a wide `DELETE` plan, we'll first create a materialized view and then delete some data.

```
CREATE OR ALTER VIEW dbo.TransactionHistoryView
WITH SCHEMABINDING
AS
SELECT COUNT_BIG(*) AS ProductCount,
       th.ProductID
FROM Production.TransactionHistory AS th
GROUP BY th.ProductID
GO
CREATE UNIQUE CLUSTERED INDEX TransactionHistoryCount
ON dbo.TransactionHistoryView(ProductID)
GO
BEGIN TRAN;
DELETE FROM Production.TransactionHistory
WHERE ProductID = 711;
ROLLBACK TRAN;
```

Listing 6-6

The resulting execution plan is much more complex than before.

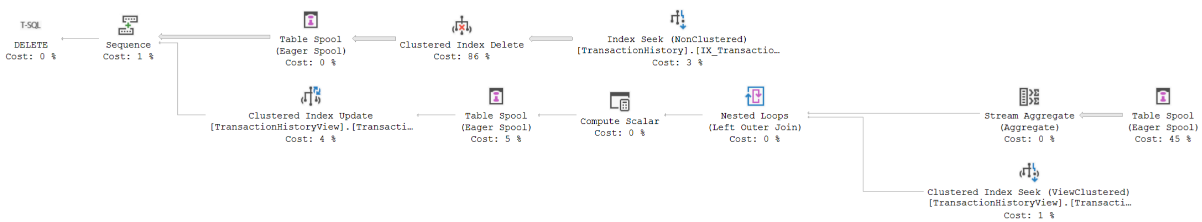


Figure 6-15: A per-index DELETE execution plan.

In reading this plan, we're going to start off on the left-hand side, following the order of execution. There are two things we must address there, before we switch back over to the data flow order of operations. Figure 6-16 shows the first two operators of the plan.

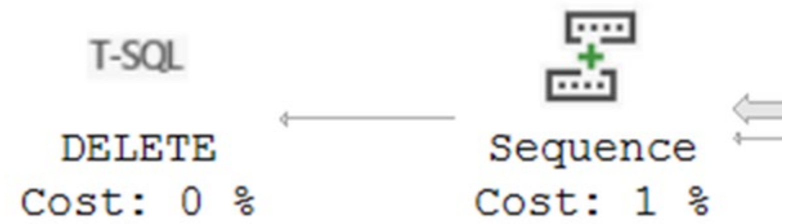


Figure 6-16: The DELETE operator receiving information from the Sequence operator.

After the `DELETE` operator, which we've already discussed in this chapter, the next operator, in order of execution, is the **Sequence** operator. It takes some number of inputs, in this case two, and processes them in precise order, from top to bottom. The inputs are related objects in which data must be modified, and the operations must be performed in the correct sequence. In our example, the optimizer needs to delete data from a clustered index, and its associated nonclustered indexes, and then from a second clustered index that defines the materialized view.

With a **Sequence** operator, almost always as part of an `UPDATE` or `DELETE`, each input represents a different object within the database. Even if multiple values can be returned from the various inputs to the **Sequence** operator, only the bottom, the final, input is passed on.

This makes the **Sequence** a partially blocking operator, since all processing for one input must be complete before the next is started. Only when all other inputs have completed, and the bottom input starts, will the **Sequence** start to pass rows it receives on to the next operator. Understanding that we're dealing with the **Sequence** operator will make the rest of the plan easier to understand.

Figure 6-17 shows the operators that comprise the top input for the **Sequence** operator.

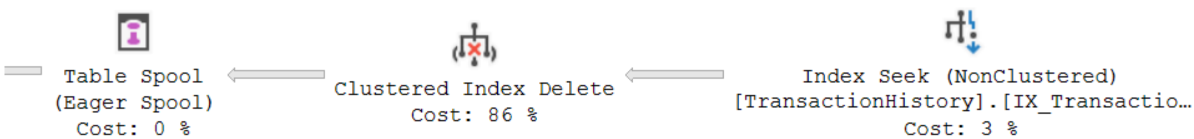


Figure 6-17: The operators in the top input to a Sequence operator.

The start of the processing in the data flow direction begins with an **Index Seek** operation against the `IX_TransactionHistory_ProductID` nonclustered index. The output from that index is a listing of `TransactionID` values that match the input value of 711, provided for the `ProductID`, from Listing 6-6.

This listing of `TransactionID` values then goes to the **Clustered Index Delete** operation which will take care of removing all data from the clustered index that defines the table. Figure 6-18 shows the output from the **Clustered Index Delete** operator.

⊟ Output List	[AdventureWorks2014].[Production].[TransactionHistory].TransactionID, [Advent
⊞ [1]	[AdventureWorks2014].[Production].[TransactionHistory].TransactionID
⊞ [2]	[AdventureWorks2014].[Production].[TransactionHistory].ProductID
⊞ [3]	[AdventureWorks2014].[Production].[TransactionHistory].ReferenceOrderID
⊞ [4]	[AdventureWorks2014].[Production].[TransactionHistory].ReferenceOrderLineID

Figure 6-18: The Output List property from a Clustered Index Delete.

If you check the output, you'll see the column, `ProductID`, which will be used elsewhere in the plan. The output is then loaded into a **Table Spool** operator for later use. Any time you start to deal with table spools, it's always a good idea to get the **NodeID** value (in this case it is 2), which you can find from the Properties or the tooltip (more on this shortly).

The Table Spool is just temporary storage for use later in the plan and nothing else is done to this data during this process except to load it into the Spool for later use. The logical operation is an **Eager Spool**. An **Eager Spool** will first collect all information from preceding operators before passing on any rows. This means that all rows that match our criteria, `ProductID = 711`, are already deleted, before the rest of the plan receives any data from this operator.

That completes the top input to the **Sequence** operator. Figure 6-19 shows the bottom input.

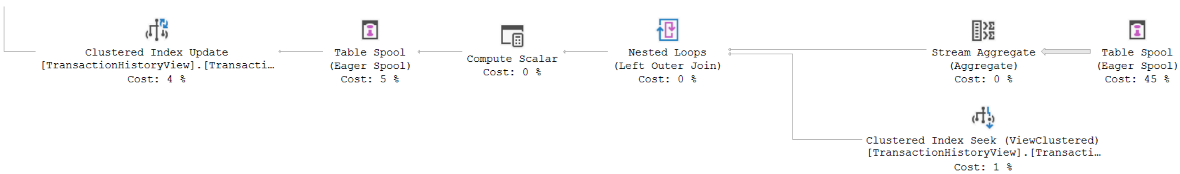


Figure 6-19: Complete bottom input of the Sequence Operator.

We'll break this down a little farther, for ease of reading, with Figure 6-20 showing the far right of the plan, up to the **Nested Loops** operator.

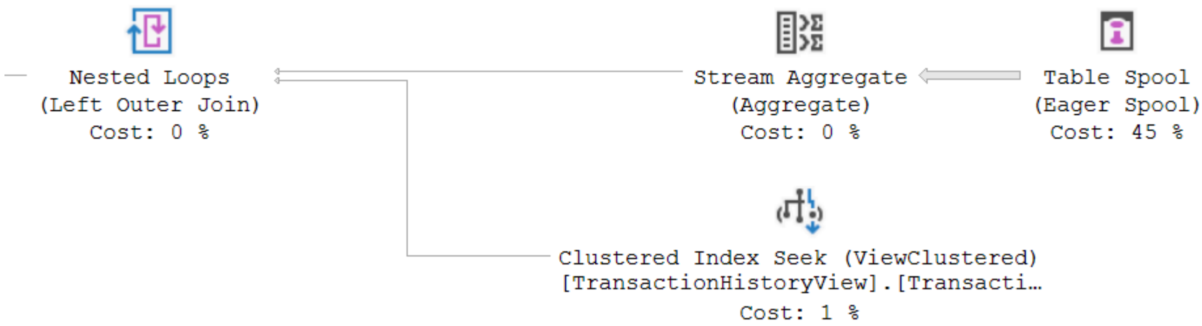


Figure 6-20: Identifying matching rows in materialized view.

We start with another **Table Spool** operator. This Table Spool operator has its own **NodeID**, showing where it falls within the processing of the plan. However, it has an additional piece of information, the **Primary Node ID**, indicating that it is reusing data stored in the Table Spool found in the top input.

Node ID	13
Number of Executions	1
⊞ Output List	[AdventureWc
Parallel	False
Physical Operation	Table Spool
Primary Node ID	2

Figure 6-21: Properties of the Table Spool operator.

All that information was loaded once from the output of the **Clustered Index Delete** operator, in the top input, and now is going to be reused in this set of operations in the bottom input.

The next operator is a **Stream Aggregate** operator (see Chapter 5), which takes the output from the deleted values in the clustered index and aggregates them in order to make them match the data in the materialized view. The **Nested Loops** join then adds the corresponding data, as it is currently stored in the materialized view.

Figure 6-22 shows the next section of the lower input of the **Sequence** operator.

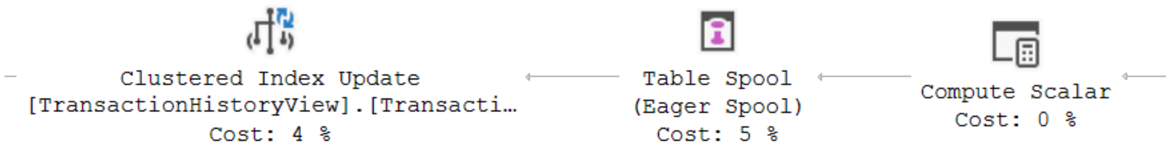


Figure 6-22: The DELETE of the materialized view.

The **Compute Scalar** computes the new value for use in the materialized view by subtracting the number of deleted rows by `ProductID` (as computed in the **Stream Aggregate**) from the originally stored data. The **Table Spool** operator has its own **NodeID**, and no **Parent NodeID**, so isn't reusing data from elsewhere. In this case, it's again protecting against the Halloween Problem. Finally, we see a **Clustered Index Update** that modifies the data in the materialized view itself.

This example illustrates the alternative way to maintain indexes in data modification plans. It is up to the optimizer to decide to use either method, or a mix. This decision is as always based on estimations on the cost of maintaining indexes in random order, versus the cost of saving the rows in a **Table Spool**, sorting them, and then maintaining the indexes with pre-ordered data. Though this example showed a `DELETE` plan, the same options apply to `INSERT`, `UPDATE`, and `MERGE` plans.

Drop the materialized view before we continue.

```
DROP INDEX TransactionHistoryCount ON dbo.TransactionHistoryView;
GO
DROP VIEW dbo.TransactionHistoryView;
GO
```

Listing 6-7

Plans for MERGE queries

With SQL Server 2008, Microsoft introduced the `MERGE` query. This is a method for modifying data in your database in a single query, instead of one query for `INSERT`s, one for `UPDATE`s, and another for `DELETE`s. The nickname for this is an "upsert." The simplest application of the `MERGE` query is to perform an `UPDATE` if there are existing key values in a table, or an `INSERT` if they don't exist. The query in Listing 6-7 `UPDATE`s or `INSERT`s rows to the `Purchasing.Vendor` table.

```
DECLARE @BusinessEntityId INT = 42,
        @AccountNumber NVARCHAR(15) = N'SSHI',
        @Name NVARCHAR(50) = N'Shotz Beer',
        @CreditRating TINYINT = 2,
        @PreferredVendorStatus BIT = 0,
        @ActiveFlag BIT = 1,
        @PurchasingWebServiceURL NVARCHAR(1024) = N'http://
shotzbeer.com',
        @ModifiedDate DATETIME = GETDATE();
BEGIN TRANSACTION;
MERGE Purchasing.Vendor AS v
USING
(   SELECT @BusinessEntityId,
           @AccountNumber,
           @Name,
           @CreditRating,
           @PreferredVendorStatus,
           @ActiveFlag,
           @PurchasingWebServiceURL,
           @ModifiedDate) AS vn (BusinessEntityId, AccountNumber,
NAME, CreditRating, PreferredVendorStatus, ActiveFlag,
PurchasingWebServiceURL, ModifiedDate)
ON (v.AccountNumber = vn.AccountNumber)
WHEN MATCHED THEN
    UPDATE SET v.Name = vn.NAME,
               v.CreditRating = vn.CreditRating,
               v.PreferredVendorStatus = vn.PreferredVendorStatus,
               v.ActiveFlag = vn.ActiveFlag,
               v.PurchasingWebServiceURL =
vn.PurchasingWebServiceURL,
               v.ModifiedDate = vn.ModifiedDate
WHEN NOT MATCHED THEN
    INSERT (BusinessEntityID,
            AccountNumber,
            Name,
            CreditRating,
            PreferredVendorStatus,
            ActiveFlag,
            PurchasingWebServiceURL,
            ModifiedDate)
```

```
    VALUES (vn.BusinessEntityId, vn.AccountNumber, vn.NAME,
vn.CreditRating, vn.PreferredVendorStatus, vn.ActiveFlag,
            vn.PurchasingWebServiceURL, vn.ModifiedDate);
ROLLBACK TRANSACTION;
```

Listing 6-8

This query uses the alternate key, the `AccountNumber` column, on the `Purchasing.Vendor` table. If the value supplied (in this case `'SSHI'`) matches a key value in this column, then the query will run an `UPDATE`, and if it doesn't, it will perform an `INSERT`. Figure 6-22 shows the execution plan.

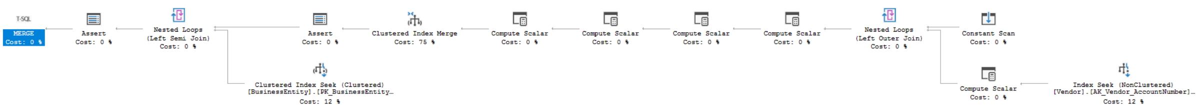


Figure 6-23: Full plan for the MERGE query.

As you can see, that plan is a bit large for the book, so I'll break this plan down in right-to-left order.

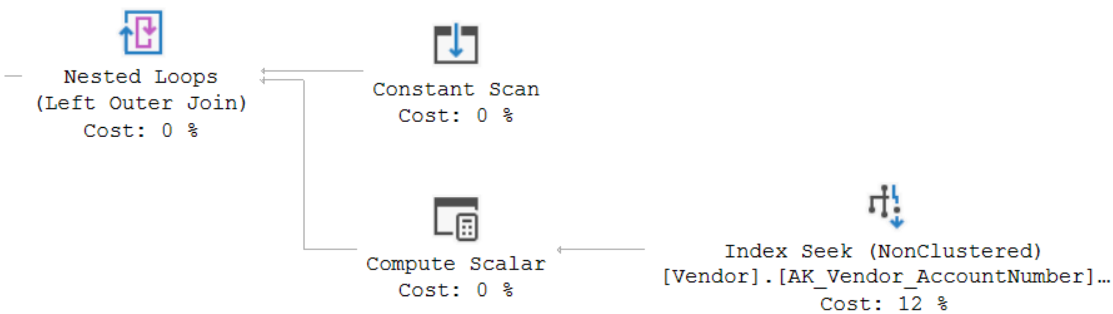


Figure 6-24: Loading the Constant Scan and checking for a row.

This first section of the plan contains a series of steps to prepare for the main operations to come. The **Constant Scan** generates one empty row, a place holder for data so that all the operators will have information to work with, even if it's an empty set. The **Nested Loops** operator uses this empty row to drive a single execution of its inner input, where the **Index Seek** against the `Vendor.AK_Vendor_AccountNumber` nonclustered index will pull back any rows to be updated (i.e. that match the supplied **Seek Predicate**). We'd expect one row at most, since it's a `UNIQUE` index but, in this case, the **Properties** for the data flow between the **Index Seek** and the first **Compute Scalar** reveals zero rows returned.

Actual Number of Rows	0
Estimated Data Size	30 B
Estimated Number of Rows	1
Estimated Row Size	30 B

Figure 6-25: Properties of the Compute Scalar operator.

For every row it receives, the **Compute Scalar** operator creates a value `TrgPrb1001` and sets it to a value of 1, as you will see in the **Defined Values** property value for the operator.

The **Nested Loops** operator combines the empty column from the **Constant Scan** with the data (if any) from the **Compute Scalar**, by using a **Left Outer Join**. If, as in our case, no data is returned by the **Compute Scalar**, it still returns a row, using `NULL` values. The effect of this is that the value 1 is passed into `TrgPrb1001` if the Index Seek finds a row, or `NULL` if it doesn't. This is used later in the plan to determine if any rows exist for `UPDATE` or `DELETE`.

The next part of the plan is a series of **Compute Scalar** operations, as shown in Figure 6-25.

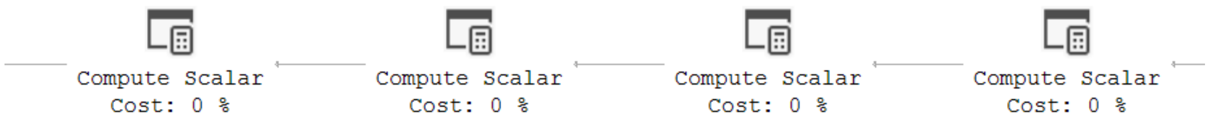


Figure 6-26: Multiple calculations against the data to determine what to do with it.

The hard part of reading a plan like this is trying to figure out what each of the **Compute Scalar** operators does. This is revealed by the **Defined Values** and **Output List** property values. Working from the right again, the first **Compute Scalar** operator in Figure 6-25 performs a calculation:

```
[Action1003] = Scalar Operator(ForceOrder(CASE WHEN [TrgPrb1001] IS
NOT NULL THEN (1) ELSE (4) END))
```

This **Compute Scalar** operator creates a new value, called `Action1003` in my case, and since `TrgPrb1001` is null, the value is set to "4." Depending on your SQL Server version, and the updates applied, you may see different values for `Action1003` or `Expr1005`, or any of the various generated values within the plan, even though you may have an otherwise identical plan. This simply reflects minor changes within the optimizer and the order in which it initializes each of these expressions.

The next **Compute Scalar** operator loads all the variable values into the row, and performs two other calculations:

```
[Expr1004] = Scalar Operator([@ActiveFlag]), [Expr1005] =
Scalar Operator([@PurchasingWebServiceURL]), [Expr1006] =
Scalar Operator([@PreferredVendorStatus]), [Expr1007] = Scalar
Operator([@CreditRating]), [Expr1008] = Scalar Operator(CASE
WHEN [Action1003]=(4) THEN [@BusinessEntityId] ELSE
[AdventureWorks2014].[Purchasing].[Vendor].[BusinessEntityID]
as [v].[BusinessEntityID] END), [Expr1009] = Scalar Operator([@
ModifiedDate]), [Expr1010] = Scalar Operator([@Name]), [Expr1011]
= Scalar Operator(CASE WHEN [Action1003]=(4) THEN [@AccountNumber]
ELSE [AdventureWorks2014].[Purchasing].[Vendor].[AccountNumber] as
[v].[AccountNumber] END)
```

Looking at the expression for `Expr1011`, we can begin to understand what's happening. The first **Compute Scalar** output, `TrgPrb1001`, determined if the row existed in the table. If it existed, then the second **Compute Scalar** would have set `Action1003` equal to 1, meaning that the row did exist, and this new **Compute Scalar** would have used the value from the table but, instead, it's evaluating `Action1003` and choosing the variable `@AccountNumber`, since an `INSERT` is needed. The same logic is used in `Expr1008` for the `BusinessEntityId` value. The result of this **Compute Scalar** is that all expressions hold the correct value for the `INSERT` or `UPDATE`, as determined by the `Action1003`.

Moving to the left, the next **Compute Scalar** operator validates what `Action1003` is and sets a new value, `Expr1023`, based on this formula:

```
[Expr1023] = Scalar Operator(CASE WHEN [Action1003] = (1) THEN
(0) ELSE [Action1003] END)
```

We know that `Action1003` is set to 4, so this expression will be set to 4.

The final **Compute Scalar** operator sets two values equal to themselves, for some reason that's not completely clear to me. It may be some internal process within the optimizer that is evidenced here in the execution plan. Finally, we're ready to move on with the rest of the execution plan.

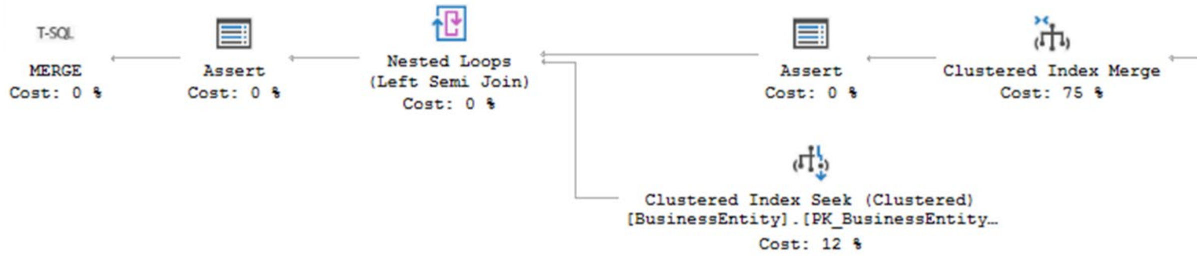


Figure 6-27: Final steps in the Merge operation.

The **Clustered Index Merge** receives all the information added to the data stream by the various operators, and uses it to determine if the action is an `INSERT`, an `UPDATE`, or a `DELETE`, and performs that action. You can see the outcome within the **Action Column** property of the operator, in Figure 6-27, which shows a value of `Action1003`.

⊟ Action Column	Action1003
Column	Action1003

Figure 6-28: Action Column values for Clustered Index Merge operator.

Of course, in this case, it's only either an `INSERT` or `UPDATE`. You can even see the information in the **Predicate** property of the operator.

⊟ Predicate	
⊟ [1]	
⊞ ScalarOperator	Scalar Ope
SetPredicateType	Insert
⊟ [2]	
⊞ ScalarOperator	Scalar Ope
SetPredicateType	Update

Figure 6-29: Predicate values of the Clustered Index Merge operator.

Appropriately, in this case, because of all the work that the **Merge** operation must perform in modifying two indexes, the optimizer estimates that this operation will account for 75% of the cost of the execution plan.

Next, an **Assert** operator runs a check against a constraint in the database, validating that the data is within a certain range. The data passes to the **Nested Loops** operator, which is used to retrieve values used for validation that the `BusinessEntityId` referential integrity is intact, through the **Clustered Index Seek** against the `BusinessEntity` table. This action is only performed in this case, since this is an `INSERT` operation, as determined earlier by the definition of the value of `Action1003`. The **Nested Loops** operator has a Pass Through function, which skips invoking the inner input, in other cases. We can see that in Figure 6-30.

Figure 6-30: The evaluation that determines if it is a Pass Through.

The information gathered by that join passes to another **Assert** operator, which validates the referential integrity, assuming that it was an `INSERT` action. The query is then completed.

As you can see, a lot of action takes place within execution plans but, with careful review, it is possible to identify most of what is going on.

Prior to the `MERGE` query, you may have done a query of this type dynamically. You either had different procedures for each of the processes, or different queries within an `IF` clause. Either way, you ended up with multiple execution plans in the cache, for each process. This is no longer the case. If you were to modify the query in Listing 6-7 and change one simple value as in Listing 6-9…

```
…
@AccountNumber NVARCHAR(15) = 'SPEEDCO0001',
…
```

Listing 6-9

…the exact same query with the exact same execution plan will now `UPDATE` the data for values where the `AccountNumber` is equal to that passed through the parameter. Therefore, this plan, with the **Merge** operator, creates a single reusable plan for all the data manipulation operations it supports.

Summary

This chapter dealt with the plans for relatively simple data modification queries. The key lessons are that you read these queries in the same way that you read a `SELECT` query and use the same tools such as properties, and the estimated costs, to try to understand how and why the optimizer has implemented the plan in this way.

Chapter 7: Execution Plans for Common T-SQL Statements

In Chapters 2 through 6, we dealt with single statement T-SQL queries. As we saw, sometimes even these relatively simple queries can generate complicated execution plans. In this chapter, we'll extend our scope to consider plans for common T-SQL statements and objects, such as stored procedures, subqueries, derived tables, common table expressions, views, and functions.

As the T-SQL statements get more complex, so the plans that the optimizer creates can get bigger, and more time-consuming to decipher. However, just as a large T-SQL statement can be broken down into a series of simple steps, large execution plans are simply extensions of the same simple plans we have already examined, just with more, and different, operators.

Again, please bear in mind that the plans you see, if you follow along, may vary slightly from what's shown in the text, due to different service pack levels, hot-fixes, differences in the `AdventureWorks` database, its statistics, and data.

Stored Procedures

The best place to get started is with stored procedures, which may comprise a single query, or a whole series of queries. In the latter case, you will see multiple execution plans, but the way you tackle each of these plans is no different than any other execution plan.

Listing 7-1 shows a `TaxRateByState` stored procedure, the intent of which is to return information on tax rates that are less than a defined value, in this case 7.5. This is a typical example of a procedure that was probably built up over time, by someone who is not an expert at T-SQL. It involves a series of steps to pull together some data, manipulate that data, then return a result set. There are circumstances where this approach is justified, but others where it is not the optimal solution.

```
CREATE OR ALTER PROCEDURE Sales.TaxRateByState @CountryRegionCode
NVARCHAR(3)
AS
SET NOCOUNT ON;
CREATE TABLE #TaxRateByState
```

```
(
    SalesTaxRateID INT NOT NULL,
    TaxRateName NVARCHAR(50) COLLATE DATABASE_DEFAULT NOT NULL,
    TaxRate SMALLMONEY NOT NULL,
    TaxType TINYINT NOT NULL,
    StateName NVARCHAR(50) COLLATE DATABASE_DEFAULT NOT NULL
);
INSERT INTO #TaxRateByState
(
    SalesTaxRateID,
    TaxRateName,
    TaxRate,
    TaxType,
    StateName
)
SELECT st.SalesTaxRateID,
       st.Name,
       st.TaxRate,
       st.TaxType,
       sp.Name AS StateName
FROM Sales.SalesTaxRate AS st
    JOIN Person.StateProvince AS sp
        ON st.StateProvinceID = sp.StateProvinceID
WHERE sp.CountryRegionCode = @CountryRegionCode;
DELETE #TaxRateByState
WHERE TaxRate < 7.5;
SELECT soh.SubTotal,
       soh.TaxAmt,
       trbs.TaxRate,
       trbs.TaxRateName
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesTerritory AS st
        ON st.TerritoryID = soh.TerritoryID
    JOIN Person.StateProvince AS sp
        ON sp.TerritoryID = st.TerritoryID
    JOIN #TaxRateByState AS trbs
        ON trbs.StateName = sp.Name;
GO
```

Listing 7-1

It would be possible to write the same logic in just a single query, without the need for a temporary table. However, this is the type of code you encounter in real-life systems, and sometimes you just need to understand the cause of the performance issue, via the plan, and decide on a fix, without necessarily having the time, or even the opportunity, to do a full rewrite.

Also, note that `NVARCHAR(3)` isn't the best data type for use for the `@CountryRegionCode` parameter; `CHAR(3)` would be far more efficient and sensible. However, `NVARCHAR(3)` is the data type used for that column, in the table, so the stored procedure follows suit, to avoid data type conversion issues.

We can execute the stored procedure by passing in a value, as shown in Listing 7-2.

```
EXEC Sales.TaxRateByState @CountryRegionCode = N'US';
GO
```

Listing 7-2

Figure 7-1 shows the resulting actual execution plan, which is a little more complex than ones we've seen previously.

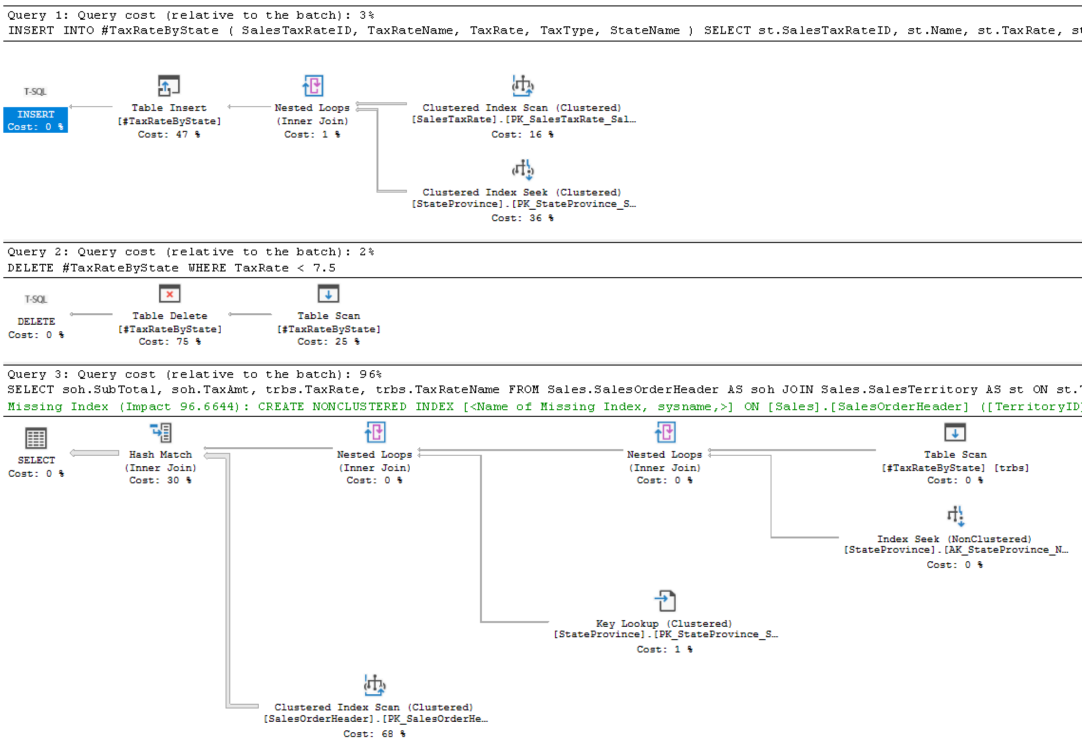


Figure 7-1: Multiple execution plans from a single stored procedure.

An interesting point is that we don't have a stored procedure in sight. Instead, the optimizer treats the T-SQL within the stored procedure in the same way as if we had written and run the `SELECT` statement, through the query window.

The more statements get added to a given stored procedure, the more execution plans you'll see. In the event of some type of looping query, you can see hundreds of execution plans. Capturing all the execution plans in such cases can cause performance problems with SSMS. If you are dealing with that situation, your approach should be to use an estimated plan where possible. If you must see an actual plan, then capture plans for individual statements using a filtered Extended Event session, or use `SET STATISTICS XML ON` and `OFF` statements, if you can modify the code.

The stored procedure in Listing 7-1 has five statements but we see only three execution plans in Figure 7-1. The Data Definition Language (DDL) statement to create the temporary table, `#TaxRateByState`, doesn't get an execution plan. A DDL statement can only be resolved one way, so they do not go through optimization, therefore there is no execution plan. We also don't see a plan for the `SET NOCOUNT` statement. An estimated plan will show a T-SQL operator for these statements, but not any kind of fuller execution plan.

Just as when we execute a batch containing two or more queries, for a stored procedure containing two or more statements, the execution plan shows the estimated cost of each query, relative to the batch. These values appear as the **Query cost (relative to the batch)**, at the head of each plan, and we can use them to identify the plan that needs the most attention, for performance tuning. As always, though, treat these estimated costs with caution, and only use them if there is no large disparity between the estimated and actual row counts.

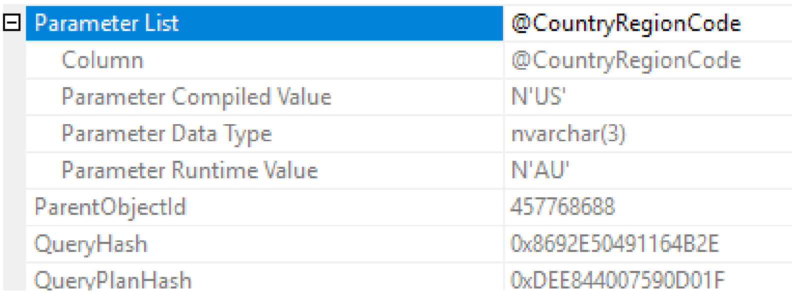
Query 1 accounts for an estimated 3% of the total cost, and it's the plan for populating the temporary table with tax rate information for each state in the supplied country, in this case the USA. We won't explore the plan in detail, but it's worth taking a peek at the properties of the `INSERT` operator.

⊟ Parameter List	@CountryRegionCode
Column	@CountryRegionCode
Parameter Compiled Value	N'US'
Parameter Data Type	nvarchar(3)
Parameter Runtime Value	N'US'
ParentObjectId	457768688
QueryHash	0x8692E50491164B2E
QueryPlanHash	0xDEE844007590D01F

Figure 7-2: Properties of the INSERT operator showing the Parameter List.

The interesting value properties here are in the **Parameter List**, which contains the **Parameter Compiled Value**, the parameter value that the optimizer used to compile the plan for the stored procedure. Below it is the **Parameter Runtime Value**, showing the value when this query was called.

When we run the batch in Listing 7-2, to execute the stored procedure, SQL Server first compiles the batch only, and sets the value of the `@CountryRegionCode` to `N'US'`. It then runs the `EXEC` command, and checks in the plan cache to see if there is a plan to execute the stored procedure. In this case there isn't, so it will then invoke the compiler again to create a plan for the procedure. At this point, the optimizer can "sniff" the parameter value, and generate a plan, using statistics for that value. If we execute the stored procedure again with a different parameter value, this time there will be a plan the optimizer can reuse, and we see a different runtime value but the same compiled value.



⊟ Parameter List	@CountryRegionCode
Column	@CountryRegionCode
Parameter Compiled Value	N'US'
Parameter Data Type	nvarchar(3)
Parameter Runtime Value	N'AU'
ParentObjectId	457768688
QueryHash	0x8692E50491164B2E
QueryPlanHash	0xDEE844007590D01F

Figure 7-3: Properties of the SELECT operator with changes to parameters.

This is only significant if the compiled value returned a row count that was very "atypical" compared to most values used to execute the procedure. The section on *Indexes and selectivity,* in Chapter 8, provides more information about parameter sniffing, and compiled values, so we won't go into further details here.

Query 2 is the plan to delete rows that fall below our tax rate threshold value, which in this case leaves only 5 rows in the temporary table.

Query 3 joins to our temporary table, and several others, to return our results. This query looks to be the place to start our serious investigation, since the optimizer thinks it accounts for the majority (96%) of the cost for executing the stored procedure, as shown in Figure 7-4.

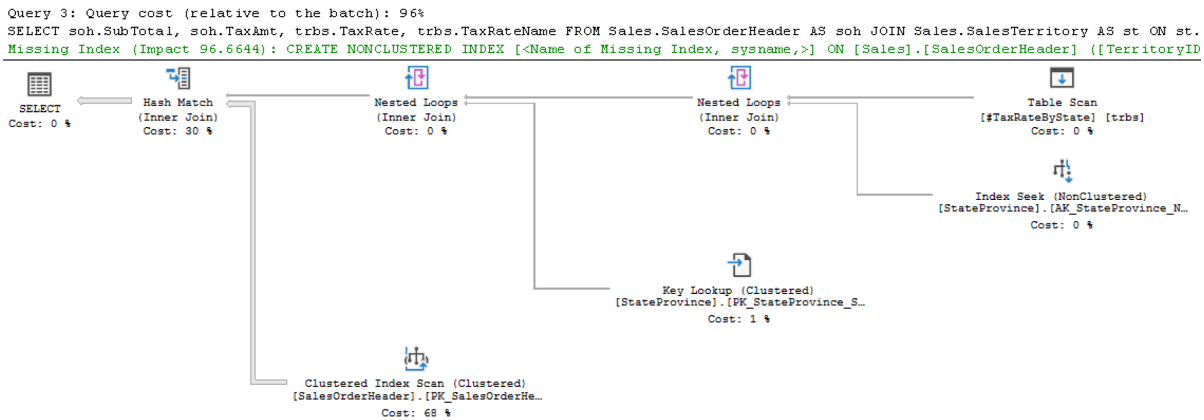


Figure 7-4: The execution plan for Query 3, 96% of the estimated cost of the batch.

Visually it's not a terribly complex plan, but there is a lot going on. Starting on the right, we have a **Nested Loops** join operator where the outer input is a scan of the temporary table, which returns 5 rows. This will incur 5 executions of the inner input, an **Index Seek** against the `StateProvince` table. The output of this **Nested Loops** join operator is the outer input from another **Nested Loops** join, so we get 5 executions on the inner input, a **Key Lookup** on the clustered index of the `StateProvince` table to retrieve the values not stored in the nonclustered index, in this case, the `TerritoryID` values.

The output of the second **Nested Loops** join is the Build input for a **Hash Match** join operator, where the Probe input is a **Clustered Index Seek** against the `SalesOrderHeader` table.

The **Hash Match** operator reads the Build input, hashes the join column (in this case `TerritoryID`), and stores the column values, and their hashes, in a hash table in memory. It then reads the rows in the Probe input one row at a time, in this case 31465 rows, and for each row, it produces a hash value for the `TerritoryID` column that it can compare to the hashes in the hash table, looking for matching values, and starts retuning the matching rows (23752 in total).

As you can see, execution plans for stored procedures are not special, and are not different from other execution plans. You just need to identify the plan, or plans, within that are causing the issue, and assess possible fixes.

Subqueries

A common and useful, but occasionally problematic, approach to querying data is to select information from other tables within the query, but not as part of a JOIN statement. Instead, we embed a SELECT statement within another SELECT, INSERT, UPDATE, or DELETE statement. We can use a subquery in any part of the query where expressions are allowed, but you'll most commonly see them in the WHERE, SELECT and FROM clauses.

Listing 7-3 illustrates a correlated subquery that accesses the Production.ProductionListPriceHistory table. This table maintains a history of prices for each product, and the date ranges for which a given price was valid. It's quite common to see subqueries used like this, for tables that hold "versioned" data. In this case, we use it to ensure we only see the most recent "version" of the list price for each product.

However, for reasons that we'll discuss as we examine the plan, it's not necessarily the optimal solution.

```
SELECT p.Name,
       p.ProductNumber,
       ph.ListPrice
FROM Production.Product AS p
    INNER JOIN Production.ProductListPriceHistory AS ph
        ON p.ProductID = ph.ProductID
           AND ph.StartDate = (   SELECT TOP (1)
                                         ph2.StartDate
                                  FROM Production.
ProductListPriceHistory AS ph2
                                  WHERE ph2.ProductID = p.ProductID
                                  ORDER BY ph2.StartDate DESC);
```

Listing 7-3

Notice that the subquery references the ProductID values from the outer query so, for each row from the outer query, that row's ProductID value is plugged into the subquery and compared with the ProductID value of the ProductListPriceHistory table. As a result, the subquery is executed once for each row returned by the outer query. The TOP (1) clause, with the ORDER BY, ensures that, in each case, the subquery only returns the most recent row (showing the current list price). Depending on the query, sometimes the optimizer can figure out a more efficient way to achieve the desired results. As we'll see, this is not one of those situations.

Figure 7-5 shows the actual execution plan.

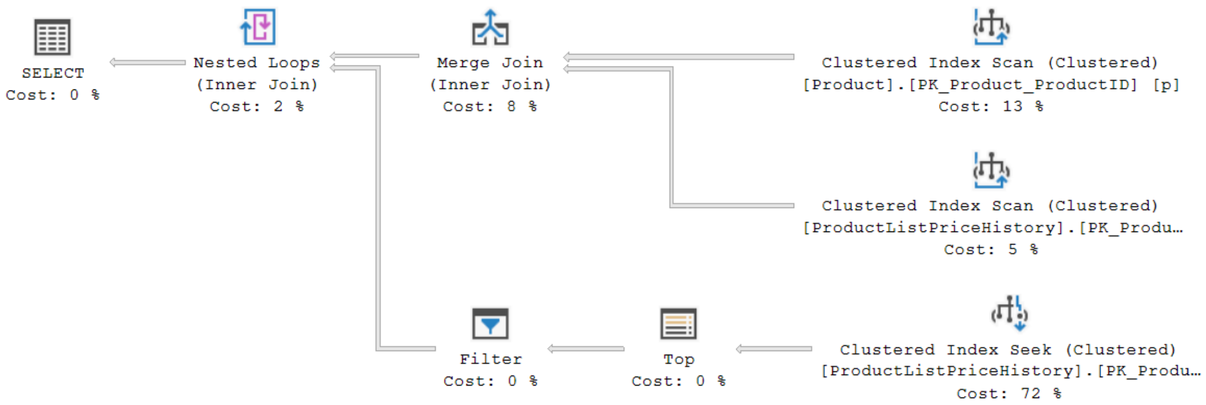
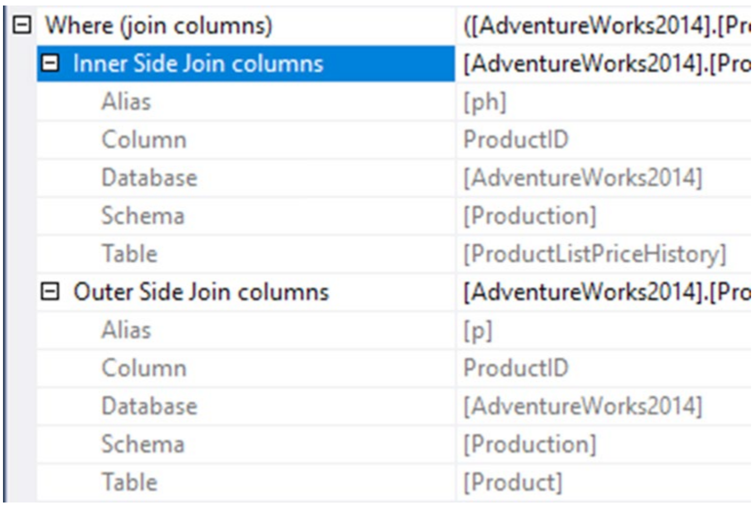


Figure 7-5: Execution plan for a subquery.

Reading the plan from right to left, we see two **Clustered Index Scans,** one on `Production.Product` and one on `Production.ProductListPriceHistory`. These two data streams are combined using the **Merge Join** operator, using `ProductID` as the join column; you can see this in the **Where (join columns)** property in the **Merge Join** operator.



⊟ Where (join columns)	([AdventureWorks2014].[Pr
⊟ Inner Side Join columns	[AdventureWorks2014].[Pro
Alias	[ph]
Column	ProductID
Database	[AdventureWorks2014]
Schema	[Production]
Table	[ProductListPriceHistory]
⊟ Outer Side Join columns	[AdventureWorks2014].[Pro
Alias	[p]
Column	ProductID
Database	[AdventureWorks2014]
Schema	[Production]
Table	[Product]

Figure 7-6: Merge Join columns defined.

Since the **Merge Join** requires that both data inputs are ordered on the join key, in this case the `ProductID`, you'll see that the **Ordered** property is set to True for each of the scans.

This means that the execution engine will use the **Ordered** retrieval method to fulfill them (see Chapter 5), and the data will be retrieved in the logical index order, in each case. In this example, both clustered indexes are ordered by `ProductID`.

Number of Executions	1
Object	[AdventureWorks2014].[Production].[Product].[PK_Product_ProductID] [p]
Ordered	True
Output List	[AdventureWorks2014].[Production].[Product].ProductID, [AdventureWorks
Parallel	False
Physical Operation	Clustered Index Scan
Scan Direction	FORWARD

Figure 7-7: Clustered Index Scan showing that it is Ordered.

So, the **Merge Join** simply takes the data from two inputs and uses the fact that the data in each input is ordered on the join column to merge them, joining rows based on the matching values. You can refer to Chapter 4 for further details on how various flavors of **Merge Join** work.

There are 395 merged rows, which are the 395 rows with list price entries. Incidentally, this is clearly an atypical data distribution, since there are 504 products in the `Products` table, and you'd generally expect there to be one or more price list entries for each product. In any event, these rows form the outer input for a **Nested Loops** join operator, which implies that the inner input will be executed 395 times. If you check the **Outer References** property of the **Nested Loops**, you'll see that values from the `ProductID` and `StartDate` column are being pushed down to the inner input.

The clustered index on the `ProductListPriceHistory` table is on (`ProductID`, `StartDate`) and for each execution, we're looking for rows matching the `ProductID` value pushed down from the outer input. However, the **TOP** operator ensures that it only reads the row with the most recent `StartDate` (remember that execution order is left to right). The **Filter** operator will either pass on or reject that single row, depending on whether there is a match on `StartDate` (the other pushed-down column value). Figure 7-8 shows the expanded Predicate property value for the **Filter** operator.

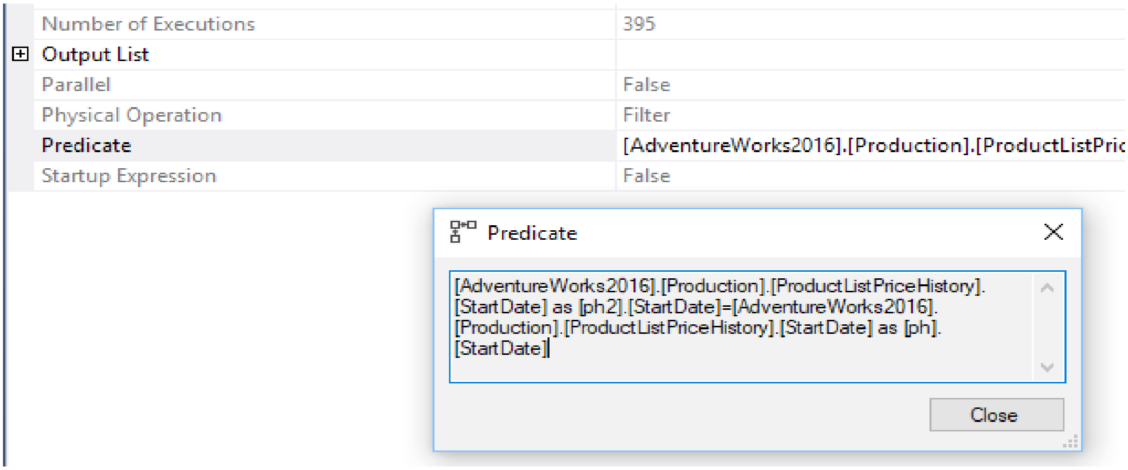


Figure 7-8: Details on the Predicate property.

A couple of other points to note here. Firstly, the **Filter** is executed 395 times (as are its child operators). It returns the most recent row for each of the 293 distinct `ProductID` values; you can see from the **Output List** that it does not return any data, just an empty row shell for each row that passes its filter criteria. The row itself is empty because the only thing that **Nested Loops** needs to make its decision is the presence or absence of a row. Finally, notice that the **Startup Expression** is False in this case, meaning the child operators will be called for every execution. If you were to see **Startup Expression Predicate**, the child operators would only be called for rows that met that Predicate condition.

Hopefully, it's clear that the fundamental problem with this plan is the number of executions of the inner input of the **Nested Loops** join. Imagine some different numbers: let's say we have 200 products and an average of 15 prices per product in the `ProductListPrice-History` history. The **Merge Join** will produce 3000 rows, so the outer input of the **Nested Loops** operator has 3000 rows, and the inner input then executes 3000 times, reading the same 200 rows repeatedly. That would cause a high number of logical reads; the optimizer is likely to choose a different plan under those conditions, if it can find one.

There are many ways you could consider trying to optimize this query and I can't cover them all here. One option would be to replace the `SELECT TOP(N)`...`ORDER BY` logic with `SELECT MAX(ph2.StartDate)`.... If you were to try this, you'd see a change from a **Nested Loops** join to two **Merge Joins** and an improvement in performance. Try it out and read through the plan. Another option is to use a derived table instead of a subquery and we'll see how that works in the next section.

Derived Tables Using APPLY

One of the ways that we can access data through T-SQL is via a **derived table**. If you are unfamiliar with them, think of a derived table as a virtual table created on the fly from within a SELECT statement.

You create derived tables by writing a subquery within a set of parentheses in the FROM clause of an outer query. Once you apply an alias, this subquery is treated as a table by the T-SQL code. Prior to SQL Server 2005, any derived table had to be fully independent of the main query. However, SQL Server 2005 introduced the APPLY operator, which allows us to create a correlation between the main query and the derived table. The APPLY operator will evaluate the subquery (or Table Valued Function) once for every row produced by the part of the FROM clause to the left of the APPLY clause. This is the logical definition; the optimizer is, of course, free to find a different, faster implementation, if it can.

There are two forms of the APPLY operator, CROSS APPLY and OUTER APPLY. The former combines each row from the left input with each row returned from the right input. The latter does the same, but also retains the row from the left input if nothing is returned from the right input, using NULL values for columns originating from the right input. If you are unfamiliar with the **Apply** operator, check out http://bit.ly/1FFmldl (it's an MSDN entry for SQL Server 2008R2, but it's still correct).

In my own code, one place where I've come to use derived tables frequently is when dealing with data that changes over time, for which I should maintain history. This query approach, shown in Listing 7-4, is an alternative to the subquery we saw in the Listing 7-3. It produces the same results as Listing 7-3, but uses the APPLY operator. The big difference is that the data becomes available to the rest of the query, when the subquery is in the FROM, making it a derived table. For a subquery used anywhere else in the query, its result is only available in the location where it is specified.

```
SELECT p.Name,
       p.ProductNumber,
       ph.ListPrice
FROM Production.Product p
    CROSS APPLY
      (
         SELECT TOP (1)
             ph2.ProductID,
             ph2.ListPrice
```

```
            FROM Production.ProductListPriceHistory ph2
            WHERE ph2.ProductID = p.ProductID
            ORDER BY ph2.StartDate DESC
        ) ph;
```

Listing 7-4

The introduction of the APPLY operator changes the execution plan substantially, as shown in Figure 7-9.

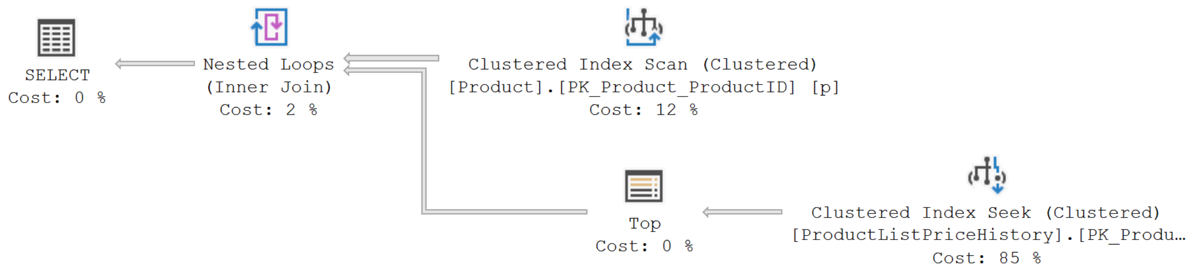


Figure 7-9: Execution plan for the APPLY command.

In this plan, we see that the outer input to the **Nested Loops** operator is a **Clustered Index Scan** of the Products tables, which produces 504 rows. This implies that the inner input will be executed 504 times. The values of the ProductID column are pushed down as Outer References, and used to seek matching rows in the ProductListPriceHistory table, and the TOP operator again ensures that each seek operation returns only the row with the most recent list price.

So, which method of writing this query do you think is the most efficient? One way to find out is to capture and compare performance metrics for each query run (duration, number of logical reads performed, and so on).

As discussed in Chapter 2, the lowest-impact way to do this is using Extended Events. Also, when you do go to measure performance (duration), it's a very good idea to stop capturing the execution plans because that introduces substantial observer effect. Figure 7-10 shows the results, captured using the Extended Events session provided in Listing 2-6.

batch_text	duration	logical_reads	cpu_time	row_count
SELECT p.Name, p.ProductNumber, ph.List...	28758	811	0	293
SELECT p.Name, p.ProductNumber, ph.List...	177413	1024	0	293

Figure 7-10: Performance results for the APPLY command.

Although both queries returned identical result sets, the subquery in the ON clause (Listing 7-3) uses fewer logical reads (811) compared to the query using `APPLY` and a derived table (Listing 7-4), which caused 1024 logical reads.

The simple explanation for the difference is that in the correlated subquery the expensive inner input of the **Nested Loops** join is executed 395 times (once per list price), and in the derived table query it's executed once per product (504 times). As noted earlier, we're dealing with a rather strange data distribution in this case, where 211 products have no list price and the remaining 293 have one or more list prices. With a more typical data distribution, consisting of multiple list prices for all, or most, products, we could easily have expected the derived table version to outperform the subquery.

Things get more interesting if we add the `WHERE` clause in Listing 7-5 to the outer query of each of the previous listings.

```
WHERE p.ProductID = 839
```

Listing 7-5

When we rerun Listing 7-3 with the added `WHERE` clause, we get the plan shown in Figure 7-11.

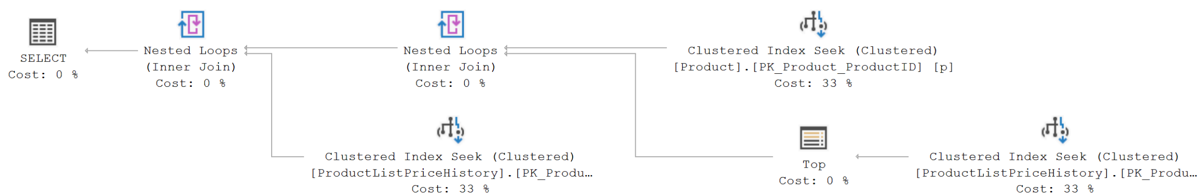


Figure 7-11: New execution plan after adding a WHERE clause.

The **Filter** operator is gone but, more interestingly, the optimizer has changed the order of evaluation; the `TOP` operator now appears in the part of the plan to resolve the outer query where, before, it was in the part of the plan to resolve the subquery. First, it finds the single requested row from the `Product` table and then immediately evaluates the subquery to find the most recent `StartDate` for that `ProductID`. If you check the properties of the right-most **Clustered Index Seek** on `ProductListPriceHistory`, you'll see that it references the **ph2** alias, which tells us it's evaluating the subquery.

The next inner join to `ProductListPriceHistory` is on both `ProductID` and `StartDate`, with `StartDate` being pushed down from the outer input (see the **Outer References** property of the **Nested Loops** join). Also, if you check out the **Seek Predicates** property of the **Clustered Index Seek** on the left, which displays each of the predicates used to define the rows that need to be read, it references both `ProductID` and `StartDate`.

The end result is that, instead of **Index Scans,** and the inefficiencies caused by executing the inner input of a **Nested Loops** join hundreds of times, we now have three **Clustered Index Seek** operations, with an equal estimated cost distribution, and two **Nested Loops** joins. The **Merge Join** we saw in Figure 7-5 was appropriate when we were dealing with scans of the data, but was not used, nor applicable, when the introduction of the `WHERE` clause reduced the data set. The inner input of each **Nested Loops** join is executed only once, since the `WHERE` clause means the outer input produces only a single row.

If we add the `WHERE` clause to Listing 7-4 (`APPLY` and a derived table), we see the plan shown in Figure 7-12.

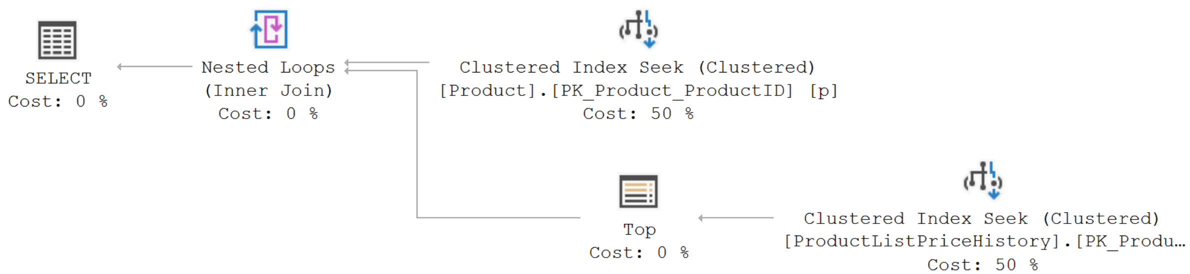


Figure 7-12: How the WHERE clause changes the APPLY plan.

This plan is almost identical to the one seen in Figure 7-9, with the only change being that the **Clustered Index Scan** has changed to a **Clustered Index Seek**. This change was possible because the inclusion of the `WHERE` clause allows the optimizer to take advantage of the clustered index to identify the row needed, rather than having to scan through them all to find the correct row to return.

Let's compare the I/O statistics for each of the queries:

batch_text	duration	logical_reads	cpu_time	row_count
SELECT p.Name, p.ProductNumber, ph.List...	125	6	0	1
SELECT p.Name, p.ProductNumber, ph.List...	121	4	0	1

Figure 7-13: Performance metrics after adding a WHERE clause.

Now, with the addition of a WHERE clause, the derived query is more efficient, with only 4 logical reads versus the sub-select query with 6 logical reads, and a marginal increase in speed. If you run the query frequently, you'll find that the APPLY query is consistently faster. If we increase the data volumes, it's very likely that you'll see the APPLY operator perform even better than the other method.

With the WHERE clause in place, the subquery became relatively costlier to maintain when compared to the speed provided by APPLY. Understanding the execution plan makes a real difference in deciding which T-SQL constructs to apply to your own code. Just remember that you should use the best possible representative data on your tests, in order to get behaviors and performance similar to your production environment. Also remember that, as data changes, so the distribution of that data may change, which can result in differences in execution plans and differences in performance. If your data is modified frequently, you may have to reevaluate queries on a regular basis.

Common Table Expressions

SQL Server 2005 introduced the Common Table Expression (CTE), a T-SQL construct with behavior that appears similar to derived tables. A CTE is a "temporary result set" that exists only within the scope of a single SQL statement. It allows access to functionality within a single SQL statement that was previously only available through the use of functions, temporary tables, cursors, and so on. Unlike a derived table, a CTE can be self-referential and referenced repeatedly within a single query. Also unlike a derived table, a CTE cannot be correlated, even when used with APPLY. For more details on CTEs, check out this article on Simple Talk: http://bit.ly/1NCr8k0.

Despite the description of a CTE as a temporary result set, don't assume that the CTE is processed in a separate manner from the rest of the T-SQL. Fundamentally, this is still a derived table, just like the other examples we've already seen. The primary difference will be when the CTE is self-referencing. A recursive CTE always uses two (or, rarely, more) queries, combined with UNION ALL. The first query, known as the "anchor member," can be executed on its own to produce a result. The second query, the "recursive member," references the CTE itself. It uses the data coming from the anchor member to produce more rows, but then recursively continues to produce even more data using the rows it produces itself. This is the logical definition; we will see how it executes shortly.

The built-in stored procedure, dbo.uspGetEmployeeManagers, in AdventureWorks, uses a CTE called EMP_cte in a classic recursive exercise, listing employees and their managers.

```
CREATE OR ALTER PROCEDURE dbo.uspGetEmployeeManagers
    @BusinessEntityID INT
AS
BEGIN
    SET NOCOUNT ON;
    -- Use recursive query to list out all Employees required for a
Manager
    WITH EMP_cte(BusinessEntityID, OrganizationNode, FirstName,
LastName, JobTitle,
         RecursionLevel) -- CTE name and columns
    AS (
        SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName,
p.LastName,
               e.JobTitle, 0 -- Get the initial Employee
        FROM HumanResources.Employee e
            INNER JOIN Person.Person AS p
            ON p.BusinessEntityID = e.BusinessEntityID
        WHERE e.BusinessEntityID = @BusinessEntityID
        UNION ALL
        SELECT e.BusinessEntityID, e.OrganizationNode, p.FirstName,
p.LastName,
               e.JobTitle, RecursionLevel + 1 -- Join recursive
member to anchor
                                                     -- and to the next
recursive member
        FROM HumanResources.Employee e
            INNER JOIN EMP_cte
            ON e.OrganizationNode = EMP_cte.OrganizationNode.
GetAncestor(1)
            INNER JOIN Person.Person p
            ON p.BusinessEntityID = e.BusinessEntityID
    )
```

```
    -- Join back to Employee to return the manager name
    SELECT EMP_cte.RecursionLevel, EMP_cte.BusinessEntityID, EMP_
cte.FirstName,
           EMP_cte.LastName, EMP_cte.OrganizationNode.ToString()
            AS OrganizationNode, p.FirstName AS 'ManagerFirstName',
            p.LastName AS 'ManagerLastName'  -- Outer select from
the CTE
    FROM EMP_cte
        INNER JOIN HumanResources.Employee e
        ON EMP_cte.OrganizationNode.GetAncestor(1) =
e.OrganizationNode
        INNER JOIN Person.Person p
        ON p.BusinessEntityID = e.BusinessEntityID
    ORDER BY RecursionLevel, EMP_cte.OrganizationNode.ToString()
    OPTION (MAXRECURSION 25)
END;
GO
```

Listing 7-6

You can see the anchor member, the first query in the UNION ALL within the CTE, which will return data based on the BusinessEntityID value that gets passed to it as a parameter. It's commented in the code as -- Get the initial Employee. The recursion then occurs in the second query within the UNION ALL. It's commented as -- Join recursive member to anchor and the next recursive member. It uses the function, GetAncestor, to retrieve additional data based on that defined within the anchor member.

Let's execute this procedure and capture the actual plan.

```
EXEC dbo.uspGetEmployeeManagers
    @BusinessEntityID = 9;
```

Listing 7-7

As Figure 7-14 shows, the execution plan is reasonably complex and will be impossible to read as is within this book.

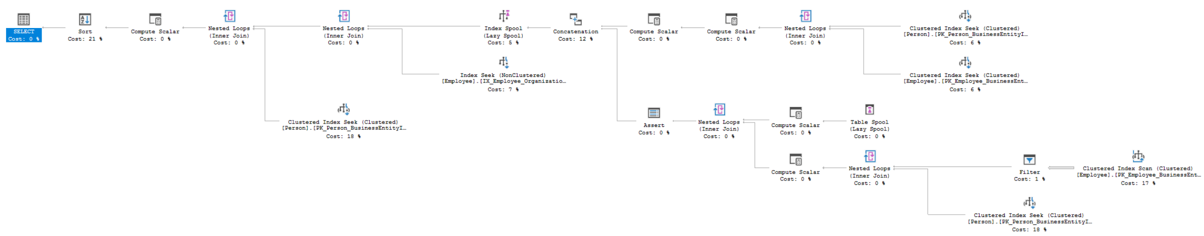


Figure 7-14: Full recursive execution plan from a CTE.

However, our hard work in previous chapters is now paying off. There aren't any operators in this plan you've not seen before, so even though it's a big plan, with patience it should be relatively easy to understand. Let's break down the plan into sections, starting with the top right section, shown in Figure 7-15.

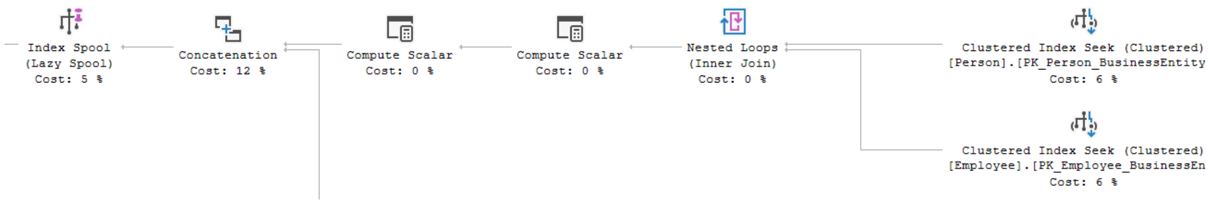


Figure 7-15: Portion of the CTE execution plan showing initial data access.

We're going to read this section of the plan from left to right (logical call order), starting with **Index Spool** operator, because this operator, in conjunction with a **Table Spool** operator that we'll encounter shortly, essentially marks the start of the recursion process, in the CTE. As discussed in Chapter 5, a **Spool** operator uses a temporary worktable to store data that may need to be used multiple times, or reused, within an execution plan. The recursive nature of the query above requires that SQL Server store the data as it recursively builds the result set. This **Index Spool** is a **Lazy Spool**, a streaming operator that requests a row from its child operator, stores it, and then passes it on immediately to its parent, the one preceding it logically passing control back to that parent.

In this case, the **Index Spool** operator has a **Node ID** value of 4, and it's storing the results from a **Concatenation** operator, which resolves the `UNION ALL` operation seen in Listing 7-6. As discussed in Chapter 4, this operator simply processes each of its inputs in order, from top to bottom, and concatenates them. A **Concatenation** operator will always have two or more inputs. It calls the top input, passing rows retrieved to its parent, until it has received all rows. After that it moves on to the second input, repeating the same process.

In this case, the top input collects the data for the "anchor member" of the CTE. It performs a **Nested Loops** join of the data from two **Clustered Index Seeks** against `HumanResources.Employee` and `Person.Person`. This produces a single row (for the employee with `BusinessEntityID` of 9). We then have two **Compute Scalar** operators, each of which returns an expression, both of which are set to zero. One is for the recursion level, and the other for the derived column, called `RecursionLevel`, in the CTE.

After all rows from the top input are processed, the **Concatenation** operator switches to its second input and never returns to the first input. Figure 7-16 displays the bottom input to the **Concatenation** operator, which resolves the recursive member.

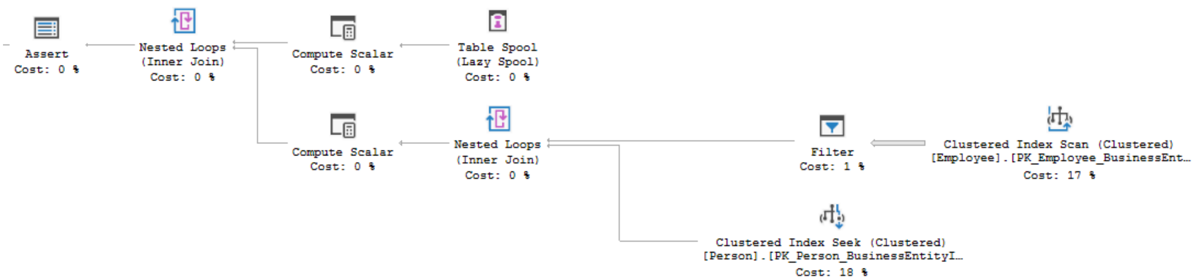


Figure 7-16: Portion of the CTE execution plan showing use of Table Spool.

This is where things get interesting. This section of the plan finds each of the managers (direct manager, manager's manager and so on). SQL Server implements the recursion method via the **Table Spool** operator, combined with the **Index Spool** in the top input. The **Primary Node ID** for the **Table Spool** is 4, indicating that it consumes the data previously loaded into the **Index Spool** operator. You can see this in Figure 7-17, along with some other property values for the **Table Spool**.

Logical Operation	Lazy Spool
Node ID	14
Number of Executions	1
⊞ Output List	Expr1022, Recr1004, Recr1005, Recr1006, Recr1007, Recr1008, Rec
Parallel	False
Physical Operation	Table Spool
Primary Node ID	4
With Stack	True

Figure 7-17: Table spool properties showing the Primary Node ID and With Stack.

The With Stack property, set to True, as shown in Figure 7-17 is a necessary part of the recursive query. Storing data as a stack means that new data is always added at the top and the data is always read from the top. After being read, the data is removed. When you see a With Stack property set to True, the behavior of the Index Spool is changed to that of a "stack." This is crucial for driving the recursive evaluation of the CTE. As the recursive member executes, the **Table Spool** reads and removes the anchor row from the spool. The rest of this plan fragment then finds the anchor value's manager. The manager is stored in the spool by the **Index Spool** operator (**NodeID** 4), and that row is then read and removed when the **Table Spool** is ready to request the next row. From there, the recursion continues. The job of the **Assert** operator, over on the left-hand side of Figure 7-16 is to verify the `MAXRECURSION(25)` in the query, aborting execution when that level is exceeded.

So, the **Table Spool** (**Node ID** 14) produces a copy of the data stored by the **Index Spool** operator (**Node ID** 4). When the operator is first called, it will produce a copy of the anchor row, and then whatever is stored later, on subsequent calls. The **Table Spool** operator loops through the rows from the **Index Spool**, and joins the data to data from the tables defined in the second part of the `UNION ALL` definition, within the CTE.

The **Table Spool** returns four rows. The **Compute Scalar** operator, next to the **Table Spool**, is used to calculate the current recursion level by adding one to the value. This data stream forms the outer input to a **Nested Loops** join, which joins to the `Employee` table on a built-in function, `GetAncestor`, which in turn joins to the `Person` table on `BusinessEntityID`. The inner input performs the **Nested Loops** join between the `Employee` and `Person` tables. Figure 7-18 shows the properties of the **Clustered Index Scan** of the `Employee` table, where you can see the number of times this scan was executed.

The **Estimated Number of Executions** is 4, and **Estimated Number of rows** is 290 and so four times 290 is 1160 rows in total, which matches exactly the **Actual Number of Rows** value.

Actual Number of Rows	1160
Actual Rebinds	0
Actual Rewinds	0
Defined Values	[AdventureWorks2014]
Description	Scanning a clustered ir
Estimated CPU Cost	0.0003975
Estimated Execution Mode	Row
Estimated I/O Cost	0.0076479
Estimated Number of Executio	4
Estimated Number of Rows	290
Estimated Operator Cost	0.0092379 (17%)
Estimated Rebinds	0
Estimated Rewinds	3
Estimated Row Size	69 B
Estimated Subtree Cost	0.0092379
Forced Index	False
ForceScan	False
Logical Operation	Clustered Index Scan
Node ID	27
NoExpandHint	False
Number of Executions	4

Figure 7-18: Clustered Index Scan of the Employee table.

We then have a **Filter** operator. The optimizer has decided to do a full scan of the `Employee` table and then, in this **Filter** operator, compare the `OrganizationNode` of each row to the `GetAncestor` of the row from the CTE, and keep only the rows that match. For the first three rows processed (the one from the anchor member and the first two returned from the recursive member), this filter keeps only one row, the employee's direct manager. The fourth row processed is the CEO, who has no manager, so the filter now returns no row at all and the recursion stops. Hence the right-most section of the plan returns four rows in total: one from the anchor member and three from the recursive member, listing the employee's managers all the way to the CEO.

So, we have one row from the anchor and three rows from the recursive member giving the four rows in total emerging from the **Concatenation** operation, but only three rows are returned in the final results. After the recursion process is finished, we do one more inner join of each row returned, to their manager, at which point, the last row returned from the recursive CTE, the CEO, fails to find data for their `ManagerFirstName` and `ManagerLastName` columns and so the row is lost.

Views

A view is essentially just a "stored query." In other words, a logical way of representing data as if it were in a table, without creating a new table. The various uses of views are well documented (preventing certain columns from being selected, reducing complexity for end-users, and so on). Here, we will just focus on what happens within an execution plan when working with a view.

One note of caution regarding views. Views are not tables, as will become clear when we examine their execution plans, but they look like tables, and so there is an inclination to use them as tables, joining one view to the next, or nesting multiple views inside of other views. This leads to horrible query performance, because the complexity of the execution plans overwhelms the optimizer. This bad practice, a common code smell, should be avoided.

Standard views

The view, `Sales.vIndividualCustomer`, provides a summary of customer data, displaying information such as their name, email address, physical address, and demographic information. A very simple query to get a specific customer would look something like Listing 7-13. While using `SELECT *` is not the best way to write queries, in this case I'm doing it to illustrate what happens when a query is run against a view and all the data referenced by that view are retrieved.

```
SELECT  *
FROM    Sales.vIndividualCustomer
WHERE   BusinessEntityId = 8743;
```

Listing 7-8

Figure 7-19 shows the resulting graphical execution plan.

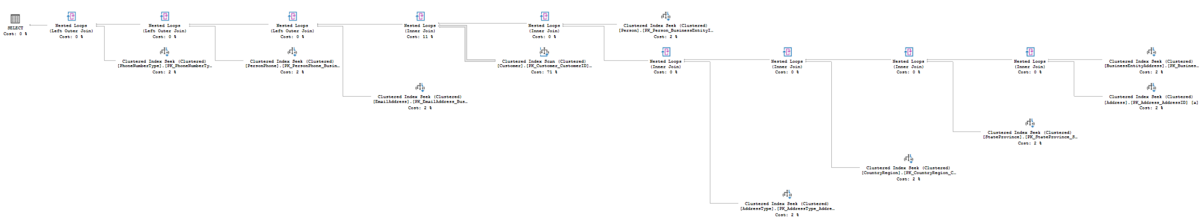


Figure 7-19: The full plan of the query against a view.

This is another plan that is very difficult to read on the printed page, so Figure 7-20 shows an exploded view of just the five operators on the right-hand side of the plan.

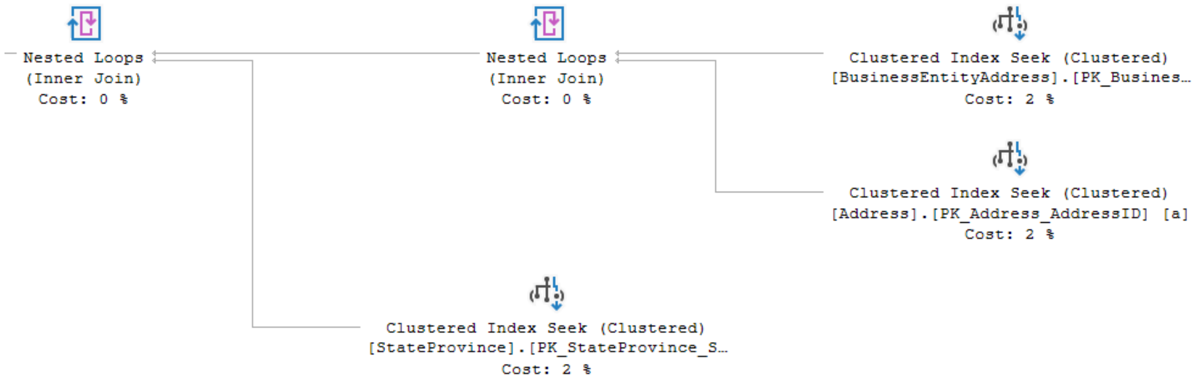


Figure 7-20: Subsection of the plan showing standard operators.

What happened to the view, `vIndividualCustomer`, which we referenced in this query? Remember that, while SQL Server treats views similarly to tables, a view is just a stored query definition, which sits on top of the base tables (and possibly other views) from which they derive. During query binding (see Chapter 1), the algebrizer "expands" the view, i.e. replaces it with its definition, and then the result is passed to the optimizer. So the optimizer never even sees the view, only the query that defines it. The optimizer simply optimizes access to the eight tables and the seven joins defined within this view.

In short, while a view can make coding easier, it doesn't in any way change the need of the query optimizer to perform the actions defined within the view. This is an important point to keep in mind, since developers frequently use views to mask the complexity of a query.

What happens if we change the query to use a list of columns in the `SELECT` statement?

```
SELECT ic.BusinessEntityID,
       ic.Title,
       ic.LastName,
       ic.FirstName
FROM Sales.vIndividualCustomer AS ic
WHERE BusinessEntityID = 8743;
```

Listing 7-9

This results in quite a different execution plan, shown in Figure 7-21.

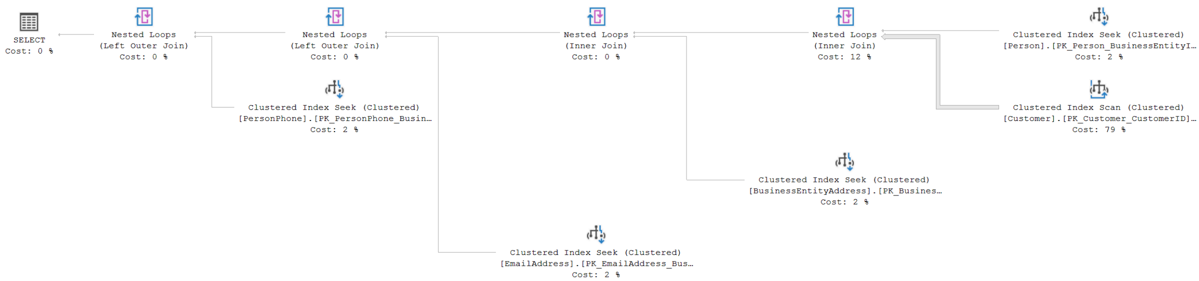


Figure 7-21: Same view, but a different execution plan.

Notice just how different the execution plan shape and the number of operators are in Figure 7-21, when compared to Figure 7-19, even though we are querying the same view. This is because a step in the process called "simplification" will eliminate tables that are not needed to satisfy the query. In this case, without referencing all the columns, the optimizer can eliminate them from the plan.

It is worth noting that you could probably write a query that references even fewer of the tables. The simplification process won't always catch every possible excess table. For example, the `EmailAddress` table is still being referenced within the plan.

Indexed views

An indexed view, also called a "materialized" view or even a "persisted" view, is essentially a "view plus a clustered index." A clustered index stores the column data as well as the index data, so creating a clustered index on a view results in what is effectively a new physical table in the database. Indexed views can often speed up the performance of many queries, as the data is directly stored in the indexed view, negating the need to join and look up the data from multiple tables each time the query is run.

Creating an indexed view is, to say the least, a costly operation. Fortunately, it's also a one-time operation, which we can schedule when our server is less busy. Indexed views also come with an internal maintenance cost for SQL Server. If the base tables in the indexed view are relatively static, there is little overhead associated with maintaining indexed views. However, it's quite different if the base tables are subject to frequent modification. For example, if one of the underlying tables is subject to a hundred `INSERT` statements a minute, then each `INSERT` will have to be updated in the indexed view. As a DBA, you must decide if the overhead associated with the internal maintenance of an indexed view is worth the gains provided by creating the indexed view in the first place.

Queries that contain aggregates are good candidates for indexed views because the creation of the aggregates only has to occur once, when the index is created, and the aggregated results can be returned with a simple SELECT query, rather than having the added overhead of running the aggregates through a GROUP BY each time the query runs. There is also a substantial I/O saving when aggregation is done within an indexed view.

For example, one of the indexed views supplied with AdventureWorks2014 is vStateProvinceCountryRegion. You can see the complete query in Listing 7-10. There I drop and recreate the view, and then create the clustered index that makes it an indexed view.

```
DROP VIEW Person.vStateProvinceCountryRegion;
GO
CREATE OR ALTER VIEW Person.vStateProvinceCountryRegion
WITH SCHEMABINDING
AS
SELECT sp.StateProvinceID,
       sp.StateProvinceCode,
       sp.IsOnlyStateProvinceFlag,
       sp.Name AS StateProvinceName,
       sp.TerritoryID,
       cr.CountryRegionCode,
       cr.Name AS CountryRegionName
FROM Person.StateProvince sp
    INNER JOIN Person.CountryRegion cr
        ON sp.CountryRegionCode = cr.CountryRegionCode;
GO
CREATE UNIQUE CLUSTERED INDEX IX_vStateProvinceCountryRegion
ON Person.vStateProvinceCountryRegion
(
    StateProvinceID ASC,
    CountryRegionCode ASC
);
GO
```

Listing 7-10

If I run the query in Listing 7-10 and try to capture the execution plan, there is one; even though each of these statements is a DDL statement. This is because, in order to satisfy the final statement which creates the index on the view, the query that defines the view must be run. Figure 7-22 shows the execution plan for this query.

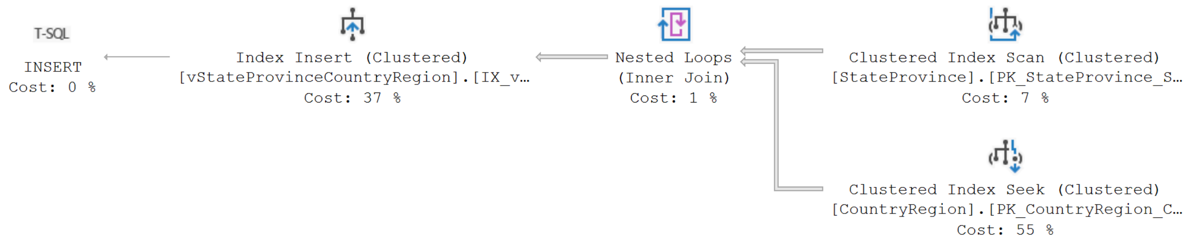


Figure 7-22: Execution plan for the creation of an Indexed View.

This looks like some of the plans we saw in Chapter 6. We're selecting from the two tables defined in the view and a **Nested Loops** operator is used to put the data together before supplying it to an **Index Insert (Clustered)** operator. This is the process of creating the indexed view.

We can run a query from the view and see the execution plan.

```
SELECT vspcr.StateProvinceCode,
       vspcr.IsOnlyStateProvinceFlag,
       vspcr.CountryRegionName
FROM Person.vStateProvinceCountryRegion AS vspcr ;
```

Listing 7-11

The execution plan that results from this query reflects, not a regular index, but an indexed view, assuming you're using either Enterprise or Developer Edition. If you're using Standard Edition, prior to SQL Server 2016 SP1, or Express Edition, where neither do indexed view matching by default, you'll need to use the `WITH NOEXPAND` hint to see the same behavior.

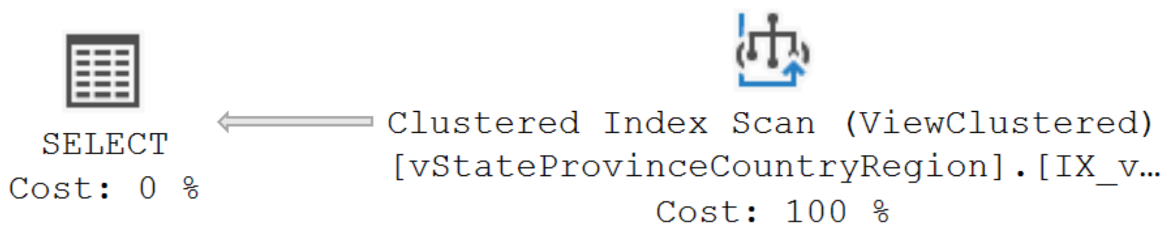


Figure 7-23: Execution plan against an indexed view.

From our previous experience with execution plans containing views, you might have expected to see two tables and the join in the execution plan. Instead, we see a single **Clustered Index Scan** operation. Rather than execute each step of the view, the optimizer went straight to the clustered index that makes this an indexed view.

Since the indexes that define an indexed view are available to the optimizer, they are also available to queries that don't even refer to the view. For example, the query in Listing 7-12 gives a very similar execution plan to the one shown in Figure 7-23, because the optimizer recognizes the index as the best way to access the data (again this assumes the use of Enterprise or Developer Edition).

```
SELECT   sp.Name AS StateProvinceName,
         cr.Name AS CountryRegionName
FROM     Person.StateProvince sp
INNER JOIN Person.CountryRegion cr
         ON sp.CountryRegionCode = cr.CountryRegionCode;
```

Listing 7-12

However, as the query grows in complexity, this behavior is neither automatic nor guaranteed. For example, consider the query in Listing 7-13.

```
SELECT   a.City,
         v.StateProvinceName,
         v.CountryRegionName
FROM     Person.Address a
JOIN     Person.vStateProvinceCountryRegion v
         ON a.StateProvinceID = v.StateProvinceID
WHERE    a.AddressID = 22701;
```

Listing 7-13

If you expected to see a join between the indexed view and the `Person.Address` table, you would be disappointed.

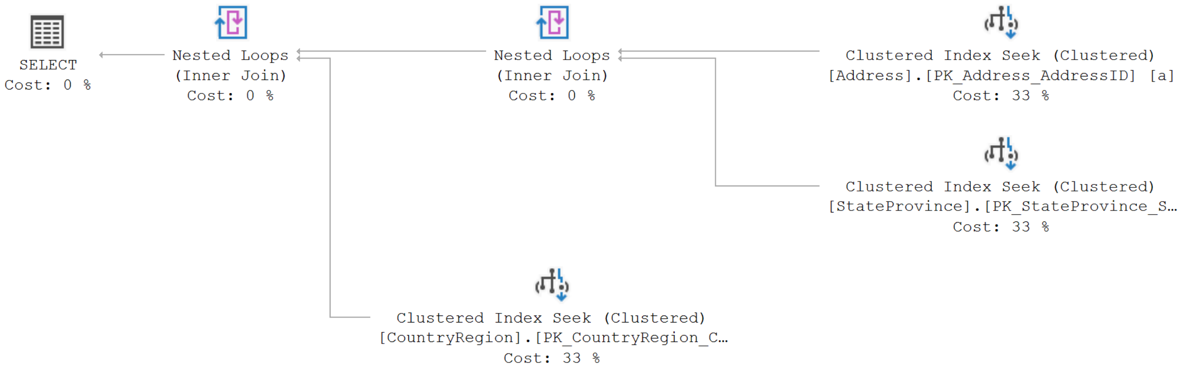


Figure 7-24: Execution plan of the expanded indexed view.

Instead of using the clustered index that supports the materialized view, as we saw in Figure 7-23, the algebrizer performs the same type of index expansion as it did when presented with a regular view. The query that defines the view is fully resolved, substituting the tables that make it up instead of using the clustered index provided with the view.

The algebrizer in SQL Server will expand views every time. The optimizer has a process that determines that direct table access will be less costly than using the indexed view. Again, there is a way around this with the `NOEXPAND` hint, covered in Chapter 10.

Functions

There are two kinds of user-defined functions within SQL Server:

- **Scalar functions** – return a single value.
- **Table valued functions** – return a table.

Their behavior within execution plans can be somewhat deceptive.

Scalar functions

Let's start with a scalar function that is part of `AdventureWorks2014`, called `dbo.ufnGetStock`. Listing 7-14 shows the query.

```
CREATE OR ALTER FUNCTION dbo.ufnGetStock(@ProductID int)
RETURNS int
AS
```

```
-- Returns the stock level for the product.
BEGIN
    DECLARE @ret int;
    SELECT @ret = SUM(p.Quantity)
    FROM Production.ProductInventory p
    WHERE p.ProductID = @ProductID
        AND p.LocationID = '6'; -- Only look at inventory in the
misc storage
    IF (@ret IS NULL)
        SET @ret = 0
    RETURN @ret
END;
GO
```

Listing 7-14

We can see the function in action with a query looking for stock levels of only black products.

```
SELECT p.Name,
       dbo.ufnGetStock(p.ProductID) AS StockLevel
FROM Production.Product AS p
WHERE p.Color = 'Black';
```

Listing 7-15

If we run the query and capture the actual execution plan, there's not much to it, as shown in Figure 7-25.

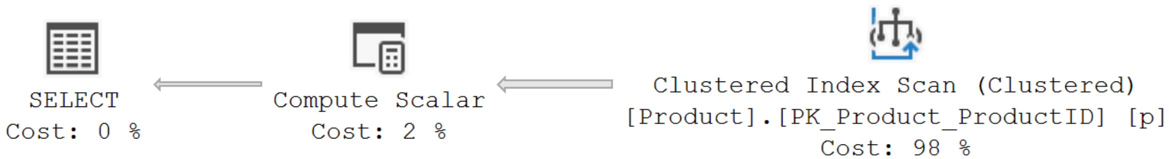


Figure 7-25: Introducing the scalar function in a plan.

The **Clustered Index Scan** makes sense because there is no index that can support the WHERE clause on the Color column. So, the entire index must be scanned and then the Predicate applied to return only the 93 rows with a Color of black. To see what the **Compute Scalar** operator is up to, we must go into the properties and look at the **Defined Values** to see the calculation.

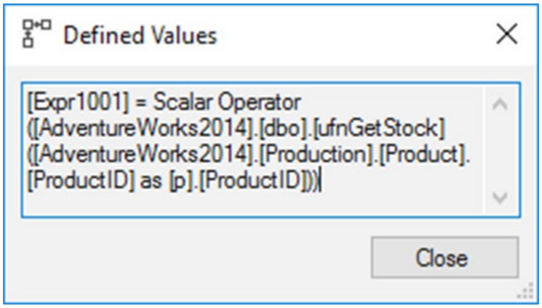


Figure 7-26: Function calculation within the Compute Scalar operator.

As you can see, that's the execution of the scalar function. So that's pretty much all we need to look at, right? Not exactly. This UDF is accessing data through the query in Listing 7-14. That access cannot be seen anywhere in Figure 7-27. Instead of capturing an actual plan for Listing 7-15, if we capture an estimated plan, different information is surfaced.

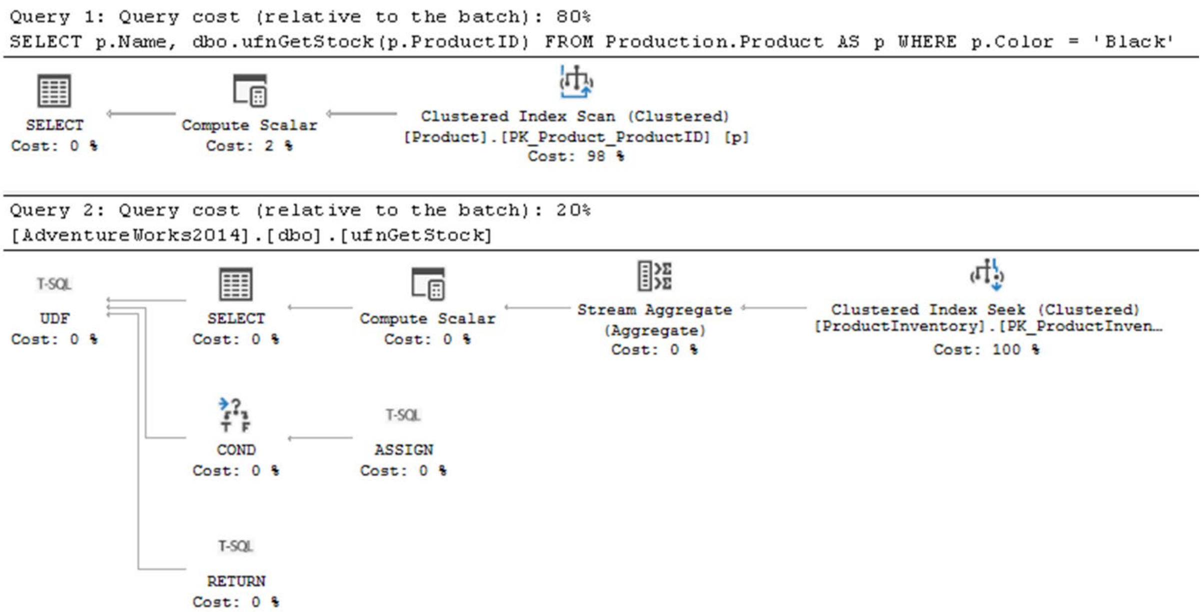


Figure 7-27: Estimated plan showing full extent of plans needed for function.

Instead of a single execution plan, there are two. The second plan represents the scalar function. This is a hidden cost behind the **Compute Scalar** operator in the plan shown in Figure 7-25. The plan in Figure 7-27 introduces a lot of functionality.

Reading the plan from the left, the first operator we see is a T-SQL operator labeled as **UDF**, representing the user-defined function. There are no properties of note beyond an estimated cost. Going to the right we see three sub-branches (in effect, three plans), one for each of the statements in the UDF.

The first operator we encounter on the top branch is a **SELECT**. We will see one **SELECT** operator for each `SELECT` statement in a UDF. If we had a UDF with three `SELECT` statements, they will each have their values for Plan Hash, Optimization Level, and so on. This sub-branch is used for the query that computes `@Ret`, by aggregating data from `ProductInventory`. It uses a **Clustered Index Seek** to find matching data, and then a **Stream Aggregate** and **Compute Scalar** to produce the desired result. We've seen all these operators before, throughout the book, but this is the first time they've been hidden away!

In the second sub-branch, we see a **COND** operator. This is a Conditional, in this case performing the `NULL` check you can see within the function in Listing 7-14. If `@ret` is `NULL`, the **COND** operator calls the **ASSIGN** operator, which sets `@ret` to 0.

The final sub-branch shows the `RETURN` operator, which represents the `RETURN` statement from Listing 7-14.

As the plan in Figure 7-30 shows, there is more going on behind the scenes with a scalar function than is immediately apparent. This is especially true of a scalar function that is accessing data. If we were to capture `STATISTICS IO` results for executing Listing 7-17, it would report only 15 logical reads to return the 93 rows. Unfortunately, as noted in Chapter 2, it fails to count additional I/O resulting from calls to the user-defined function. The user-defined function is called from the **Compute Scalar** of the "main" plan, once for each of the 93 rows returned from the `Product` table. This means that each of the steps in the execution plan for the UDF itself is executed 93 times.

If you capture the performance metrics, using our Extended Events session (Listing 2.6), you will see that in fact it performs 211 logical reads, and that the query references not 93 but 365 rows. Each of the 93 executions of the UDF does an Index Seek to find all rows for one specific `ProductID`, processing 365 rows in total, but performing a lot of unnecessary I/O to return them. If we had avoided the UDF and just written a join between the two tables, chances are that the same number of rows would have been written, but using far fewer logical reads.

Table valued functions

User-defined table valued functions come in two different varieties with two different modes of behavior. First is the inline Table Valued Function (iTVF). These are sometimes referred to as parameterized views because of how they operate. The second is the multi-statement table valued function. These allow for complex queries consisting of multiple statements. These functions are each exposed in execution plans in different ways.

Listing 7-16 shows how we could rewrite the function from Listing 7-14 as in iTVF.

```
CREATE FUNCTION dbo.GetStock (@ProductID INT)
RETURNS TABLE
AS
RETURN
(
    SELECT SUM(pi.Quantity) AS QuantitySum
    FROM Production.ProductInventory AS pi
    WHERE pi.ProductID = @ProductID
          AND pi.LocationID = '6'
);
```

Listing 7-16

To use the function in a query we'll have to modify Listing 7-15 slightly.

```
SELECT p.Name,
       gs.QuantitySum
FROM Production.Product AS p
CROSS APPLY dbo.GetStock(p.ProductID) AS gs
WHERE p.Color = 'Black';
```

Listing 7-17

The resulting actual execution plan is completely different from what we saw for the scalar function.

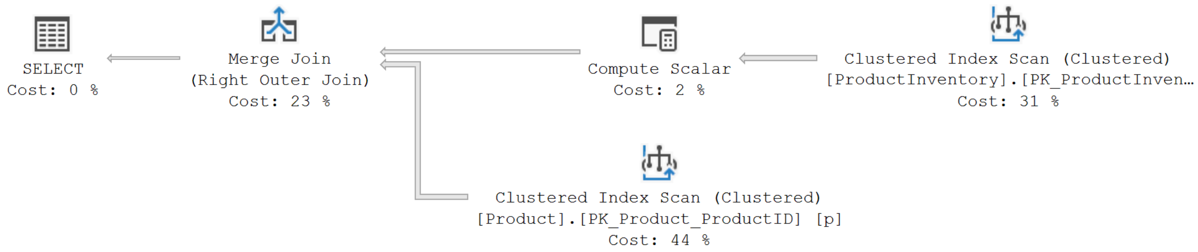


Figure 7-28: Plan for a Table Valued Function.

The most immediate question you might have is: why is there no aggregation operator in the plan? How does the `SUM` get computed? The answer is that the optimizer uses information in the query used to define the iTVF (the filter on `LocationID`) along with metadata (the fact that there is a unique index on `ProductID`) to conclude that *per product*, there will be at most one row with `LocationID` = 6. Since there can never be more than 1 row per product, aggregating by product is unnecessary.

Reading from the left we see a **Merge Join** operator, which is performing a right **Outer Join** between the `ProductInventory` and `Product` tables. We see a **Clustered Index Scan** on the `ProductInventory` table, with a pushed-down **Predicate** on `LocationID`. The **Compute Scalar** is an implicit convert of the `Quantity` value to an integer. `Quantity` is defined as `SMALLINT`, but the `SUM` aggregation automatically converts that to `INT`. Without the aggregation in the plan, the conversion must be done in a **Compute Scalar**. This data is merged with the data from a **Clustered Index Scan** of `Product`.

Unlike the scalar function earlier, the inline function is fully exposed in a single execution plan. An estimated plan of Listing 7-17 would be the same as Figure 7-28, minus the runtime values. There are no hidden costs, and rows required to satisfy the query are accurately reflected within the execution plan.

A multi-statement table valued UDF behaves completely differently. Listing 7-18 shows how we could rewrite our inline function to be a multi-statement UDF.

```
CREATE FUNCTION dbo.GetStock2 (@ProductID INT)
RETURNS @GetStock TABLE (QuantitySum int NULL)
AS
BEGIN
    INSERT @GetStock
    (
        QuantitySum
    )
    SELECT SUM(pi.Quantity) AS QuantitySum
    FROM Production.ProductInventory AS pi
    WHERE pi.ProductID = @ProductID
          AND pi.LocationID = '6';
    RETURN;
END
```

Listing 7-18

If we modify Listing 7-17 to use this function and then run the query, the execution plan changes as in Figure 7-29.

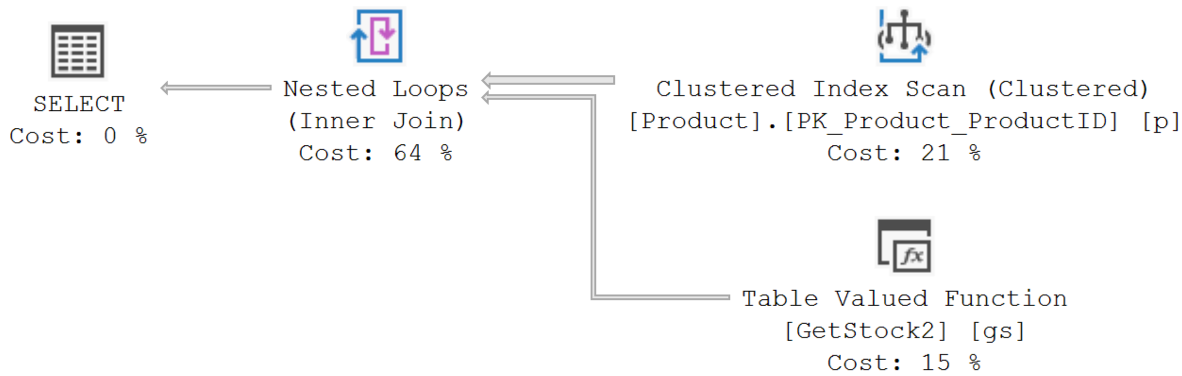


Figure 7-29: Multi-statement table valued function execution plan.

You can easily see that we are once again facing a situation where there is hidden functionality. We have a new operator, **Table Valued Function**, on the inner input of a **Nested Loops** join.

The single most important property value to examine for the **Table Valued Function** operator is the **Estimated Number of Rows**, which is 100.

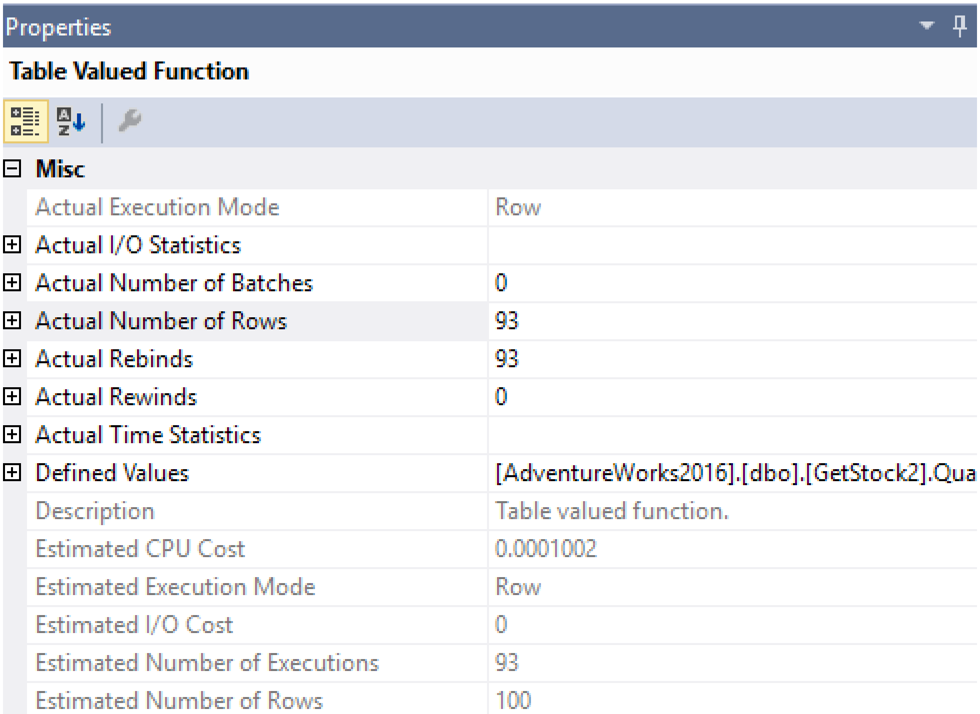


Figure 7-30: Properties of the Table Valued Function operator.

In fact, the estimated rows returned for a multi-statement table valued function will always be 100 rows. The cardinality estimator uses a hard-coded value for table variables. Prior to SQL Server 2014 this value was 1. From SQL Server 2014 onwards, this value is 100. That row count is completely separated from reality.

In this case, an estimated 100 rows returned, per execution, and an estimated 93 executions (once for each row produced by the outer input), giving a total of 9300 rows. In fact, it only returns 1 row per execution, 93 in total.

To see the functionality behind the **Table Valued Function** operator, we must look to the estimated plan again. Figure 7-31 shows the full function.

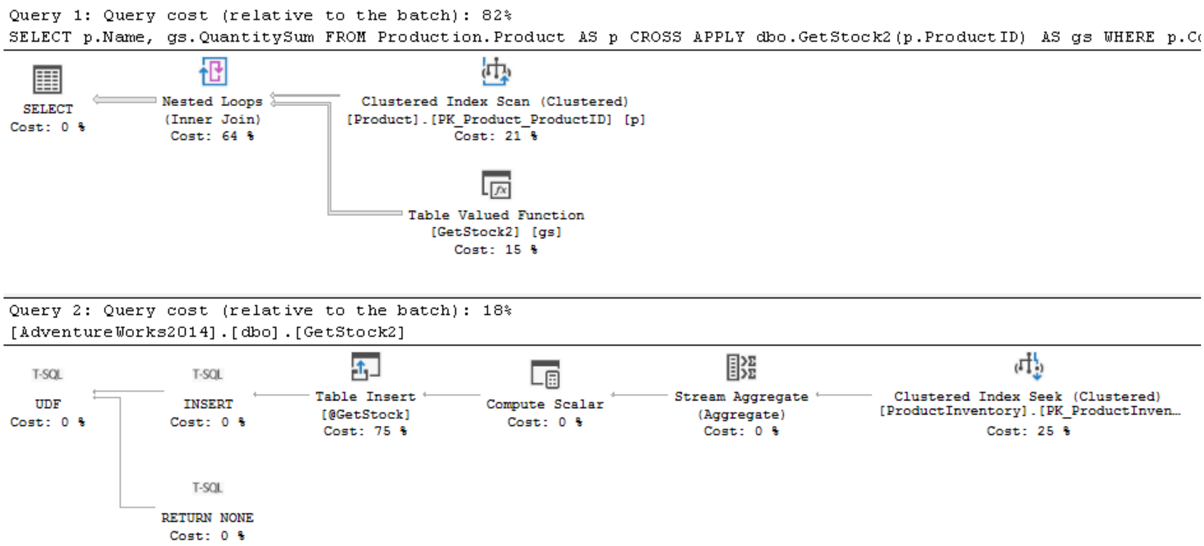


Figure 7-31: Estimated plan showing full functionality of the Table Valued Function.

You can see that, in this situation, the multi-statement function looks very similar to the original scalar function. The one addition is the **Table Insert** operator that's necessary to load the table variable within the function. Once more, this represents a hidden cost to the query. If we look at the I/O from the Extended Events for the `GetStock` function and compare it to `GetStock2` function we see them go from 44 reads to 1141 reads. The optimizer is just not given adequate information to make good choices, when dealing with a multi-statement user-defined function.

Summary

This chapter demonstrated the sort of execution plans that we can expect to see when our code uses stored procedures, views, derived tables, CTEs, and user-defined functions. They are more complex than the ones we've seen in earlier chapters, but all the principles are the same; there is nothing special about larger and more complicated execution plans except that their size and level of complexity requires more time to read them. If you follow the same patterns of using the information in the first operator to understand how the engine is resolving the query, and then reading the properties to understand how the information is flowing between the operators, you'll be fine.

Chapter 8: Examining Index Usage

It's difficult to understate the impact that a carefully selected set of indexes will have on the quality of the plans that the optimizer generates, and the performance of your queries. However, we can't always solve a performance problem just by adding an index. It is entirely possible to have too many indexes, so we must be judicious in their use.

We need to ensure that the indexes we choose to create are well designed and selective for the predicates used by your most important queries. This also means making sure that your statistics accurately reflect the data that is stored within the index.

This chapter will describe how the optimizer uses these statistics to make selectivity and cardinality estimations, and what can go wrong, either because the statistics are unreliable, or because the optimizer used accurate statistics to generate a plan that was good for some execution of a parameterized query, but bad for others.

Finally, we'll examine some of the important execution plan features you'll see for queries that use two relatively new index types, Columnstore indexes and Memory-optimized indexes.

Standard Indexes

For a typical OLTP workload, comprising the sorts of example queries seen throughout this book, our indexing strategy will primarily rely on standard clustered and nonclustered indexes:

- **Clustered indexes** – the primary means of storing and accessing most tables within the standard relational storage of SQL Server.
- **Nonclustered indexes** – a secondary method of accessing data, in support of the clustered index on a table, designed to improve the performance of frequent and expensive queries in the workload.

Generally, if a suitable index is available, then the query optimizer will choose an effective plan that uses it. If there isn't, then you risk poor execution plans and poor query performance.

When a table is altered to add a clustered index, it replaces the heap table with an index that stores all the table's data, ordered such that it is easy to access rows based on the clustering key value, or a range of consecutive key values. Most tables will have a clustered index, plus one or more nonclustered indexes. A nonclustered index is similar in that its intent is to make it easy to access data by certain key values but, instead of storing all data, it stores only the index key values, with a pointer to the location of the full data, usually the values of the clustered index key or, for a heap table, an internal value known as the row identifier. A nonclustered index can also store additional data columns at the leaf level with the use of the `INCLUDE` operator.

An important part of any tuning effort involves choosing the right clustered index, and then a set of supporting nonclustered indexes, for each table in the database. As we've discussed throughout the book, we are not trying to cover every query with an index. Instead, our goal is to create the minimal set of indexes that will be most beneficial to the optimizer in helping it resolve, as cheaply as possible, the most important, expensive and frequent queries in our workload.

How the optimizer selects which indexes to use

We've already seen plenty of examples of the optimizer choosing to use certain indexes to locate and retrieve the data the query needs to read or modify. Sometimes, however, the optimizer will, perplexingly, choose a different plan that ignores what appears to be a useful index. There is always a reason for this, revealed by the execution plan, often by examining the estimated costs for the operators, estimated and actual row counts, as well as other behaviors and properties of each index-reading operator, and their interaction with other operators in the execution plan, as we'll see shortly.

First, we need to recap a little on how the optimizer chooses which indexes to use (it's essentially the same process for any operator).

Estimated costs and statistics

As we discussed way back in Chapter 1, the optimizer will choose the lowest-cost plan, based on **estimated cost** values. It will choose the plan that its calculations suggest will have the lowest total cost, in terms of the sum of the estimated CPU and I/O processing costs. Each operator's estimated cost contributes to the overall estimated cost of the plan.

The accuracy of the optimizer's estimated costs depends largely on the accuracy of its statistical knowledge of the data: its *data about the data*. These statistics, collected automatically for each index, and many columns as well, provide aggregated information to the optimizer, based on a sample of the data. They describe, hopefully accurately, the volume and distribution of all the data in the table.

For example, the statistics used by the optimizer include a **density graph**, which predicts the "uniqueness" of the data in a column (the number of different values present) and a **histogram**, which predicts the number of occurrences of each value. The optimizer needs to know this information accurately, because it is a key factor in its decisions on which indexes to use, and how.

Selectivity and cardinality estimations

The key measure for the optimizer in determining whether to use an index, and how to read that index, is the likely **selectivity** of a query predicate that the index could support. The selectivity of a predicate, for a given index, is the *expected ratio of matching rows*. Count the total number of rows in the table (z), count the number of distinct values (x) for a given column, or combination of columns, across all the rows, and then (x/z) gives the selectivity of the index, for an equality predicate comparing the column (or columns) against unknown values.

A highly selective index will have a low selectivity value. For example, a selectivity of 0.01 (1%) means that the optimizer expects 1% of the total rows in the table to match the predicate. Conversely, the worst possible selectivity is 1.0 (or 100%) meaning that every row will match the predicate condition.

The **cardinality** for a given operator in a plan, shown in the **Estimated Number of Rows** property, is computed based on the selectivity of each predicate in the filter, some other data available from the statistics, and some assumptions about the data in the tables. The nature of calculations varies depending on the operator. For example, for a **Merge Join**, the **Estimated Number of Rows** is based on the estimated cardinalities of the two input streams and some very complex calculations on the histograms of those two input streams (if available).

Indexes and selectivity

Essentially, a query is resolved by a chain of successive operations on the data, as described in its execution plan. Therefore, an indexing strategy that can help the optimizer reduce the amount of data being manipulated, as soon as possible in the chain, is likely to work best.

To do this, we need an index to be selective, for the filtering predicates used by the queries you intend it to help. If an index exists that matches a predicate column used by certain queries in the workload, and if the optimizer gauges that, for a given query, the selectivity of the predicate is sufficiently high, then it will consider the index to be a good candidate to use in the plan. Usually, this means that the estimated cardinality will be low, meaning only a few rows will be accessed, which will lower the overall estimated cost of the operator.

To demonstrate how the optimizer makes decisions on how to read data from tables, we'll create a copy of the `SalesOrderDetail` table, in AdventureWorks. We'll assume that at some point a developer added a couple of nonclustered indexes that he or she thought might help certain queries.

```
DROP TABLE IF EXISTS NewOrders ;
GO
SELECT SalesOrderID,
       SalesOrderDetailID,
       CarrierTrackingNumber,
       OrderQty,
       ProductID,
       SpecialOfferID,
       UnitPrice,
       UnitPriceDiscount,
       LineTotal,
       rowguid,
       ModifiedDate
INTO dbo.NewOrders
FROM Sales.SalesOrderDetail;
GO
ALTER TABLE dbo.NewOrders
ADD  CONSTRAINT PK_NewOrders_SalesOrderID_SalesOrderDetailID
PRIMARY KEY CLUSTERED
(
   SalesOrderID,
   SalesOrderDetailID
);
CREATE NONCLUSTERED INDEX IX_NewOrders_ProductID
ON dbo.NewOrders (ProductID);
GO
CREATE NONCLUSTERED INDEX IX_NewOrders_OrderQty
ON dbo.NewOrders(OrderQty);
GO
```

Listing 8-1

We'll run the following simple query to return order details for a known order quantity (20) and capture the actual execution plan.

```
SELECT OrderQty,
       SalesOrderID,
       SalesOrderDetailID,
       LineTotal
FROM dbo.NewOrders
WHERE OrderQty = 20;
```

Listing 8-2

Figure 8-1 shows the execution plan. We see that the optimizer chose to use an **Index Seek** on our nonclustered index on `OrderQty`, even though this index is not covering for this query. A total of 46 rows are returned from the **Index Seek** and, because the index is not covering, this results in 46 executions of the **Key Lookup**.

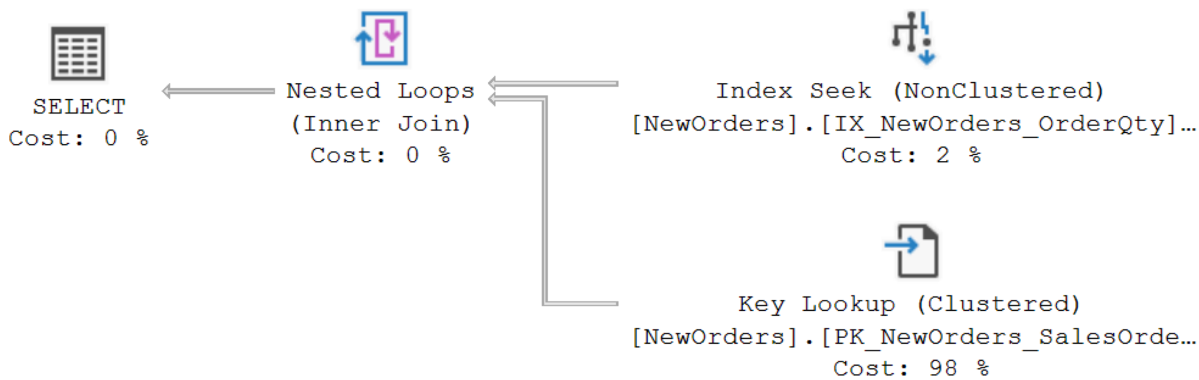


Figure 8-1: The index selection process.

To help us understand the decisions that the optimizer has made, we can look at the statistics for the `IX_NewOrders_OrderQty` index, using the `DBCC SHOW_STATISTICS` command.

```
DBCC SHOW_STATISTICS('dbo.NewOrders',
                     'IX_NewOrders_OrderQty');
```

Listing 8-3

This returns three result sets, the first showing the **header**, with general details about the statistics, the second the **density graph**, and finally the **histogram** with the tabulation of counts for each indexed column value that's sampled in the statistics.

Statistics header

The header displays the name of the index, the number of rows in the table, and the number of rows sampled by the create/update statistics algorithm to generate the statistics, in this case all 12317 rows. It also shows that there are 40 rows, or steps, in this histogram.

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	IX_NewOrders_OrderQty	Feb 15 2018 11:10AM	121317	121317	40	0.5	10	NO	NULL	121317

Figure 8-2: The header information in the statistics for IX_NewOrders_OrderQty.

There are only ever up to 200 data points or steps in the histogram. In this case, there are 40 steps. Since there are 41 distinct values in the `OrderQty` column, that may appear surprising, but this is simply a consequence of how the algorithm for building the histogram works; it simply tries to identify the most "interesting" data points, with a maximum of 200, in a single pass of the data.

Density graph

The density graph provides the optimizer with its estimations of the number of distinct values in a column or index. The lower the density, the higher the "uniqueness," and the more selective is the index. A unique column in a 10000-row table has a density of 1/10000 or 0.0001. An equality predicate on this column has a selectivity of 0.0001 (or 0.01 percent), the exact same number, because they are computed in the same way.

However, density and selectivity aren't the same thing. For example, density is also used to estimate the number of rows after an aggregation operator: if the same 10000-row table has 5 distinct values for `Color`, then the density of `Color` will be 1/5, or 0.2; the estimated number of rows when you group by `Color` is then computed as 1/0.2 which brings us back to 5.

Figure 8-3 shows that the density for the `OrderQty` column is 0.02439024.

	All density	Average Length	Columns
1	0.02439024	2	OrderQty
2	2.18055E-05	6	OrderQty, SalesOrderID
3	8.242868E-06	10	OrderQty, SalesOrderID, SalesOrderDetailID

Figure 8-3: The density graph for IX_NewOrders_OrderQty.

The optimizer can use the density graph to estimate the selectivity of a predicate, for an equality predicate comparing the column (or columns) against unknown values. If a query uses a predicate on `OrderQty` and the optimizer cannot "sniff" the parameter or variable value, it simply takes the density value for the `OrderQty` column, which is 0.02439024, multiplies it by the total number of rows in the table (121317) and estimates a cardinality of 2958.95 rows.

If we're performing an inequality predicate against unknown values, then the optimizer always uses a default estimated selectivity of 30%, and no density is used.

The other rows in the density graph refer to the density for predicates that use a combination of `OrderQty` and the clustered index key column values, also stored in the index. As you can see, for this index the density for a predicate on a combination of `OrderQty` and `SalesOrderID` is about 1000 times less that for `OrderQty` alone, meaning that an equality predicate on this combination of columns is about 1000 times more selective than a predicate on `OrderQty`. This density level makes the index a very attractive option for the optimizer, for an equality predicate on these columns, comparing to unknown values.

The histogram

Often, the optimizer knows the parameter or variable value to which it is comparing, either because it sniffed it, or because we hard-coded it. In such cases, the optimizer uses the histogram to get a better estimate of the cardinality for the predicate.

In Listing 8-2, where we supplied a hard-coded `OrderQty` value of 20 and in the histogram, this value matches exactly one of the ranges defined by the `RANGE_HI_KEY`. The optimizer reads a cardinality value (row count) of 46, from the `EQ_ROWS` column for that row.

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
12	12	0	466	0	1
13	13	0	230	0	1
14	14	0	265	0	1
15	15	0	119	0	1
16	16	0	133	0	1
17	17	0	94	0	1
18	18	0	101	0	1
19	19	0	53	0	1
20	20	0	46	0	1
21	21	0	31	0	1
22	22	0	12	0	1
23	23	0	23	0	1
24	24	0	19	0	1
25	25	0	17	0	1
26	26	0	15	0	1
27	27	0	9	0	1
28	28	0	6	0	1
29	29	0	3	0	1
30	30	0	1	0	1
31	31	0	5	0	1
32	32	0	7	0	1
33	33	0	6	0	1
34	34	0	3	0	1
35	36	2	2	1	2
36	38	0	1	0	1

Figure 8-4: An extract from the histogram for IX_NewOrders_OrderQty.

If there is no exact match, the optimizer uses a slightly different approach to the row count estimates. For example, if we changed the literal value for `OrderQty` to 35, in Listing 8-2, we can see that there is a match for 34 and 36 in the `RANGE_HI_KEY` column, but no match for 35. Since the `RANGE_HI_KEY` defines the top of a range, the value of 35 lies within the range defined by 36, and the optimizer uses the `AVG_RANGE_ROWS` value for that row as the row count estimate, 2 rows. It derives the `AVG_RANGE_ROWS` value simply by dividing `RANGE_ROWS` (the estimated number of rows that make up the range defined by the `RANGE_HI_KEY`) by `DISTINCT_RANGE_ROWS` (number of distinct values within the range). You may see a different row number estimate, depending on your version of SQL Server or AdventureWorks, or on whether you modified your database structures, rebuilt indexes, or updated your statistics.

Armed with its cardinality estimate (46 rows), the optimizer calculates the total estimated cost of performing a seek followed by 46 lookups, and compares it to its alternatives, (in this case simply performing a single scan of the clustered index), and chooses the cheapest option. The higher the estimated row count, the more lookups will need to be performed, and there will be a tipping point where the optimizer decides to simply scan the clustered index.

In this example, the tipping point is somewhere around 400 rows. If you execute Listing 8-2 with a literal value of 11 (estimated 392 rows), we still see the seek/lookup plan, but use a value of 12 (estimated 466 rows) and it tips, and we see the clustered index scan.

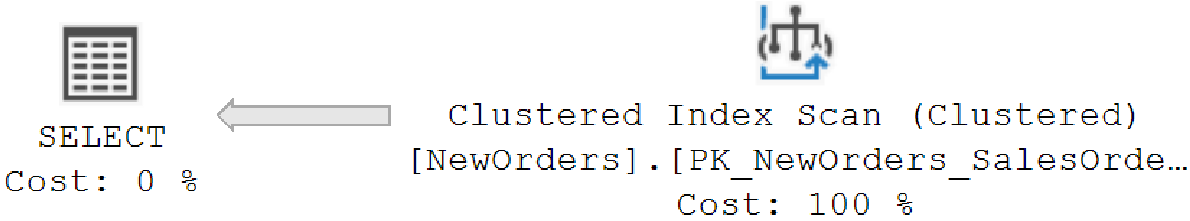


Figure 8-5: A clustered index scan caused by a change in estimated rows.

What if we were to rewrite Listing 8-2 to use a local variable, instead of a hard-coded literal?

```
DECLARE @OrderQuantity SMALLINT
SET @OrderQuantity = 20
SELECT OrderQty,
       SalesOrderID,
       SalesOrderDetailID,
       LineTotal
FROM dbo.NewOrders
WHERE OrderQty = @OrderQuantity;
```

Listing 8-4

When we execute this, we'll see the plan with the clustered index scan, even though in terms of actual number of rows returned, we are below the tipping point. The reason is that the optimizer cannot sniff the value supplied, when we use local variables (unless statement-level recompile takes place because of an `OPTION (RECOMPILE)` hint), and so it simply uses the density graph to estimate a cardinality of 2958.95 rows, as described earlier, which we can confirm from the Properties sheet for the **Clustered Index Scan**. This estimated number of rows is way above the tipping point for the optimizer to choose a scan in preference to the seeks plus lookups.

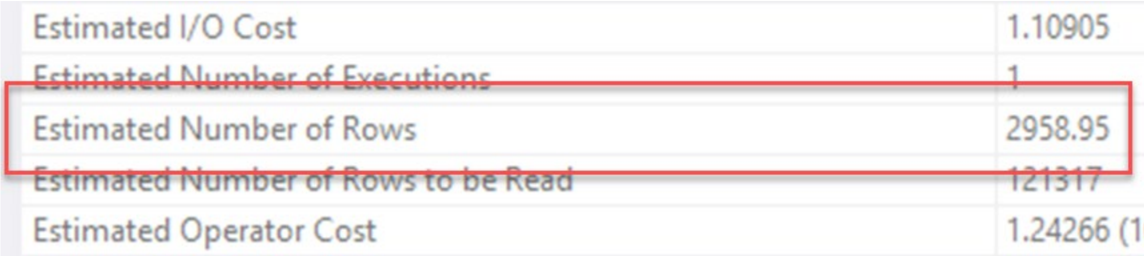


Figure 8-6: Properties showing the Estimated Number of Rows.

If we were to modify the `WHERE` clause in Listing 8-4 to use an inequality search condition, `OrderQty > @OrderQuantity`, then you'll see that the optimizer reverts to using a hard-coded cardinality estimation of 30% of the rows in the table, estimating 36,395.1 rows when only 164 are returned. This will always result in the plan with the scan whereas, for a `OrderQty` value of 20, the optimizer would choose the seek/lookup plan in cases where it knows or can sniff the value, since it can once again use the histogram to get accurate cardinality estimations.

Using covering indexes

In the previous examples, our index on the `OrderQty` column did not cover any of our queries. When the optimizer chose to use the index, the plans incurred the extra cost of performing lookups on the clustered index, to retrieve the column values not contained in the nonclustered index.

As discussed in Chapter 3, we create a covering index either by having all the columns necessary as part of the key of the index, or by using the `INCLUDE` operation to store extra columns at the leaf level of the index so that they're available for use with the index.

A lookup always adds some extra cost, but when the number of rows is small then that extra cost is also small, and the extra cost may be an acceptable tradeoff against the total cost for the entire application of adding a covering index.

Remember that adding an index, however selective, comes at a price during `INSERT`s, `UPDATE`s, `DELETE`s and `MERGE`s as the data within each index is reordered, added, or removed. We need to weigh the importance, frequency of execution, and actual run time of the query, against the overhead caused by adding an extra index, or by adding an extra column to the `INCLUDE` clause of an existing index.

If this were a critical or frequent query, we might consider replacing the existing index with one that included the `LineTotal` column to cover the query, and perhaps other columns, if it meant that the same index would then also cover several other queries in the workload.

What can go wrong?

There are many reasons why the optimizer might be unable to use what looks like a very suitable index, or appears to ignore it, and we can't cover them in this book.

Sometimes, it's a problem with the code. For example, a mismatch between the parameter data type and the column type forces implicit conversion on the indexed column, and this will prevent the optimizer from seeking the index. Sometimes, a query contains logic that defeats accurate estimations. Complex predicates are harder to estimate than simple predicates. Inequality predicates are sometimes harder to estimate than equality predicates and, in cases where the parameter or variable values can't be sniffed, the optimizer simply uses a hard-coded selectivity estimation (30%). Expressions with a column embedded are harder to estimate than expressions where the column is by itself and the expression is on the other side.

Sometimes, the optimizer chooses what appears to be a less ideal index because it is, in fact, cheaper overall, perhaps because that index presents the data in an order that facilitates a merge join or stream aggregate later in the plan, instead of its more expensive counterparts. Or, because it allows the optimizer to observe `ORDER BY` without having to add a **Sort** operator.

We can't cover every case, so in this section we'll focus only on problems that occur when the optimizer's selectivity and cardinality estimations don't match reality. The optimizer thinks an operator will only need to process 10 rows, but it processes 10,000, or vice versa.

If the optimizer cannot accurately estimate how many rows are involved in each operation in the plan, or it reuses a plan with estimated row counts that are no longer valid, then it may ignore even well-constructed and highly selective indexes, or use inappropriate indexes, and therefore create suboptimal execution plans. These problems often manifest in large discrepancies between actual and estimated row counts in the plan, and the potential causes are numerous.

Problems with statistics

Regarding statistics, the optimizer can use a suboptimal plan for several possible reasons:

- **Missing statistics** – no statistics are available on the column used in the predicate, perhaps because certain database options prevent their creation, such as the `AUTO_CREATE_STATISTICS` option being set to `OFF`.
- **Stale statistics** – it had to generate a plan for a query containing a predicate on a column with statistics that have not recently updated, and no longer reflect accurately the true distribution.
- **Reusing a suboptimal cached plan**– the optimizer reused a plan that was good when it was created, but the data volume or distribution has changed significantly since then, and the plan is no longer optimal.
- **Skewed data distribution** – the optimizer had to generate a plan for a query containing a predicate on a column where the data distribution was very non-uniform, making accurate cardinality estimations difficult.

Let's see an example. Listing 8-5 captures an actual execution plan for a simple query against our `NewOrders` table. It then inserts new rows. It only inserts 5% of the total number currently in the table, which is below the threshold required to trigger an automatic statistics update, but it does it in a way designed to skew the data distribution.

Next, it recaptures the plan for the same query. Finally, it manually updates the statistics, and captures the plan a final time. If you're following along, you might also consider creating and starting the Extended Events session I show in Chapter 2 (Listing 2-6), to capture the I/O and timing metrics for each query.

```
SET STATISTICS XML ON;
GO
SELECT OrderQty,
       CarrierTrackingNumber
FROM dbo.NewOrders
WHERE ProductID = 897;
GO
SET STATISTICS XML OFF;
GO
--Modify the data
BEGIN TRAN;
INSERT INTO dbo.NewOrders (SalesOrderID,
                           CarrierTrackingNumber,
                           OrderQty,
                           ProductID,
                           SpecialOfferID,
                           UnitPrice,
                           UnitPriceDiscount,
                           LineTotal,
                           rowguid,
                           ModifiedDate)
SELECT TOP (5) PERCENT
       SalesOrderID,
       CarrierTrackingNumber,
       OrderQty,
       897,
       SpecialOfferID,
       UnitPrice,
       UnitPriceDiscount,
       LineTotal,
       rowguid,
       ModifiedDate
FROM Sales.SalesOrderDetail
ORDER BY SalesOrderID;
GO
SET STATISTICS XML ON;
GO
SELECT OrderQty,
       CarrierTrackingNumber
FROM dbo.NewOrders
WHERE ProductID = 897;
GO
SET STATISTICS XML OFF;
GO
```

```
--Manually update statistics
UPDATE STATISTICS dbo.NewOrders
GO
SET STATISTICS XML ON;
GO
SELECT OrderQty,
       CarrierTrackingNumber
FROM dbo.NewOrders
WHERE ProductID = 897;
GO
SET STATISTICS XML OFF;
GO
ROLLBACK TRAN;
--Manually update statistics
UPDATE STATISTICS dbo.NewOrders;
GO
```

Listing 8-5

By using `SET STATISTICS XML` statements, along with separating the code into batches, we can capture just the execution plans for those specific batches, and omit the other plans such as the one that is generated for the `INSERT` statement. First, here is the plan for the query before inserting the extra rows.

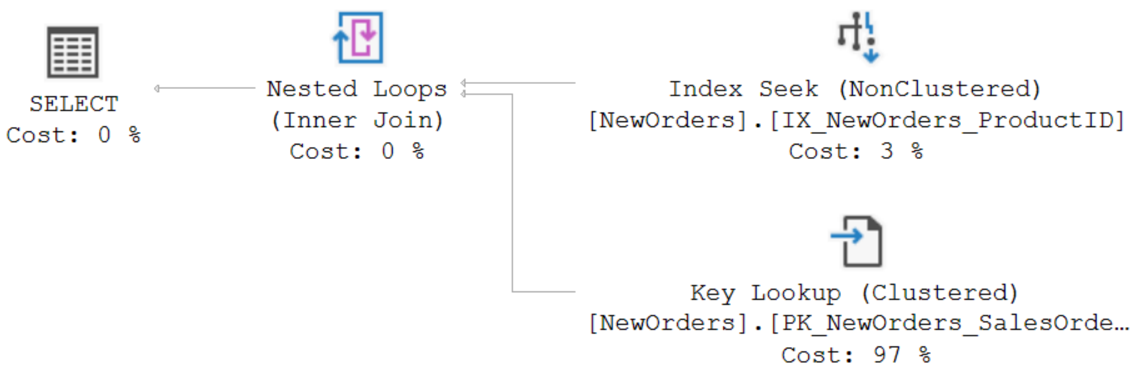


Figure 8-7: The initial execution plan before statistics are updated.

The optimizer chose to seek the nonclustered index on `ProductID`. The index does not cover the query, but it estimates that the seek will return only 50.817 rows. It gets this estimate from the `AVG_RANGE_ROWS` value column of the histogram for the `IX_ProductID_NewOrders` index, as described earlier.

In fact, it returns only two rows, but even so the optimizer estimates that the extra overhead of the **Key Lookup** operator, for around 51 rows, is small enough to prefer this route over scanning the clustered index.

Figure 8-7 shows the plan after we "skewed" the data with our `INSERT` statement.

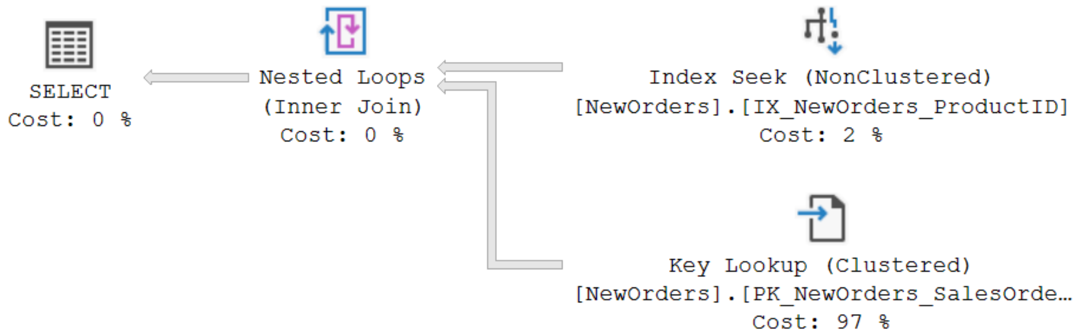


Figure 8-8: Inefficient execution plan for out-of-date statistics.

We see the same plan. The optimizer has simply encountered a query it has seen before, selected the existing plan from the cache and passed it on to the execution engine.

However, now the **Actual Number of Rows** for the **Index Seek** is 6068, so the **Key Lookup** is executed 6068 times. The initial query had 52 logical reads, but the subsequent query had 19385, as measured in Extended Events.

Finally, we update the statistics, so the plan in cache will be invalidated, causing a new one to be compiled. With up-to-date statistics, the plan is now reflected in Figure 8-7.

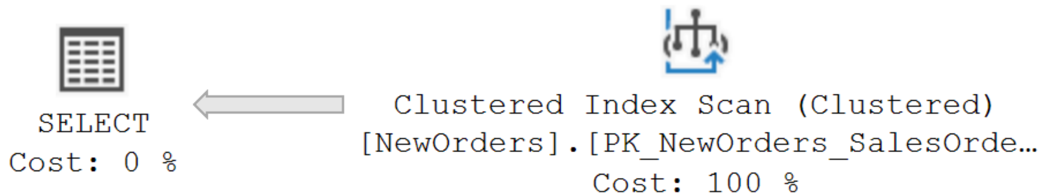


Figure 8-9: Correct execution plan for up-to-date statistics.

This is a good and appropriate strategy for the query on this table, as it is now. Since a large percentage of the table now matches the criteria defined in the `WHERE` clause of Listing 8-2, the **Clustered Index Scan** makes sense. Further, the number of reads has dropped to 1,723 even though the exact same number of rows is being returned.

This example illustrates the importance of statistics in helping the optimizer to make good choices, and how those choices affect the behavior of indexes that we can see within the execution plans generated. Bad statistics will result in bad choices of plan. A discussion on maintaining statistics is outside the scope of this book, but certainly you should always leave `AUTO_UPDATE_STATISTICS` enabled, and possibly consider running `UPDATE STATISTICS` as a scheduled maintenance job for big tables, if required. For data skews that affect important queries, you might consider investigating filtered statistics.

Problems with parameter sniffing

In correctly-parameterized queries, and when we use correctly-written objects such as stored procedures and functions, the optimizer can peek at the value passed to a parameter, and use it to compare to the statistics of the index key (or the column), specifically the histogram. This is known as parameter sniffing and it allows the optimizer to get accurate cardinality estimates, rather than relying on "averages," based on statistical density of the index or column, or on hard-coded estimates (such as 30%).

When SQL Server runs the batch to execute a stored procedure, for example, it first compiles the batch. At this point, it sets the value of any variables, and evaluates any expressions. It then runs the `EXEC` command, checking in the plan cache to see if there is a plan to execute the stored procedure. If there isn't one, it invokes the compiler again to create a plan for the procedure. At this point, the optimizer can "sniff" the parameter value it detected when running the `EXEC` command in the batch.

In some cases, parameter sniffing is unequivocally our friend. For example, let's say we have a million-row **Orders** table that we query using an inequality predicate (such as a date range), and only ever return a small subset of the data, typically results for the last week. Without parameter sniffing, we'll always get a plan generated to accommodate an estimated row count of 300,000 (30% of 1 million), which is likely to be a bad plan, if the queries typically only return tens or hundreds of rows.

In other cases, such as if our queries filter on the `PRIMARY KEY` column, or on a key with an even data distribution, then parameter sniffing is largely irrelevant.

Often, we're somewhere in between, and problematic parameter sniffing occurs when queries filter on keys with uneven data distribution, and the optimizer reuses a cached plan generated for a sniffed input parameter value with an estimated row count that turns out to be atypical of the row counts for subsequent input values.

Stored procedures and parameter sniffing

In Listing 8-6, we simply turn our NewOrders query from Listing 8-2 into a stored procedure but, to keep things interesting, with the slight kink that the @OrderQty parameter is optional.

```
CREATE OR ALTER PROCEDURE dbo.OrdersByQty
    @OrderQty SMALLINT = NULL
AS
SELECT SalesOrderID,
       SalesOrderDetailID,
       OrderQty,
       LineTotal
FROM dbo.NewOrders
WHERE
      (
          OrderQty = @OrderQty
          OR @OrderQty IS NULL
      );
GO
```

Listing 8-6

We already know what if we supply a literal value of OrderQty=20 for the original query, the optimizer will create a plan with the nonclustered index seek and the key lookups (see Figure 8-1). Figure 8-10 shows the actual plan when we execute this procedure supplying an OrderQty value of 20.

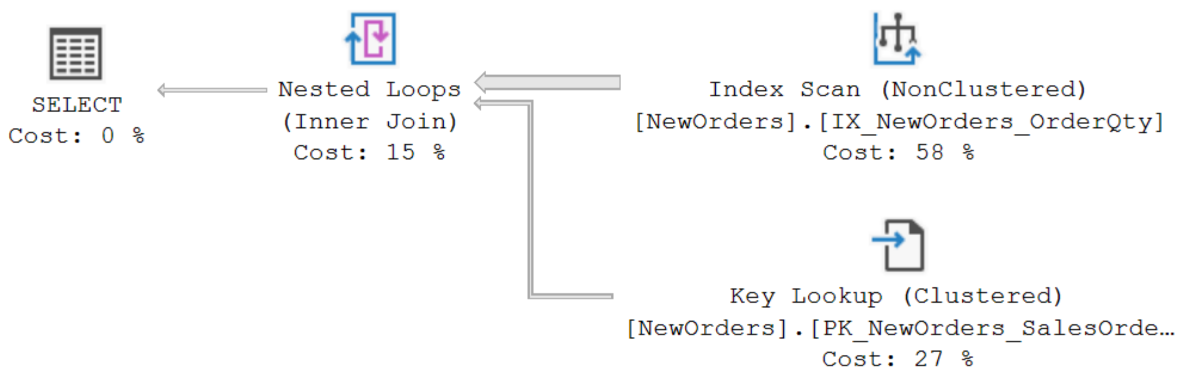


Figure 8-10: Parameter sniffing results in a plan with Key Lookups.

The optimizer has used parameter sniffing and created a plan optimized for a parameter value of 20, which we can see from the properties of the **SELECT** operator.

⊟ Parameter List	@OrderQty
Column	@OrderQty
Parameter Compiled Value	(20)
Parameter Data Type	smallint
Parameter Runtime Value	(20)

Figure 8-11: Parameter List showing the same runtime and compile time parameter values.

This means that we see the same nonclustered index and key lookup combination, but with the difference that here the optimizer *scans* rather than seeks the nonclustered index (I'll explain why, shortly).

The timing and I/O metrics tell us that SQL Server performs 424 logical reads and the execution time was about 10 milliseconds.

If the optimizer had not been able to sniff the parameter value, we know that it would have used the density graph for the nonclustered index to estimate a cardinality of 2958.95 rows, and chosen a clustered index scan (see Figure 8-3). So, this is an example of the optimizer making good use of its ability to sample the data directly through parameter sniffing to arrive at a more efficient execution plan; scanning the smaller nonclustered index and performing a few key lookups is cheaper than scanning the clustered index.

However, parameter sniffing can have a darker side. Let's re-execute the stored procedure and pass it a different value.

```
EXEC dbo.OrdersByQty @OrderQty = 1;
GO
```

Listing 8-7

It reuses the execution plan from the cache, but now 74954 rows match the parameter value, rather than 46, which means 74954 executions of the **Key Lookup**, instead of 46. It performs 239186 logical reads, and takes about 1400 ms.

If you see performance issues with stored procedures, it's worth checking the properties of the first operator for the plan to see if the compile and runtime values for any parameters are different.

Parameter List	@OrderQty
Column	@OrderQty
Parameter Compiled Value	(20)
Parameter Data Type	smallint
Parameter Runtime Value	(1)

Figure 8-12: Parameter List showing different runtime and compile time parameter values.

If they are, that's your cue to investigate 'bad' parameter sniffing as the cause. Of course, here, we know the optimizer would choose a different plan for Listing 8-7 if it were starting from scratch. Listing 8-8 retrieves the `plan_handle` value for our stored procedure, from the `sys.dm_exec_procedure_stats` DMV and uses it to flush just that single plan from the procedure cache.

```
DECLARE @PlanHandle VARBINARY(64);
SELECT  @PlanHandle = deps.plan_handle
FROM    sys.dm_exec_procedure_stats AS deps
WHERE   deps.object_id = OBJECT_ID('dbo.OrdersByQty');
IF @PlanHandle IS NOT NULL
    BEGIN
        DBCC FREEPROCCACHE(@PlanHandle);
    END
GO
```

Listing 8-8

Run Listing 8-7 again and the optimizer uses the histogram to get an estimated row count of 74954 (spot on), and you'll see the clustered index scan plan, and only 1512 logical reads instead of 239186.

Finally, why does the optimizer use an **Index Scan**, rather than **Seek** operator in Figure 8-10? If we check the properties of the **Index Scan**, we'll see that the **Predicate** condition is `OrderQty = @OrderQty OR @OrderQty IS NULL`. The reason is simply that the optimizer must always ensure that a plan is safe for reuse. If it has selected the expected **Index Seek** with a Seek Predicate of `OrderQty = @OrderQty`, then what would happen if that plan were reused when no value for `@OrderQty` was supplied? The seek predicate would be an equality with `NULL` and no rows would be returned, when of course the intent would be to return rows for all order quantities.

What to do if parameter sniffing causes performance problems

There are many possible ways to address problems relating to parameter sniffing, depending on the exact situation. If the data distribution is "jagged" with lots of variations in row counts returned, depending on the input parameter value, then this will often increase the likelihood of problematic parameter sniffing.

In such cases, you might consider adding the `OPTION (RECOMPILE)` hint to the end of the affected query (or queries). For example, if a stored procedure has three queries and only one of them suffers from bad sniffing, then only add the hint to the affected query; recompiling all three is a waste of resources.

This will force SQL Server to recompile the plan for that query every time, and optimize it for the specific value passed in. Use of this hint within our `OrderByQty` stored procedure would both fix the problem with problematic parameter sniffing, and mean that the optimizer could choose a plan with the usual **Index Seek** / **Key Lookup** combination (instead of the **Index Scan** / **Key Lookup** seen in Figure 8-10, since it will then know that the plan will never be reused.

However, the downside with the `OPTION (RECOMPILE)` solution, generally, is the extra compilations it causes. For stored procedures and other code modules, all statements, including the one with `OPTION(RECOMPILE)`, will still be in the plan cache, but the plan for the `OPTION(RECOMPILE)` statement will still recompile for every execution, which means that its plan is not reused. When we use the hint for ad hoc queries, the optimizer marks the plan created so that it is not stored in cache at all.

An alternative is to persuade the optimizer to always pick a specific plan; since the problem is caused by the optimizer optimizing the query based on an inappropriate parameter value, the solution might be to specify what parameter value the optimizer *must* use to create the plan, using the `OPTION (OPTIMIZE FOR <value>)` query hint. We'll cover hints in detail in Chapter 10. Yet another alternative, is to use a plan-forcing technique, discussed in Chapter 9.

Of course, this relies on us knowing the best parameter value to pick, one that will most often result in an efficient or at least good-enough execution plan. For example, from the previous example, we might choose to optimize for an `OrderQty` value of 20, if we felt the plan in Figure 8-10 would generally be the best plan. The issue you can hit here is that data changes over time and that value may no longer work well in the future.

Yet another alternative is to generate a generic plan, by optimizing for an unknown value. We can do this, in this case, by adding the `OPTION (OPTIMIZE FOR (@OrderQty UNKNOWN))` hint to the query in our stored procedure. The optimizer will use the density graph to arrive at a cardinality estimation (in this case, always estimating that 2958.95 rows will return), and we'll see the plan in Figure 8-9.

The issue comes when good enough just isn't, for certain values, such that performance suffers unduly where a more specific plan would work better. In short, everything is a trade-off. There isn't always a single correct answer.

Columnstore Indexes

Columnstore indexes were a new index type introduced in SQL Server 2012, in addition to the existing index types. With a columnstore index, the storage architecture is different. It doesn't use the B-tree as a primary storage mechanism (although part of the data can be stored in a B-tree), and it stores data by column instead of by row. So, rather than storing as many rows as will fit on a data page, the columnstore index takes all values for a single column and stores them in one or more pages.

A clustered columnstore index replaces the heap table with an index that stores all the table's data in a column-wise structure. A nonclustered columnstore index can be applied to any table, alongside traditional "rowstore" clustered and nonclustered indexes.

CS indexes achieve high data compression and are designed to improve the performance of analysis, reporting, and aggregation queries such as those found in a data warehouse. In other words, typical workloads for CS indexes involve a combination of large tables (millions, or even billions, of rows), and queries that operate on all rows or on large selections. In fact, simple queries that retrieve a single row or small subsets of rows usually perform much worse with the columnstore indexes than they do with traditional indexes, because in the former case SQL Server needs to read a page for each column in the table to reconstruct each row.

Further, the nature of the storage of the columnstore index makes putting less than 100,000 rows into the index much less efficient than storing greater than that value of rows. Quoting from the Microsoft documentation, you should consider using a clustered columnstore index on a table when: ***Each partition has at least a million rows. Columnstore indexes have rowgroups within each partition. If the table is too small to fill a rowgroup within each partition, you won't get the benefits of columnstore compression and query performance.***

As well as having a different architecture, columnstore indexes also support a new kind of query execution model, optimized for modern hardware, called **batch mode**, the traditional model being **row mode**. We won't cover plans for queries that use the batch mode execution model until Chapter 12.

This section is going to focus purely on how columnstore indexes are exposed within the execution plans and some important properties that you need to pay attention to when working with these indexes. For further detail regarding columnstore indexes, and their use in query tuning, their behavior and storage mechanisms, and maintenance, I suggest the following resources:

- **Columnstore indexes: overview** – the Microsoft documentation: http://bit.ly/1djYOCW
- **SQL Server Central Stairway to Columnstore Indexes** – written by Hugo Kornelis, the technical reviewer of this book: http://bit.ly/2CBiXoQ
- **Columnstore indexes: what's new** – includes a useful table summarizing support for various CS features from SQL Server 2012 onwards: http://bit.ly/2oD9keB
- **Niko Neugebauer's Columnstore series** – extensive and comprehensive coverage of all aspects of using columnstore indexes, though the early articles cover the basics: http://www.nikoport.com/columnstore/

Using a columnstore index for an aggregation query

Despite being designed for analytics queries on very large tables, you can occasionally see improved performance using the columnstore index even within an OLTP system, if, for example, you have reporting queries that pull data from very large tables. A minimum number of recommended rows to really see big performance gains is one million.

We'll start with a simple query on the `TransactionHistory` table, with no columnstore indexes created. This table is not an ideal candidate for a columnstore index, since it contains only 113 K rows, and is subject to OLTP-style, rather than DW-style, workloads. However, columnstore indexes are well suited to aggregation queries, so this simple example serves perfectly well as a first demo of how columnstore indexes work.

```
SELECT p.Name,
       COUNT(th.ProductID) AS CountProductID,
       SUM(th.Quantity) AS SumQuantity,
       AVG(th.ActualCost) AS AvgActualCost
FROM Production.TransactionHistory AS th
    JOIN Production.Product AS p
        ON p.ProductID = th.ProductID
GROUP BY th.ProductID,
         p.Name;
```

Listing 8-9

The execution plan shown in Figure 8-13 illustrates some of the potential load on the server from this query.

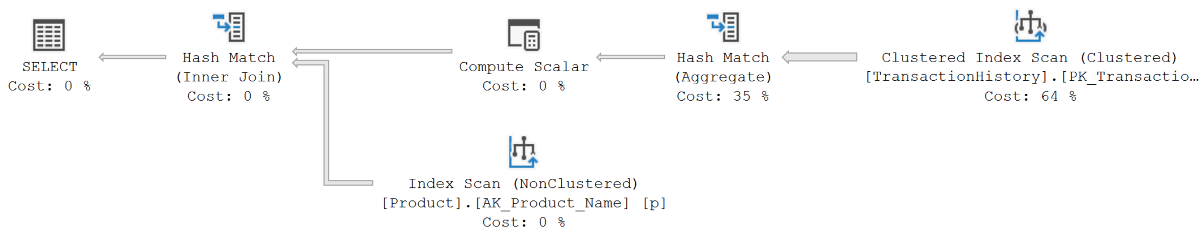


Figure 8-13: Execution plan for an aggregation query (no Columnstore index).

Our query has no `WHERE` clause, so the optimizer sensibly decides to scan the clustered index to retrieve all the data from the `TransactionHistory` table. We then see a **Hash Match (Aggregate)** operator. As discussed in Chapter 5, SQL Server creates a temporary hash table in memory in which it stores the results of all aggregate computations. In this case, the hash table is created on the `ProductID` column, and for each distinct `ProductID` value it stores a row count tally, total `Quantity`, and total `ActualCost`, increasing the counts and totals whenever it processes a row with the same `ProductID`. A **Compute Scalar** computes the requested `AVG`, by dividing the row tally for each `ProductID` by the total `Actual-Cost` (it also performs some data type conversions). This data stream forms the Build input for a **Hash Match** (inner join) operator, where the Probe input is an **Index Scan** against the `Product` table, to join the `Name` column.

This simple query returns 441 rows and in my tests returned them in 127ms, on average, with 803 logical reads. Let's see what happens when we add a nonclustered columnstore to the table.

```
CREATE NONCLUSTERED COLUMNSTORE INDEX ix_csTest
ON Production.TransactionHistory
(
    ProductID,
    Quantity,
    ActualCost,
    ReferenceOrderID,
    ReferenceOrderLineID,
    ModifiedDate
);
```

Listing 8-10

If we rerun the query from Listing 8-9, we'll see significant changes in performance. In my tests, the query time dropped from an average of 127ms to an average of 55ms, and the number of logical reads plummeted from 803 to 84, because the columnstore structure allows the engine to read only the requested columns and skip the other columns in the table. You'll likely see variance on the number of reads, because of how columnstore builds the index and compresses the data.

Figure 8-14 shows the execution plan.

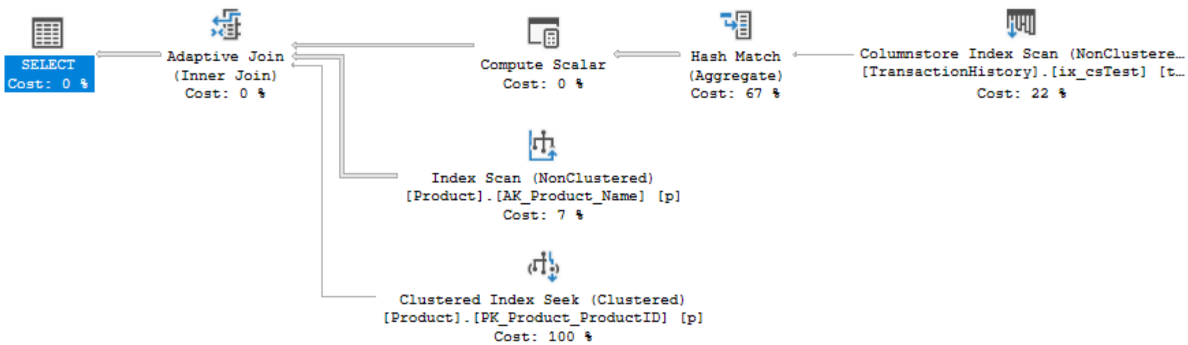


Figure 8-14: Execution plan for an aggregation query (with Columnstore index).

We've seen the **Adaptive Join** before, in Chapter 4, so we won't describe that part of the plan again here. Note that you'll only see this operator if your database compatibility level is set to 140 or higher.

We'll use this plan, and one for a similar query with a `WHERE` clause filter, to explore differences you'll encounter in execution plans, when the optimizer chooses to access data using a columnstore index.

Aggregate pushdown

The first difference from the plan we saw before creating the CS index is that we now see a **Columnstore Index Scan**. If we look at its property sheet, some of the values may seem confusing at first, since it seems to suggest that the estimated number of rows returned is 113443, but the actual number of rows is 0!

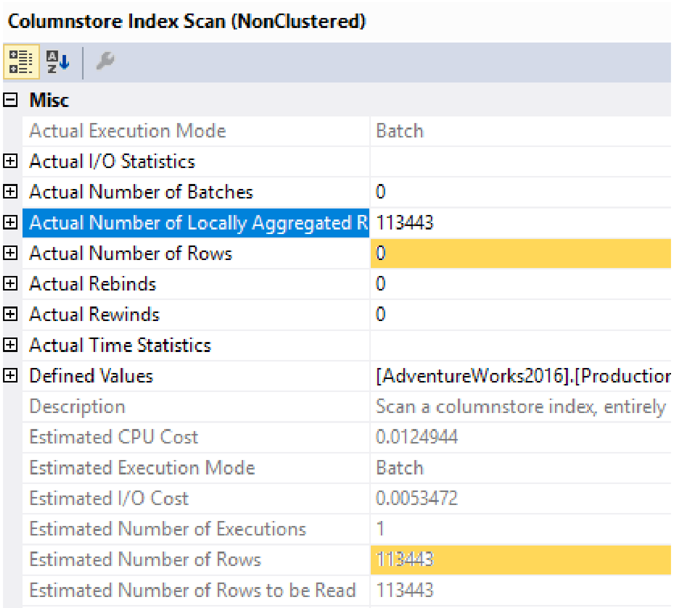


Figure 8-15: Columnstore Index Scan properties showing locally aggregated rows.

This is a special feature of CS indexes in action, called "aggregate pushdown," introduced in SQL Server 2016, where some, or all, of the aggregation is done by the scan itself. This is possible because of the pivoted storage mechanisms of the columnstore index. The aggregation results are "injected directly" into the aggregation operator, in this case the **Hash Match (Aggregate)** operator. The arrow from the operator displays only rows that cannot be locally aggregated. This explains why the **Hash Match (Aggregate)** operator appears to make the arrow thicker (effectively adding rows).

In Figure 8-15, the **Actual Number of Locally Aggregated Rows** value indicates the number of rows that were aggregated within the scan and not returned in "the normal way" to the **Hash Match (Aggregate)**, in this case all the rows (113443). On a **Columnstore Index Scan** operator, the **Actual Number of Rows** is the number of rows that were not aggregated in the scan and were hence returned "normally," in this case, zero rows.

No seek operation on columnstore index

Let's add a simple filter to our previous aggregation query.

```
SELECT p.Name,
       COUNT(th.ProductID) AS CountProductID,
       SUM(th.Quantity) AS SumQuantity,
       AVG(th.ActualCost) AS AvgActualCost
FROM Production.TransactionHistory AS th
    JOIN Production.Product AS p
        ON p.ProductID = th.ProductID
WHERE th.TransactionID > 150000
GROUP BY th.ProductID,
         p.Name;
```

Listing 8-11

The plan is the same shape, and has the same operators as the one in Figure 8-14; we still see the **Columnstore Index Scan**. There is no **Seek** operator for a columnstore index, simply due to how the index is organized; the data in a columnstore index is not sorted in any way, so there is no way to find specific values directly.

Predicate pushdown in a columnstore index

If we examine the properties of the **Columnstore Index Scan** in the plan for Listing 8-11, we see that the WHERE clause predicate was pushed down.

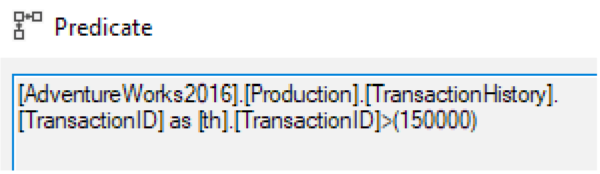


Figure 8-16: Predicate within the columnstore index.

Predicate pushdown in a **Columnstore Index Scan** is even more important than in a **Rowstore Index Scan**, because pushed predicates can result in **rowgroup elimination** (sometimes, for historic reasons, incorrectly called **segment elimination**). In a columnstore index, each partition is divided into units called rowgroups, and each rowgroup contains up to about a million rows that are compressed into columnstore format at the same time.

Rowgroup elimination is visible in `SET STATISTICS IO`, by looking at the "**segment skipped**" count.

```
(434 rows affected)
Table 'TransactionHistory'. Scan count 2, logical reads 0, physical
reads 0, read-ahead reads 0, lob logical reads 63, lob physical
reads 0, lob read-ahead reads 0.
Table 'TransactionHistory'. Segment reads 1, segment skipped 0.
Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0,
read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob
read-ahead reads 0.
Table 'Product'. Scan count 1, logical reads 6, physical reads 0,
read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob
read-ahead reads 0.
```

In this example, on a small table, all data is in a single rowgroup, so we don't see rowgroup elimination, of course. However, if you have a 60 million row table then predicate pushdown can lead to rowgroup elimination and you will see an improvement in query performance.

Batch mode versus row mode

We cover Batch mode in detail in Chapter 12, and the only details I want to call out here are the **Actual Execution Mode** and **Estimated Execution Mode** for the **Columnstore Index Scan** operator, both of which are **Batch** in this case (see Figure 8-15).

This indicates that the plan was optimized for batch mode operation, and so we're seeing the full potential of the columnstore index. If a query is unexpectedly slow when using a columnstore index then it's worth comparing the actual and estimated execution modes. If the former shows row and the latter, batch, then you have a plan optimized for batch mode that for some reason had to fall back into row mode during execution. This is very bad for query performance, but is only an issue on SQL Server 2012, where a batch mode plan can fall back to row mode when a hash operation spills to tempdb.

Memory-optimized Indexes

Indexes perform the same purpose for memory-optimized tables as for the disk-based tables that we've used up to now. However, they are very different structures, representing a complete redesign of the data access and locking structures, and specifically designed to get the best possible performance from being in-memory.

Memory-optimized tables, introduced in SQL Server 2014, support two new types of nonclustered index:

- **Hash indexes** – a completely new type of index, for memory-optimized tables, used for performing lookups on specific values. It's essentially an array of hash buckets, where each bucket points to the location of a data row, in memory.
- **Range indexes** – used for retrieving ranges of values, and more akin to the familiar B-tree index, except these memory-optimized counterparts use a different, Bw-tree storage structure.

Again, memory-optimized tables and indexes are designed to meet the specific performance requirements of very-high-throughput OLTP systems, with many inserts per second, but as well as inserts, updates, and deletes. In other words, the sort of situation where you're likely to experience the bottleneck of page latches in memory, when accessing disk-based tables.

Even if you’re not hitting the memory latch issues, but you have an extremely write-heavy database, you could see some benefits from memory-optimized tables. Otherwise, the only other regular use of memory-optimized tables is to enhance the performance of table variables.

Again, our goal in this section is purely to examine some of the main features of execution plans for queries that access memory-optimized tables and indexes. For further details of their design and use, as well as the various caveats that may prevent you from using them, I'd suggest the Microsoft online documentation (http://bit.ly/2EQl2Lc) and Kalen Delaney's book on the topic (http://bit.ly/2BpDxXI).

Using memory-optimized tables and indexes

Listing 8-12 creates a test database and, in it, three memory-optimized tables (copied from AdventureWorks 2014), and then fills them with data. Please adjust the values of the file properties, `FILENAME`, `SIZE` and `FILEGROWTH`, as suitable for your system.

```
CREATE DATABASE InMemoryTest
ON PRIMARY (NAME = InMemTestData,
            FILENAME = 'C:\Data\InMemTest.mdf',
            SIZE = 10GB,
            FILEGROWTH = 10GB),
   FILEGROUP InMem CONTAINS MEMORY_OPTIMIZED_DATA (NAME = InMem,
                                                   FILENAME = 'c:\
data\inmem.ndf')
LOG ON (NAME = InMemTestLog,
        FILENAME = 'C:\Data\InMemTestLog.ldf',
        SIZE = 5GB,
        FILEGROWTH = 1GB);
GO
--Move to the new database
USE InMemoryTest;
GO
--Create some tables
CREATE TABLE dbo.Address (AddressID INTEGER NOT NULL IDENTITY
PRIMARY KEY NONCLUSTERED HASH
                                                                  WITH
(BUCKET_COUNT = 128),
                          AddressLine1 VARCHAR(60) NOT NULL,
                          City VARCHAR(30) NOT NULL,
                          StateProvinceID INT NOT NULL)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
GO
CREATE TABLE dbo.StateProvince (StateProvinceID INTEGER NOT NULL
PRIMARY KEY NONCLUSTERED,
                                StateProvinceName VARCHAR(50) NOT
NULL,
                                CountryRegionCode NVARCHAR(3) NOT
NULL)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
CREATE TABLE dbo.CountryRegion (CountryRegionCode NVARCHAR(3) NOT
NULL PRIMARY KEY NONCLUSTERED,
                                CountryRegionName NVARCHAR(50) NOT
NULL)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```

```
--Add Data to the tables
--Cross database queries can't be used with in-memory tables
SELECT a.AddressLine1,
       a.City,
       a.StateProvinceID
INTO dbo.AddressStage
FROM AdventureWorks2014.Person.Address AS a;
INSERT INTO dbo.Address (AddressLine1,
                         City,
                         StateProvinceID)
SELECT a.AddressLine1,
       a.City,
       a.StateProvinceID
FROM dbo.AddressStage AS a;
DROP TABLE dbo.AddressStage;
SELECT sp.StateProvinceID,
       sp.Name,
       sp.CountryRegionCode
INTO dbo.ProvinceStage
FROM AdventureWorks2014.Person.StateProvince AS sp;
INSERT INTO dbo.StateProvince (StateProvinceID,
                               StateProvinceName,
                               CountryRegionCode)
SELECT ps.StateProvinceID,
       ps.Name,
       ps.CountryRegionCode
FROM dbo.ProvinceStage AS ps;
DROP TABLE dbo.ProvinceStage;
SELECT cr.CountryRegionCode,
       cr.Name
INTO dbo.CountryStage
FROM AdventureWorks2014.Person.CountryRegion AS cr;
INSERT INTO dbo.CountryRegion (CountryRegionCode,
                               CountryRegionName)
SELECT cs.CountryRegionCode,
       cs.Name
FROM dbo.CountryStage AS cs
DROP TABLE dbo.CountryStage;
GO
```

Listing 8-12

Before we dive in, let's first run a query that accesses the standard, disk-based Adventure-Works tables, for comparison.

```
SELECT a.AddressLine1,
       a.City,
       sp.Name,
       cr.Name
FROM Person.Address AS a
    JOIN Person.StateProvince AS sp
        ON sp.StateProvinceID = a.StateProvinceID
    JOIN Person.CountryRegion AS cr
        ON cr.CountryRegionCode = sp.CountryRegionCode
WHERE a.AddressID = 42;
```

Listing 8-13

It produces a standard execution plan with no real surprises, or new lessons to be learned.

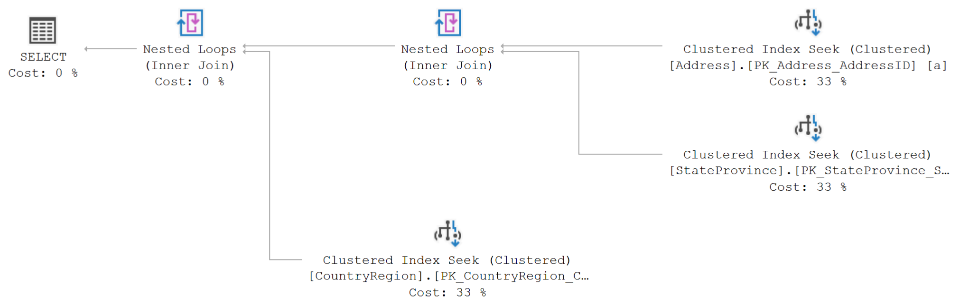


Figure 8-17: Execution plan for query accessing standard tables.

We can run essentially the same standard query against our `InMemoryTest` table, thanks to the **Query Interop** component of in-memory OLTP, which allows interpreted T-SQL to reference memory-optimized tables.

```
USE InMemoryTest;
GO
SELECT a.AddressLine1,
       a.City,
       sp.StateProvinceName,
       cr.CountryRegionName
FROM dbo.Address AS a
    JOIN dbo.StateProvince AS sp
        ON sp.StateProvinceID = a.StateProvinceID
    JOIN dbo.CountryRegion AS cr
        ON cr.CountryRegionCode = sp.CountryRegionCode
WHERE a.AddressID = 42;
```

Listing 8-14

The execution plan it produces is not very abnormal looking either.

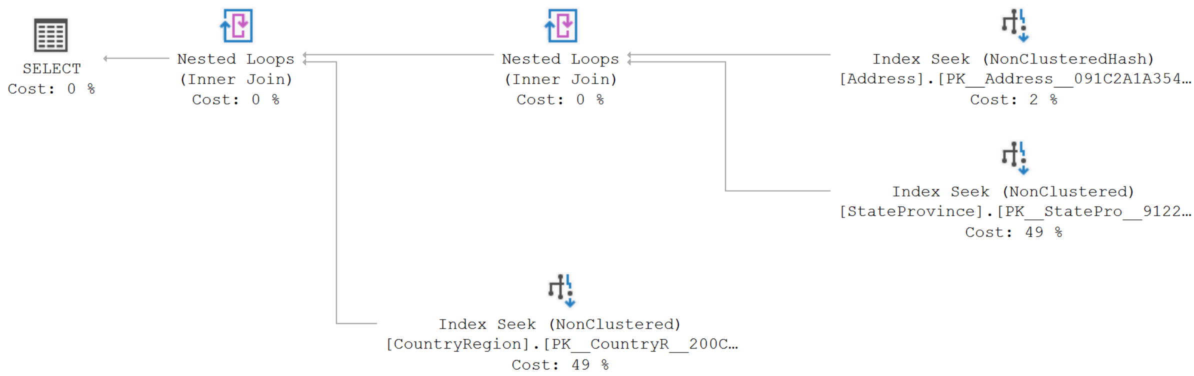


Figure 8-18: Execution plan for query accessing memory-optimized tables.

However, there are a few differences:

- We can see a new **Index Seek (NonClusteredHash)** operator for accessing the Address table.
- Examine the Storage property of any of the **Index Seek** operators, and you'll see it's **MemoryOptimized** instead of **RowStore**.
- **Estimated costs** for the seeks are lower because the memory-optimized index is assumed to be more efficient.

On the last point, remember that lower cost estimated doesn't necessarily mean that these operations cost more or less. You can't effectively compare the costs of operations within a given plan with the costs of operations within another plan. They're just estimates. Estimates for a regular plan account for the fact that some of the costs will be accessing data from the disk whereas the cost estimates for in-memory plans will only be retrieving data from memory.

Standard queries against memory-optimized tables will generate a completely standard execution plan. You'll be able to understand which indexes have been accessed, and how they're accessed. Internally there's a lot going on, but visibly, in the graphical plan, there's just nothing much to see.

It gets more interesting when we look at a slightly different query.

No option to seek a hash index for a range of values

Let's modify the query just a little bit, looking for a range of addresses rather than just one.

```
SELECT a.AddressLine1,
       a.City,
       sp.StateProvinceName,
       cr.CountryRegionName
FROM dbo.Address AS a
    JOIN dbo.StateProvince AS sp
        ON sp.StateProvinceID = a.StateProvinceID
    JOIN dbo.CountryRegion AS cr
        ON cr.CountryRegionCode = sp.CountryRegionCode
WHERE a.AddressID BETWEEN 42
                  AND     52;
```

Listing 8-15

If I run the equivalent query against the normal AdventureWorks database I'll get an execution plan as shown in Figure 8-19.

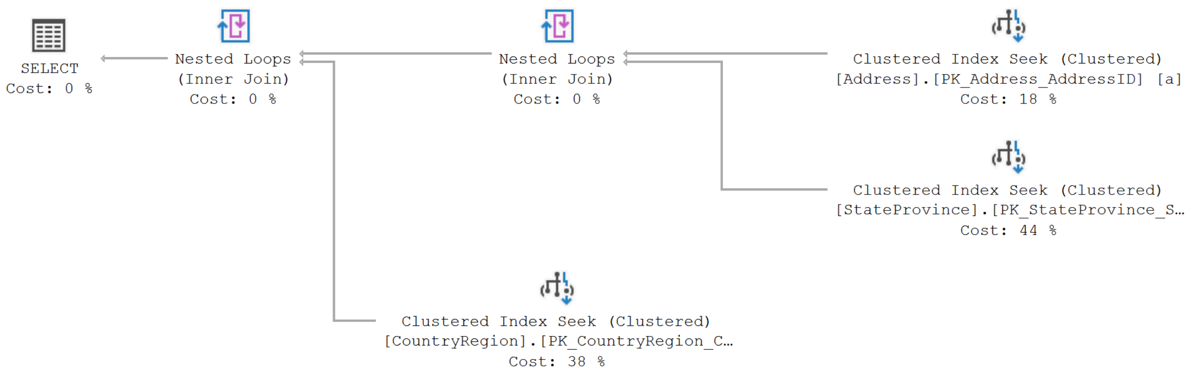


Figure 8-19: Standard execution plan with Index Seek operators.

The `BETWEEN` operator doesn't affect whether the clustered index is used for a seek operation. It's still an efficient mechanism for retrieving data from the clustered index on the `Address` table. Contrast this with the execution plan against the memory-optimized hash index.

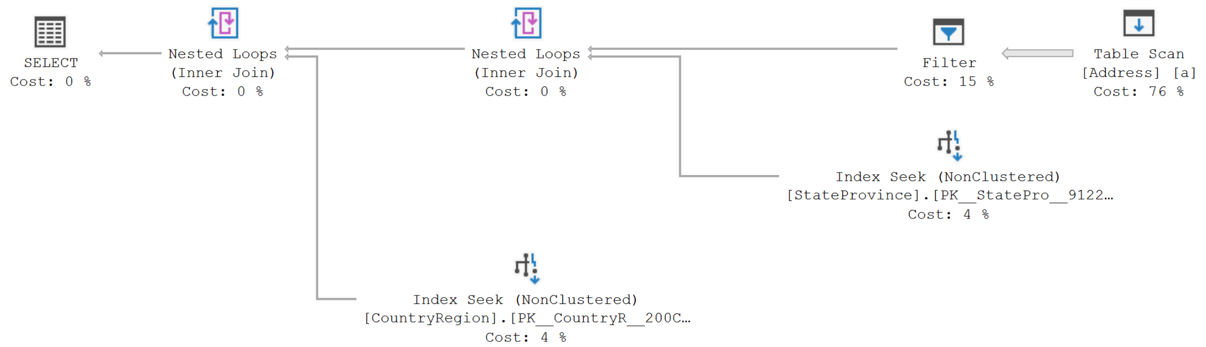


Figure 8-20: Execution plan for query using memory-optimized hash index.

Instead of a seek against the hash index, we see a table scan against the `Address` table. This is because the hash index isn't conducive to selections of range values, but instead is optimized for point lookups. Notice also that the optimizer cannot push down a search predicate into a scan when running in **Interop** mode, so it must pass all 19,614 rows to the **Filter** operator.

If this was the common type of query being run against this table, we'd need to have a memory-optimized nonclustered index on the table to better support this type of query. You can use your execution plans to evaluate this type of information within memory-optimized tables and queries.

Plans with natively-compiled stored procedures

One additional object that was introduced with memory-optimized tables is the natively-compiled stored procedure. Currently, the behavior here is different than the standard queries as demonstrated above. Listing 8-17 creates a natively-compiled stored procedure from the query in Listing 8-15.

```
CREATE OR ALTER PROC dbo.AddressDetails @AddressIDMin INT, @
AddressIDMax INT
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER AS
BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE
= N'us_english')
 SELECT a.AddressLine1,
       a.City,
       sp.StateProvinceName,
       cr.CountryRegionName
```

```
FROM dbo.Address AS a
    JOIN dbo.StateProvince AS sp
        ON sp.StateProvinceID = a.StateProvinceID
    JOIN dbo.CountryRegion AS cr
        ON cr.CountryRegionCode = sp.CountryRegionCode
WHERE a.AddressID BETWEEN @AddressIDMin
                  AND     @AddressIDMax;
END
GO
EXECUTE dbo.AddressDetails @AddressIDMin = 42, -- int
                           @AddressIDMax = 52;  -- int
```

Listing 8-16

We cannot execute the query and get an actual execution plan. That's a limitation with the compiled procedures. We can get an estimated plan.

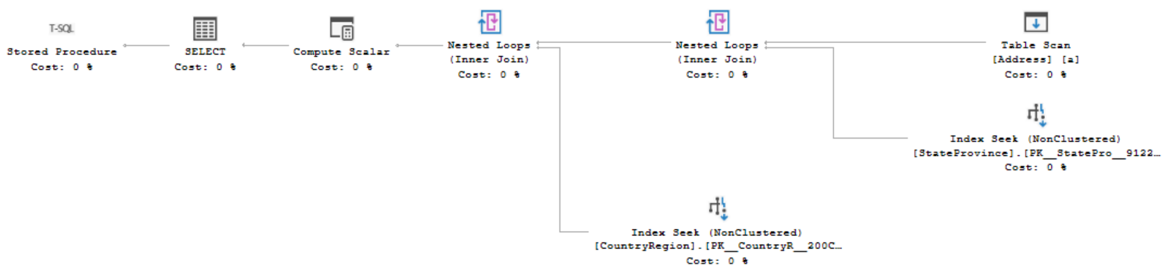


Figure 8-21: Execution plan for query accessing a natively-compiled stored procedure.

We still see the **Table Scan** on the `Address` table, because there is no supporting index, but this time, but if we examine its properties, we see that predicate pushdown is supported in natively-compiled code.

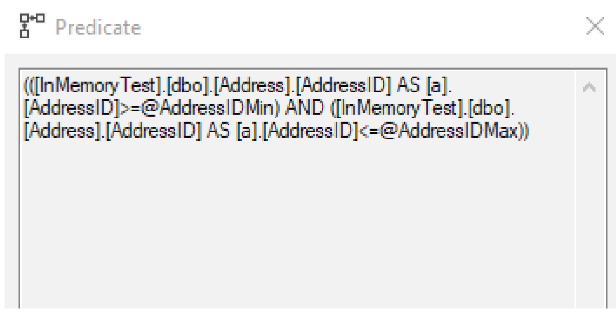


Figure 8-22: Predicate pushdown in natively-compiled code.

A scan within a memory-optimized table is faster, and different internally, than a standard table, but if the table has a few million rows it will still take time to scan all of them, and a Bw-tree index would still be useful for this query. Even if did choose to alter the table to supply an index, the plan itself won't recompile and show us differences, it'll just choose the index at runtime.

Note that all the estimated costs are zero because Microsoft are costing these procedures in a new way that isn't reflected externally. There is not a single value beyond zero in any of the estimated costs inside any of the properties for any of the operators. Let's look at the properties of the **SELECT** operator.

Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0
IsNativelyCompiled	True
Procedure Name	dbo.AddressDetails

Figure 8-23: SELECT operator properties showing estimated costs of zero for natively-compiled code.

That represents the complete set of properties available. None of the useful properties we've discussed earlier in the book such as the **Reason for Early Termination** exist here. This is because of differences in how these plans are stored (for example, this plan is not in the plan cache) and how they are generated.

As of this writing, SQL Server 2017 execution plans, when used with the compiled memory-optimized stored procedures, are less useful. Missing the row counts and costs affects your ability to make decisions based on the plans, but they still provide good information, which should allow you to see the actions taken when the query executes, and figure out why a query is slow.

Summary

It's difficult to overstate the impact of indexes and their supporting statistics on the quality of the plans that the optimizer generates.

You can't always solve a performance problem just by adding an index. It is entirely possible to have too many indexes, so you must be judicious in their use. You need to ensure that the index is selective, and you must make appropriate choices regarding the addition or inclusion of columns in your indexes, both clustered and nonclustered.

You will also need to be sure that your statistics accurately reflect the data that is stored within the index because the choice of index used in plan is based on the optimizer's estimated row count and estimated operator costs, and the estimated row counts are based on statistics. If you use hard-coded input parameter values, then the optimizer can use statistics for that specific value, but SQL Server loses the ability to reuse plans for those queries. If the optimizer can sniff parameters, such as when we use a stored procedure, it can use accurate statistics, but a reused plan based on a sniffed parameter can backfire if the next parameter has a hugely different rowcount.

Chapter 9: Exploring Plan Reuse

All the processes the optimizer needs to perform to generate execution plans, come at a cost. It costs time and CPU resources to devise an execution strategy for a query. For simple queries, SQL Server can generate a plan in less than a millisecond, but on typical OLTP systems there are lots of these short, fast queries and the costs can add up. If the workload also includes complex aggregation and reporting queries, then it will take the optimizer longer to create an execution plan for each one.

Therefore, it makes sense that SQL Server wants to avoid paying the cost of generating a plan every single time it needs to execute a query, and that's why it tries its best to reuse existing query execution strategies. The optimizer saves them as reusable plans, in an area of memory called the **plan cache**. Ideally, if the optimizer encounters a query it has seen before, it grabs a ready-made execution strategy for it from the plan cache, and passes it straight to the execution engine. That way, SQL Server spends valuable CPU resources *executing* our queries, rather than always having to first devise a plan, and *then* execute it.

SQL Server will try its best to promote plan reuse automatically, but there are limits to what it can do without our help as programmers. Fortunately, armed with some simple techniques, we can ensure that our queries are correctly parameterized, and that plans get reused as often as possible; I'm going to show you exactly what you need to do. We'll also explore some of the problems that can occur with plan reuse and what you can do about them.

Querying the Plan Cache

As discussed in Chapter 1, when we submit any query for execution, the optimizer generates a plan if one doesn't already exist that it can reuse, and stores it in an area of the buffer pool called the plan cache. Our goal as programmers, DBAs and database developers, is to help promote efficient use of this memory, which means that the plan for a query gets reused from cache, and not created or recreated each time the query is called, unless changes in structures or statistics necessitate recompiling the plan.

The plan cache has four cache stores that store plans (see https://bit.ly/2mgrS6s for more detail). The **compiled plans** in which we're interested will be stored in either the **SQL plans** cache store (`CACHESTORE_SQLCP`) or the **Object plans** store (`CACHESTORE_OBJCP`), depending on object type (`objtype`):

- **SQL plans** store contains plans for ad hoc queries, which have an `objtype` of `Adhoc`, as well as plans for auto-parameterized queries, and prepared statements, both of which have an `objtype` of `Prepared`.
- **Object plans** store contains plans for procedures, functions, triggers, and some other types of object, and each plan will have an associated Object ID value. Plans for stored procedures, scalar user-defined functions, or multi-statement table-valued functions have an `objtype` of `Proc`, and triggers have an `objtype` of `Trigger`.

To examine plans currently in the cache, as well as to explore plan reuse, we can query a set of execution-related Dynamic Management Objects (DMOs). Whenever we execute an ad hoc query, a batch, or an object such as a stored procedure, the optimizer stores the plan. An identifier, called a `plan_handle`, uniquely identifies the cached query plan for every query, batch, or stored procedure that has been executed.

We can supply the `plan_handle` as a parameter to the `sys.dm_exec_sql_text` function to return the SQL text associated with a plan, as well as to the `sys.dm_exec_query_plan` function, to return the execution plan in XML format. Several DMOs store the `plan_handle`, but in this chapter, we'll primarily use:

- **`sys.dm_exec_cached_plans`** – returns a row for every cached plan and provides information such as the type of plan, the number of times it has been used, and its size.
- **`sys.dm_exec_query_stats`** – returns a row for every query statement in every cached plan, and provides execution statistics, aggregated over the time the plan has been in cache. Many of the columns are counters, and provide information about how many times the plan has been executed, and the resources that were used.

There are also a few DMOs that provide similar aggregated execution statistics to `sys.dm_exec_query_stats`, but for specific objects, each of which will have a separate plan, with an associated `object_id` value. We have `sys.dm_exec_procedure_stats` for stored procedures, `sys.dm_exec_trigger_stats` for triggers and `sys.dm_exec_function_stats` for user-defined scalar functions. Even though multi-statement table-valued functions do get a plan, with an `object_id` value, these plans only appear in `sys.dm_exec_query_stats`. Inline views and table-valued functions do not get a separate plan because their behavior is incorporated into the plan for the query referencing them.

All the previous DMOs are for investigating plans for queries that have completed execution. However, since the execution plan is already stored in the cache when execution starts, we also can look at the plan for queries that are still executing, using the `sys.dm_exec_requests` DMV. This is useful if your system is experiencing resource pressure right now, due to currently-executing, probably long-running, queries. This DMV stores the `plan_handle` and a range of other information, including execution stats, for any currently executing query, whether it's ad hoc, or a prepared statement, or part of a code module.

Using these DMOs, we can construct simple queries that for each `plan_handle` will return, for example, the associated query text, and an XML value representing the cached plan for that query, along with a lot of other useful information. We'll see some examples as we work through the chapter, though I won't be covering the DMOs in detail.

More on the DMOs

You can refer to the Microsoft documentation (http://bit.ly/2m1F6CA), or Louis Davidson and Tim Ford's excellent book, *Performance Tuning with SQL Server Dynamic Management Views* (https://bit.ly/2Je3evr), which is available as a free eBook. Glenn Berry's diagnostic queries (http://bit.ly/Q5GAJU) include lots of examples on using DMOs to query the cache in. Finally, you can skip writing your own queries and use Adam Machanic's **sp_WhoIsActive** (http://whoisactive.com/).

Plan Reuse and Ad Hoc Queries

When a query is submitted, the engine first computes the **QueryHash** and looks for matching values in the plan cache. If any are found, it does a detailed comparison of the full SQL text. If they are identical then, assuming there are also no differences in `SET` options or database ID, it can bypass the compilation process and simply submit the cached plan for execution. This is efficient plan reuse at work, and we'd like to promote this as far as possible. Unfortunately, use of ad hoc queries with hard-coded literals, to cite one example, defeats plan reuse.

Listing 9-1 clears out the plan cache and then executes a batch consisting of three ad hoc queries, which concatenate the name columns in the `Person` table of AdventureWorks. The first and second queries are identical in all but the value supplied for `BusinessEntityID`, and the second and third differ only in white space formatting.

```
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
GO
SELECT ISNULL(p.Title, '') + ' ' + p.FirstName + ' ' + p.LastName
FROM Person.Person AS p
WHERE p.BusinessEntityID = 5;
SELECT ISNULL(p.Title, '') + ' ' + p.FirstName + ' ' + p.LastName
FROM Person.Person AS p
WHERE p.BusinessEntityID = 6;
SELECT ISNULL(p.Title, '') + ' ' + p.FirstName + ' ' + p.LastName
FROM Person.Person AS p WHERE p.BusinessEntityID = 6;
GO
```

Listing 9-1

The plans for each query are the same in each case, consisting of only three operators. If you examine the **QueryHash** and **QueryPlanHash** values of the `SELECT` operator, you'll see that these are identical for each plan. However, let's see what's stored in the plan cache. All the DMOs used in this query are server-scoped, so the database context for the query is irrelevant.

```
SELECT cp.usecounts,
       cp.objtype,
       cp.plan_handle,
       DB_NAME(st.dbid) AS DatabaseName,
       OBJECT_NAME(st.objectid, st.dbid) AS ObjectName,
       st.text,
       qp.query_plan
FROM sys.dm_exec_cached_plans AS cp
    CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) AS st
    CROSS APPLY sys.dm_exec_query_plan(cp.plan_handle) AS qp
WHERE st.text LIKE '%Person%'
      AND st.dbid = DB_ID('AdventureWorks2014')
      AND st.text NOT LIKE '%dm[_]exec[_]%' ;
```

Listing 9-2

Figure 9-1 shows the result set, with one entry.

usecounts	objtype	plan_handle	Databas...	ObjectN...	text	query_plan
1	Adhoc	0x0600050000C...	Advent...	NULL	SELECT ISNULL(Person...	<ShowPlanXML xmlns="http://schemas.r

Figure 9-1: Results from querying the plan cache.

When we submit to the query processor a batch, or a stored procedure or function, containing multiple statements, the whole batch will be compiled at once, and so the optimizer has produced a plan for the whole `Adhoc` batch. If you check the value of the `text` column, you'll see it's the SQL text of the entire batch. The final column in the result set, **query_plan**, contains the XML representation of the query execution plan. When viewing the results in grid view, these XML values are displayed as hyperlinks, and we can click on one to show the graphical form of the execution plan. As you can see, the optimizer produces a plan for the batch, which contains individual plans for every statement in the batch.

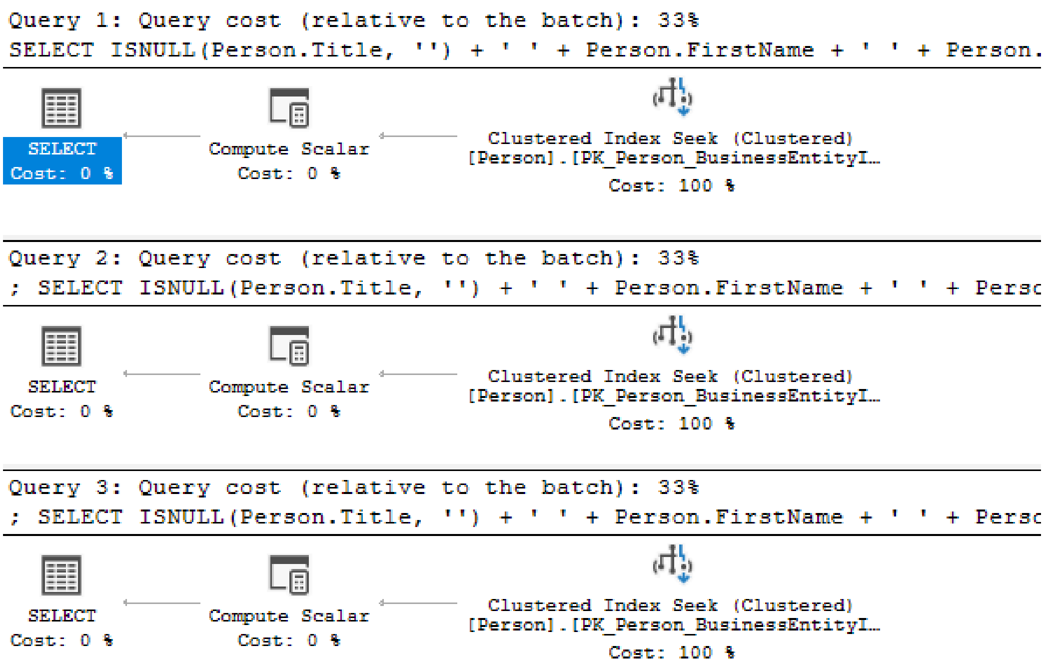


Figure 9-2: Three execution plans that look the same despite being from three queries.

The first column of the result set, in Figure 9-1, **usecounts**, tells us the number of times a plan has been looked up in the cache. In this case it's once, and the only way the plan for this batch will be reused is if we submit the exact same batch again; same formatting, same literal values. If we re-execute just part of the same batch, such as the last query then, after rerunning Listing 9-2, we'll see a new entry, and a new plan generated.

The `sys.dm_exec_query_stats` DMV shows us a slightly different view on this, since it returns one row for every query statement in a cached plan.

```
SELECT SUBSTRING(
                          dest.text,
                          (deqs.statement_start_offset / 2) + 1,
                          (CASE deqs.statement_end_offset
                                WHEN -1 THEN
                                    DATALENGTH(dest.text)
                                ELSE
                                    deqs.statement_end_offset - deqs.
statement_start_offset
                           END
                          ) / 2 + 1
                     ) AS QueryStatement,
       deqs.creation_time,
       deqs.execution_count,
       deqp.query_plan
FROM sys.dm_exec_query_stats AS deqs
    CROSS APPLY sys.dm_exec_query_plan(deqs.plan_handle) AS deqp
    CROSS APPLY sys.dm_exec_sql_text(deqs.plan_handle) AS dest
WHERE dest.text LIKE '%Person%'
      AND deqp.dbid = DB_ID('AdventureWorks2016')
      AND dest.text NOT LIKE '%dm[_]exec[_]%'
ORDER BY deqs.execution_count DESC,
         deqs.creation_time;
```

Listing 9-3

To see some differences in counts and batches, execute the final statement in the batch from Listing 9-1 two times. Figure 9-3 shows the results after executing the whole of Listing 9-1 once, and then those additional two executions.

	text	creation_time	execution_count	query_plan
1	SELECT ISNULL(Person.Title, '') + ' ' + Person....	2018-04-16 18:41:49.020	2	<ShowPlanXML xmlns="http://sc
2	SELECT ISNULL(Person.Title, '') + ' ' + Person...	2018-04-16 18:41:43.253	1	<ShowPlanXML xmlns="http://sc
3	SELECT ISNULL(Person.Title, '') + ' ' + Person...	2018-04-16 18:41:43.257	1	<ShowPlanXML xmlns="http://sc
4	SELECT ISNULL(Person.Title, '') + ' ' + Person...	2018-04-16 18:41:43.257	1	<ShowPlanXML xmlns="http://sc

Figure 9-3: Multiple executions from the plan cache.

Of course, I could have opted, in Listing 9-3, to return many other columns containing useful execution statistics, such as the aggregated physical and logical reads and writes, and CPU time, resulting from all executions of each plan, since that information was stored in the cache.

The cost of excessive plan compilation

One of the worst offenders for misuse of the plan cache is the unnecessary overuse of ad hoc, unparameterized queries. These are sometimes generated dynamically by a poorly-written application library, or by an incorrectly-configured Object-Relational Mapping (ORM) layer between the application and the database. You also see a lot more plan compiles when an ORM tool is coded poorly so that it creates different parameter definitions based on the length of the string being passed, for example `VARCHAR(3)` for 'Dog' or `VARCHAR(5)` for 'Horse'.

Dynamic SQL is any SQL declared as a string data type, and an ad hoc query is any query where the query text gets submitted to SQL Server directly, rather than being included in a code module (stored procedure, scalar user-defined function, multi-statement user-defined function, or trigger). Examples include unparameterized queries typed in SSMS, and dynamic SQL queries submitted through `EXEC(@sql)` or through `sp_excutesql`, as well as any query that is submitted and sent from a client program, which may be parameterized, in a prepared statement, or may just be an unparameterized string, depending on how the client code is built.

In extreme cases, unparameterized queries run iteratively, row by row, instead of a single set-based query. Listing 9-4 uses our previous query in a couple of iterations. The first iteration hard codes the `@id` value (for `BusinessEntityID`) into a dynamic SQL string and passes the string into the `EXECUTE` command.

The second iteration uses the `sp_executesql` procedure to create a prepared statement containing a parameterized string, to which we pass in parameter values. This approach allows for plan reuse. Don't worry too much about the details here, as we'll discuss prepared statements later in the chapter. The key point here is that we want to compare the work performed by SQL Server to execute the same ad hoc SQL multiple times, in one case where it can't reuse plans, and in one where it can.

Of course, both iterative approaches are still highly inefficient, given that we can achieve the desired result set in a set-based way, with a single execution of one query.

```
DECLARE @ii INT;
DECLARE @IterationsToDo INT = 500;
DECLARE @id VARCHAR(8);
SELECT @ii = 1;
WHILE @ii <= @IterationsToDo
```

```
BEGIN
    SELECT @ii = @ii + 1,
           @id = CONVERT(VARCHAR(5), @ii);
    EXECUTE ('SELECT ISNULL(Title, '''') + '' '' + FirstName + ''
'' + LastName FROM Person.Person WHERE BusinessEntityID =' + @id);
END;
GO
DECLARE @ii INT;
DECLARE @IterationsToDo INT = 500;
DECLARE @id VARCHAR(8);
SELECT @ii = 1;
WHILE @ii <= @IterationsToDo
BEGIN
    SELECT @ii = @ii + 1,
           @id = CONVERT(VARCHAR(5), @ii);
    EXEC sys.sp_executesql N'
  SELECT ISNULL(Title, '''') + '' '' + FirstName + '' '' + LastName
FROM Person.Person WHERE BusinessEntityID = @id',
N'@id int',
@id = @ii;
END;
GO
```

Listing 9-4

If you capture performance metrics using Extended Events, you'll see that the first iteration performs about 3,500 logical reads and takes 368,890 microseconds, the second performs 1,500 logical reads and takes 26,329 microseconds. Note that `STATISTICS IO` doesn't show the extra work; you see only work done directly by the query, not the extra work done on behalf of the query, for plan cache management.

The approach, using ad hoc, dynamic, unparameterized strings, floods the plan cache with 500 single-use copies of the same plan (you can run Listing 9-2 to verify). The extra logical reads this requires, over the iterative approach that reuses the plan, is extra work associated with compiling and storing these plans. It's only an extra 4 logical reads per iteration, but if your system is inundated with unparameterized ad hoc queries, all this extra work adds up quickly.

It causes bigger problems, too. It increases the amount of CPU processing the server must perform, in continuously and unnecessarily compiling and storing new plans. It also wastes memory resources, using buffer cache memory to store plans that will only ever be used once. Unless you have the luxury of enough server memory to accommodate every parameter combination of every query, it can lead to "cache churn," where older plans, ones that might be useful, reusable plans, are continuously evicted to make room for the flood of ad hoc query plans. In severe cases, it can lead to memory pressure.

If you're experiencing such problems, there are various ways to query the plan cache to confirm or disprove that it's related to excessive use of ad hoc queries. For example, the simple query in Listing 9-5 will tell you the proportion of each type of compiled plan in the cache.

```
SELECT decp.objtype,
       CAST(100.0 * COUNT(*) / SUM(COUNT(*)) OVER () AS DECIMAL(5,
2)) AS plans_In_Cache
FROM sys.dm_exec_cached_plans AS decp
GROUP BY decp.objtype
ORDER BY plans_In_Cache;
```

Listing 9-5

The results of this query don't mean much, as a one-off execution. You will need to monitor the values over time, and understand what the expected numbers are for your system, alongside metrics such as **Batch Requests/sec** and **SQL Compilations/sec**, using Perfmon, or track events directly with Extended Events. You can also retrieve the plan types from the Query Store.

Various online resources provide more detailed scripts to examine use and abuse of the plan cache; see, for example, https://bit.ly/2EfYOkl.

Simple parameterization for "trivial" ad hoc queries

For very simple, one-table queries, the optimizer might recognize that, if a query supplied a parameter instead of a literal, it would be able to create an execution plan it could reuse. In such cases, the optimizer will try to automatically create a parameter for you, through a process called "simple parameterization." This only works for execution plans that qualify as trivial plans (see Chapter 1), because it is only for these that the optimizer can be certain that the same plan will work well, regardless of the parameter value supplied.

Simple parameterization in action

We encountered simple parameterization back in Chapter 2, but didn't cover it in any detail, so let's see it in action again. Execute Listing 9-6 and capture the actual plan.

```sql
SELECT a.AddressID,
       a.AddressLine1,
       a.City
FROM Person.Address AS a
WHERE a.AddressID = 42;
```

Listing 9-6

Figure 9-4 shows the very simple execution plan. I've highlighted the first, visible indication that the optimizer has performed simple parameterization. You can see the query that is highlighted is different than the query I wrote and executed, because the hard-coded value for `AddressID` has been replaced by a parameter called `@1`.

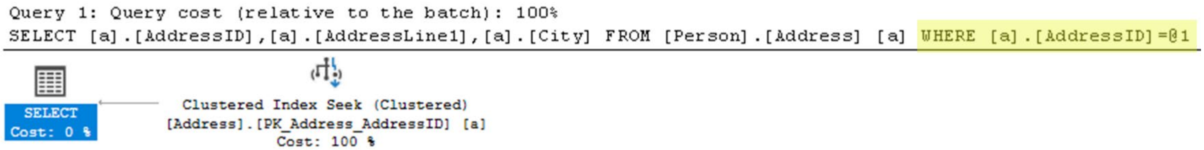


Figure 9-4: First visual evidence of simple parameterization.

If the query text is longer, you might not see this clue in the graphical execution plan. The best place to look is in the properties of the **SELECT** operator, specifically the **Parameter List**.

Optimization Level	TRIVIAL
⊞ OptimizerHardwareDependentProperties	
⊟ Parameter List	@1
Column	@1
Parameter Compiled Value	(42)
Parameter Data Type	tinyint
Parameter Runtime Value	(42)
ParentObjectId	0
QueryHash	0xFEF15E0EF2341AB8
QueryPlanHash	0xFC03D880C5ECF2D1
⊞ QueryTimeStats	
RetrievedFromCache	true
SecurityPolicyApplied	False
⊞ Set Options	ANSI_NULLS: True, ANSI_PADDIN
Statement	SELECT [a].[AddressID],[a].[Addr
StatementParameterizationType	2

Figure 9-5: SELECT properties showing evidence of simple parameterization.

Just as we see for stored procedures, or any other parameterized query, the **Parameter List** shows the name of any parameters, their compile-time and runtime values, and their data types. We have no control over the naming of these parameters; they will be simply listed in the order that the optimizer creates them. We also have no control over the data types; the optimizer chooses the data type for simple parameterization based on the size of the value passed to it. You can also see that the query engine respected the parameterization, by looking at the value at the bottom of Figure 9-5, **StatementParameterizationType**. If this value is 0, no parameterization occurred. In this case the value is 2, indicating simple parameterization.

Re-execute Listing 9-6, but with a hard-coded value of 100, and you'll see that the compile-time value remains at 42, but the runtime value changes to 100. If we query `sys.dm_exec_cached_plans` (see Listing 9-2), then we see the following output.

	usecounts	objtype	plan_handle	size_in_bytes	Datab...	Obj...	text	query_plan
1	1	Adhoc	0x06000500F...	16384	Adve...	N...	SELECT a.AddressID, a.AddressLine1, ...	<ShowPlanXML xmlns...
2	1	Adhoc	0x060005009...	16384	Adve...	N...	SELECT a.AddressID, a.AddressLine1, ...	<ShowPlanXML xmlns...
3	2	Prepared	0x060005004...	40960	Adve...	N...	(@1 tinyint)SELECT [a].[AddressID],[a].[AddressL...	<ShowPlanXML xmlns...

Figure 9-6: Plan usecounts from the plan cache.

The bottom entry in the output shows that the optimizer reused the existing plan that it created for the auto-parameterized query, effectively turning it into a prepared statement. In the **text** column, we can see the parameter it used (`@1`) and its data type, in this case `tinyint`. For integers, the optimizer uses the smallest data type that can fit the value. If we'd passed in a value of, say, 300 instead of 42, then the data type would be a `smallint` instead of a `tinyint`. This can mean that even when simple parameterization occurs, we can still have more than one plan in cache for the same trivial query, but with differences in the size of the parameter. This is not a major concern, but it's something to be aware of.

The first two entries in Figure 9-6 are for the individual ad hoc queries (with hard-coded literals). However, if you click on the links to the query plans for each of these entries, you'll see that they consist only of a **SELECT** operator. The first thing SQL Server does when we issue a query is search for an exact textual match in the plan cache. This is done before simple parameterization, and obviously requires that the pre-parameterization query is stored. However, these "placeholder" plans are never completed or executed. You can confirm this by querying `sys.dm_exec_query_stats` (Listing 9-3), which shows just a single plan for this query, executed twice.

You can also use the Query Store to retrieve the execution counts, compile counts, the type of plan, and the type of parameterization. Listing 9-7 shows the information available.

```
SELECT qsqt.query_sql_text,
       qsq.query_parameterization_type_desc,
       qsq.count_compiles,
       qsp.is_trivial_plan,
       qsrs.count_executions
FROM sys.query_store_query AS qsq
    JOIN sys.query_store_query_text AS qsqt
       ON qsqt.query_text_id = qsq.query_text_id
    JOIN sys.query_store_plan AS qsp
       ON qsp.query_id = qsq.query_id
    JOIN sys.query_store_runtime_stats AS qsrs
       ON qsrs.plan_id = qsp.plan_id
WHERE qsqt.query_sql_text LIKE '%@1%';
```

Listing 9-7

The results would look like Figure 9-7.

query_sql_text	query_parameterization_type_desc	count_compiles	is_trivial_plan	count_executions
(@1 tinyint)SELECT [a].[AddressID],[a].[AddressL...	Simple	1	1	2

Figure 9-7: Results from the Query Store showing multiple executions.

The optimizer must be sure that any possible query that could use the auto-parameterized plan will be executed safely, and it won't apply it in cases that could cause plan instability. In short, it is very cautious in its application of simple parameterization, and is easily deterred.

As noted earlier, a prerequisite is that the plan is trivial, as it was for our query in Listing 9-6, and as indicated by an **Optimization Level** of **TRIVIAL** in Figure 9-5 and the `is_trivial_plan` indicator in Figure 9-7. However, that doesn't mean any trivial plan will be auto-parameterized. If you capture the actual plans for the queries in Listing 9-1, and check the properties of the **SELECT** operator, you'll see that they also get trivial plans, but you'll see no parameter list. In this case, simple parameterization is defeated by our inclusion of the `ISNULL` function in the query (remove it, and it works). In Chapter 3 (Listing 3-4), we saw a similar case, where simple parameterization was defeated by use of a `LIKE` predicate.

What happens if we need to join to another table in our query?

```
SELECT a.AddressID,
       a.AddressLine1,
       a.City,
       bea.BusinessEntityID
FROM Person.Address AS a
    JOIN Person.BusinessEntityAddress AS bea
        ON bea.AddressID = a.AddressID
WHERE a.AddressID = 42;
```

Listing 9-8

Figure 9-8 shows the relevant properties from the resulting plan. As you can see, the **Optimization Level** will be **FULL**, rather than **TRIVIAL**. Since a trivial plan is a pre-condition of simple parameterization, we'll see no parameters.

Optimization Level	FULL
OptimizerHardwareDependentProperties	
Physical Operation	
QueryHash	0xE0697EE0DD488957
QueryPlanHash	0xC0E0E174A73E1CEC
Reason For Early Termination Of Statement	Good Enough Plan Found

Figure 9-8: SELECT properties showing the Optimization Level.

There are many other clauses and conditions that will defeat simple parameterization if included in a query, such as `GROUP BY`, `DISTINCT`, `TOP`, `UNION`, `INTO`, `BULK INSERT`, `COMPUTE`, and others. For more details, refer to Microsoft documentation at https://bit.ly/2LS6Api.

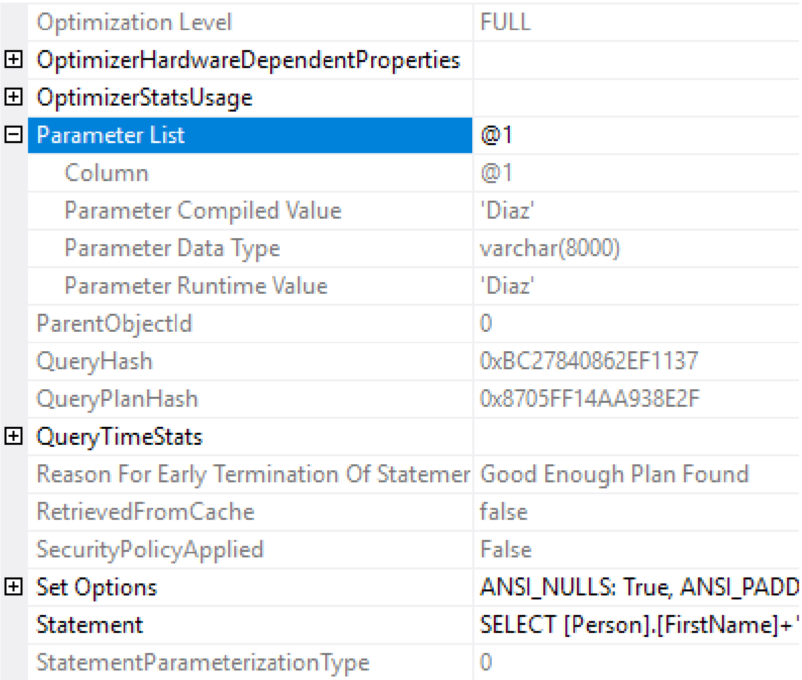
"Unsafe" simple parameterization

Simple parameterization, and the rules that govern it, are not quite as simple as they might seem. Try capturing an actual plan for the query in Listing 9-9.

```
SELECT Person.FirstName + ' ' + Person.LastName,
       Person.Title
FROM Person.Person
WHERE Person.LastName = 'Diaz';
```

Listing 9-9

The properties of the **SELECT** operator do show a **Parameter List**, apparently indicating that the optimizer did simple parameterization. But did it? Look higher, and you'll see that the **Optimization Level** is **FULL**, and earlier I said that **TRIVIAL** was a prerequisite for simple parameterization.



Optimization Level	FULL
⊞ OptimizerHardwareDependentProperties	
⊞ OptimizerStatsUsage	
⊟ Parameter List	@1
Column	@1
Parameter Compiled Value	'Diaz'
Parameter Data Type	varchar(8000)
Parameter Runtime Value	'Diaz'
ParentObjectId	0
QueryHash	0xBC27840862EF1137
QueryPlanHash	0x8705FF14AA938E2F
⊞ QueryTimeStats	
Reason For Early Termination Of Statemer	Good Enough Plan Found
RetrievedFromCache	false
SecurityPolicyApplied	False
⊞ Set Options	ANSI_NULLS: True, ANSI_PADD
Statement	SELECT [Person].[FirstName]+'
StatementParameterizationType	0

Figure 9-9: SELECT properties showing parameterization, but not really.

In fact, simple parameterization has not occurred. Change 'Diaz' to 'Brown' in Listing 9-9, rerun it, and then query either the `sys.dm_exec_cached_plans` or `sys.dm_exec_query_stats` DMO. You will see two plans, one for each execution, each unparameterized. We can also see that the `StatementParameterizationType` property, only visible if Query Store is enabled in the database, and a value only found in actual plans because it's a runtime metric, is set to the value of 0. This indicates that no parameters were used in the execution of the query.

The query plan as captured in the Query Store also shows the attempt to parameterize, including the parameterized version of the statement. However, the `query_parameterization_type` column value will be zero, indicating that there was no parameterization, and the `query_sql_text` column shows the original query text, not the parameterized version it would show if the parameterization had been successful.

Not all details of the simple parameterization process are fully documented, so the following is merely an "educated speculation," based on current understanding and observations. It appears that there are two phases. The first phase, prior to actual compilation, looks at only the query text to determine whether the query might qualify for simple parameterization. A long list of keywords is checked and, if none of them occur in the query, it will be parameterized and handed to the optimizer. Otherwise the query is sent to the optimizer unchanged, with all the constants in place.

The optimizer will, as always, first check whether **TRIVIAL** optimization applies. Apart from the same list of keywords checked for simple parameterization, this now also considers other database objects such as constraints, indexes, and so on. At this stage, the optimizer might conclude that simple parameterization is unsafe. The parameterization is undone and the original, unparameterized query is compiled.

Unfortunately, this series of events results in SSMS showing (and Query Store capturing) the execution plan as if it were parameterized. The fact that the `StatementParameterizationType` property has a value of zero (see Figure 9-9) is the only indicator that the displayed execution plan is not the plan that was used.

Of course, when a query does qualify for simple parameterization in the first check, and then also qualifies for trivial optimization in the second check, the parameterized version of the query will be compiled, and all plans shown in SSMS, in Query Store, and in the DMOs, will show the parameterized version.

If you simply omit the `Title` column from Listing 9-9 and rerun it, you'll see that simple parameterization now succeeds.

Inclusion of the `Title` column, in Listing 9-9 necessitated a **Key Lookup**, which means that is a threshold at which a clustered index scan is the better option; without `Title`, the index is covering and will always be used. Probably, this explains why simple parameterization is now "safe."

Finally, you'll see from the **Parameter Data Type** value that, for strings, the optimizer chooses a very long maximum length, and so will be able to reuse this plan for input strings that are much longer.

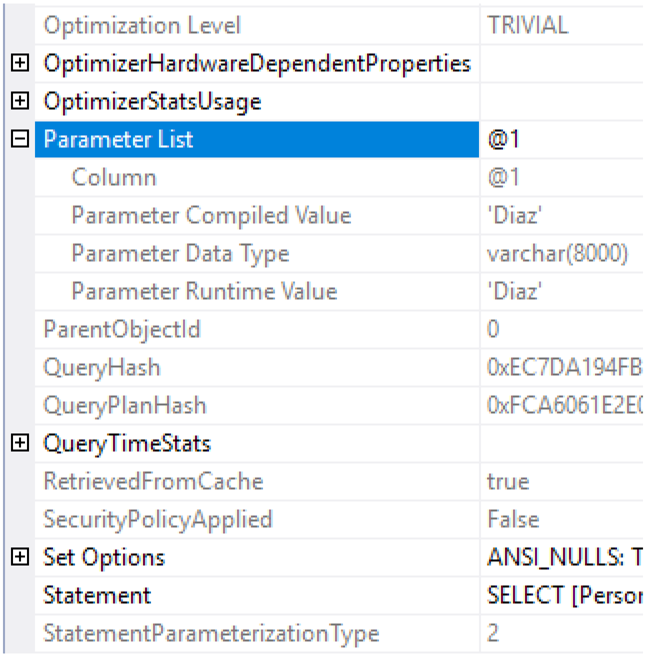


Figure 9-10: SELECT properties showing successful simple parameterization.

Programming for Plan Reuse: Parameterizing Queries

As we saw earlier, if we simply hardcode values directly into a dynamic SQL string and then pass it directly into SQL Server for execution using the `EXECUTE` command, or by any other method, the optimizer cannot reuse a cached plan for a subsequent execution where the SQL string differs only by the coded value. While plan reuse is our focus here, a far bigger problem with this approach is its vulnerability to SQL Injection attacks. I cannot cover the latter topic here, but will happily refer you to Erland Sommarskog (http://www.sommarskog.se/dynamic_sql.html) for further details.

To avoid this vulnerability when issuing dynamic SQL, and to ensure your plans will be reused rather than regenerated each time, we need to parameterize the SQL text, so that the optimizer sees the exact same SQL text each time you execute the query. However, as the previous discussion indicates, we can't rely on the optimizer's simple parameterization for anything other than the most trivial queries, and sometimes not even those.

As T-SQL coders, we need to promote plan reuse, by using parameters in our queries. From application code, we can do this by creating a **prepared statement**, using the ODBC ADO.NET and OLEDB APIs. This parameterizes the query, and then we pass in the parameter values, for each execution of the parameterized SQL text.

In SQL Server, the best approach, especially for more complex queries to which we need to pass parameters in (and out), and that we wish to reuse, we use code modules such as stored procedures or functions. However, we can also create prepared statements using `sp_executesql`, or even `sp_prepare`.

Prepared statements

Listing 9-10 shows how to create a parameterized statement in SQL Server using `sp_executesql` (see Listing 9-4 for another example).

```
DECLARE @sql NVARCHAR(400);
DECLARE @param NVARCHAR(400);
SELECT @sql =
  N'SELECT  p.Name,
            p.ProductNumber,
            th.ReferenceOrderID
    FROM    Production.Product AS p
    JOIN    Production.TransactionHistory AS th
            ON th.ProductID = p.ProductID
    WHERE   th.ReferenceOrderID = @ReferenceOrderID;';
SELECT @param = N'@ReferenceOrderID int';
EXEC sys.sp_executesql @sql, @param, 53465;
```

Listing 9-10

When SQL Server compiles the batch containing the prepared statement, it will set the values of any variables, and then run the `EXECUTE` command, and at this point can sniff the parameter values. This means that it can use statistics to come up with a very accurate row count estimate for the predicate (72 rows). Figure 9-11 shows the resulting plan.

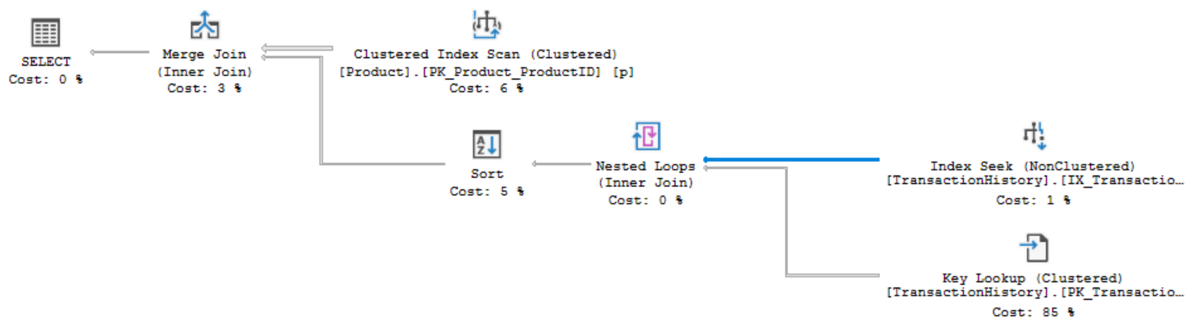


Figure 9-11: Execution plan showing sniffed parameters.

In a similar fashion, Listing 9-11 shows how to define parameters through prepared statements in your application (this example uses C#), making use of the API of OLEDB or ODBC.

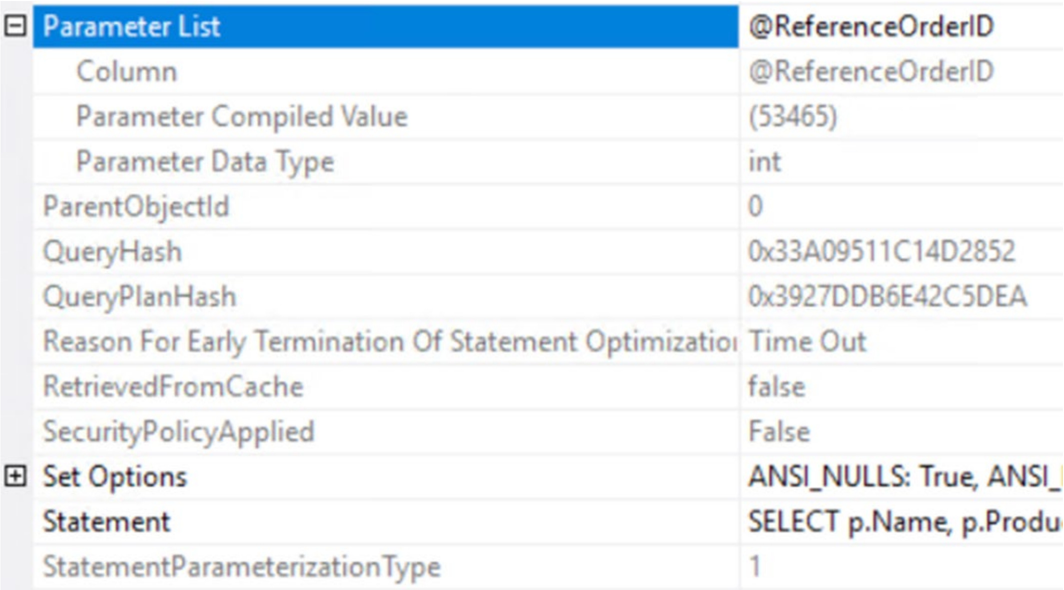
```
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.SqlClient;
namespace ExecuteSQL
{
    class Program
    {
        static void Main(string[] args)
        {
            string connectionString = "Data Source=MySQLInstance;Da
tabase=AdventureWorks2014;Integrated Security=true";
            try
            {
                using (SqlConnection myConnection = new SqlConnecti
on(connectionString))
                {
                 myConnection.Open();
                 SqlCommand prepStatement = myConnection.
CreateCommand();
                 prepStatement.CommandText = @"SELECT p.Name,
p.ProductNumber,
```

```
                                         th.ReferenceOrderID
                                         FROM Production.Product AS p
                                         JOIN Production.TransactionHistory
AS th
                                         ON th.ProductID = p.ProductID
                                         WHERE th.ReferenceOrderID = @
ReferenceOrderID";
                prepStatement.Parameters.Add("@ReferenceOrderID,"
SqlDbType.Int);
                prepStatement.Prepare();
                prepStatement.Parameters["@ReferenceOrderID"].Value
= 53465;
                prepStatement.ExecuteReader ();
                }
            }
            catch (SqlException e)
            {
                Console.WriteLine(e.Message);
                Console.Read();
            }
        }
    }
}
```

Listing 9-11

If you execute this and examine the plan cache (Listing 9-2 or 9-3) you'll find the plan shown in Figure 9-11. If you look at the **SELECT** operator as we have done throughout this chapter, you'll see that the `@ReferenceOrderID` was parameterized and that the value was sniffed, with a compile value of 53465 and that the `StatementParameterization-Type` has a value of 1, which means the user explicitly parameterized the query, as shown in Figure 9-12.

Different types of prepared statement behave differently. A .NET application can build a query in a `StringBuilder` object, then prepare and execute it; technically, that's a prepared statement, but it would have all the characteristics of dynamic SQL.



⊟ Parameter List	@ReferenceOrderID
Column	@ReferenceOrderID
Parameter Compiled Value	(53465)
Parameter Data Type	int
ParentObjectId	0
QueryHash	0x33A09511C14D2852
QueryPlanHash	0x3927DDB6E42C5DEA
Reason For Early Termination Of Statement Optimizatio	Time Out
RetrievedFromCache	false
SecurityPolicyApplied	False
⊞ Set Options	ANSI_NULLS: True, ANSI_
Statement	SELECT p.Name, p.Produ
StatementParameterizationType	1

Figure 9-12: SELECT properties showing the prepared statement parameterization.

Similarly, we can also create a prepared statement in SQL using the built-in `sp_prepare` stored procedure, although there's not much practical need for it and, again, it behaves somewhat differently.

```
DECLARE @sql NVARCHAR(400);
DECLARE @param NVARCHAR(400);
DECLARE @PreparedStatement INT;
DECLARE @MyID INT;
SELECT @sql =
  N'SELECT  p.Name,
            p.ProductNumber,
            th.ReferenceOrderID
    FROM    Production.Product AS p
    JOIN    Production.TransactionHistory AS th
            ON th.ProductID = p.ProductID
    WHERE   th.ReferenceOrderID = @ReferenceOrderID;';
SELECT @param = N'@ReferenceOrderID int';
SELECT @MyID = 53465;
EXEC sp_prepare @PreparedStatement OUTPUT, @param, @sql;
EXEC sp_execute @PreparedStatement, @MyID;
EXEC sp_unprepare @PreparedStatement;
```

Listing 9-12

Using this technique, the compilation occurs in two steps: first, prepare (without values) and then execute (with values). The plan is generated during the prepare step and, since there are no values, the parameters cannot be sniffed and are treated as normal local variables. This is different than what we showed with the C# code from Listing 9-11.

Therefore, prepared statements created in this fashion always cause optimization for unknown values, and so the optimizer will use the density graph to arrive at a cardinality estimation, in this case 3.05 rows, and will generate an appropriate plan, which is rather different from the one we saw in Figure 9-10. You'll have to clear the cache to see this plan, else you'll see a reuse of the plan for Listing 9-9, because the SQL text is identical in each case.

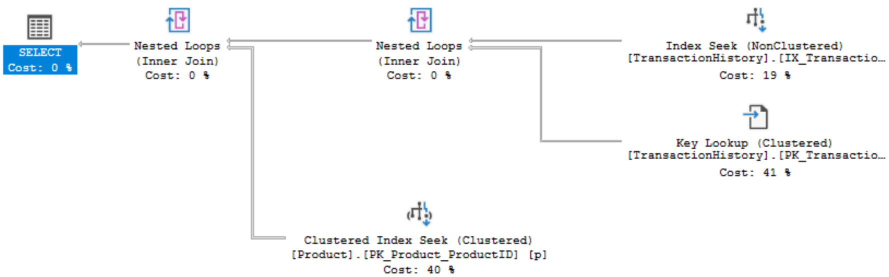


Figure 9-13: Execution plan from the sp_prepare statement query.

This is the same plan as we'd see if we'd simply set the value of `@ReferenceOrderID` using a local variable (`DECLARE @ReferenceOrderID INT`). While it may look like a parameter, the two behave differently and are handled in different ways by the optimizer, as we saw in Chapter 8.

In this case, the efficiency of this plan will decrease, the more rows are returned by the top inputs into each of the **Nested Loops** joins. However, in this case, it isn't a significant performance issue, and the plan is good enough for all values that can be passed in.

As we saw, when we parameterize SQL using `sp_executesql`, use a code-based prepared statement, or a stored procedure based on this query, we get the optimizer plan for the sniffed parameter value, but we may see erratic performance as a result.

Stored procedures

We've already seen plenty of examples in this book, especially in Chapter 7, of encapsulating a parameterized query in a stored procedure. When you call a stored procedure, a plan is created and placed in a cache that is associated with the object ID of the procedure. This makes plan reuse straightforward and simple, both to work with and to understand.

Listing 9-13 uses the same query as the previous two listings, but this time in a stored procedure.

```
CREATE OR ALTER PROC dbo.ProductTransactionHistoryByReference (@
ReferenceOrderID INT)
AS
BEGIN
    SELECT p.Name,
           p.ProductNumber,
           th.ReferenceOrderID
    FROM Production.Product AS p
        JOIN Production.TransactionHistory AS th
            ON th.ProductID = p.ProductID
    WHERE th.ReferenceOrderID = @ReferenceOrderID;
END
GO
```

Listing 9-13

I can execute the stored procedure with the command in Listing 9-14.

```
EXEC dbo.ProductTransactionHistoryByReference @ReferenceOrderID =
41798;
```

Listing 9-14

A big advantage of investigating cached plans for stored procedures is that I can now retrieve its plan directly from cache. In this case, it will be the plan that is optimized for low estimated row counts, where the leftmost join is a **Nested Loops** (Figure 9-13).

```
SELECT DB_NAME(deps.database_id) AS DatabaseName,
       deps.cached_time,
       deps.min_elapsed_time,
       deps.max_elapsed_time,
       deps.last_elapsed_time,
       deps.total_elapsed_time,
       deqp.query_plan
FROM sys.dm_exec_procedure_stats AS deps
    CROSS APPLY sys.dm_exec_query_plan(deps.plan_handle) AS deqp
WHERE deps.object_id = OBJECT_ID('AdventureWorks2014.dbo.ProductTra
nsactionHistoryByReference');
```

Listing 9-15

This query will return all the various runtimes (in microseconds), which are stored with the cached plan and are updated for as long as the object remains in cache, and doesn't get recompiled. The `cached_time` shows when the object was added to the cache. Figure 9-14 shows the results of running Listing 9-15, after two executions of Listing 9-14.

	DatabaseName	cached_time	min_elapsed_time	max_elapsed_time	last_elapsed_time	total_elapsed_time	query_plan
1	AdventureWorks2014	2018-05-30 14:53:15.737	80	6900	80	6980	<ShowPlanXML xmlns="http://scl

Figure 9-14: Execution metrics of the stored procedure.

The compile time is included in the `*_elapsed_time` metrics, so the first execution (6900 microseconds) is substantially slower than the second (80). If we execute the procedure a third time, but with a parameter value of 53465, you'll see that the `last_elapsed_time` is longer (about 12 K microseconds, in my case) because the plan optimized for returning 3 rows is now returning 72. This is not a significant performance issue, but would be more of a concern if there were parameter values that returned significantly more rows.

Listing 9-15, using the `object_id` as a filter, is the best way to investigate plans for stored procedures. However, we can also examine the plans for individual statements within a stored procedure, using `sys.dm_exec_query_stats`.

```
SELECT  dest.text,
       deqp.query_plan,
       deqs.execution_count,
       deqs.max_worker_time,
       deqs.max_logical_reads,
       deqs.max_logical_writes
FROM sys.dm_exec_query_stats AS deqs
    CROSS APPLY sys.dm_exec_query_plan(deqs.plan_handle) AS deqp
    CROSS APPLY sys.dm_exec_sql_text(deqs.sql_handle) AS dest
WHERE dest.text LIKE 'CREATE PROC dbo.ProductTransactionHistoryByRe
ference%';
```

Listing 9-16

I used the `LIKE` statement, and the `'CREATE…'` filter, because the `text` column in this case shows the object definition of the procedure (or function or trigger) that was called.

What can go wrong with plan reuse for parameterized queries?

Once the optimizer generates a plan for a prepared statement or stored procedure, all subsequent executions will use that plan, until the plan is, for whatever reason, removed from cache. As we discussed briefly above, and in more detail in Chapter 8, if the distribution of rows in an index is very uneven, the optimizer will choose very different plans, depending on the parameter value supplied. In these cases, parameter sniffing can sometimes cause you performance problems.

If you can alter the query text, then the common solutions include use of various query hints, such as `OPTION (RECOMPILE)` if you want the optimizer to produce a new plan on every execution of the statement to which it's applied. For stored procedures and other code modules, all statements will still be in the plan cache, but the plan for the `OPTION(RECOMPILE)` statement will still recompile for every execution, which means that its plan is not reused. For ad hoc parameterized queries (including prepared statements), use of this hint means the plan is not stored at all. In either case, this means that you lose out on reducing recompiles, but at least you do still save space in the plan cache.

The alternative if you don't want to recompile is to use Query Store to force a plan. Another option is to use various forms of the `OPTION (OPTIMIZE FOR…)` hint, if you want the optimizer to always use a plan for specific parameter value, or to always use a "generic" plan, based on average statistics.

We'll see a few of these hints briefly later, when we discuss plan guides and plan forcing. Hints will be covered in full detail in Chapter 10, and the Query Store in Chapter 16.

Fixing Problems with Plan Reuse if You Can't Rewrite the Query

There are two distinct types of problem that we may need to fix, and that are especially hard to fix with third-party vendor code that you can't change. One is pressure on memory and CPU resources, caused by the optimizer compiling a very high volume of ad hoc query plans that it cannot reuse, because of a workload consisting of unparameterized ad hoc queries.

The second is erratic performance of parameterized queries when reusing cached plans, caused by cases of "bad" parameter sniffing.

Optimize for ad hoc workloads

Let's imagine that a third-party application, where you have no control over the submitted SQL text, is generating a huge number of ad hoc queries, many of which are only ever executed once. Another possibility is that an ORM tool, which should be using parameterized queries, is instead badly configured and generates ad hoc queries instead. Either of these situations results in plan cache bloat, and is a contributing factor to memory pressure on the server.

Probably the first option you should consider in this type of situation is to enable the *server-wide* setting `optimize for ad hoc workloads`. I emphasize *server-wide* because this setting will affect all databases on the server, and you'll need to test its impact carefully before choosing to enable it in production. Starting with SQL Server 2016, though, you can use the database scoped configuration settings to enable, or disable, this setting at the database level.

With this setting enabled, the query optimizer still optimizes each query in the usual way, but with one critical difference. Rather than immediately storing a plan in cache, it instead stores a plan stub, or placeholder. If the same query is executed a second time, then the plan must be compiled again, and now it is added to the cache for future reuse. This reduces significantly the amount of memory the plan cache uses for managing execution plans that are only ever executed once, at the cost of one additional compile for queries that are called more than once.

Listing 9-17 initializes the `optimize for ad hoc workloads` setting, and then clears out the entire plan cache. I'm using the `DBCC` command just for demonstration purposes. It's better to either use targeted plan cache removal by passing a plan handle, or to only remove plans for a single database using `ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE`.

```
EXECUTE sp_configure 'show advanced options', '1';
RECONFIGURE;
GO
EXECUTE sp_configure 'optimize for ad hoc workloads', 1;
RECONFIGURE;
DBCC FREEPROCCACHE;
GO
```

Listing 9-17

Listing 9-18 shows how to initialize the setting at the database level in Azure SQL Database, using database scoped configuration changes.

```
ALTER DATABASE SCOPED CONFIGURATION SET OPTIMIZE_FOR_AD_HOC_
WORKLOADS = ON;
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
```

Listing 9-18

To see `optimize for ad hoc` in action, let's execute a query. This one uses several literals in a search to find email addresses that start with "david" belonging to people from the state of Washington.

```
SELECT 42 AS TheAnswer,
       em.EmailAddress,
       a.City
FROM Person.BusinessEntityAddress AS bea
    JOIN Person.Address AS a
        ON bea.AddressID = a.AddressID
    JOIN Person.StateProvince AS sp
        ON a.StateProvinceID = sp.StateProvinceID
    JOIN Person.EmailAddress AS em
        ON bea.BusinessEntityID = em.BusinessEntityID
WHERE em.EmailAddress LIKE 'david%'
      AND sp.StateProvinceCode = 'WA';
```

Listing 9-19

Figure 9-15 shows the actual execution plan. If you were to inspect the properties of the **SELECT** operator, you'd see that the text of the **Statement** is identical to the text we submitted, and there is no **Parameter List**. In other words, no parameterization occurred.

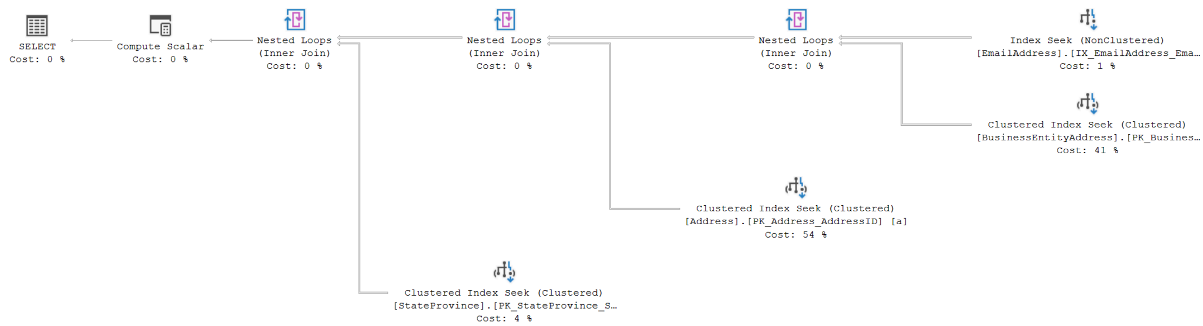


Figure 9-15: Execution plan for the query in Listing 9-19.

Now let's see what's in the plan cache, by querying `sys.dm_exec_cached_plans`. I used the query in Listing 9-2, adapted slightly so that it also returns the `cp.size_in_bytes` column.

	usecounts	objtype	plan_handle	size_in_bytes	DatabaseN...	Obje...	text	query_plan
1	1	Adhoc	0x06000500943A6D27603A99DCEE...	424	Adventure...	NULL	SELECT 42 AS TheAns...	NULL

Figure 9-16: Output from sys.dm_exec_cached_plans showing no execution plan.

Having enabled `optimize for ad hoc workloads`, and run this ad hoc for the first time, the optimizer compiles the plan, but it doesn't store it in the plan cache. There is just a small (424 byte) plan "stub" with an associated `plan_handle`.

If you were to run Listing 9-19 one more time and re-query `sys.dm_exec_cached_plans`, the results will be different. The optimizer has compiled the plan again, and this time stored it.

usecounts	objtype	plan_handle	size_in_bytes	DatabaseN...	Obje...	text	query_plan
1	Adhoc	0x06000500943A6D27203299DCEE...	73728	Adventure...	NULL	SELECT 42 AS TheAn...	<ShowPlanXML xmlns="h...

Figure 9-17: Output from sys.dm_exec_cached_plans with an execution plan.

Notice that the **usecount** didn't go up by one, because this is effectively a new query plan in cache. Subsequent executions of the same query will result in the execution count ticking over as normal, with no further compilations. If we execute the same query, but this time looking for emails starting with "paul" then we'll see a new "stub" entry for that query, then a normal plan the next time the exact same text is submitted.

Before we move on, let's disable the setting to avoid confusion.

```
EXECUTE sp_configure 'show advanced options', 1;
RECONFIGURE;
GO
EXECUTE sp_configure 'optimize for ad hoc workloads', 0;
RECONFIGURE;
GO
EXECUTE sp_configure 'show advanced options', 0;
RECONFIGURE;
GO
```

Listing 9-20

Forced parameterization

The `'optimize for ad hoc workloads'` reduces the memory required in the plan cache for plans that will only ever be used once, but it does not help promote plan reuse. If your OLTP system is subject to a heavy workload comprising ad hoc queries, and the sheer number of plan compilations is contributing heavily to existing CPU pressure, then you may need a different approach. If you can't rewrite the queries to parameterize them, then you may consider using forced parameterization, although there can be substantial drawbacks, as we'll discuss later in this section.

We saw earlier that the optimizer applies simple parameterization very cautiously, occasionally replacing literals with parameters, in trivial plans, based on a complex set of rules.

If we enable forced parameterization, then the optimizer attempts to replace *all* literal values with a parameter, with the following important exceptions (among others, see https://bit.ly/2JhrIb2):

- literals in the select list of any `SELECT` statement are not replaced
- parameterization does not occur within individual T-SQL statements inside stored procedures, triggers, and UDFs, which get execution plans of their own
- The *pattern* and *escape_character* arguments of a `LIKE` clause
- XQuery literals are not replaced with parameters.

Normally, forced parameterization is set at the database level, by setting the `PARAMETERIZATION` option to `FORCED`, and will apply to all queries on that basis. You also have the option of choosing to set it only for a single query using the query hint, `PARAMETERIZATION FORCED`, but this hint is only available as a plan guide, which we cover later in this chapter.

Listing 9-21 shows a simple ad hoc query like the one we encountered earlier in the chapter, and which does not get simple parameterization.

```
SELECT ISNULL(Person.Title, '') + ' ' + Person.FirstName + ' ' +
Person.LastName
FROM Person.Person
WHERE Person.BusinessEntityID = 278;
```

Listing 9-21

Figure 9-18 shows the results of running Listing 9-3, to see what's in the plan cache.

	text	creation_time	execution_count	query_plan
1	SELECT ISNULL(Person.Title, '') + ' ' + Person.First...	2018-05-25 17:36:42.703	1	<ShowPlanXML xmlns="http://schemas.microsoft.com...

Figure 9-18: Query without parameterization from cache.

Let's now enable forced parameterization and clean out the buffer cache, which happens automatically when you change the parameterization option.

```
ALTER DATABASE AdventureWorks2014 SET PARAMETERIZATION FORCED;
GO
```

Listing 9-22

Now run Listing 9-21 again. If you capture the actual plan and examine the properties of the **SELECT** operator, you'll see that, this time, they were parameterized. We see a **Parameter List**, and a `StatementParameterizationType` of 3, indicating forced parameterization.

Optimization Level	TRIVIAL
⊞ OptimizerHardwareDependentPr	
⊟ Parameter List	@0
Column	@0
Parameter Compiled Value	(278)
Parameter Data Type	int
Parameter Runtime Value	(278)
ParentObjectId	0
QueryHash	0x10C57723825D609D
QueryPlanHash	0x1A5FBD082F13EED3
⊞ QueryTimeStats	
RetrievedFromCache	true
SecurityPolicyApplied	False
⊞ Set Options	ANSI_NULLS: True, ANSI_PADD
Statement	select IsNull (Person . Title , ''
StatementParameterizationType	3

Figure 9-19: SELECT properties showing that forced parameterization occurred.

Just as for simple parameterization, with forced parameterization we still have no control over the parameter names, which are just based on the order in which parameters are created, which in turn is driven by the order in which the literal values appear in the query. Crucially, we can't control the data types picked for parameterization, either.

Figure 9-20 shows the plan cache after executing Listing 9-21 one more time, but with a different literal value, proving that the plan was reused.

text	creation_time	plan_handle	execution_count	query_plan
(@0 int)select IsNull (Person . Title , '') +...	2018-05-17 19:06:13.987	0x0600050022F26234F03799DCE...	2	<ShowPlanXML xmlns=

Figure 9-20: A parameterized query is now in cache.

Is this a good thing? For this query, yes. The plan uses a **Seek** of the clustered index, and will always produce the same plan, regardless of parameter value. However, the problem with enforcing parameterization is that it is a very blunt instrument. It will force the optimizer to parameterize all queries running on the database, for better or worse. If some queries get parameterized that otherwise would have many different plans, according to the exact value supplied then, while you might reduce compilations, you're possibly heading for bad parameter sniffing problems.

Forced parameterization also has limitations. What if your OLTP system is subject to many wildcard searches, with hard-coded literals? Rerun Listing 9-19, which contains just such a wildcard search for email addresses. You'll see that the execution plan is the same as that shown in Figure 9-10. However, the query text stored with the plan is no longer the same. It now looks as below (formatted for legibility).

```
SELECT 42 AS TheAnswer,
       em.EmailAddress,
       a.City
FROM Person.BusinessEntityAddress AS bea
    JOIN Person.Address AS a
        ON bea.AddressID = a.AddressID
    JOIN Person.StateProvince AS sp
        ON a.StateProvinceID = sp.StateProvinceID
    JOIN Person.EmailAddress AS em
        ON bea.BusinessEntityID = em.BusinessEntityID
WHERE em.EmailAddress LIKE 'david%'
      AND sp.StateProvinceCode = @0
```

Instead of the two-character string we supplied in the original query definition, the parameter `@0` is used in the comparison to the `StateProvinceCode` field. If this query is called again with a different two- or three-character state code, the plan will be reused. This could affect performance, either positively or negatively. Also, because `LIKE` is in the exception list for forced parameterization, this plan will only be reused for a search for email addresses that start with `'david'`, in any state.

As a small side note, the query stored with the plan did not include the semicolon statement terminator that I had in my original query.

Before proceeding, be sure to reset the parameterization of the database.

```
ALTER DATABASE AdventureWorks2014 SET PARAMETERIZATION SIMPLE;
GO
```

Listing 9-23

Plan guides

The `optimize for ad hoc workloads` setting and forced parameterization, at the database level, may be useful options for fixing problems related to ad hoc query workloads, especially where you don't have the option of fixing the code. However, they are both broad-reaching in their impact.

Plan guides offer you a way to control certain aspects of the optimizer's behavior, and therefore "guide" towards the plan you want, in cases where you can't modify the database code or schema. They allow us to apply valid query hints to the code, without editing the T-SQL code in any way. They're available on all SQL Server Editions except Express Edition.

We can create plan guides for stored procedures and other database objects (**object plan guides**), or for SQL statements that are not part of a database object (**SQL plan guides** and **template plan guides**). Their advantage over the `optimize for ad hoc workloads` setting and forced parameterization is that they affect only the specific objects or queries to which we apply them. I'll offer typical examples of how you might use each of these types of plan guide to tackle problems related to plan reuse (of course, they have broader applications, too).

Before we start, my customary words of caution: exercise due care when implementing plan guides, because changing how the optimizer deals with a query can degrade its performance, if used incorrectly. As I stress heavily in Chapter 10, hints and therefore plan guides, can be dangerous. They are not suggestions that the optimizer might consider, they are commands that the optimizer *must* obey. Also, any performance advantage a plan guide offers today may soon start to work against you, as the database and its data change over time.

As with hints, plan guides should be a last resort, not a standard tactic. As code, structures, or the data change, the forced plan may become suboptimal, hurting performance. Proper testing and due diligence must be observed prior to applying forcing with plan guides, or with Query Store. Then, over time, you should reevaluate the plans being forced in this fashion. Finally, plan guides are a tool for dealing with some types of issues around plan reuse, but plan forcing through the Query Store, covered later in this chapter, is now a preferred mechanism over plan guides.

You can monitor the success or failure of any of the plan guides using the Extended Events `plan_guide_successful` and `plan_guide_unsuccessful`.

Template plan guides

Let's say there are only a few problematic ad hoc queries that you'd like the optimizer to parameterize, while not affecting the optimization behaviors for any other queries on the database. In other words, you'd like a solution like forced parameterization, but localized to just those problem queries. This is where template plan guides can be useful.

Let's suppose that we decide that our query from Listing 9-17 must have its `PARAMETERIZATION` set to `FORCED`, but the query comes from vendor code that we can't edit. We can simply create a template plan guide to implement forced parameterization, just for that query, rather than changing the settings on the entire database. A template plan guide will override parameterization settings in queries.

The first step is to use the `sp_get_query_template` stored procedure to retrieve the template. We use the query text as input, and the outputs, which "mimic the parameterized form of a query that results from using forced parameterization," we store in variables and then pass to the `sp_create_plan_guide` procedure, to create the template plan guide.

The `@templatetext` output parameter will contain the parameterized form of the query text, as a string, and the `@parameters` output parameter will contain a comma-separated list of parameter names and data types.

```
DECLARE @templateout NVARCHAR(MAX),
        @paramsout NVARCHAR(MAX);
EXEC sys.sp_get_query_template @querytext = N'SELECT  42 AS
TheAnswer
       ,em.EmailAddress
       ,e.BirthDate
       ,a.City
FROM    Person.Person AS p
        JOIN HumanResources.Employee e
            ON p.BusinessEntityID = e.BusinessEntityID
        JOIN Person.BusinessEntityAddress AS bea
            ON p.BusinessEntityID = bea.BusinessEntityID
        JOIN Person.Address a
            ON bea.AddressID = a.AddressID
        JOIN Person.StateProvince AS sp
            ON a.StateProvinceID = sp.StateProvinceID
        JOIN Person.EmailAddress AS em
        ON e.BusinessEntityID = em.BusinessEntityID
WHERE   em.EmailAddress LIKE ''david%''
        AND sp.StateProvinceCode = ''WA'';',
                              @templatetext = @templateout OUTPUT,
                              @parameters = @paramsout OUTPUT;
EXEC sys.sp_create_plan_guide
    @name = N'MyTemplatePlanGuide',
    @stmt = @templateout,
    @type = N'TEMPLATE',
    @module_or_batch = NULL,
    @params = @paramsout,
    @hints = N'OPTION(PARAMETERIZATION FORCED)';
```

Listing 9-24

The input parameters for `sp_create_plan_guide` are as follows:

- `@name` – the plan guide name will operate within the context of the database, not the server, which also means the guide only works within that database.
- `@stmt` – must be an exact match to the query that the query optimizer will be called on to match, although white space and carriage returns don't matter. When the optimizer finds code that matches, it will look up and apply the correct plan guide. In this case, we supply the variable storing the `@templatetext` output.
- `@type` – the type of plan guide, in this case a template plan guide.

- `@module_or_batch` – we'd specify the name of the target object if we were creating an object plan guide. We'd supply `NULL` otherwise.
- `@params` – only applicable to template plan guides, and is a comma-separated list of parameter names and data types.
- `@hints` – specifies any hints that need to be applied, in this case `OPTION(PARAMETERIZATION FORCED)`.

Run Listing 9-24, and then rerun Listing 9-19 and you'll see that this query is now subject to forced parameterization, as indicated in the properties of the **SELECT** operator. Unlike for other types of plan guides, the template plan guide itself isn't identified within the execution plan. You can use the Extended Event `plan_guide_successful` to ensure that the plan guide was applied.

SQL plan guides

Rather than having problems with unparameterized queries, perhaps your system executes lots of parameterized queries, and you're getting performance problems with some of them, due to bad parameter sniffing.

In the earlier section on *Prepared statements*, we encountered a parameterized query where the optimizer's choice of plan depended on the input value. When the cardinality estimation was just a few rows, we saw a simple plan consisting of **Nested Loops** joins (Figure 9-13). For higher estimated rows returned, we saw a more complex-looking plan with a **Merge Join** (Figure 9-11).

We've decided that the simpler plan is the best plan for most possible input values, and so we want to apply the `OPTIMIZE FOR` hint to get that plan. However, again, we can't add a hint because we have no control over the SQL executed. This is one example of where a SQL plan guide can be useful.

One option would be to force the optimizer to produce a plan for a specific value, one that we know results in the simpler plan, for example `OPTIMIZE FOR (@ReferenceOrderID = 41798)`. However, what if the data changes and suddenly this input value returns many rows? The plan will change, and this could impact the performance of other executions of the prepared statement.

Instead, we'll create a SQL plan guide that uses the `OPTIMIZE FOR` hint with a value of `UNKNOWN` to force a more generic plan on the optimizer, based on average statistics, which results in the simple plan we want and is less susceptible to instability over time.

```
EXEC sys.sp_create_plan_guide
  @name = N'MySQLPlanGuide',
  @stmt = N'SELECT  p.Name,
            p.ProductNumber,
            th.ReferenceOrderID
    FROM    Production.Product AS p
    JOIN    Production.TransactionHistory AS th
            ON th.ProductID = p.ProductID
    WHERE   th.ReferenceOrderID = @ReferenceOrderID;',
  @type = N'SQL',
  @module_or_batch = NULL,
  @params = N'@ReferenceOrderID int',
  @hints = N'OPTION (OPTIMIZE FOR UNKNOWN)';
```

Listing 9-25

Now if we rerun the prepared statement in Listing 9-10, the optimizer will no longer do parameter sniffing and arrive at the plan with the **Merge Join**, but will instead create the plan based on average statistics, shown in Figure 9-21.

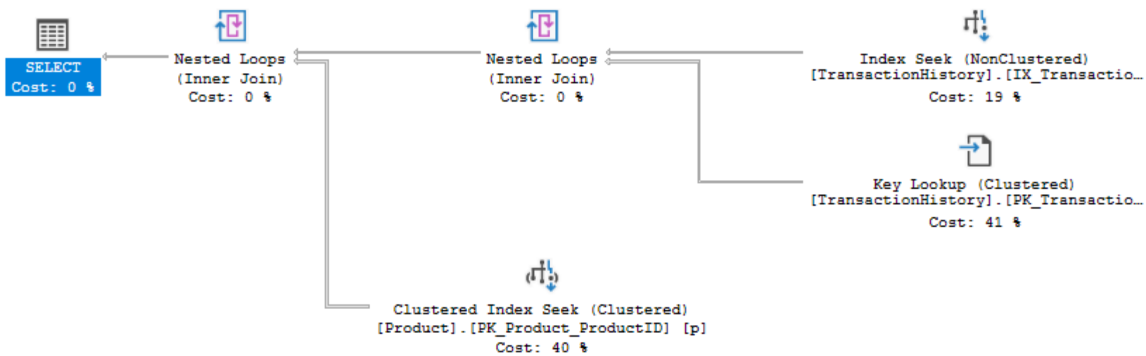


Figure 9-21: Execution plan resulting from the hint in the plan guide.

The properties of the **SELECT** operator show that the plan guide was applied.

Optimization Level	FULL
OptimizerHardwareDependentProperties	
OptimizerStatsUsage	
Parameter List	@ReferenceOrderID
ParentObjectId	0
PlanGuideDB	AdventureWorks2016
PlanGuideName	MySQLPlanGuide
QueryHash	0x0C28038006BB2B4B
QueryPlanHash	0x0168677A0A8D89FC
QueryTimeStats	
Reason For Early Termination Of Statement Optim	Good Enough Plan Found
RetrievedFromCache	true
SecurityPolicyApplied	False
Set Options	ANSI_NULLS: True, ANSI_PADD
Statement	SELECT p.Name, p.Prod
StatementParameterizationType	1

Figure 9-22: SELECT properties showing the plan guide in use.

This means you have a method to see if a plan guide was accurately applied to a stored procedure, and to identify plans where a plan guide affected the optimizer, when troubleshooting an inherited database.

Object plan guides

Perhaps your system executes lots of parameterized queries, in stored procedure form, and again you're getting performance problems with some of them, due to bad parameter sniffing. You've identified a stored procedure, `dbo.uspGetManagerEmployees` (which is a built-in stored procedure in `AdventureWorks`), where you're willing to take the hit of having SQL Server compile a plan for every execution, by applying the `RECOMPILE` hint. However, this isn't a procedure you can edit. So you decide to create an object plan guide to apply the `RECOMPILE` hint. We can only use object plan guides for queries that execute in the context of T-SQL stored procedures, scalar user-defined functions, multi-statement table-valued user-defined functions, and DML triggers.

```
EXEC sys.sp_create_plan_guide
    @name = N'MyObjectPlanGuide',
    @stmt = N'WITH [EMP_cte]([BusinessEntityID],
[OrganizationNode],
                              [FirstName], [LastName],
[RecursionLevel])
                              -- CTE name and columns
AS (
SELECT e.[BusinessEntityID], e.[OrganizationNode], p.[FirstName],
       p.[LastName], 0 -- Get initial list of Employees for Manager
n
FROM [HumanResources].[Employee] e
     INNER JOIN [Person].[Person] p
            ON p.[BusinessEntityID] = e.[BusinessEntityID]
WHERE e.[BusinessEntityID] = @BusinessEntityID
UNION ALL
SELECT e.[BusinessEntityID], e.[OrganizationNode], p.[FirstName],
       p.[LastName], [RecursionLevel] + 1
-- Join recursive member to anchor
FROM [HumanResources].[Employee] e
     INNER JOIN [EMP_cte]
            ON e.[OrganizationNode].GetAncestor(1) =
                  [EMP_cte].[OrganizationNode]
    INNER JOIN [Person].[Person] p
           ON p.[BusinessEntityID] = e.[BusinessEntityID]
)
SELECT [EMP_cte].[RecursionLevel],
       [EMP_cte].[OrganizationNode].ToString() as
[OrganizationNode],
       p.[FirstName] AS ''ManagerFirstName'',
       p.[LastName] AS ''ManagerLastName'',
       [EMP_cte].[BusinessEntityID], [EMP_cte].[FirstName],
       [EMP_cte].[LastName] -- Outer select from the CTE
FROM [EMP_cte]
     INNER JOIN [HumanResources].[Employee] e
             ON [EMP_cte].[OrganizationNode].GetAncestor(1) =
                  e.[OrganizationNode]
     INNER JOIN [Person].[Person] p
            ON p.[BusinessEntityID] = e.[BusinessEntityID]
ORDER BY [RecursionLevel], [EMP_cte].[OrganizationNode].ToString()
OPTION (MAXRECURSION 25) ',
    @type = N'OBJECT',
    @module_or_batch = N'dbo.uspGetManagerEmployees',
```

```
    @params = NULL,
    @hints = N'OPTION(RECOMPILE,MAXRECURSION 25)';
```

Listing 9-26

Again, the `@stmt` parameter must contain SQL text that is an exact match to that which the query optimizer sees (barring white space and carriage returns). Remember that a procedure could have more than one statement and you want to apply the hint to the correct one within the procedure.

This time, the `@type` parameter is a database object, and in the `@module_or_batch` parameter we specify the name of the target object.

For the `@hints` parameter, we apply the `RECOMPILE` hint, but notice that this query already had a hint, `MAX RECURSION`. That hint had also to be part of my `@stmt` in order to match what was inside the stored procedure. The plan guide replaces the existing `OPTION`, so if we need it to be carried forward, we must add it to the plan guide.

From this point forward, without making a single change to the actual definition of the stored procedure, when we execute it, the optimizer will recompile the plan for the specified query every time, and optimize it for the specific value provided. Note that you cannot alter a stored procedure that has a plan guide.

Again, you can identify that a guide has been used by looking at the **SELECT** operator of the resulting execution plan.

Viewing, validating, disabling, and removing plan guides

To see a list of plan guides within the database, just `SELECT` from the dynamic management view, `sys.plan_guides`.

```
SELECT  *
FROM    sys.plan_guides;
```

Listing 9-27

After you apply cumulative updates, upgrade your instance of SQL Server, or even deploy changes to your database, it's a good idea to ensure that your plan guides, if any, are intact. You can validate the plan guides using `fn_validate_plan_guide`.

```
SELECT pg.plan_guide_id,
       pg.name,
       fvpg.message,
       fvpg.severity,
       fvpg.state
FROM sys.plan_guides AS pg
    OUTER APPLY sys.fn_validate_plan_guide(pg.plan_guide_id) AS
fvpg;
```

Listing 9-28

The value being passed is the `plan_guide_id`, retrieved from the `sys.plan_guides` system view. If the plan guide is valid, nothing is returned. If the plan guide is invalid you'll get the first error found by the validation process. This query, then, will list all the plan guides and show any that have errors.

Aside from the procedure to create plan guides, a second one, `sp_control_plan_guide`, allows you to drop, disable (which saves the definition but stops SQL Server from using it), or enable a specific plan guide; or drop, disable, or enable all plan guides in the database.

Simply run execute the `sp_control_plan_guide` procedure, changing the `@operation` parameter appropriately. Listing 9-29 will remove all the plan guides created in this chapter.

```
EXEC sys.sp_control_plan_guide @operation = N'DROP ALL', @name =
N'*';
```

Listing 9-29

Plan forcing

There may be situations where adding hints using plan guides does not produce consistent results. While hints dictate how the optimizer deals with certain aspects of a query (such as dictating use of a join operator), sometimes they still allow the optimizer room to pick from multiple candidate plans, of which some are good and some are bad. You cannot control which one is picked.

In such cases, where you can't touch the code, and you want to "strong-arm" the optimizer into picking the plan you want, you can use plan forcing. I'll show you how to use a plan guide to force the use of *your* plan for a query, by applying the `USE PLAN` query hint. I'll then show an alternative approach to plan forcing using Query Store (a topic we'll cover in detail in Chapter 16). As you will see, it is much easier to use plan forcing within Query Store than it is to implement a plan guide.

As with hints, and plan guides, and for all the reasons discussed previously, plan forcing should be a final attempt at solving an otherwise unsolvable problem. As the data and statistics change, or new indexes are added, plan guides can become outdated and the exact thing that saved you so much processing time yesterday will be costing you more and more tomorrow.

Using plan guides to do plan forcing

The `USE PLAN` query hint, introduced in SQL Server 2005, allows you to come as close as you can to gaining total control over a query execution plan. This hint allows you to take an execution plan, captured as XML, and then use that plan on a given query from that point forward. This doesn't stop the optimizer from doing its job. You'll still get full optimization depending on the query, but the optimization is used only to verify that the forced plan will be valid for the query.

With plan guides, you cannot force a plan on:

- `INSERT`, `UPDATE`, `DELETE`, or `MERGE` queries
- Queries that use cursors other than `static`, `fast_forward`, `forward_only` or `insensitive`.

While you can simply attach an XML plan directly to the query in question, XML execution plans are very large. If your attached plan exceeds 8 K in size, then SQL Server can no longer cache the query, because it exceeds the 8 K string literal cache limit. For this reason, you should employ `USE PLAN`, within a plan guide, so that the query in question will be cached appropriately, enhancing performance. It also means that you avoid thousand-line queries, improving the readability and maintainability of the code, and you avoid having to deploy and redeploy the query to your production system, if you want to add or remove a plan.

Listing 9-30 shows a simple `CreditInfoBySalesPerson` stored procedure, for reporting some information from the `SalesOrderHeader` table.

```
CREATE PROCEDURE Sales.CreditInfoBySalesPerson (@SalesPersonID INT)
AS
SELECT soh.AccountNumber,
       soh.CreditCardApprovalCode,
       soh.CreditCardID,
```

```
        soh.OnlineOrderFlag
FROM Sales.SalesOrderHeader AS soh
WHERE soh.SalesPersonID = @SalesPersonID;
```

Listing 9-30

When the procedure is run using the value for `@SalesPersonID = 277`, a **Clustered Index Scan** results.

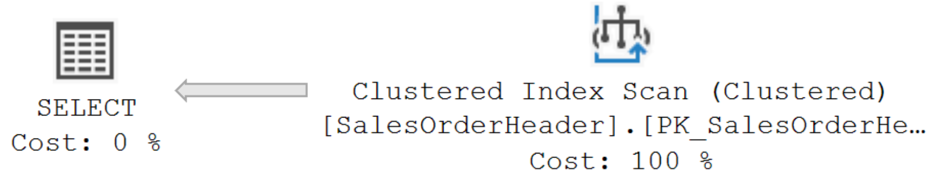


Figure 9-23: Execution plan with a scan for a large data set.

If we remove the plan from cache and change the value to 285, we see an **Index Seek** with a **Key Lookup**.

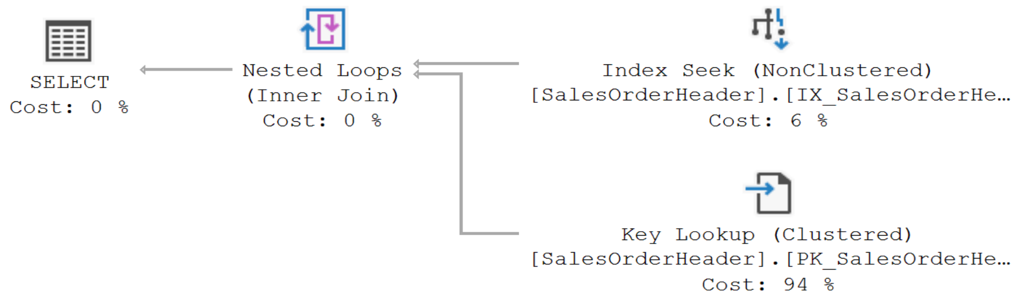


Figure 9-24: Execution plan with a Seek and Key Lookup for a smaller data set.

In situations like this, you might generally choose to recompile, using the `RECOMPILE` hint, but let's assume that is not acceptable, in this case. The next valid option is to add a plan guide that uses the `OPTIMIZE FOR` hint, as described previously. The **Clustered Index Scan** has the advantage of predictable and consistent performance, whereas the plan with the **Index Seek** and **Key Lookup** will likely have more erratic performance patterns.

However, your testing suggests that, for most values of `SalesPersonID`, the **Index Seek** with a **Key Lookup** is much faster than the **Clustered Index Scan** and, rather than use a plan guide and `OPTIMIZE FOR` hint, you're going to force the optimizer to always use your preferred plan.

First, we need to create an XML plan that behaves the way we want. We do this by taking the SQL text out of the stored procedure and modifying it to behave the correct way. This results in the desired plan, which we capture by wrapping it within `STATISTICS XML`, which will generate an actual execution plan in XML. You can also use a graphical plan and then right-click to capture the XML.

```
SET STATISTICS XML ON;
GO
SELECT soh.AccountNumber,
       soh.CreditCardApprovalCode,
       soh.CreditCardID,
       soh.OnlineOrderFlag
FROM Sales.SalesOrderHeader AS soh
WHERE soh.SalesPersonID = 285;
GO
SET STATISTICS XML OFF;
GO
```

Listing 9-31

This simple query generates a 117-line XML plan, which I won't show here. With the XML plan in hand, we'll create a plan guide to apply it to the stored procedure. You can just right-click on the **XML Showplan** link, select **Copy** and paste it in as the value for the `@hints` parameter.

```
EXEC sys.sp_create_plan_guide
    @name = N'UsePlanPlanGuide',
    @stmt = N'SELECT soh.AccountNumber,
       soh.CreditCardApprovalCode,
       soh.CreditCardID,
       soh.OnlineOrderFlag
FROM Sales.SalesOrderHeader AS soh
WHERE soh.SalesPersonID = @SalesPersonID;',
    @type = N'OBJECT',
    @module_or_batch = N'Sales.CreditInfoBySalesPerson',
    @params = NULL,
    @hints = N'<ShowPlanXML xmlns="http://sche...
```

Listing 9-32

If we supply a valid XML plan to `@hints`, then `sp_create_plan_guide` automatically interprets this as a `USE PLAN` hint. Now, we execute the query using the value that generates the non-preferred plan.

```
EXEC Sales.CreditInfoBySalesPerson @SalesPersonID = 277;
```

Listing 9-33

However, we still get the execution plan we want, as shown in Figure 9-25.

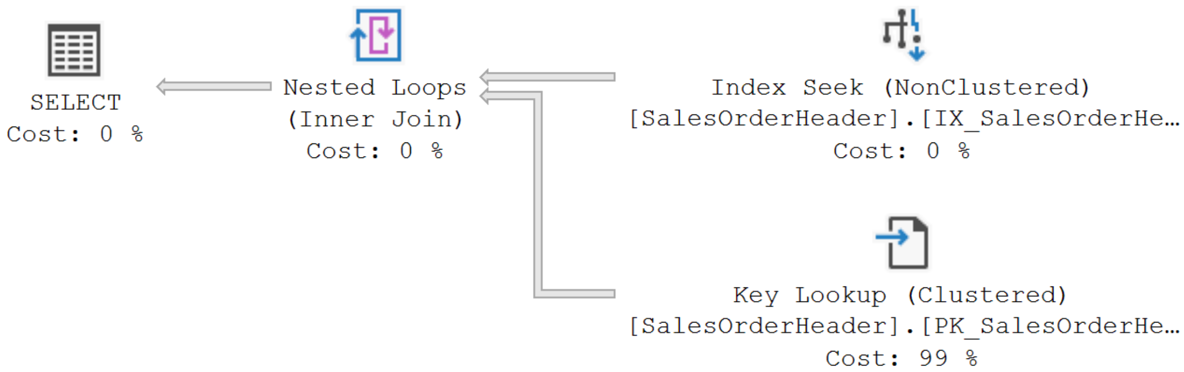


Figure 9-25: Execution plan using the Seek because of the plan guide.

The fatter data transfer pipes between the operators in Figure 9-25, compared to Figure 9-24, tells us that more data is being moved through the plan, as expected. You can also inspect the properties of the **SELECT** operator to verify that the plan guide was used.

Using Query Store to do plan forcing

If you're working on Azure SQL Database or SQL Server 2016 and later, an easier way to solve the same problem is to use plan forcing in Query Store. I'm assuming you have Query Store enabled. If not, go to Chapter 16 to learn how.

Execute the query in Listing 9-34.

```
SELECT Object_Name(qsq.object_id) AS ObjectName,
  Cast(qsp.query_plan AS XML) AS xmlplan, qsq.query_id, qsp.plan_id
  FROM sys.query_store_query AS qsq
    JOIN sys.query_store_plan AS qsp
      ON qsp.query_id = qsq.query_id
  WHERE qsq.object_id = Object_Id('Sales.CreditInfoBySalesPerson');
```

Listing 9-34

You should see three execution plans, all with the same `query_id` but different values for `plan_id`. The first two are the plans for the initial two executions of the stored procedure, with `@SalesPersonID` values of 277 and 285, and the third is technically a different plan because it is now a forced plan.

	ObjectName	xmlplan	query_id	plan_id
1	CreditInfoBySalesPerson	<ShowPlanXML xmlns="http://schemas.microsoft.com...	5004	5111
2	CreditInfoBySalesPerson	<ShowPlanXML xmlns="http://schemas.microsoft.com...	5004	5113
3	CreditInfoBySalesPerson	<ShowPlanXML xmlns="http://schemas.microsoft.com...	5004	5165

Figure 9-26: Three plans in the Query Store.

If we had edited the query text directly, to add the hint, then the `query_id` would have been different as well. However, in this case we used a plan guide so the query text was still exactly the same.

Let's say that this time we want to force the **Clustered Index Scan** plan for this procedure (Figure 9-23), then we can pull a plan directly out of the Query Store and put it into the plan cache. In my case, `plan_id` 5111 is the one I want.

```
EXEC sys.sp_query_store_force_plan @query_id = 5004, @plan_id =
5111;
```

Listing 9-35

Execute `CreditInfoBySalesPerson` with a parameter value of 285, and you'll see the **Clustered Index Scan** plan instead of the **Index Seek** and **Key Lookup** plan. And remember, unless you dropped it, the `UsePlanPlanGuide` guide, forcing the latter plan, is still in place. Query Store plan forcing will take precedence over a plan guide.

To unforce the plan, run Listing 9-36.

```
EXEC sp_query_store_unforce_plan @query_id = 5004, @plan_id = 5111;
```

Listing 9-36

You many also want to run Listing 9-29 one more time, if you still have plan forcing with a plan guide in place.

Again, plan forcing is a quick, although temporary, method for addressing bad parameter sniffing. I call the fix temporary because, as with any of the other bad-parameter-sniffing fixes, you'll want to reassess it over time as the data, your systems, and your code change.

Summary

Creating execution plans is a costly operation for SQL Server. Because of this, you want to reuse plans as often as you can, and in as many ways as you can. Using parameterized queries, whether stored procedures or prepared statements, is a great way to get this done. Other methods of controlling plan use and reuse such as forced parameterization and `Optimize For Ad Hoc Workloads` can also help reduce the load placed on the server by the optimization process.

Using plan guides and plan forcing, you can take direct control away from the optimizer and attempt to achieve better performance for your queries. However, by taking control of the optimizer you can introduce problems as big as those you're attempting to solve. Be very judicious in the use of some of the methods outlined in this chapter. Take your time and test everything you do to your systems. You will also need to regularly retest your systems wherever you've taken direct control using plan guides. Use the information that you've gleaned from the other chapters in this book to be sure that the choices you're making are the right ones.

Chapter 10: Controlling Execution Plans with Hints

The query optimizer gets it right most of the time, but occasionally it chooses a plan that isn't the best one possible. As discussed in Chapter 8, the optimizer bases its plan choices on selectivity and cardinality estimates that are derived from statistics. If a column has a particularly "jagged" distribution, even statistics that are as good, and as up to date, as SQL Server can make them can't accurately describe it. Sometimes, our queries use complex predicates that are hard to estimate, or that force the optimizer to use a hard-coded selectivity estimation. These issues could cause the optimizer to err in its choice of plan, resulting in suboptimal query performance.

In such cases, we might decide to force the optimizer's hand, by applying **hints** that tell it how to access certain tables, or which join strategy to use, or how it should optimize a whole set of operations for a given query. This, of course, will result in a different plan from the one the optimizer would have chosen if given a free hand.

I'll describe those query, join, and table hints that directly affect the choice of execution plan. I won't cover hints that affect the strategy for executing rather than compiling the query (such as locking hints), or any that have minimal impact on plan choice. I'll also explain why it's a very good idea, generally, to be extremely cautious when applying hints to your queries, and I'll point out the specific dangers associated with certain hints.

The Dangers of Using Hints

While you may find situations where a hint does indeed help performance, you should use them sparingly, because hints can be dangerous. Even their name is misleading; hints are not suggestions that the optimizer *might* consider, they are commandments that the optimizer *must* follow. Even if you supply a hint with which it is technically impossible for the optimizer to comply, it will still attempt to apply the hint, and throw an error. You'll see an example of that later, when we discuss the `INDEX()` hint.

While hints allow you to control the behavior of the optimizer, it doesn't mean your choices are necessarily better than the optimizer's choices. If you find yourself putting hints on most of your queries and stored procedures, then you're *doing something wrong*. Yes, the right hint on the right query can improve query performance. However, the exact same hint used on another query can create more problems than it solves, radically slowing your query and leading to severe blocking and timeouts in your application. Even a hint that is "good" right now can turn out to be very bad with time, because it removes the optimizer's subsequent ability to make a better plan choice, in response to changes in the data distribution, or in response to an upgrade to a new SQL Server version, or the application of a new service pack.

Over the coming sections, I'll describe the various hints we can use, and problems that we're hoping to solve by applying that hint. You'll see examples where a hint improves performance, or changes the behavior in a positive manner, and also some where a hint degrades performance. Again, this is not a chapter about hints, per se, but rather their effect on execution plans. For more details on hints, please refer to the Microsoft documentation (http://bit.ly/2pt7UF2).

For any hint, only apply it after copious testing, and with thorough documentation. You need to make it as easy as possible for others to find where hints are used, to understand the intent of the hint, and therefore to schedule regular tests to verify that its use is still valid, as the system and its data change over time.

Query Hints

Query hints take control of an entire query and can affect all operators within the execution plan. We can use query hints to force the use of a specific operator for all aggregations in a query, or for all joins. We can use them to instruct the optimizer to optimize a query for a defined parameter value, or to compile a new plan on every execution of that query, to control use of parallelism for that query, and more. Some query hints are useful occasionally, while a few are for rare circumstances. As with all hints, injudicious use of query hints can cause you more problems than they solve!

We specify query hints in the `OPTION` clause. Listing 10-1 shows the basic syntax.

```
SELECT ...
OPTION (<hint>,<hint>...);
```

Listing 10-1

We can't apply query hints to data manipulation statements INSERT, except as part of an associated SELECT operation, and we can't use query hints in subqueries since the hint must apply to the entire query.

HASH | ORDER GROUP

The HASH GROUP and ORDER GROUP hints apply to all aggregations in the query caused by GROUP BY or DISTINCT. Generally, the optimizer will choose the most appropriate of the two aggregation mechanisms it has available, **Hash Match** (which is hash based) or **Stream Aggregate** (which is order based). The HASH GROUP hint forces it to use the former, and the ORDER GROUP hint, the latter.

In Listing 10-2, we have a simple GROUP BY query that returns a count of the number of occurrences of each distinct value in the Suffix column of the Person table.

```
SELECT p.Suffix,
       COUNT(*) AS SuffixUsageCount
FROM Person.Person AS p
GROUP BY p.Suffix;
```

Listing 10-2

Let's suppose that you, as the DBA, maintain a high-end shop where the sales-force submits many queries against an ever-changing set of data. One of the sales applications frequently calls the query in Listing 10-2 and your job is to make this query run as fast as possible.

The first thing you'll do, of course, is look at the execution plan, as shown in Figure 10-1.

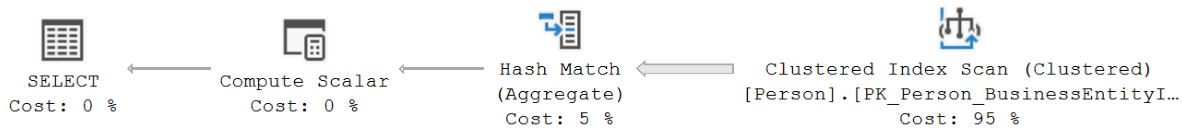


Figure 10-1: Unforced execution plan using a Hash Match for aggregation.

As you can see, the optimizer has chosen to use hashing for this query. The "unordered" data from the **Clustered Index Scan** is grouped within the **Hash Match (Aggregate)** operator. This operator builds a hash table, creating entries for each of the distinct values in the data supplied by the **Clustered Index Scan**, and maintains a count of each of those values.

As a reference point, on my system, and on my version of AdventureWorks, the scan on the `Person` table caused 3,819 reads, the plan had an estimated cost of 2.99727, and the query ran in about 9.7ms.

Although not the most expensive operation in the plan (that's the **Clustered Index Scan**), you may have read that the **Hash Match** could cause problems because of the overhead of building and populating a table in memory, and because this is a "blocking" operation. Therefore, let's see what happens if we force the optimizer to use a **Stream Aggregate** instead, by adding the `ORDER GROUP` hint to the query.

```
SELECT p.Suffix,
       COUNT(p.Suffix) AS SuffixUsageCount
FROM Person.Person AS p
GROUP BY p.Suffix
OPTION (ORDER GROUP);
```

Listing 10-3

Figure 10-2 shows the new plan.

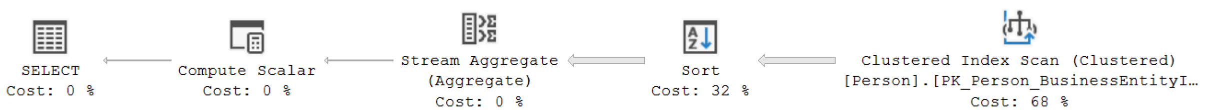


Figure 10-2: Execution plan forced to use Stream Aggregate operator.

Since stream aggregation requires sorted data (See Chapter 5), and since there is no index that SQL Server can use to directly produce rows ordered by `Suffix`, the optimizer introduced a **Sort** operator to enforce the required ordering, and the estimated cost of the plan jumped 39% to 4.17893, with the source of the increased cost being the **Sort** operation. As a result, this query now runs in 18ms, instead of the original 9.7ms, a 100% increase.

The broader problem with this hint, as with all hints, is that it forces a certain behavior, regardless of changes to the database structure, such as addition or removal of indexes, or to the data. Instead of adding the hint, it's much better to find out why the optimizer doesn't use stream aggregation, and then fix the root cause. For example, if appropriate for the query workload, you might consider adding a new nonclustered index, or modifying an existing index.

MERGE | HASH | CONCAT UNION

These query hints affect how the optimizer deals with UNION operations in your queries, instructing the optimizer to use either merging, hashing, or concatenation of the data sets. If a UNION operation is causing performance issues, you may be tempted to use these hints to guide the optimizer's behavior. As discussed in Chapter 4, the optimizer will never use a **Hash Match** operator for a UNION ALL concatenation, and so the HASH UNION hint doesn't work for UNION ALL queries.

The example query in Listing 10-4 is not running fast enough to satisfy the demands of the application.

```
SELECT pm1.Name,
       pm1.ModifiedDate
FROM Production.ProductModel AS pm1
UNION
SELECT p.Name,
       p.ModifiedDate
FROM Production.Product AS p;
```

Listing 10-4

When a query has been identified as running slow, it's time to look at the execution plan, as seen in Figure 10-3.

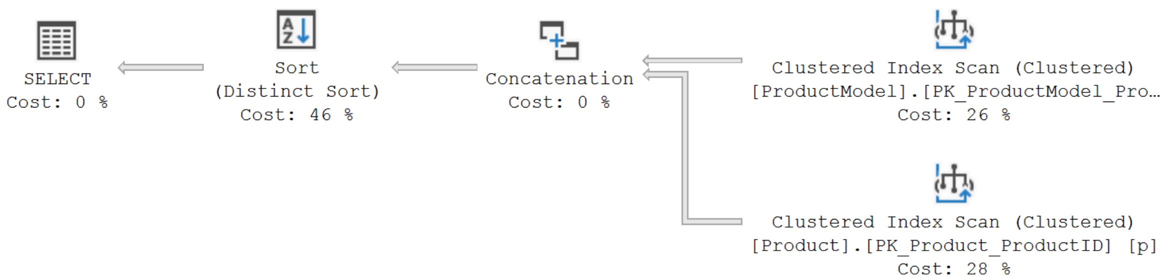


Figure 10-3: An execution plan for a UNION operation using concatenation.

The **Concatenation** operator simply concatenates the 128 rows from the top input with the 504 rows from the bottom and, in the context of the plan, it is very cheap. The **Sort** operator, specifically **a Distinct Sort** (see Chapter 5), is in the plan to remove duplicates, as required by the UNION clause, and is relatively expensive. The query took about 121ms to run with 29 reads.

Perhaps forcing the use of a join operator to implement the `UNION` clause, instead of concatenation, might enable the optimizer to remove the expensive **Sort** operator, and improve performance? As a first test, you apply the `MERGE UNION` hint.

```
SELECT pm1.Name,
       pm1.ModifiedDate
FROM Production.ProductModel AS pm1
UNION
SELECT p.Name,
       p.ModifiedDate
FROM Production.Product AS p
OPTION (MERGE UNION);
```

Listing 10-5

The plan confirms that you have forced the `UNION` operation to use the **Merge Join (Union)** instead of the **Concatenation** operator.

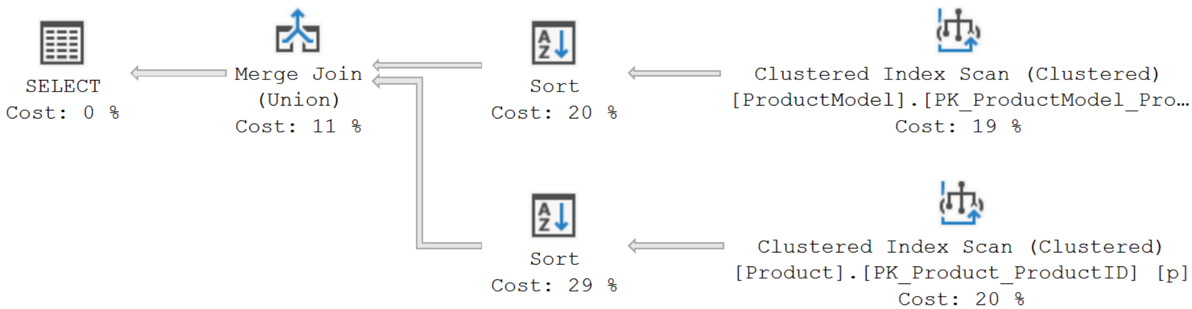


Figure 10-4: Forcing the execution plan to use a Merge Join for the UNION.

Now that we're joining rather than concatenating the rows, we no longer see the **Distinct Sort**. However, since the **Merge Join** only works with sorted data feeds, we've also forced the optimizer to use two **Sort** operators to sort each of the inputs. The execution time went up to 193ms from 121ms and the reads went to 41 from 29. Clearly, this didn't work.

What if you tried the `HASH UNION` hint? Note that use of this hint will only work if the probe (bottom) input is guaranteed to have no duplicates, as is true here.

```
SELECT pm1.Name,
       pm1.ModifiedDate
FROM Production.ProductModel AS pm1
UNION
```

```
SELECT p.Name,
       p.ModifiedDate
FROM Production.Product AS p
OPTION (HASH UNION);
```

Listing 10-6

Figure 10-5 shows the new execution plan, with the **Sort** operations eliminated although, if the bottom input had had duplicates, the optimizer would have needed to add a **Sort (Distinct Sort)** or other operator to the input to remove them. You can verify this by removing the `Name` column from Listing 10-6.

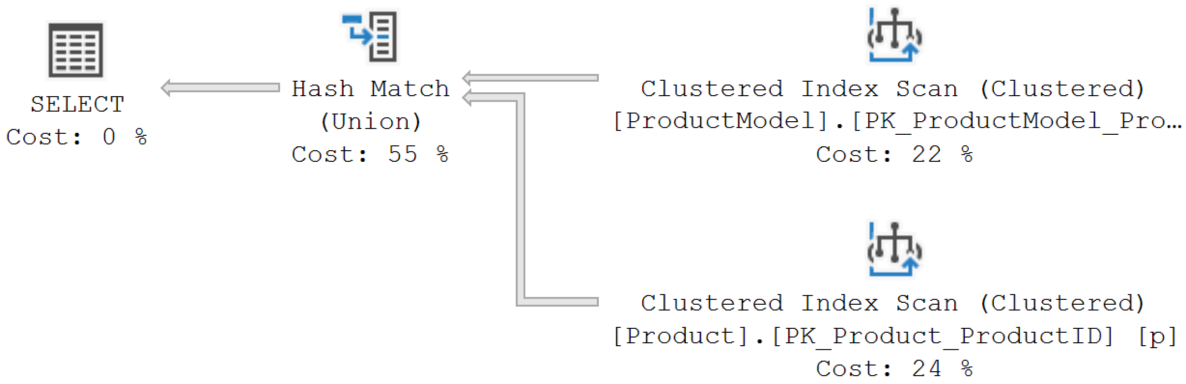


Figure 10-5: Execution plan forced to use a Hash Match Union operator.

We achieved our initial goal of eliminating the post-union **Sort** operator without introducing any new **Sort** operators. It turns out that, in this case, using a **Hash Match** to perform the `UNION` operation is less expensive than performing a **Concatenation** followed by a **Distinct Sort**, and the execution time has decreased from 121ms on average to 99ms, while the reads remained the same. Of course, it's possible that with bigger, or different, tables the dynamic might change.

LOOP | MERGE | HASH JOIN

These query hints make all the join operations in the query, including the semi-joins used to fulfill the `EXISTS` or `IN` clauses, use the method supplied by the hint. However, note that, if we also apply a join hint (covered later) on a specific join, then the more granular join hint takes precedence over the general query hint.

Let's say that our system is suffering from poor disk I/O, so we need to reduce the number of reads that our queries generate. By collecting data from Extended Events and Performance Monitor, we identify the query in Listing 10-7 as one that needs some tuning.

```
SELECT pm.Name,
       pm.CatalogDescription,
       p.Name AS ProductName,
       i.Diagram
FROM Production.ProductModel AS pm
    LEFT JOIN Production.Product AS p
        ON pm.ProductModelID = p.ProductModelID
    LEFT JOIN Production.ProductModelIllustration AS pmi
        ON p.ProductModelID = pmi.ProductModelID
    LEFT JOIN Production.Illustration AS i
        ON pmi.IllustrationID = i.IllustrationID
WHERE pm.Name LIKE '%Mountain%'
ORDER BY pm.Name;
```

Listing 10-7

Figure 10-6 shows the plan

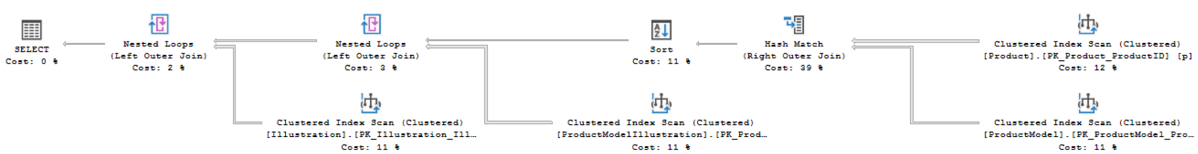


Figure 10-6: A mix of Nested Loops and Hash Match joins.

The query predicate, `WHERE pm.name LIKE '%Mountain%'`, is non-SARGable, a term used for predicates that can't be used by the optimizer in an **Index Seek**, and so the **Clustered Index Scan** operator on the `ProductModel` table makes sense. The query has no filter on the `Product` table, so the scan is the only option. The optimizer uses a **Hash Match** operator to join the `Product` and `ProductModel` tables , accounting for 39% of the estimated cost of the plan. It then performs the required **Sort** which, because the optimizer estimates only about 99 matching rows, should be cheap. It then uses **Nested Loops** joins to construct the rest of the data set. The optimizer chooses to scan the `ProductModelIllustration` and `Illustration` tables rather than seek them, probably because they're both so small that the cost estimates are all too small to make a significant difference to the total query cost.

In my tests, this query ran in about 74ms, requiring 485 logical reads, as measured using Extended Events (see Chapter 2, Listing 2-6).

Again, let's say you've read that **Hash Match** joins incur the overhead of creating an in-memory worktable that is prone to spilling to `tempdb`. Maybe it will be cheaper if we force the use of **Nested Loops** joins, by adding the `LOOP JOIN` hint to the end of the query?

```
…OPTION ( LOOP JOIN );
```

Listing 10-8

Figure 10-7 shows the new plan.

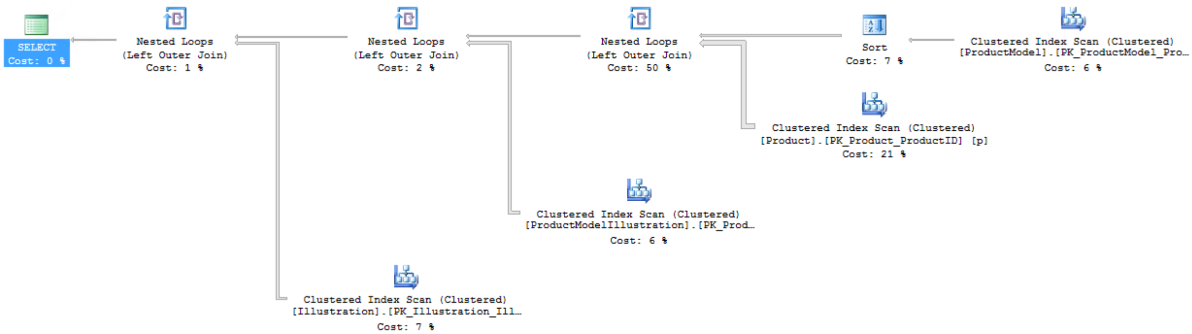


Figure 10-7: Forcing the execution plan to use only Nested Loops joins.

As expected, we've forced the optimizer to use **Nested Loops** joins throughout. As a result, it's moved the **Sort** operation to directly after the scan of the `ProductModel` table, which it can do because a **Nested Loops** join will always preserve the order of the outer input, so now will sort only an estimated 40 rows (actual number is 37). Also, we should have eliminated the need for in-memory worktables. But has it reduced the I/O?

Sadly, no. The query now performs 1250 logical reads, and ran in about 73ms. This is due to the increased logical reads on the `Product` table. Thanks to us forcing the use of **Nested Loops** joins, this table is now scanned 37 times, once for every row returned by our **Sort** operator. On the plus side, if you check the **MemoryGrantInfo** property of the **Select** operator, for Figure 10-7, you'll see that the query has a significantly smaller memory grant compared to the original plan, which may be a consideration if this were a frequently-executed query.

What if we modify the query to use the MERGE JOIN hint, instead?

```
OPTION  ( MERGE JOIN );
```

Listing 10-9

Figure 10-8 shows the new plan.

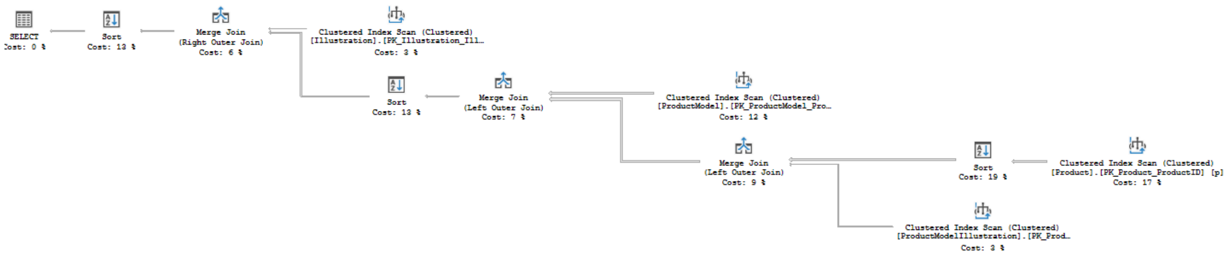


Figure 10-8: An execution plan that only contains Merge Joins.

The plan is a different shape, and looks more complicated mainly because, inauspiciously, we now see three **Sort** operators rather than one. The **Sort** on the Name column is now the final operation, before returning the results. The two new **Sort** operators are required because, as discussed in Chapter 4, the data in each input must be ordered on the join column, and the data stream from the Product table, and the one emerging from the second **Merge Join**, are not in the required order.

Did we manage to reduce logical reads? In fact, yes, this plan performs only 116 logical reads. However, in my tests, performance did not improve (around 83ms in my tests). The first problem is the extra overhead of the sorting operations; the memory grant is almost double that of the original query. The second problem is the rightmost **Merge Join** is a many-to-many join, which requires the creation of a worktable in tempdb, and is far less efficient (see Chapter 4, Listing 4-3 and subsequent discussion).

Given that we said we were worried about the overhead of worktables, we'd be unlikely to try the final option, the HASH JOIN hint, but let's see what it might do.

```
OPTION  ( HASH JOIN );
```

Listing 10-10

Figure 10-9 shows the new plan.

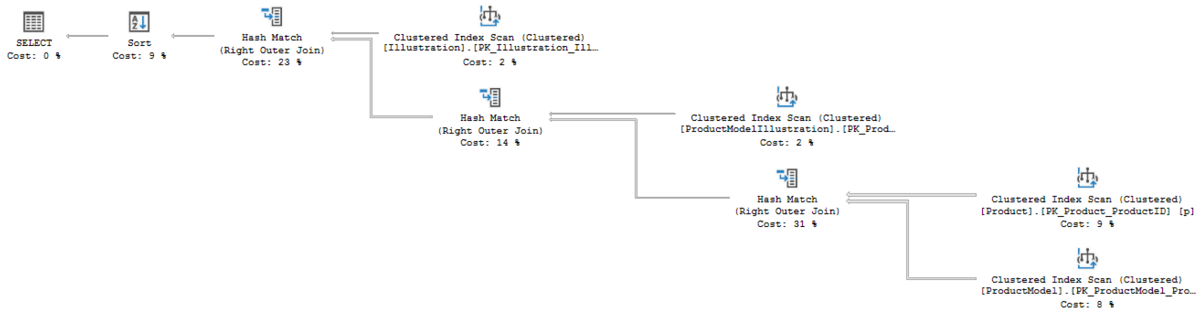


Figure 10-9: Forcing the plan to use Hash Joins.

We now see three **Hash Match** joins, and we're back down to only one **Sort** (on `name`), but it's over on the left-hand side. This is the only place the optimizer can safely put it since the **Hash Match** joins are not guaranteed to preserve the order of the probe input (if they were, then the sort could go directly after the scan of `ProductModel`).

How does it perform? Well, we've reduced logical reads to 97, the best so far, but the query runs in about the same time as the original query. If we are seeing lots of I/O contention, this could be a possible win, but you'd need to test this in an environment with additional load to understand if there are contention issues. Also, we've significantly increased the memory cost; the memory grant is up to about 6080 KB, due to the overhead of hashing values in all tables and creating hash tables for the build inputs.

Overall, our efforts have reaped minimal rewards, and whether you chose to use one of these hints would depend on the contention points in your system. More significantly, all our efforts with hints have ignored the bigger problem with this query, which is the use of the `LIKE '%Mountain%'` in the `WHERE` clause. This is an operator that can only be resolved by scans against the table, and it's those scans that are our primary problem. The best solution for this query could be to modify the database structure so that the need for the `LIKE` query, using wild cards, is removed. When modifying the code or structure is not possible, you may have to resort to query hints to attempt to gain improvements where you can.

FAST n

Let's assume for a moment that we are less concerned about the overall performance of the database, generally a very poor proposition, than we are about perceived performance of the application. The users would like an immediate return of data to the screen, even if it's not the complete result set, and even if they end up waiting longer for the complete result set. This could be a handy way to get a little bit of information in front of people quickly, so that they can decide whether it's important, and either move on or wait for the rest of the data.

The `FAST n` hint provides this ability by getting the optimizer to focus on finding the execution plan that will return the first "n" rows as fast as possible, where "n" is a positive integer value. Consider the following query and execution plan.

```
SELECT soh.SalesOrderNumber,
       soh.OrderDate,
       soh.DueDate,
       sod.CarrierTrackingNumber,
       sod.OrderQty
FROM Sales.SalesOrderDetail AS sod
    JOIN Sales.SalesOrderHeader AS soh
        ON sod.SalesOrderID = soh.SalesOrderID
ORDER BY soh.DueDate DESC;
```

Listing 10-11

Figure 10-10 shows the plan. The **Estimated Subtree Cost** of this plan is 11.4, so if your `cost threshold for parallelism` setting (see Chapter 11) is at 11.4 or higher, you'll see the parallelized version of this plan.

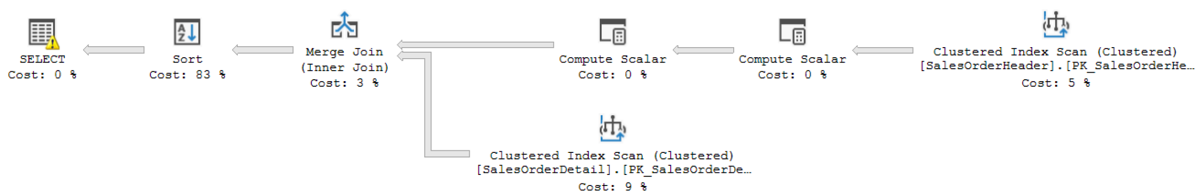


Figure 10-10: An execution plan optimized to return all data quickly.

I won't explain this plan in any detail, except to point out the warning visible on the `SELECT` operator. If you look at the **Warnings** property of the **SELECT** operator, you'll find the following:

```
Type conversion in expression (CONVERT(nvarchar(23),[soh].
[SalesOrderID],0)) may affect "CardinalityEstimate" in query plan
choice
```

This is caused by a calculated column in the `SalesOrderHeader` table. This is an example of a false warning. It doesn't affect our query in any way because we're not referring to that column in any filtering clause.

This query performs adequately considering the fact that it's selecting all the data from the tables without any sort of filtering operation, but let's try to get some, but not all, rows back faster from this query by adding the `FAST n` hint to return the first 10 rows as quickly as possible.

```
OPTION  ( FAST 10 );
```

Listing 10-12

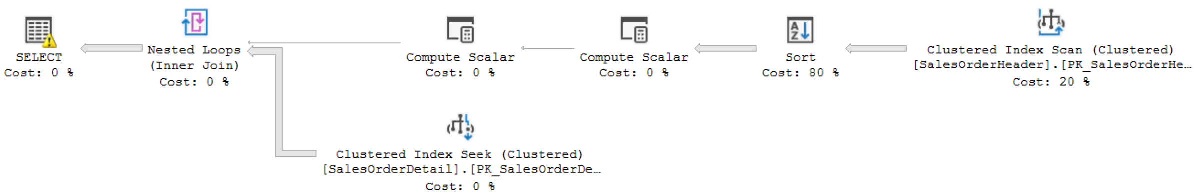


Figure 10-11: An execution plan optimized to return only 10 rows.

Now, the optimizer chooses a **Nested Loops** operator to perform the join, rather than a **Merge Join**. This plan returns first rows very fast, but the rest of the processing was somewhat slower, which is perhaps to be expected, since the optimizer focuses its efforts on getting just the first ten rows back as soon as possible. The way this works, internally, is that the optimizer treats this query as if it had a `TOP (10)` clause and was only ever going to return 10 rows. That changes completely the execution plan choices; the plan you get will usually be the same as the plan for a query that uses `TOP`, but without the operators that implement the `TOP` clause.

The total estimated cost for the original query was 11.3573. The hint reduced that cost to 2.72567. While that sounds great, remember that is the estimated cost for only the first 10 rows. This is also why the plan in Figure 10-11 shows some "bad" row estimates. For example, if you were to check the properties of the **Sort** operator, you'd see that the optimizer estimated that it would return 2.6 rows (the actual number of rows was 31465).

We've made the choice that we don't care about overall performance impact on the system, we just want to see the first 10 rows very fast. However, we can't ignore the fact that the number of logical reads increases dramatically, from 1,935 for the un-hinted query to 106,505 for the hinted query. Depending on the load on your system and the contention on your disk, getting a responsive appearance on your application could seriously negatively impact the overall system.

FORCE ORDER

Once again, our monitoring tools have identified a query that is performing poorly. It's a long query with a higher number of tables being joined, as shown in Listing 10-13, which could be a concern, because the more tables there are involved, the harder the optimizer has to work.

Normally, the optimizer will determine the order in which the joins occur, rearranging them as it sees fit. However, the optimizer can make incorrect choices when the statistics are not up to date, when the data distribution is less than optimal, or if the query has a high degree of complexity, with many joins. In the latter case, the optimizer may even time out when trying to rearrange the tables because there are so many of them for it to try to deal with.

Using the `FORCE ORDER` hint, you can make the optimizer use the order of joins as you have defined them in the query. This might be an option if you are sure that your join order is better than that supplied by the optimizer, if you're experiencing timeouts in the optimization process, or if you see lots of compiles or recompiles from a query, and system performance is suffering as a result (although, testing is, as always, in order).

```
SELECT pc.Name AS ProductCategoryName,
       ps.Name AS ProductSubCategoryName,
       p.Name AS ProductName,
       pdr.Description,
       pm.Name AS ProductModelName,
       c.Name AS CultureName,
       d.FileName,
       pri.Quantity,
       pr.Rating,
```

```
        pr.Comments
FROM Production.Product AS p
    LEFT JOIN Production.ProductModel AS pm
        ON p.ProductModelID = pm.ProductModelID
    LEFT JOIN Production.ProductSubcategory AS ps
        ON p.ProductSubcategoryID = ps.ProductSubcategoryID
    LEFT JOIN Production.ProductInventory AS pri
        ON p.ProductID = pri.ProductID
    LEFT JOIN Production.ProductReview AS pr
        ON p.ProductID = pr.ProductID
    LEFT JOIN Production.ProductDocument AS pd
        ON p.ProductID = pd.ProductID
    LEFT JOIN Production.Document AS d
        ON pd.DocumentNode = d.DocumentNode
    LEFT JOIN Production.ProductCategory AS pc
        ON ps.ProductCategoryID = pc.ProductCategoryID
    LEFT JOIN Production.ProductModelProductDescriptionCulture AS
pmpdc
        ON pm.ProductModelID = pmpdc.ProductModelID
    LEFT JOIN Production.ProductDescription AS pdr
        ON pmpdc.ProductDescriptionID = pdr.ProductDescriptionID
    LEFT JOIN Production.Culture AS c
        ON c.CultureID = pmpdc.CultureID;
```

Listing 10-13

Based on your knowledge of the data, you're confident that you've put the joins in the correct order. Figure 10-12 shows the current execution plan.

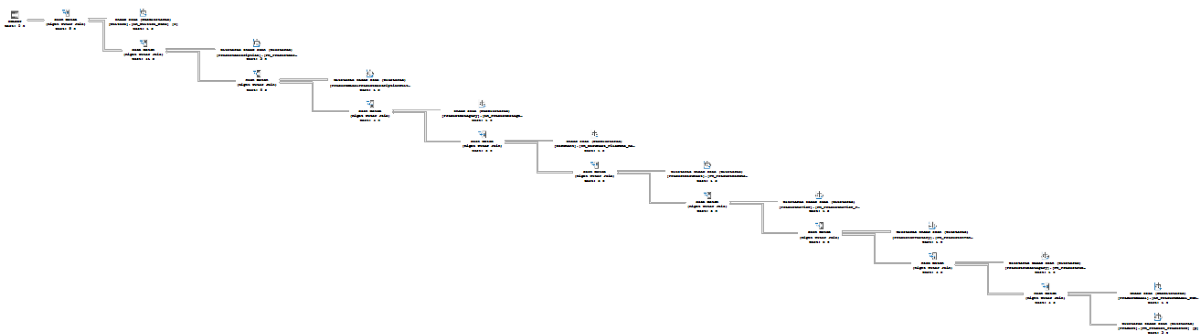


Figure 10-12: Large execution plan with more tables.

This plan is far too large to review on this page in the book. The image in Figure 10-12 gives you a good idea of the overall structure and shape of the execution plan. Figure 10-13 shows an exploded view of the bottom right of the plan, showing just a few of the tables and the order in which they are being joined.

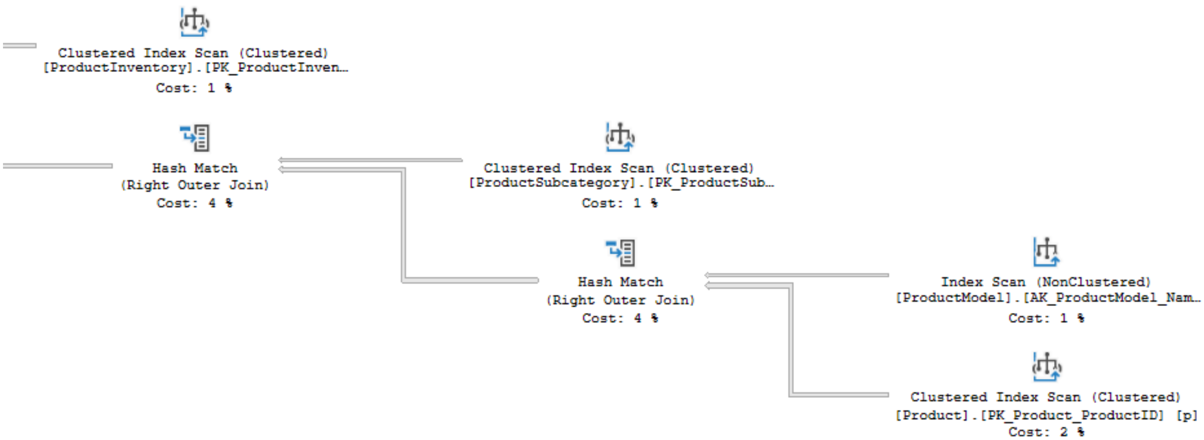


Figure 10-13: Subset of execution plan in Figure 10-12 showing table join order.

Following the data flow, we first see the **Hash Match** join between `ProductModel` and `Product`. This data forms the bottom input to a **Hash Match** join to `ProductSubcate-gory` and this joined data stream forms the bottom input to **Hash Match** join to `Product-Inventory`, and so on. However, in execution order, the optimizer starts right at the other end, with `Culture`, then `ProductDescription`, then `Product-ModelProduct-DescriptionCulture` and so on.

If you check the properties of the **SELECT** operator, you'll see that the optimizer timed out when generating this execution plan.

Figure 10-14: SELECT property showing the Reason For Early Termination.

With a larger number of tables, and a timeout in the optimizer, there's a good chance that not all possible permutations of the join order were attempted. If we had exhausted other attempts at tuning this query, we might attempt to wrest control from the optimizer by using a query hint. Take the same query and apply the `FORCE ORDER` query hint.

```
OPTION (FORCE ORDER);
```

Listing 10-14

It results in the plan shown in Figure 10-15.

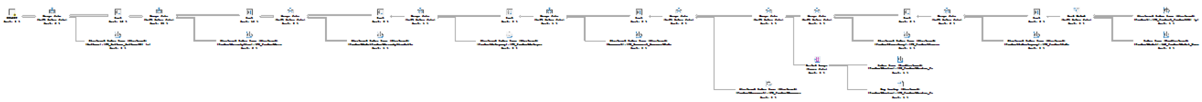


Figure 10-15: A new execution plan shape because of the FORCE ORDER hint.

You can tell, just by comparing the shapes of the plan in Figure 10-12 to the one in Figure 10-15 that a substantial change has occurred. The optimizer is now accessing the tables exactly in the order specified by the query. Again, we'll zoom in on the set of operators on the right-hand side of the plan, so that you can see how the join order has changed.

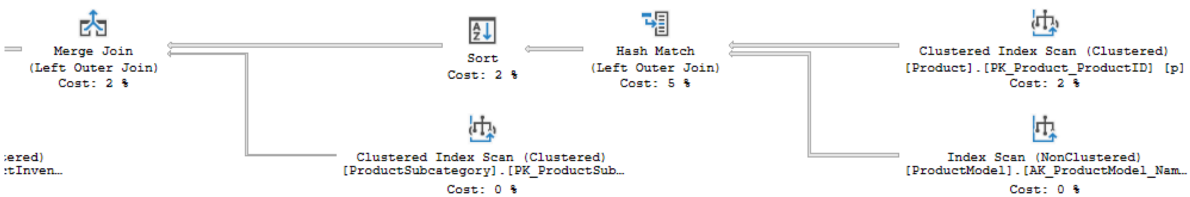


Figure 10-16: Subset of Figure 10-15 showing a different table order in the joins.

Now the join order is from the `Product` table, followed by the `ProductModel`, exactly as specified in the query. This data forms the top input to a **Merge Join** to `ProductSub-category`, which forms the top input to **Merge Join** to `ProductInventory`, and so on. This order forces the optimizer to do more **Sort** operations, and the execution time went from 149ms in the first query to 166ms in the second. While it is possible to get direct control over the optimizer to achieve positive results, this is not one of those cases.

MAXDOP

In this example, we have one of those nasty problems where a query that sometimes runs just fine, sometimes runs incredibly slowly. We have investigated the issue, using Extended Events or the Query Store to capture the execution plan of a query, over time, with various parameters. We finally arrive at two execution plans. Figure 10-17 shows the execution plan that results in better performance on my system.

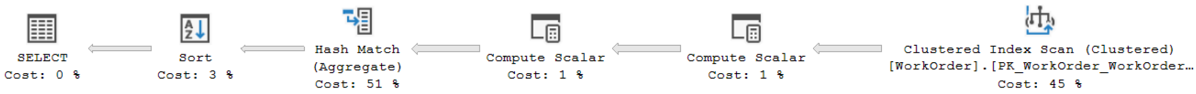


Figure 10-17: A serial execution plan that runs quickly.

Figure 10-18 shows the slower execution plan (I modified this image for readability).

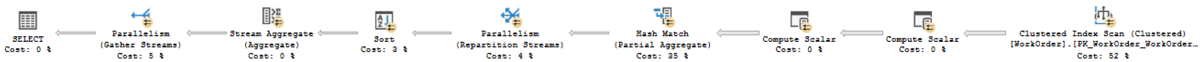


Figure 10-18: A parallel execution plan that, in this case, doesn't run as fast.

This is an example of where the optimizer has estimated that the cost of executing the plan in a serial fashion might exceed the `'cost threshold for parallelism' sp_configure` option, and so produces a parallel plan, whereby the work required to execute the query is split across multiple CPUs (see Chapter 11 for more detail). Ideally, this should be helping the performance of your system, but it seems to be hurting it in this specific case.

Of course, the first question to ask here is why we have two plans with two different costs. What caused the new compile in the first place, and why are the costs different? If this were a parameterized query, then parameter sniffing might be a likely culprit (see Chapter 8), and we'd investigate that possibility first. However, in this case we're dealing with a simple query and, for this discussion, we've decided to fix the problem the "easy" way, with a hint.

We can control parallelism by setting the `Max Degree of Parallelism` value at the server level. You can also control this setting at the database level, and this is generally considered the better approach. A properly configured system will benefit from parallel execution, so you shouldn't simply turn it off. We'll also assume that you've tuned the value of `cost threshold for parallelism`, on your server, in order to be sure that only high-cost queries are experiencing parallelism. (A strong recommendation: don't leave it at the default value of 5; for details see this blog post: http://bit.ly/2DM92sc.)

However, having done this work, you still have the occasional outliers where the execution engine chooses to use the parallel plan. It's for cases like this that the `MAXDOP` hint becomes useful, since it controls the use of parallelism within an individual query, rather than working using the server-wide setting of `max degree of parallelism`.

For example, we can suppress parallelism altogether for this query by setting `MAXDOP` to 1. More commonly, we'd use it to set `MAXDOP` to a value greater than 1, but less than the number of processors, to ensure that a long-running query doesn't hog all resources.

This example is somewhat contrived in that, as part of the query, I'm going to reset the `cost threshold for parallelism` for my system to a low value, to enable this query to be run in parallel.

```
--enable advanced options
EXEC sys.sp_configure 'show advanced options', 1
GO
RECONFIGURE WITH OVERRIDE
GO
--change the cost threshold to 1
EXEC sp_configure 'cost threshold for parallelism', 1;
GO
RECONFIGURE WITH OVERRIDE;
GO
--Execute the query which will go parallel
SELECT wo.DueDate,
       MIN(wo.OrderQty) AS MinOrderQty,
       MIN(wo.StockedQty) AS MinStockedQty,
       MIN(wo.ScrappedQty) AS MinScrappedQty,
       MAX(wo.OrderQty) AS MaxOrderQty,
       MAX(wo.StockedQty) AS MaxStockedQty,
       MAX(wo.ScrappedQty) AS MaxScrappedQty
FROM Production.WorkOrder AS wo
GROUP BY wo.DueDate
ORDER BY wo.DueDate;
GO
--reset the cost threshold to the default value
--if your cost threshold is set to a different value, change the 5
EXEC sys.sp_configure 'cost threshold for parallelism', 5;
GO
RECONFIGURE WITH OVERRIDE;
GO
--disable advanced options
EXEC sys.sp_configure 'show advanced options', 0
GO
RECONFIGURE WITH OVERRIDE
GO
```

Listing 10-15

This will result in an execution plan that takes full advantage of parallel processing, as shown in Figure 10-18.

Let's now modify the query to include the MAXDOP hint.

```
OPTION  ( MAXDOP 1 );
```

Listing 10-16

The use of the hint makes the new execution plan use a single processor, so no parallelism occurs at all. Add the hint to the end of the query in Listing 10-15 and then rerun the code. The plan will be the same as Figure 10-17.

Generally, you'd expect the performance of certain operators, such as the **Sort** arising from our ORDER BY clause in Listing 10-15, to benefit greatly from parallelism, as it reduces both CPU cost and runtime. Balancing these kinds of savings is the extra overhead associated with the parallelism operators that take the data from a single stream to a set of parallel streams, and then bring it all back together again. On my system, is seems that these extra costs outweighed the savings. However, with a properly configured cost threshold for parallelism setting, you'd expect most queries that cross that threshold to benefit from parallel execution.

OPTIMIZE FOR

You can use the OPTIMIZE FOR hint in any situation where you want to attempt to control how the optimizer deals with parameter values. Let's say that you have identified a query that will run at an adequate speed for hours or days, and then it suddenly performs horribly. With a lot of investigation and experimentation, you find that the parameters supplied by the application to run the procedure or parameterized query usually result in an execution plan that performs very well. Sometimes, though, a certain value or subset of values supplied to the parameters after a recompile event, results in an execution plan that performs extremely poorly. This is an instance of the **bad parameter sniffing** problem, as discussed in Chapter 8.

When you're hitting a bad parameter sniffing situation, you can use the OPTIMIZE FOR hint, which instructs the optimizer to optimize the query for the value that you supply, rather than a sniffed parameter value. Starting with SQL Server 2008, we can also use the OPTIMIZE FOR hint with a value of UNKNOWN to force a more generic plan on the optimizer, rather than a specific plan for a specific value.

We can demonstrate the utility of this hint with a very simple set of queries.

```
SELECT  AddressID,
        AddressLine1,
        AddressLine2,
```

```
        City,
        StateProvinceID,
        PostalCode,
        SpatialLocation,
        rowguid,
        ModifiedDate
FROM    Person.Address
WHERE   City = 'Mentor';
SELECT  AddressID,
        AddressLine1,
        AddressLine2,
        City,
        StateProvinceID,
        PostalCode,
        SpatialLocation,
        rowguid,
        ModifiedDate
FROM    Person.Address
WHERE   City = 'London';
```

Listing 10-17

We'll run these at the same time, and we get two different execution plans.

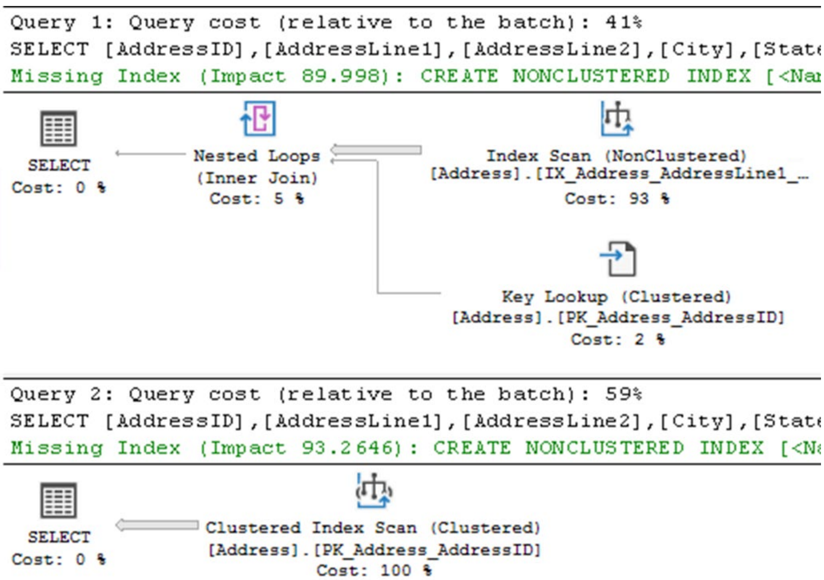


Figure 10-19: Two different execution plans for two different values.

Each query is returning the data from the table in a way that is optimal for the value passed to it, based on the indexes and the statistics of the table. The first execution plan, for the first query, where City = 'Mentor' scans the Address table to find matching values. Next, it must perform a **Key Lookup** operation to get the rest of the data. The data is joined through the **Nested Loops** operation. The value of London is much less selective, so the optimizer decides to perform a scan of the clustered index only, which you can see in the second execution plan in Figure 10-19.

If this query were in a stored procedure, which was executed first with a value of Mentor, then the next time we executed it with a value of London, the plan would be reused (unless it was recompiled for some reason), and we'd likely see a lot of key lookups and very poor performance.

We might consider adding a OPTIMIZE FOR (@City = 'London') query hint. While this might seem a sensible option in this case, the more general problem with the OPTIMIZE FOR <value> hint, is that it's susceptible to "turning bad," as data in the table changes over time.

Let's now see what happens if we use local variables in our T-SQL, as shown in Listing 10-18.

```
DECLARE @City NVARCHAR(30)
SET @City = 'Mentor'
SELECT  AddressID,
        AddressLine1,
        AddressLine2,
        City,
        StateProvinceID,
        PostalCode,
        SpatialLocation,
        rowguid,
        ModifiedDate
FROM    Person.Address
WHERE   City = @City;
SET @City = 'London'
SELECT  AddressID,
        AddressLine1,
        AddressLine2,
        City,
        StateProvinceID,
        PostalCode,
        SpatialLocation,
```

```
        rowguid,
        ModifiedDate
FROM    Person.Address
WHERE   City = @City;
```

Listing 10-18

Now, we see the same plan, with a clustered index scan, for both queries.

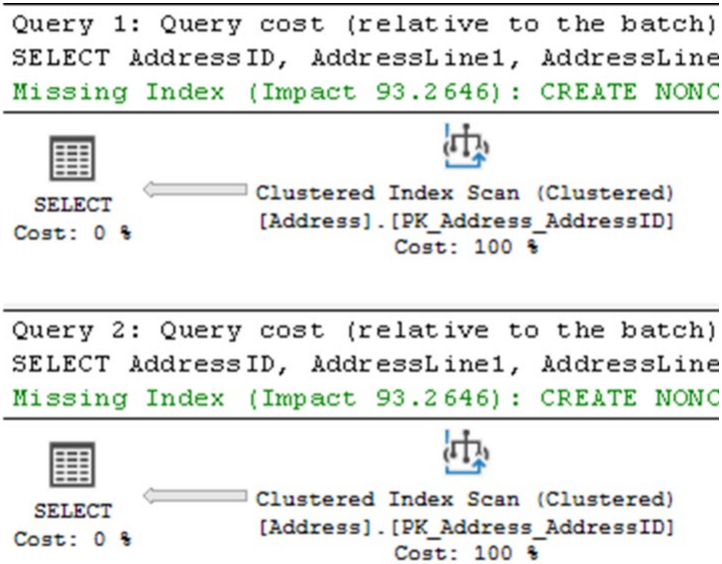


Figure 10-20: Identical execution plans for queries using a local variable.

As described in Chapter 8, the optimizer cannot sniff the value supplied, when we use local variables, unless statement-level recompile takes place because of an `OPTION (RECOMPILE)` hint (covered later). It optimizes for the average distribution, using the density value, to arrive at a cardinality estimation (it's the ratio of number of rows in the table to number of distinct values). If we know that the resulting plan will be good enough for most executions, then we might consider using the `OPTIMIZE FOR UNKNOWN` hint to force the optimizer to produce that generic plan. Listing 10-19 shows an example (I've simply moved the query into a stored procedure).

```
CREATE OR ALTER PROCEDURE dbo.AddressByCity @City NVARCHAR(30)
AS
SELECT  AddressID,
        AddressLine1,
        AddressLine2,
        City,
        StateProvinceID,
        PostalCode,
        SpatialLocation,
        rowguid,
        ModifiedDate
FROM    Person.Address
WHERE   City = @City
OPTION (OPTIMIZE FOR UNKNOWN);
GO
EXEC dbo.AddressByCity @City = N'Mentor';
```

Listing 10-19

Even though `Mentor` is an uncommon city, and so our nonclustered index is selective for this predicate, we still see the "generic" plan.

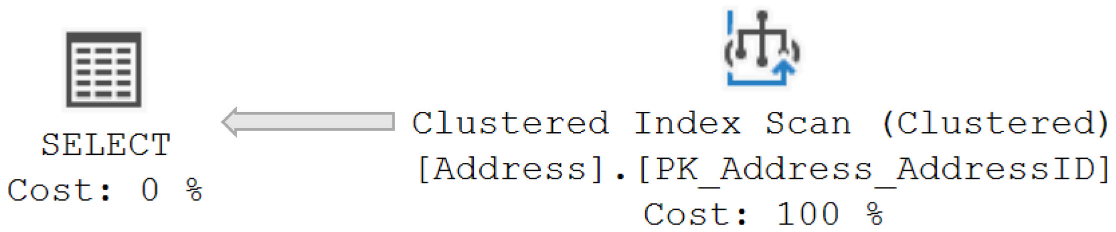


Figure 10-21: The plan once the OPTIMIZE FOR hint has been applied.

Use of the `OPTIMIZE FOR` hint requires intimate knowledge of the underlying data. Choosing the wrong value for `OPTIMIZE FOR` will not only fail to help performance, but could have a very serious negative impact. It's also very important that you maintain the hint, and adapt it as necessary, as the data changes over time.

In the example above, there was only a single variable, so there was only a single hint needed. If you need to control the value used for optimization for more than a single variable in a query, you can set as many hints as necessary. Listing 10-20 shows an example of the necessary syntax.

```
CREATE OR ALTER PROCEDURE dbo.AddressDetails
    @City NVARCHAR(30),
    @PostalCode NVARCHAR(15),
    @AddressLine2 NVARCHAR(60) NULL
AS
SELECT a.AddressLine1,
       a.AddressLine2,
       a.SpatialLocation
FROM Person.Address AS a
WHERE a.City = @City
      AND a.PostalCode = @PostalCode
      AND (   a.AddressLine2 = @AddressLine2
              OR @AddressLine2 IS NULL)
OPTION (OPTIMIZE FOR (@City = 'London', @PostalCode = 'W1Y 3RA'));
```

Listing 10-20

The `OPTIMIZE FOR` hint is one of the few that I use regularly, though still not often. Even so, I strongly recommend you exercise caution and perform lots of tests before applying the `OPTIMIZE FOR` hint. As the data changes over time, you will need to re-evaluate whether the choice you made is still the correct one. In my experience, the `OPTIMIZE FOR UKNOWN` hint is generally more stable than optimizing for a particular value, because of those data changes.

RECOMPILE

We discussed use of the `RECOMPILE` hint in Chapter 8, as a common cure for bad parameter sniffing when using stored procedures or other forms of parameterized SQL, such as prepared statements. We apply the hint to any of the individual queries within the procedure, and it will force SQL Server to recompile the plan for that query every time. The new compile will optimize the plan for the current values of all variables and parameters used in the query (rather than reuse the plan for a previously sniffed value).

The `RECOMPILE` query hint was introduced in SQL Server 2005 along with statement-level recompiles. For stored procedures and other code modules, all statements including the one with `OPTION(RECOMPILE)` will still be in the plan cache, but the plan for the `OPTION(RECOMPILE)` statement will still recompile for every execution, which means that the plan is not reused in any way.

When we use the hint for ad hoc queries, the optimizer marks the plan created so that it is not stored in the cache at all. We discussed the problems that ad hoc queries can cause, such as cache bloat, in Chapter 9. If the problem is caused by lack of parameterization, then the most common fix is to enable the **Optimize for Ad Hoc Workloads** setting. However, if your system executes lots of parameterized ad hoc queries, and you're getting performance problems with bad parameter sniffing, then you might opt to take the hit of having SQL Server compile a plan for every execution, by applying the `RECOMPILE` hint.

Consider the pair of queries in Listing 10-21.

```
SELECT  soh.SalesOrderNumber ,
        soh.OrderDate ,
        soh.SubTotal ,
        soh.TotalDue
FROM    Sales.SalesOrderHeader soh
WHERE   soh.SalesPersonID = 279;
GO
SELECT  soh.SalesOrderNumber ,
        soh.OrderDate ,
        soh.SubTotal ,
        soh.TotalDue
FROM    Sales.SalesOrderHeader soh
WHERE   soh.SalesPersonID = 280;
```

Listing 10-21

This results in the mismatched set of query plans in Figure 10-22, once again demonstrating the optimizer's "tipping point" between choosing a plan with a seek and lookups, versus scanning the clustered index (as discussed in detail in Chapter 8).

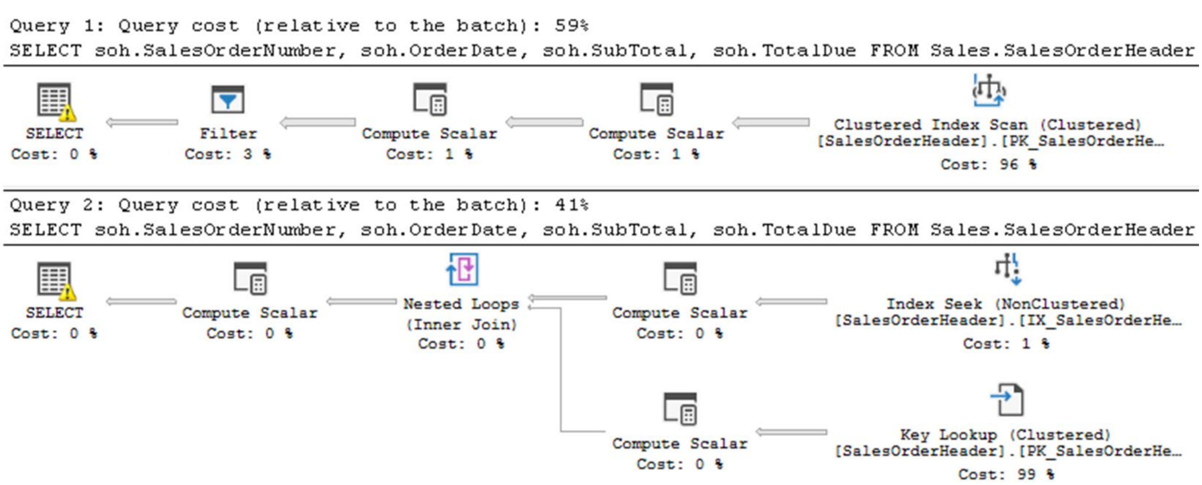


Figure 10-22: The execution plans change radically when recompiled.

If you examine the **Parameter List** property of either **SELECT** operator, it appears that both these queries have gone through Simple Parameterization (covered in Chapter 9). However, the value of the **StatementParameterizationType** property, lower down, tells us that, in fact, they were not parameterized.

Parameter List	@1
Column	@1
Parameter Compiled Value	(279)
Parameter Data Type	smallint
Parameter Runtime Value	(279)
ParentObjectId	0
QueryHash	0x0E8395
QueryPlanHash	0xB192FA
QueryTimeStats	
Reason For Early Termination Of Statement Optimizatio	Good En
RetrievedFromCache	false
SecurityPolicyApplied	False
Set Options	ANSI_NU
Statement	SELECT [
StatementParameterizationType	0

Figure 10-23: A failed attempt at Simple Parameterization.

If this query runs as a prepared statement, we'll see different behavior. Using `sp_prepare` always causes optimization for unknown values (see Chapter 9), and so the optimizer will use the density graph to arrive at a cardinality estimation and generate an appropriate plan, which will then be reused for subsequent executions.

```
DECLARE @IDValue INT;
DECLARE @MaxID INT = 280;
DECLARE @PreparedStatement INT;
SELECT @IDValue = 279;
EXEC sp_prepare @PreparedStatement OUTPUT,
                N'@SalesPersonID INT',
                N'SELECT  soh.SalesPersonID, soh.SalesOrderNumber,
        soh.OrderDate,
        soh.SubTotal,
        soh.TotalDue
FROM    Sales.SalesOrderHeader soh
WHERE   soh.SalesPersonID =  @SalesPersonID';
WHILE @IDValue <= @MaxID
BEGIN
    EXEC sp_execute @PreparedStatement, @IDValue;
    SELECT @IDValue = @IDValue + 1;
END;
EXEC sp_unprepare @PreparedStatement;
```

Listing 10-22

If you query the plan cache (as shown in Chapter 9), or the query store (see Chapter 16), you'll see a single plan, the clustered index scan plan, used twice. This is what you'll see, regardless of whether you execute using the value of 280 first instead of 279, because the optimizer isn't doing parameter sniffing, it's optimizing for an unknown value.

If this lack of parameter sniffing is causing performance issues for one of the queries, and so you prefer to optimize for sniffed variables, then you might consider simply adding `OPTION (RECOMPILE)` to the end of the prepared statement.

```
…
EXEC sp_prepare @PreparedStatement OUTPUT,
                N'@SalesPersonID INT',
                N'SELECT  soh.SalesPersonID, soh.SalesOrderNumber,
        soh.OrderDate,
        soh.SubTotal,
        soh.TotalDue
FROM    Sales.SalesOrderHeader soh
```

```
WHERE   soh.SalesPersonID =  @SalesPersonID
OPTION (RECOMPILE)';
…
```

Listing 10-23

If you execute Listing 10-23 and capture the plans in SSMS, you'll see the two different plans again, but if you check the plan cache you'll see that neither is cached.

EXPAND VIEWS

The EXPAND VIEWS query hint eliminates the use of the indexed, or materialized, views within a query and forces the optimizer to go directly to tables for the data. The optimizer replaces the referenced indexed view with the view definition (in other words, the query used to define the view) just like it normally does with a standard view; but when the EXPAND VIEWS hint is used it will then not try to match the expanded queries with usable indexed views. This behavior can be overridden on a view-by-view basis by adding the WITH (NOEXPAND) clause to any indexed views within the query. Indexed view matching is Enterprise only, so this hint has no effect in a Standard system.

In some instances, the plan generated by referencing the indexed view performs worse than the one that uses the view definition. In most cases, the reverse is true. Test this hint to ensure its use doesn't negatively affect performance.

Using one of the indexed views supplied with AdventureWorks2014, we can run the following simple query.

```
SELECT vspcr.StateProvinceCode,
       vspcr.StateProvinceName,
       vspcr.CountryRegionName
FROM Person.vStateProvinceCountryRegion AS vspcr;
```

Listing 10-24

Figure 10-24 shows the resulting execution plan.

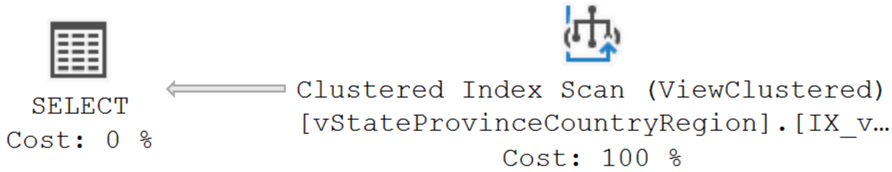


Figure 10-24: Execution plan using an indexed view.

A view is changed into an indexed view by creating a clustered index on it, which stores the data defined by the query in the view. This execution plan makes perfect sense, since the data needed to satisfy the query is available in the indexed view. Things change, as we see in Figure 10-25, if we add the query hint, `OPTION (EXPAND VIEWS)`.

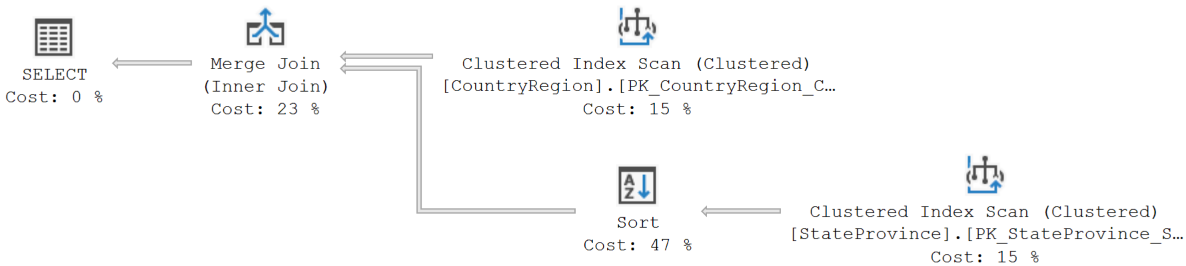


Figure 10-25: View definition expanded out because of the query hint.

Now we're no longer scanning the indexed view. Within the compilation process (before the optimizer is invoked), the view has been expanded into its definition, and so the effect of the hint is that the view matching phase of optimization is skipped. As a result, we see the **Clustered Index Scan** against the `Person.CountryRegion` and `Person.StateProvince` tables. These are then joined using a **Merge Join**, after the data in the `StateProvince` stream is run through a **Sort** operation. The first query ran in about 54ms, but the second ran in about 189ms, so we're talking a substantial decrease in performance to use the hint in this situation.

IGNORE_NONCLUSTERED_COLUMNSTORE_INDEX

As discussed in Chapter 8, the optimizer can choose to use a columnstore index, where appropriate. Columnstore indexes are extremely efficient when assisting aggregation queries, but much less efficient for traditional point lookup queries. As with all the other choices made by the optimizer, the choice of a columnstore index may not always be appropriate.

You can use this query hint to ensure that any existing nonclustered columnstore index is ignored, for the entire query. If the table in question has a clustered columnstore index, this hint does not affect its use within the execution plan.

Join Hints

A **join hint** provides a means to force SQL Server to use one of the three standard join methods that we detailed in Chapter 4, but for a specific join operation rather than all join operations, as we saw when we applied the query hints earlier.

By incuding one of the join hints in your T-SQL, you will potentially override the optimizer's choice of the most efficent join method. Also, as soon as you force a particular join, you're also forcing the join order, effectively the same as using `OPTION (FORCE ORDER)`. In general, this is not a good idea, and if you're not careful you could seriously impede performance.

Application of the join hint applies to any query (`SELECT`, `INSERT`, or `DELETE`) where joins can be applied. Join hints are specified as part of the `JOIN` clause between two inputs (such as tables). You can use the `LOOP`, `HASH`, or `MERGE` join hints in the same fashion. The core behavior won't change. You'll just get a different join depending on the hint you use. Worth noting is that you can't force an **Adaptive Join** using hints, at time of writing.

There is a fourth join method, the **Remote** join, that is used when dealing with data from a remote server. The `REMOTE` join hint forces the join operation from your local machine onto the remote server. This has no effect on execution plans, so we won't be drilling down on this functionality here.

Since all join hints work basically the same, I'm only going to demonstrate the `HASH` join hint, to force use of a **Hash Join** operator. We'll reuse the simple query from an earlier query (Listing 10-7) that lists `Product Models`, `Products`, and `Illustrations`.

```
SELECT pm.Name,
       pm.CatalogDescription,
       p.Name AS ProductName,
       i.Diagram
FROM Production.ProductModel AS pm
    LEFT JOIN Production.Product AS p
        ON pm.ProductModelID = p.ProductModelID
    LEFT JOIN Production.ProductModelIllustration AS pmi
```

```
        ON p.ProductModelID = pmi.ProductModelID
    LEFT JOIN Production.Illustration AS i
        ON pmi.IllustrationID = i.IllustrationID
WHERE pm.Name LIKE '%Mountain%'
ORDER BY pm.Name;
```

Listing 10-25

Once again, we'll get the execution plan shown in Figure 10-26.

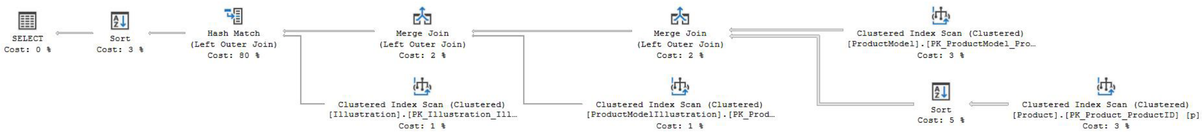


Figure 10-26: An execution plan with joins chosen by the optimizer.

As discussed earlier, this plan (I won't describe it again) entails 485 logical reads and the query ran in about 74ms.

The top input to the final **Nested Loops** join returns 455 rows, which means that the **Clustered Index Scan** on the `Illustration` table executes 455 times. What happens if we decide that we're smarter than the optimizer and that it really should be using a **Hash Match** join instead of that **Nested Loops** join? We can force the issue by adding the `HASH` hint to the join condition between `Illustration` and `ProductModelIllustration`.

```
SELECT pm.Name,
       pm.CatalogDescription,
       p.Name AS ProductName,
       i.Diagram
FROM Production.ProductModel AS pm
    LEFT JOIN Production.Product AS p
        ON pm.ProductModelID = p.ProductModelID
    LEFT JOIN Production.ProductModelIllustration AS pmi
        ON pm.ProductModelID = pmi.ProductModelID
    LEFT HASH JOIN Production.Illustration AS i
        ON pmi.IllustrationID = i.IllustrationID
WHERE pm.Name LIKE '%Mountain%'
ORDER BY pm.Name;
```

Listing 10-26

If we execute this new query, we'll see the plan shown in Figure 10-27.

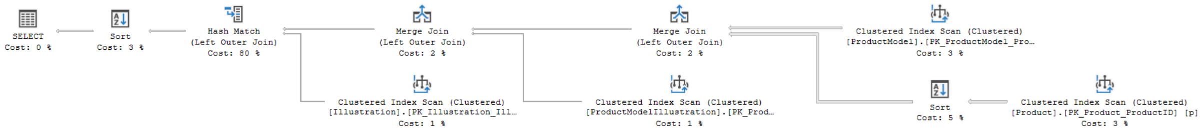


Figure 10-27: The new plan with a forced Nested Loops join.

Sure enough, where previously we saw a **Nested Loops** operator, we now see the **Hash Match** operator. However, the rest of the plan has changed shape as well. The optimizer has decided that the most efficient way to deal with **Hash Match** (which it has no choice but to implement due to our hint), is to change the other joins to **Merge**. This adds the requirement to Sort the data from the `Product` table.

Interestingly, in this case, we drop to 34 logical reads and the execution time drops, just a little, to 74.1ms on average. It's entirely possible that by eliminating the loops we are getting superior performance. The actual difference between 77 and 74 is small, but the reads going from 485 to 34 is a substantial saving. Additional testing on a system under load would be required to determine if, for certain, this hint resulted in superior performance.

Table Hints

Table hints enable you to control how the optimizer "uses" a table when generating an execution plan for the query to which the table hint is applied. For example, you can force the use of a **Table Scan** for that query, or specify which index you want the optimizer to use.

As with the query and join hints, using a table hint circumvents the normal optimizer processes and can lead to serious performance issues. Further, since table hints can affect locking strategies, they could possibly affect data integrity leading to incorrect or lost data. Use table hints sparingly and judiciously!

Most of the table hints are primarily concerned with locking strategies. Since they don't affect execution plans, we won't be covering them. The table hints covered below have a direct impact on the execution plans. For a full list of table hints, please refer to Books Online.

The correct syntax is to use the `WITH` keyword, and then list the hints within a set of parentheses. Listing 10-27 shows an example of applying table hints when the table name directly follows the `FROM` clause, but they can also be used when the table name follows a `JOIN` or `APPLY` keyword.

```
FROM TableName WITH (hint, hint,…)
```

Listing 10-27

The WITH keyword is not required in all cases, nor are the commas required in all cases but, rather than attempt to guess or remember which hints are the exceptions, all hints can be placed within the WITH clause. As a best practice, separate hints with commas to ensure consistent behavior and future compatibility. Even with the hints that don't require the WITH keyword, it must be supplied if more than one hint is to be applied to a given table.

NOEXPAND

When one or more indexed views are referenced within a query, the use of the NOEXPAND table hint will prevent view expansion, roughly the opposite of the EXPAND VIEW hint we used earlier. The query hint affects all views in the query. The table hint will prevent the indexed view to which it applies from being "expanded" into its underlying view definition. The primary use of this hint is to get indexed views to be used inside the plans on Standard Edition systems, because they won't use the materialized view otherwise.

SQL Server Enterprise and Developer editions use the indexes in an indexed view if the optimizer determines that index is best for the query. This is **indexed view matching**, and it requires the following settings for the connection:

- ANSI_NULL set to **On**
- ANSI_WARNINGS set to **On**
- CONCAT_NULL_YIELDS_NULL set to **On**
- ANSI_PADDING set to **On**
- ARITHABORT set to **On**
- QUOTED_IDENTIFIER set to **On**
- NUMERIC_ROUNDABORT set to **Off**.

Using the NOEXPAND hint forces the optimizer to use one of the indexes from the indexed view. In Chapter 7 (Listing 7-11), we used a query that referenced one of the indexed views, vStateProvinceCountryRegion**,** in AdventureWorks2014. During the compilation process, the indexed view was replaced with its definition and then the optimizer did not undo that during view matching, and we saw an execution plan that featured a three-table join. Via use of the NOEXPAND table hint, in Listing 10-28, we change that behavior.

```
SELECT a.City,
       v.StateProvinceName,
       v.CountryRegionName
FROM Person.Address AS a
    JOIN Person.vStateProvinceCountryRegion AS v WITH (NOEXPAND)
        ON a.StateProvinceID = v.StateProvinceID
WHERE a.AddressID = 22701;
```

Listing 10-28

Now, instead of a three-table join, we get the execution plan in Figure 10-28.

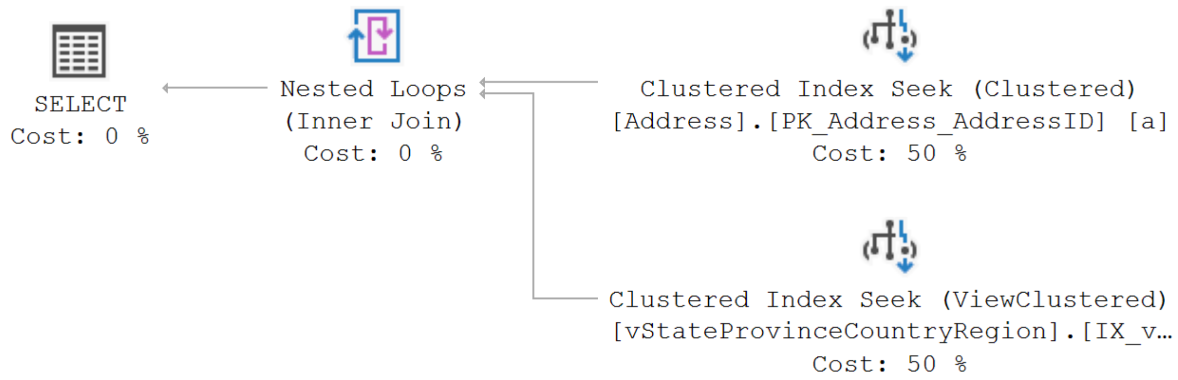


Figure 10-28: A smaller execution due to the use of the NOEXPAND hint.

Now, not only are we using the clustered index defined on the view, but we're also seeing a performance increase, albeit a very small one, from 189ms to 162ms on average on my system. The reads dropped from 6 to 4. In this situation, eliminating the overhead of the extra join resulted in improved performance. That will not always be the case, so you must test the use of hints very carefully.

INDEX()

The `INDEX()` table hint allows you to specify the index to be used when accessing a table. The syntax supports two methods, or four if you include the `WITH (INDEX = (name or number))`, although this syntax doesn't support multiple indexes, so is generally not used.

We can specify the index to use by its number or its name. Indexes are numbered within the `sys.indexes` table. You'll have to look up any given index there. The numbers 0 and 1 cause different behaviors. 0 forces a scan of the clustered index or the heap, while 1 forces either a scan or a seek on a clustered index and produces an error on a heap. The syntax is as follows.

```
…FROM dbo.TableName WITH (INDEX(2))…
```

Listing 10-29

Alternatively, we can simply refer to the index by name, which I recommend, because the order in which indexes are applied to a table can change, so you can't guarantee the value for the number of the index.

```
…FROM dbo.TableName WITH (INDEX (IndexName))…
```

Listing 10-30

You can only have a single `INDEX()` hint for a given table, but you can define multiple indexes within that one hint. This is applicable when you're attempting to perform index joins to retrieve data, forcing an intersection between all indexes on the table, i.e. forcing the optimizer to use all listed indexes, in listed order.

```
…FROM TableName WITH (INDEX (IndexName1,IndexName2))…
```

Listing 10-31

This does not cause the optimizer to pick among only the mentioned indexes, but forces it to use all of them, in the order specified. Within the comma-separated list of indexes, you can match the index number and index name formats. For a quick demo, examine the plan for the following query.

```
CREATE TABLE dbo.IndexSample (ID INT NOT NULL IDENTITY(1, 1),
                              ColumnA INT,
                              ColumnB INT,
                              ColumnC INT,
                              CONSTRAINT IndexSamplePK
                                  PRIMARY KEY
                                  (
                                      ID
                                  ));
```

```
CREATE INDEX FirstIndex ON dbo.IndexSample (ColumnA);
CREATE INDEX SecondIndex ON dbo.IndexSample (ColumnB);
CREATE INDEX ThirdIndex ON dbo.IndexSample (ColumnC);
SELECT isa.ID,
       isa.ColumnA,
       isa.ColumnB,
       isa.ColumnC
FROM dbo.IndexSample AS isa WITH (INDEX(FirstIndex, SecondIndex,
ThirdIndex));
DROP TABLE dbo.IndexSample;
```

Listing 10-32

Now, let's take a simple query that lists department, job title, and employee name.

```
SELECT de.Name,
       e.JobTitle,
       p.LastName + ', ' + p.FirstName
FROM HumanResources.Department AS de
    JOIN HumanResources.EmployeeDepartmentHistory AS edh
        ON de.DepartmentID = edh.DepartmentID
    JOIN HumanResources.Employee AS e
        ON edh.BusinessEntityID = e.BusinessEntityID
    JOIN Person.Person AS p
        ON e.BusinessEntityID = p.BusinessEntityID
WHERE de.Name LIKE 'P%';
```

Listing 10-33

We get a reasonably straightforward execution plan, as shown in Figure 10-29.

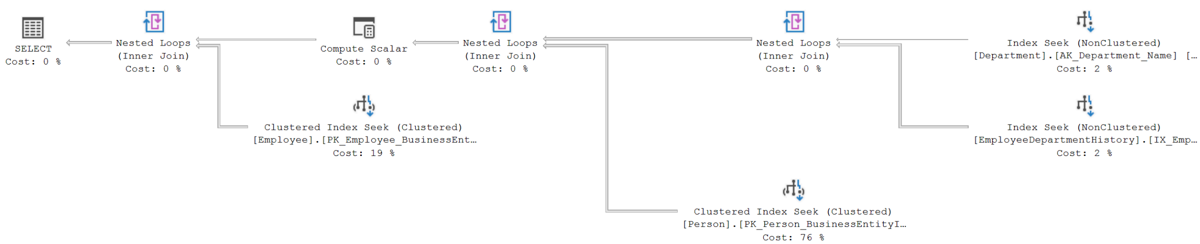


Figure 10-29: Execution plan using indexes chosen by the optimizer.

We see a series of **Index Seek** and **Clustered Index Seek** operators, joined together by **Nested Loops** operators. Suppose we're convinced that we can get better performance if we could eliminate the **Index Seek** on the `HumanResources.Department` table, and instead use that table's clustered index, `PK_Department_DepartmentID`. We could accomplish this using the `INDEX()` hint, as shown in Listing 10-34.

```
SELECT de.Name,
       e.JobTitle,
       p.LastName + ', ' + p.FirstName
FROM HumanResources.Department AS de WITH (INDEX(PK_Department_
DepartmentID))
    JOIN HumanResources.EmployeeDepartmentHistory AS edh
        ON de.DepartmentID = edh.DepartmentID
    JOIN HumanResources.Employee AS e
        ON edh.BusinessEntityID = e.BusinessEntityID
    JOIN Person.Person AS p
        ON e.BusinessEntityID = p.BusinessEntityID
WHERE de.Name LIKE 'P%';
```

Listing 10-34

Figure 10-30 shows the resulting execution plan.

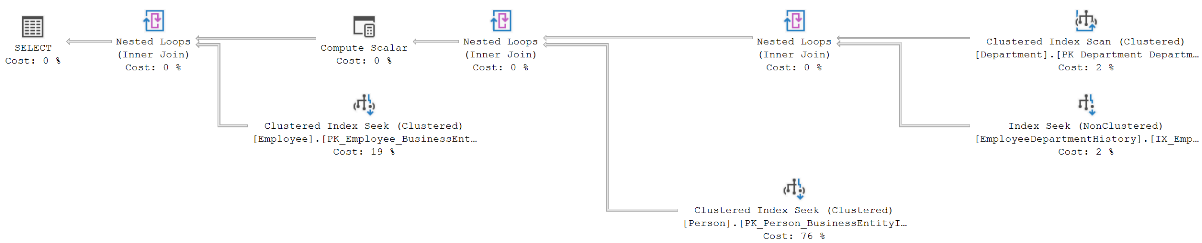


Figure 10-30: An execution plan with forced index choices.

After the hint is added, we can see a **Clustered Index Scan** of one index replacing the **Index Seek** of the other index, just as we told the optimizer to do, although we didn't specify either seek or scan, through the use of the table hint. This change results in a slight improvement in performance in the query, with the execution time coming in at 103ms as opposed to 217ms without the hint. Interestingly, the number of reads for the query overall remained consistent at 1042, regardless of the index used.

FORCESEEK/FORCESCAN

As we have seen throughout this chapter, it is possible to make some choices for the optimizer, which can either hurt or enhance performance. One area that lots of people worry about is the use of indexes. Seeing an index scan leads many people to want to force an index seek in its place, working under the assumption that seeks are always better than scans. However, this is not always the case.

Nevertheless, we can use the `FORCESEEK` or `FORCESCAN` table hints to force the specified type of operator, without forcing the index used. It's rather like the reverse of an index hint, which forces the index but allows the optimizer to choose between scan or seek.

Let's take the query in Listing 10-35 as an example.

```
SELECT p.Name AS ComponentName,
       p2.Name AS AssemblyName,
       bom.StartDate,
       bom.EndDate
FROM Production.BillOfMaterials AS bom
    JOIN Production.Product AS p
        ON p.ProductID = bom.ComponentID
    JOIN Production.Product AS p2
        ON p2.ProductID = bom.ProductAssemblyID;
```

Listing 10-35

As you can probably guess from looking at the query, without a `WHERE` clause to provide any sort of filtering, scans have been used to retrieve the data from the tables in question. You can see this in the execution plan shown in Figure 10-31.

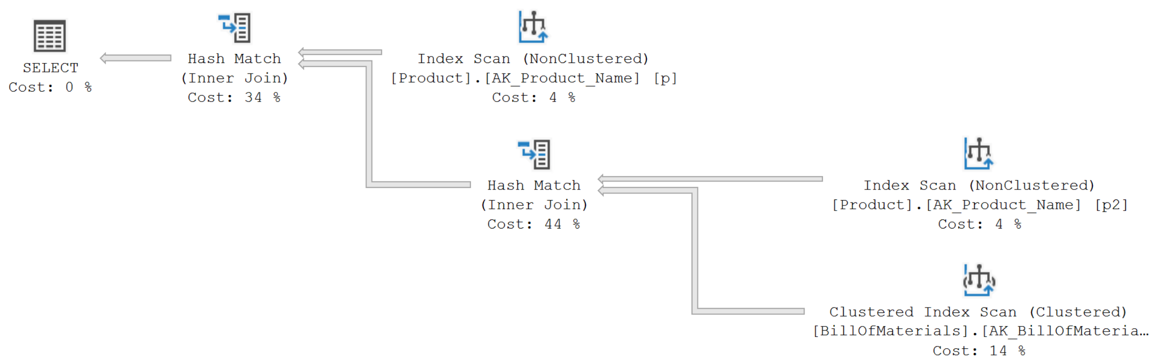


Figure 10-31: An execution plan using scans because of a lack of a WHERE clause.

This is completely normal behavior considering the query in question. However, people really like to see those seeks. Looking at the plan, you can see that the highest estimated cost of any of the index operations is the scan against the `BillOfMaterials` table. Let's see if forcing a seek operation will improve performance.

```
SELECT p.Name AS ComponentName,
       p2.Name AS AssemblyName,
       bom.StartDate,
       bom.EndDate
FROM Production.BillOfMaterials AS bom WITH (FORCESEEK)
    JOIN Production.Product AS p
        ON p.ProductID = bom.ComponentID
    JOIN Production.Product AS p2
        ON p2.ProductID = bom.ProductAssemblyID;
```

Listing 10-36

Taking the choices for a scan away from the optimizer, it is forced to use a seek operation and that also forces other changes on the execution plan, as you can see in Figure 10-32.

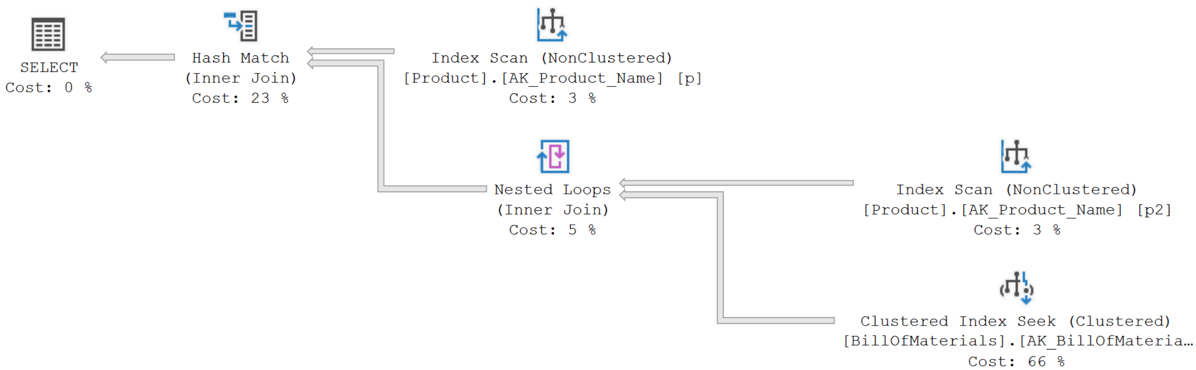


Figure 10-32: Execution plan forcing a Seek operation through the table hint.

The scan of the `BillOfMaterials` table has been replaced with a seek. Also, the **Hash Match** operator has been replaced with a **Nested Loops**. The question is not what changes occurred in the plan, however. The question is, what happened to performance. The execution time went from about 145ms on average to about 290ms. The reads jumped from 34 to 1160. Not only was the query slower because of the seek and the loops join, but the number of reads means that there will be a marked increase in contention for resources on a system under load.

The FORCESCAN operator can be used to go the other way, changing a seek to a scan. Either of these table hints may be useful, depending on the circumstances. However, you must exercise extreme caution in the use of all the table, query, and join hints.

Summary

While the optimizer makes very good decisions most of the time, it may sometimes make less than optimal choices. Taking control of the queries using table, join, and query hints, when appropriate, can often be the right choice. However, remember that the data in your database is constantly changing. Any choices you force on the optimizer through hints today, to achieve whatever improvement you're hoping for, may become a major pain in the future.

If you decide to use hints, test them prior to applying them, and remember to document their use in some manner so that you can come back and test them again periodically as your database grows and changes. As Microsoft releases patches and service packs, the behavior of the optimizer can change. Be sure to retest any queries using hints after an upgrade to your server. I intentionally demonstrated cases where the query hints hurt as well as help, as this simply reflects reality. Hints more often hurt performance than they help it. Use of these hints should be a last resort, not a standard method of operation.

Chapter 11: Parallelism in Execution Plans

SQL Server can take advantage of a server's multiple processors, by spreading the processing of certain operations across the CPUs available to it. Firstly, lots of small queries can run at the same time, each on their own thread. These queries will just have normal execution plans. Secondly, a single query can execute across multiple threads. This latter case, the parallel execution of a single query, will result in a different execution plan, and these differences are our focus in this chapter.

Essentially, when the optimizer detects that its estimated cost for a plan exceeds the "cost threshold," beyond which parallelization of the query will benefit performance, it produces a parallel version of the plan. The work performed by any "parallelized" operators in the parallel plan can be distributed across multiple CPUs, the goal being that, by dividing the work into smaller chunks, the overall operation performs quicker.

For large-scale queries, and for queries using columnstore indexes, query parallelism is extremely desirable for performance. For smaller, OLTP-style queries, it can cause more problems than it solves. By understanding how to read parallelized plans, you'll start to understand how it affects the overall cost of the plan, which operators benefit most, and where the added overhead of parallelism might come into play.

This chapter focuses on the details of parallel execution of a single plan, and only on plans that use the traditional row mode execution model, where the operators pass around data row by row. As mentioned briefly in Chapter 8, columnstore indexes support a new type of query execution model, called **batch mode**, where operators pass around batches of rows rather than single rows. Chapter 12 will cover batch mode in detail, including parallel execution plans that use columnstore indexes.

Controlling Parallel Query Execution

SQL Server has two instance-wide configuration options that determine if, or when, the optimizer might generate parallel execution plans, and also control the parallel execution of queries by the engine. The `max degree of parallelism` (I'll sometimes use `MAXDOP` for brevity) setting determines the maximum number of processors that the SQL Server execution engine can use when executing a parallel query, and the `cost threshold for parallelism` setting, which specifies the threshold, or minimum cost, at which SQL Server creates and runs parallel plans; the cost being measured in this case is the estimated cost of the execution plan.

Of course, parallel query execution requires SQL Server to have access to more than one processor. At compilation time, if the optimizer determines that only one processor is available, or that `MAXDOP` is set to 1, then it will not produce parallel plans. Otherwise, the optimizer will select a plan in the usual fashion and, if the estimated cost of that plan exceeds the `cost threshold for parallelism` value, it will produce a parallel version of the plan.

At runtime, the execution engine then determines across how many processors to parallelize the query, up to the maximum value defined by the instance-level `MAXDOP` setting, or by use of the `MAXDOP` query hint (see Chapter 10). Also, the engine must check with the OS to determine if sufficient threads (an operating system construct that allows multiple concurrent operations) are available for use, prior to launching a parallel process. Plans that are eligible for parallelism may not go parallel. If the execution engine decides that, even though a plan qualifies for parallel execution, there aren't enough resources to support it, then it will simply strip out the parallelism and run a serial version of the plan (Query Store is the only place you'll see both versions of the plan).

Max degree of parallelism

By default, `MAXDOP` is set to 0 (zero), which means that SQL Server can use all available processors to execute a query. If you wish to suppress parallel execution, you set this option to a value of 1. If you wish to specify the number of processors to use for a query execution, then you can set a value of greater than 1, and up to 64.

Without thorough measurement, and tested proof that query parallelism is always going to cause issues, I recommend leaving parallelism on, for most systems. However, I also recommend that you don't leave `MAXDOP` set to zero. Instead, you'll want to set it to a value greater than 1, but less than the total number of available processors, to prevent an expensive, parallelized query from blocking other queries, by "hogging" all available processors.

A very general recommendation is to set this value to half the number of physical cores on your machine, but this doesn't begin to cover all the subtlety and nuances of this topic. Determining a precise setting for `MAXDOP` requires precise knowledge of your operating system, your hardware, whether your system is virtualized and the type of workload that your system runs. Microsoft offers some recommendations on how to determine the right MAXDOP setting for your system: https://bit.ly/2uwvUeI. Paul Randal and the SQLskills team also provide some very detailed recommendations, and punch holes in common myths on the topic: https://bit.ly/2GwQ9Pu. Between these two resources, you should be able to determine the right answer for your system.

You can query the current setting and determine the configuration of this option via the following scripts shown in Listing 11-1.

```
EXEC sys.sp_configure @configname = 'show advanced options',
                      @configvalue = 1;
GO
RECONFIGURE WITH OVERRIDE;
GO
--show the current value
EXEC sys.sp_configure @configname = 'max degree of parallelism'
--change value
EXEC sys.sp_configure @configname = 'max degree of parallelism',
                      @configvalue = 4;
GO
RECONFIGURE WITH OVERRIDE;
GO
EXEC sys.sp_configure @configname = 'show advanced options',
                      @configvalue = 0;
GO
RECONFIGURE WITH OVERRIDE;
GO
```

Listing 11-1

The first statement turns on the advanced options, necessary to access the degree of parallelism. The system is then reconfigured, necessary to actually activate the new setting. Then we query the configuration by passing the first parameter value and not the second to the system procedure, sys.sp_configure.

	name	minimum	maximum	config_value	run_value
1	max degree of parallelism	0	32767	0	0

Figure 11-1: The max degree of parallelism set to the default value of 0.

The `run_value` shows the current setting which, in this case, is the default value of 0. To change the value we call `sys.sp_configure`, passing two values, the setting we wish to change, `max degree of parallelism`, and the value we wish to change it to, 4. The script then resets the advanced options display.

If we were to query the setting again after running the script, we would see the `run_value` had changed to 4.

Cost threshold for parallelism

The optimizer assigns estimated costs to operators within the execution plan. These costs, at one point in the past, represented an estimation of the number of seconds each operation would take. Today, simply think of the cost as just that, estimated cost units. The accumulated values of each of the costs assigned to the operators is the estimated cost of the plan itself. If that estimated cost is greater than the `cost threshold for parallelism`, then that operation may be executed as a parallel operation.

The default value for the `cost threshold for parallelism` is 5. This was probably a good default value back in 1998, when it was first established for SQL Server 7. The number and power of processors, and the type of processors, all have changed radically since then, and I strongly advise you to change that value to something much higher. My rough recommendation would be 25 or more for a reporting system or data warehouse, and 50 for an OLTP system. I make these choices because, in general, you're more likely to see large-scale data movement in reporting systems, where a parallel plan is more likely to benefit query processing. An OLTP system generally only deals with smaller data sets and therefore should be using its processors for lots of queries, not a single query.

Regardless, you should not leave the `cost threshold for parallelism` at the default value, and Listing 11-2 shows how to change it, using the same `sys.sp_configure` function as previously.

```
EXEC sys.sp_configure @configname = 'show advanced options',
                      @configvalue = 1;
GO
RECONFIGURE WITH OVERRIDE;
GO
EXEC sys.sp_configure @configname = 'cost threshold for
parallelism',
                      @configvalue = 50;
GO
```

```
RECONFIGURE WITH OVERRIDE;
GO
EXEC sys.sp_configure @configname = 'show advanced options',
                      @configvalue = 0;
GO
RECONFIGURE WITH OVERRIDE;
GO
```

Listing 11-2

Blockers of parallel query execution

A few code statements can force the entire plan to be serial, regardless of your settings for your `MAXDOP` or cost threshold:

- Scalar functions using T-SQL
- CLR multi-statement, table-valued, or user-defined functions that access data
- Some internal functions in SQL Server such as `ERROR_NUMBER()`, `IDENT_CURRENT()`, `@@TRANCOUNT` and others
- Accessing system tables
- Dynamic cursors.

There are also some T-SQL functions and objects that lead to parts of a plan executing in serial mode (this list can vary depending on the version of SQL Server):

- Recursive CTEs
- TOP
- Paging functions such as `ROW_NUMBER`
- Backward scans
- Multi-statement, table-valued, user-defined functions
- Global scalar aggregates.

The parts of any T-SQL statement using these objects and functions will prevent parallel execution within the plan for the parts of the plan that satisfy these functions.

Parallel Query Execution

When the optimizer determines that a query could benefit from parallelization, it creates a version of the plan optimized for parallel execution. In this parallel plan, you'll see all the familiar operators you've seen previously in the book, except with the yellow "double arrow" icon, indicating that the work performed by the operator will be split across processors. In effect, these operators do the same work as in a serial plan, but on less data. You'll also see extra operators, which handle distribution of data across threads. In plans, these are called **Parallelism** operators, but they are often referred to as **Exchange** operators. These do the "marshaling" job of partitioning the workload into multiple streams of data, passing it through the various parallel operators, and gathering all the streams back together again. You can see an example of these in Figure 11-2.

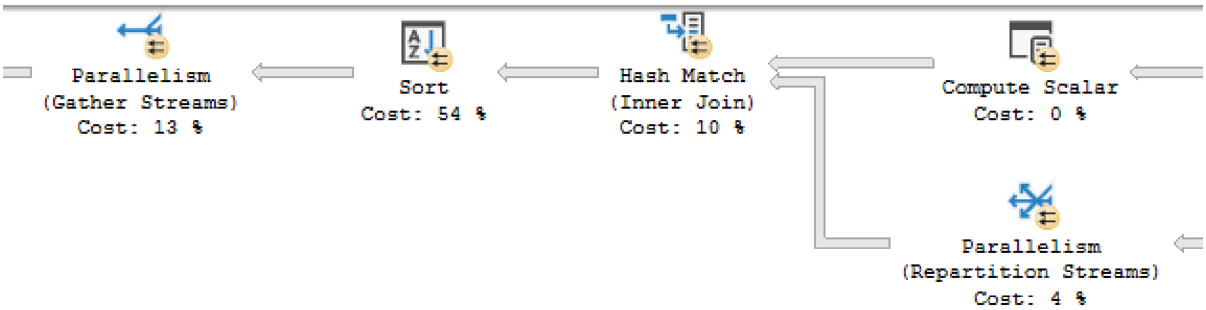


Figure 11-2: Examples of parallel operators in an execution plan.

Most operators are not parallelism aware; they just do their normal work on whatever data they get; the only difference is that they will only process some proportion of the rows, rather than all of them, as they would in a serial plan. In fact, scans, and seeks when used to return ranges of consecutive rows, are the only operators that change their behavior between parallel and serial plans, and we'll discuss that in more detail shortly.

Examining a parallel execution plan

We'll start with an aggregation query, of the sort that you might find in a data warehouse. If the data set this query operates against is very large, it might benefit from parallelism.

```
SELECT   so.ProductID,
         COUNT(*) AS Order_Count
FROM     Sales.SalesOrderDetail so
WHERE    so.ModifiedDate >= '20140301'
         AND so.ModifiedDate < DATEADD(mm, 3, '20140301')
GROUP BY so.ProductID
ORDER BY so.ProductID;
```

Listing 11-3

Figure 11-3 shows the estimated execution plan, which seems straightforward.

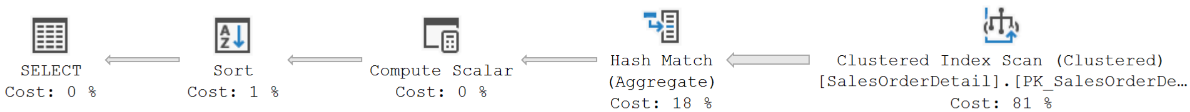


Figure 11-3: A plan that is executing in serial fashion.

There is nothing in this plan that we haven't seen before. One interesting point is that the optimizer decided to use the **Hash Match** operator, and then **Sort** the aggregated data, rather than the alternative, which would be to **Sort** the data emerging from the scan on `SalesOrderDetail` by `ProductID` and then use the **Stream Aggregate** operator. The reason is that the extra cost of sorting about 24 K rows in the latter case, rather than 178 in the former, outweighed any savings from using the cheaper aggregation operator.

Let's move on to see what happens if the optimizer decided to produce a parallelized version of his plan. In this simple example, the total cost of the plan is only 1.3 (you can see this from the **Estimated Subtree Cost** property of the **SELECT** operator, so I'll need to artificially lower the `cost threshold for parallelism` to 1.

```
EXEC sys.sp_configure @configname = 'show advanced options',
                      @configvalue = 1;
GO
RECONFIGURE WITH OVERRIDE;
GO
```

```
EXEC sys.sp_configure @configname = 'cost threshold for
parallelism',
                      @configvalue = 1;
GO
RECONFIGURE WITH OVERRIDE;
GO
SET STATISTICS XML ON;
SELECT so.ProductID,
       COUNT(*) AS Order_Count
FROM Sales.SalesOrderDetail AS so
WHERE so.ModifiedDate >= 'March 3, 2014'
      AND so.ModifiedDate < DATEADD(mm,
                                    3,
                                    'March 1, 2014')
GROUP BY so.ProductID
ORDER BY so.ProductID;
SET STATISTICS XML OFF;
GO
EXEC sys.sp_configure @configname = 'cost threshold for
parallelism',
                      @configvalue = 5; --your value goes here
GO
RECONFIGURE WITH OVERRIDE;
GO
EXEC sys.sp_configure @configname = 'show advanced options',
                      @configvalue = 0;
GO
RECONFIGURE WITH OVERRIDE;
GO
```

Listing 11-4

Figure 11-4 shows the execution plan.

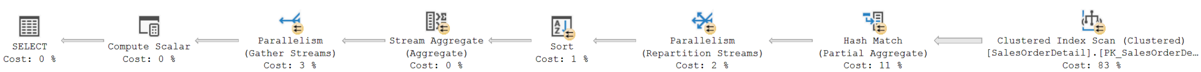


Figure 11-4: A plan that has gone to parallel execution.

Let's start on the left, with the **SELECT** operator. If you look at its **Properties** sheet, you can see the **Degree of Parallelism** property, which in this case is 4, indicating that the execution of this query was split between each of the four available processors. If there had been excessive load on the system at the time of execution, the plan might not have gone parallel, or, it might have used fewer processors. The **Degree of Parallelism** property is a performance metric, captured at runtime and displayed with an actual plan, and so will reflect accurately the parallelism used at runtime.

CompileMemory	416
CompileTime	11
Degree of Parallelism	4
Estimated Number of Rows	265.69
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	1.18784

Figure 11-5: Properties of the SELECT operator showing the Degree of Parallelism.

Looking at the graphical execution plan, we'll start from the right and follow the data flow. First, we find a **Clustered Index Scan** operator. Figure 11-6 shows part of its **Properties** sheet.

Actual Execution Mode	Row
⊞ Actual Number of Batches	0
⊟ Actual Number of Rows	23883
Thread 0	0
Thread 1	0
Thread 2	455
Thread 3	13484
Thread 4	9944
⊞ Actual Rebinds	0
⊞ Actual Rewinds	0
⊞ Defined Values	[AdventureWorks2014].[Sales].[SalesOrder

ForceScan	False
Logical Operation	Clustered Index Scan
Node ID	7
NoExpandHint	False
Number of Executions	4
⊞ Object	[AdventureWorks2014].[Sales].[SalesOrder
Ordered	False
⊞ Output List	[AdventureWorks2014].[Sales].[SalesOrder
Parallel	True
Physical Operation	Clustered Index Scan
Predicate	[AdventureWorks2014].[Sales].[SalesOrder

Figure 11-6: Properties of the Clustered Index Scan showing parallel artifacts.

The **Parallel** property is set to **True**. More interesting is that the **Number of Executions** value indicates that this operator was called 4 times, once for each thread. At the very top of the sheet, you can that 23883 rows matched our predicate on `ModifiedDate`, and we can see how these rows were distributed across four threads, in my case quite unevenly.

Scans and seeks are among the few operators that change their behavior between parallel and serial plans. In parallel plans, rows are provided to each worker thread using a demand-based system where the operator requests rows from a Storage Engine feature called the **Parallel Page Supplier**, which responds to each request by supplying a batch of rows to any thread that asks for more work (this feature is not part of the query processor, so it doesn't appear in the plan).

The data passes on to a **Hash Match** operator, which is performing an aggregate count for each `ProductID` value, as defined by the `GROUP BY` clause within the T-SQL, but only for each row on its thread (the **Hash Match** is not parallelism aware). The result will be one row for each `ProductID` value that appears on a thread (plus its associated count). It is likely that there will be other rows for the same `ProductID` in the other threads, so the resulting aggregates are not the final values, which is why, in the execution plans shown in Figure 11-4, the logical operation performed by the Hash Match is listed as a **Partial Aggregate**, although in every other respect the operator functions in the same way as a Hash Match (Aggregate).

If you inspect the **Properties** of the Hash Match (Partial Aggregate) operator (Figure 11-7), you'll see that it was called 4 times, and again you will see the distribution of the partially aggregated rows across the threads.

Remember that you can, and likely will, see different row counts at this stage of the plan, depending on the degree of parallelism you see in your tests, and on how the rows are distributed across those threads. There are 178 distinct `ProductID` values in the selected data. If all rows for each `ProductID` ended up on the same thread, then you'd see the theoretical minimum total of 178 rows, because the partial aggregate would already be the final aggregate. The theoretical maximum number of rows occurs when every `ProductID` value occurs on every thread. If there are 4 threads, as in my case, the theoretical maximum is 4*178 = 712 rows. I see 470 rows, nicely in between the theoretical minimum and maximum.

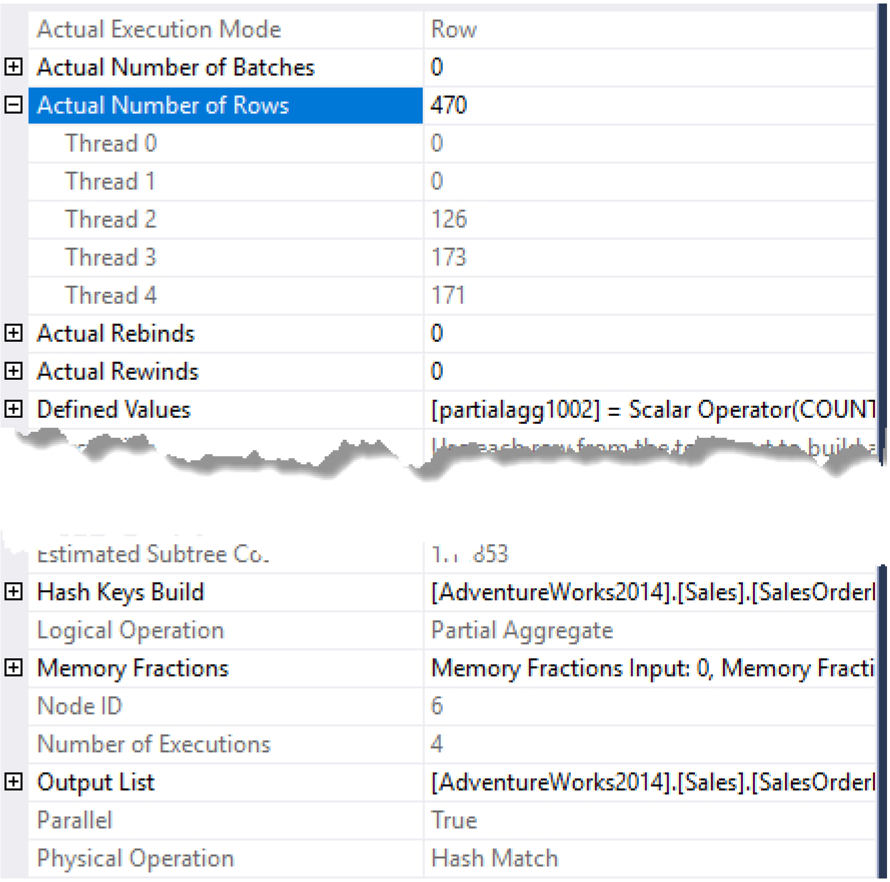


Figure 11-7: Properties of the Hash Match showing parallel artifacts.

The rows pass to a **Parallelism** operator (often referred to, remember, as an Exchange operator), which implements the **Repartition Streams** operation. You can think of this operator, generally, as being responsible for routing rows to the right thread. Sometimes this is done just to balance the streams, trying to make sure that a roughly equal amount of work is performed by each stream. Other times, its main function is to ensure that all rows that need to be processed by a single instance of an operator are on the same thread. This is an example of the latter; the operator is used to ensure that the columns with matching `ProductID` values are all on the same thread, so that the final, global aggregation can be performed. We can see that in the properties the partitioning type is **Hash** (there are other partitioning types too, such as **Round Robin** and **Broadcast**) and the partition column is `ProductID`.

⊟ Partition Columns	[AdventureWorks2014].[S
Alias	[so]
Column	ProductID
Database	[AdventureWorks2014]
Schema	[Sales]
Table	[SalesOrderDetail]
Partitioning Type	Hash
Physical Operation	Parallelism

Figure 11-8: Partition Column properties of the Repartition Streams operation.

Figure 11-9 shows the results of this "rerouting" in the operator properties.

Actual Execution Mode	Row
Actual Number of Batches	0
Actual Number of Rows	470
Thread 0	0
Thread 1	123
Thread 2	98
Thread 3	121
Thread 4	128
Actual Rebinds	0
Actual Rewinds	0
Description	Repartition streams.
Estimated CPU Cost	0.0294116
Estimated Execution Mode	Row
Estimated I/O Cost	0
Estimated Number of Executions	1
Estimated Number of Rows	531.38
Estimated Operator Cost	0.02941 (2%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	19 B
Estimated Subtree Cost	1.14794
Logical Operation	Repartition Streams
Node ID	5
Number of Executions	4

Figure 11-9: Rows rearranged inside the threads.

You can see the **Actual Number of Rows** property, and how the threads have been rearranged with a roughly even distribution of rows. A more even distribution of data across threads was a happy side-effect in this case. However, if the `ProductID` values had not been spread equally in the hash algorithm used, then this could just as easily have added more skew.

Conceptually, you can imagine the plan, up to this point, as looking like Figure 11-10. Again, the exact row counts will differ for you, but it demonstrates the execution of the query on multiple threads, and the distribution, then repartitioning, of the rows across those threads.

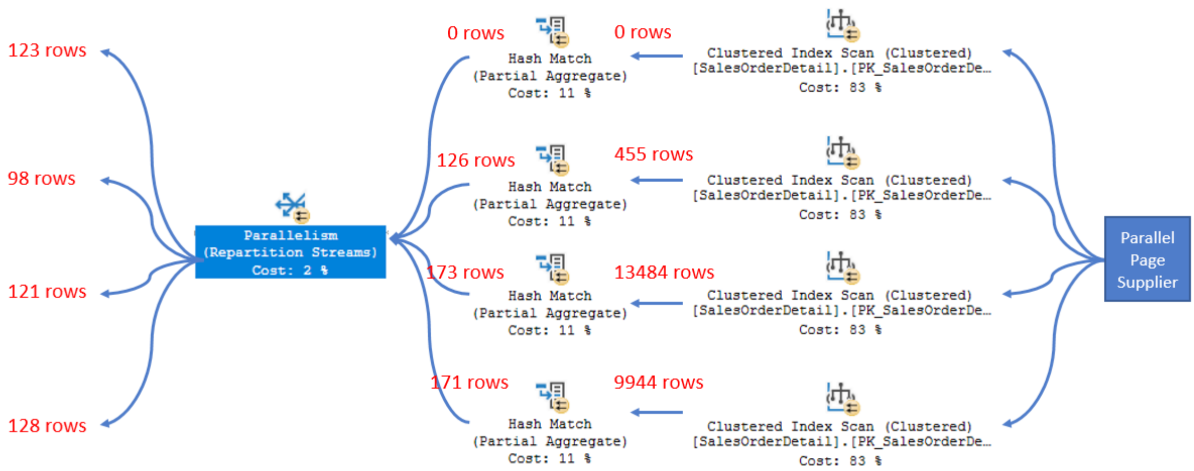


Figure 11-10: An imaginary example of what is happening in parallel execution.

After the partial aggregation and repartitioning, all rows for a given `ProductID` value will be on the same thread, which means that each of the four threads will have up to four rows per `ProductID`. The rows need to be aggregated again to complete the "local-global aggregation."

Now that the number of rows is reduced substantially by the partial aggregation, the optimizer estimates that it is cheaper to sort the data into the correct order so that a **Stream Aggregate** operator can do the final aggregation, rather than use another **Hash Match**. A **Sort** operator is one that benefits greatly from parallelization, and often shows a significant reduction in total cost, compared to the equivalent serial **Sort**.

The next operator is another **Parallelism** operator, performing the **Gather Streams** operation. The function of this operator is somewhat self-explanatory, in that it gathers the streams back together, to present the data as a single data set to the query or operator calling it. The output from this operator is now a single thread of data. The one, very important, property that I will call out here is the **Order By** property, as shown in Figure 11-11.

Order By	[AdventureWorks2014].[Sales
Ascending	True
⊟ Column Reference	[AdventureWorks2014].[Sales
Alias	[so]
Column	ProductID
Database	[AdventureWorks2014]
Schema	[Sales]
Table	[SalesOrderDetail]

Figure 11-11: Properties of the Parallelism operator showing the Order By property.

In the previous **Parallelism** operator (**Repartition Streams**), this property was absent, meaning that it just read each of the input threads and sent packets of rows on to each output thread as soon as it could. The incoming order of the data is not guaranteed to be preserved.

However, if the order is preserved, then you will see different behavior. If the data in each thread is already in the correct order, then an order-preserving exchange operator will wait for data to be available on all inputs, and merge them into a single stream that is still in the correct order. This means that an order-preserving Exchange can be a little slower than one that doesn't preserve order. However, because parallelized sorting is so efficient, the optimizer will usually favor a plan with a parallel sort, and an order-preserving **Parallelism** operator, over a plan with a non-order-preserving **Parallelism** operator, and a serial sort of all the data.

From this point on, the plan is just a normal, "serial" plan, working on a single thread of data, which passes next to the **Compute Scalar** operator, which converts the aggregated column to an `int`. This implies that internally, during the aggregation phases of the plan, that value was a `bigint`, but it's unclear. Finally, the data is returned through the **SELECT** operator.

Are parallel plans good or bad?

Parallelism comes at a cost. It takes processing time and power to divide an operation into various threads, coordinate the execution of each of those threads, and then gather all the data back together again. If only a few rows are involved, then that cost will far outweigh the benefits of putting multiple CPUs to work on the query.

However, given how quickly operator costs increase, and the cost of certain operators in particular (such as **Sort**s), with the number of rows to process, it's likely that parallelism will make a lot of sense for any long-running, processor-intensive, large-volume queries, including most queries that use columnstore indexes. You'll see this type of activity mainly in reporting, warehouse, or business intelligence systems.

In an OLTP system, where the majority of the transactions are small and fast, parallelism can sometimes cause a query to run slower that it would have run with a serial plan. Sometimes, it can cause the parallelized query to run a bit faster, but the extra resources used cause queries on all other connections to run slower, reducing overall performance of the system. Most of the time the optimizer does a good job of avoiding these situations, but it can sometimes make poor choices. However, even in OLTP systems, some plans, such as for reporting queries, will still benefit from parallelism. The general driving factor here is the estimated costs of these plans, which is why setting the `cost threshold for parallelism` setting becomes so important.

There is no hard-and-fast rule for determining when parallelism may be useful, or when it will be costlier. The best approach is to observe the execution times and wait states of queries that use parallelism, as well as the overall workload, using metrics such as "requests per second." If the system deals with an especially high level of concurrent requests, then allowing one user's query to parallelize and occupy all available CPUs will probably cause blocking problems. Where necessary, either change the system settings to increase the cost threshold and `MAXDOP`, or use the `MAXDOP` query hint in individual cases.

It all comes down to testing to see if you are gaining a benefit from the parallel processes, and query execution times are usually the surest indicator of this. If the time goes down with `MAXDOP` set to 1 during a test, that's an indication that the parallel plan is hurting you, but it doesn't mean you should disable parallelism completely. You need to go through the process of choosing appropriate settings for `Max Degree of Parallelism` and `Cost Threshold for Parallelism`, and then measure your system performance and behaviors with parallel plans executing.

Summary

The chapter explained the basics of how you can read through a parallel execution plan. Parallelism doesn't fundamentally change what you do when reading execution plans, it just requires additional knowledge and understanding of a few new **Parallelism** operators, and the potential impact on other operators in the plan, so you can start to see what types of queries really benefit, and to spot the cases where the added overhead of parallelism becomes significant.

Parallel execution of queries can be a performance enhancer. It can also hurt performance. You need to ensure that you've set your system up correctly, both the `Max Degree of Parallelism` and the `Cost Threshold for Parallelism`. With those values correctly set, you should benefit greatly by limiting the execution of parallel queries to those that really need it.

Chapter 12: Batch Mode Processing

Introduced with the columnstore index in 2012, batch mode processing is a new way for the query engine to process queries, allowing it to pass batches of rows between operators, rather than individual rows, which can radically improve performance in some situations.

For many queries that use columnstore indexes, parallel execution is desirable for performance. As a result, batch mode processing tends to be discussed together with parallel processing, but in fact parallel execution is not required for all types of batch mode processing, and batch mode is available in non-parallel execution in SQL Server 2016 and later, as well as in Azure SQL Database.

Elsewhere in the book, such as when we discussed Adaptive Joins back in Chapter 4, you've seen some evidence of the row or batch mode processing in the properties of operators, but here we're going to discuss in detail what it is and how it works, the characteristics of execution plans for queries that use batch mode and, finally, some of its limitations.

At the time of writing, only tables with columnstore indexes support this new batch mode execution model, and so this chapter will only discuss execution plans for queries that access tables with a columnstore index, and execute in batch mode. However, Microsoft recently announced that an upcoming release of SQL Server will also introduce batch mode to rowstore queries, so this type of processing is going to expand.

Batch Mode Processing Defined

The traditional processing mechanism, row mode, has been described throughout the book. An operator will request rows from the preceding operator, process each row that it receives and then pass that row on to the next operator as it requests rows (or, in the case of a blocking operator, request all input rows one by one, and then return all result rows one by one). This constant request negotiation is a costly operation within SQL Server. It can, and does, slow things down, especially when we start dealing with very large data sets.

Batch mode processing reduces the frequency of the negotiation process, thereby increasing performance. Instead of passing along individual rows, operators pass rows on in batches, generally 900-row batches, and then only the batches are negotiated. So, if we assume 9,000 rows moving between operators, instead of 9,000 negotiations to move the rows, you'll see 10 negotiations (9,000 rows / 900 = 10 batches), radically reducing the overhead for processing the data.

The batch mode size won't always be 900 rows; that is the value provided by Microsoft as guidance. However, it can vary as it's largely dependent on the number and size of the columns being passed through the query. You'll see some examples where the batch size is less than 900, but I've yet to see a case where it is more than 900 rows in a batch, although I've seen no evidence that 900 is a hard maximum, and you may see different behaviors, depending on the SQL Server version.

Plan for Queries that Execute in Batch Mode

As discussed in Chapter 8, columnstore indexes are designed to improve workloads that involve a combination of very large tables (millions of rows), and analysis, reporting and aggregation queries that operate on all rows, or on large selections. It is for these types of queries that batch mode execution can really improve performance, rather than typical OLTP queries that process single rows or small collections of rows.

We're going to focus on how row and batch mode processing appear in execution plans, and how you can determine which processing mode you're seeing, again based on information supplied through the execution plan.

To get started with batch mode, *and* demonstrate the resulting changes in behavior within execution plans, we'll need to create a columnstore index on a pretty big table. Fortunately, Adam Machanic has posted a script that can create a couple of large tables within Adventure-Works for just this sort of testing. You can download the script from http://bit.ly/2mNBIhg.

With the larger tables in place, Listing 12-1 creates a nonclustered columnstore index on the `bigTransactionHistory` table.

```
CREATE NONCLUSTERED COLUMNSTORE INDEX TransactionHistoryCS
ON dbo.bigTransactionHistory
(
    ProductID,
    TransactionDate,
    Quantity,
    ActualCost,
    TransactionID
);
```

Listing 12-1

We'll start off with a simple query that groups information together for analysis, as shown in Listing 12-2.

```
SELECT th.ProductID,
       AVG(th.ActualCost),
       MAX(th.ActualCost),
       MIN(th.ActualCost)
FROM dbo.bigTransactionHistory AS th
GROUP BY th.ProductID;
```

Listing 12-2

Figure 12-1 shows the actual execution plan.

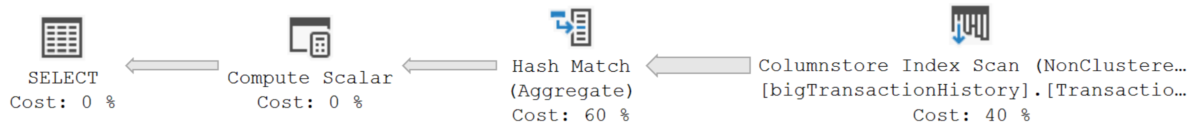


Figure 12-1: The Columnstore Index Scan operator.

On my system, the database compatibility level is 140, and the `cost threshold for parallelism` is 50 (the estimated cost of the serial plan is just under 25; see the **Estimated Subtree Cost** property of the **SELECT** operator). If your compatibility level setting is different, or your cost threshold setting is below 25, then you may see a parallelized version of the plan.

Following the flow of data from right to left, the first operator is the **Columnstore Index Scan** (described in Chapter 8). There is nothing in Figure 12-1 to indicate visually whether this operator is using batch mode or row mode, but the Properties sheet, shown in Figure 12-2 reveals the pertinent pieces of information.

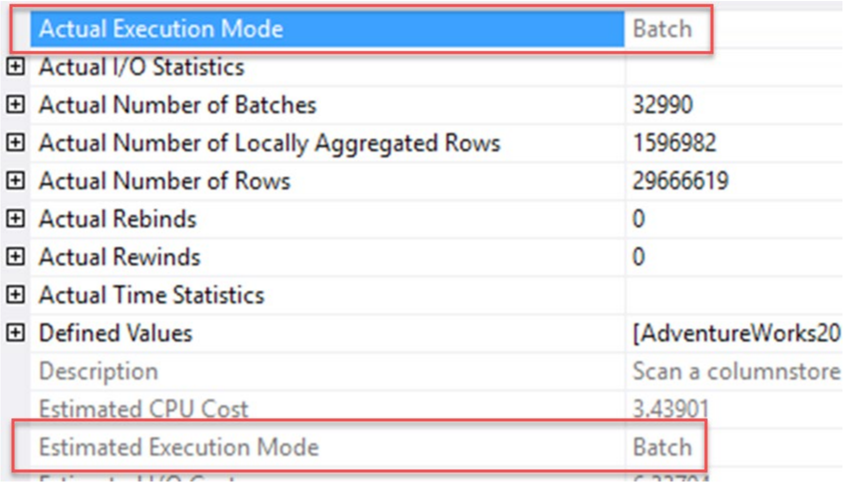


Figure 12-2: Batch mode in the properties of the Columnstore Index Scan operator.

As you can see in Figure 12-2, there is an estimated and actual execution mode that will designate an operator as performing in batch mode or not. This operator was estimated to use batch mode and then, when the query ran, batch mode was used. Prior to SQL Server 2016, with a non-parallel plan such as this, batch mode was not available for serial plans such as this (more on this shortly).

This operator scanned the whole table (over 31 million rows) and returned to the **Hash Match (Aggregate)** operator 29,666,619 rows, in 32,990 batches. The remaining 1,596,982 were aggregated locally (due to aggregate pushdown, as described in Chapter 8); the results of this local aggregation were injected directly into the **Hash Match (Aggregate)** operator's results. Aggregation was on the `ProductID`. Figure 12-3 shows the tooltip for the **Hash Match**, which will also reveal whether operators in the plan used batch mode processing.

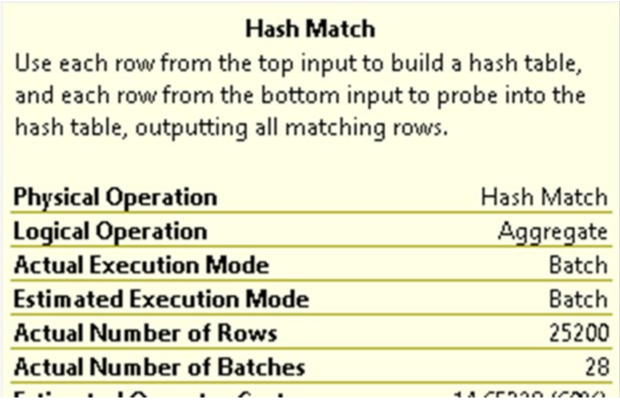


Figure 12-3: Portion of the Hash Match operator's tooltip.

The **Hash Match** operator also used batch mode processing. It received almost 30 million rows in 32990 batches and, after aggregation, returned 25,200 rows in 28 batches. So, there were about 899 rows per batch coming in and 846 going out, both close to the 900 value stated earlier.

Batch mode prior to SQL Server 2016

If we change the compatibility level of the database from the SQL Server 2017 value of 140 to the SQL Server 2014 value of 120, it can change the behavior of our batch mode operations, because fewer operations supported batch mode in earlier versions of SQL Server.

```
ALTER DATABASE AdventureWorks2014
SET COMPATIBILITY_LEVEL = 120;
```

Listing 12-3

Now, when we rerun the query from Listing 12-2, we'll get a parallelized execution plan, as shown in Figure 12-5.

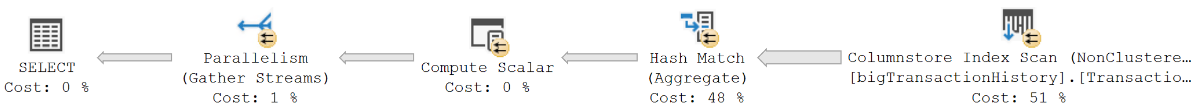


Figure 12-4: A parallel execution plan against a columnstore index.

The primary, visible difference between the plan in Figure 12-4 and the one in Figure 12-1 is the addition of the **Parallelism (Gather Streams)** operator that pulls parallel execution back into a single data stream.

Prior to SQL Server 2016, it was very common for queries to never go into batch mode unless they were costly enough to run in parallel, because it was a requirement of batch mode processing that the plan be parallel. In this instance, because batch mode was not available to the serial plan, the cost of that serial plan was high enough that the `cost threshold for parallelism` was exceeded, and the plan went parallel. If you want to verify this, you can add the `OPTION (MAXDOP 1)` hint to Listing 12.2, and capture the actual plan, and you'll see that the cost of the serial plan is now approximately 150. You'll also notice that the optimizer no longer chooses the columnstore index, and that's because the estimated cost of using it in a serial plan is even higher (approximately 200). You can verify this by adding an index hint (see Chapter 10) to force use of the columnstore index.

More generally, depending on the query, you may also see different costs in different SQL Server versions and compatibility modes, due to changes in both the options that the query optimizer can use and the cardinality estimation engine in use.

In SQL Server 2012 and 2014 (and corresponding compatibility levels), a query below the threshold for parallelism would never use batch mode. In those earlier SQL Server versions, if you wanted to see batch mode within your queries, you would need to lower the cost threshold for parallelism. If that wasn't viable, you would be forced to modify the query to add an undocumented trace flag, 8649, that artificially lowers the cost threshold for parallelism to zero, making sure that any query will run in parallel. Listing 12-4 shows how to use the `QUERYTRACEON 8649` hint to force parallel execution.

```
SELECT th.ProductID,
       AVG(th.ActualCost),
       MAX(th.ActualCost),
       MIN(th.ActualCost)
FROM dbo.bigTransactionHistory AS th
GROUP BY th.ProductID
OPTION(QUERYTRACEON 8649);
```

Listing 12-4

Before we continue, let's change the compatibility mode back to 140.

```
ALTER DATABASE AdventureWorks2014
SET COMPATIBILITY_LEVEL = 140;
```

Listing 12-5

Mixing columnstore and rowstore indexes

The previous section showed how execution plans exposed batch mode processing. Your next question might be: what happens when you're mixing columnstore and rowstore indexes within the same query? In fact, batch mode works regardless of the type of index used to read the rows. The only requirement for batch mode is that at least one of the tables in the query must have a columnstore index (even if it's not useful for the query). As long as this is true then you may see plans with some operators using row mode and some using batch mode processing, depending on the operators involved. Let's see an example that joins rowstore and columnstore data.

```
SELECT bp.Name,
       AVG(th.ActualCost),
       MAX(th.ActualCost),
       MIN(th.ActualCost)
FROM dbo.bigTransactionHistory AS th
    JOIN dbo.bigProduct AS bp
      ON bp.ProductID = th.ProductID
GROUP BY bp.Name;
```

Listing 12-6

Figure 12-5 shows the resulting execution plan (if your `cost threshold for parallelism` is 26 or more).

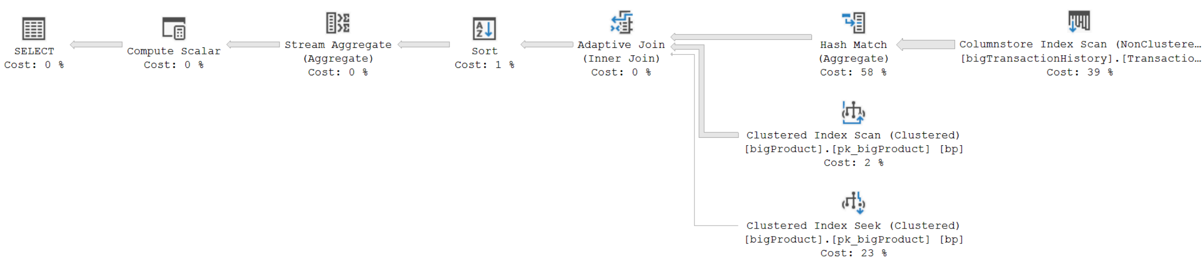


Figure 12-5: An execution plan combining rowstore and columnstore data.

Once again, if you inspect the properties of the **Columnstore Index Scan**, you'll see that it is using batch mode execution, and that it again uses an early aggregation enhancement called aggregate pushdown, where some (or sometimes all) of the aggregation is done by the scan itself, as the data is read. Doing this reduces the number of rows returned to the **Hash Match** by about 1.5 million.

The data passes to a **Hash Match (Aggregate)** operator which, again, is using batch mode execution. You might be surprised to see not one but two aggregation operators in this plan, for what is a relatively simply query. This is another example of the optimizer opting to use both local and global aggregation. We also saw a "local-global" aggregation in Chapter 11 (Listing 11-4), as part of a row-mode parallel plan. In that case, the **Hash Mash** operator was clearly marked as **(Partial Aggregate)**, because it was only working on the data in this one thread, but it behaved in the same way as a normal aggregate operator.

Here, the **Defined Values** and **Hash Key Build** properties of the **Hash Match (Aggregate)** offer some insight into what is occurring. Figure 12-6 shows the **Defined Values**.

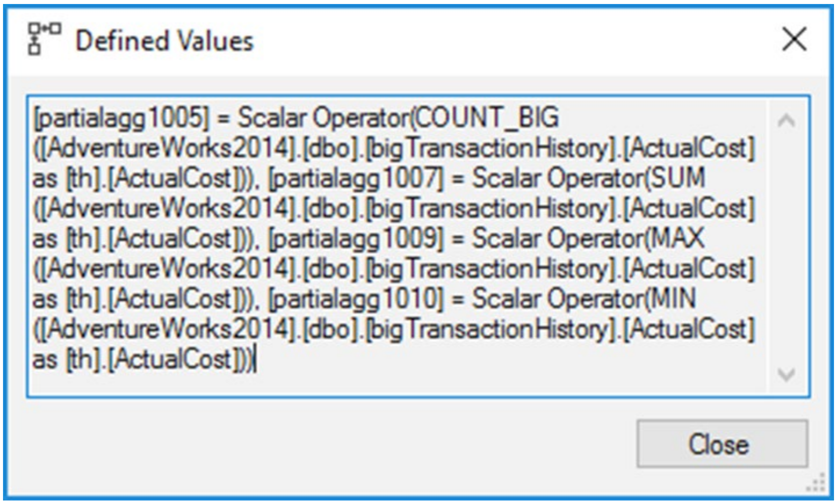


Figure 12-6: Defined Values showing partial aggregation.

You can see that a value called **[partialagg1005]** is created, consisting of the aggregation of several columns in the data set. The aggregation is being performed on the `ProductID` column, as shown in the **Hash Keys Build** property.

Hash Keys Build	[AdventureWorks2014].[dt
Alias	[th]
Column	ProductID
Database	[AdventureWorks2014]
Schema	[dbo]
Table	[bigTransactionHistory]

Figure 12-7: Local aggregation based on ProductID.

Based on the output from the **Columnstore Index Scan** index, the optimizer has decided that an early aggregation on `ProductID` will make the later aggregation, by the product `Name` as we defined in the T-SQL, more efficient. Consequently, the number of rows returned to the subsequent join operation is reduced from approximately 30 million rows (from the base table) to just 25200 (the number of distinct `ProductID` values), as shown by the **Actual Number of Rows** property.

This data stream is joined with rows in the `bigProduct` table, based on matching `ProductID` values. It uses an **Adaptive Join** (see Chapter 4), again executing in batch mode. The **Actual Join Type** used is Hash Match, with the **Clustered Index Scan** as the lower input (chosen because the number of rows returned exceeds the **Adaptive Threshold Rows** property value). The **Clustered Index Scan** used row mode processing.

At this point the product `Name` column values are available and, after a batch mode **Sort** operator, we see the **Stream Aggregate** operator, which uses the partial aggregates to perform the final "global" aggregation on `Name`.

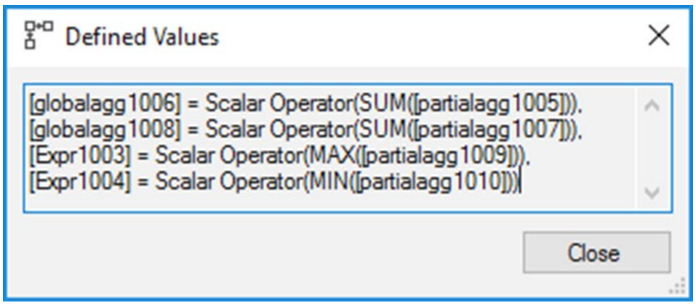


Figure 12-8: Global aggregation for final values.

The **Stream Aggregate** operator used row mode processing, since it does not support batch mode.

Batch mode adaptive memory grant

Finally, for batch mode processing, let's look at one more query, a stored procedure as shown in Listing 12-7.

```
CREATE OR ALTER PROCEDURE dbo.CostCheck (@Cost MONEY)
AS
SELECT p.Name,
       AVG(th.Quantity)
FROM dbo.bigTransactionHistory AS th
    JOIN dbo.bigProduct AS p
        ON p.ProductID = th.ProductID
WHERE th.ActualCost = @Cost
GROUP BY p.Name;
```

Listing 12-7

Listing 12-8 shows how we could execute the `CostCheck` procedure.

```
EXEC dbo.CostCheck @Cost = 0;
```

Listing 12-8

Figure 12-9 shows the execution plan.

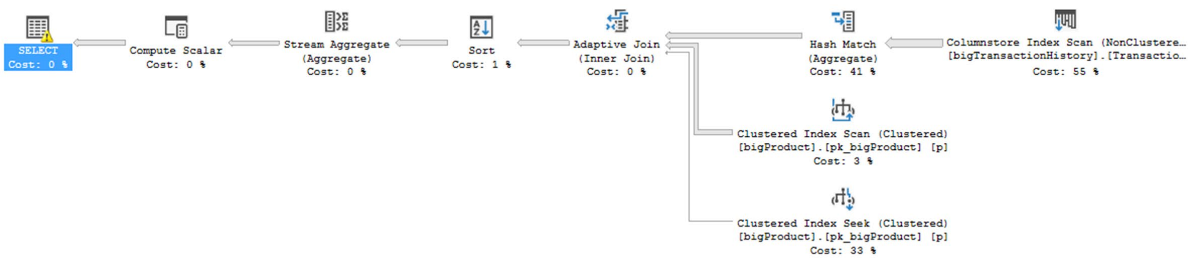


Figure 12-9: Execution plan with a Warning indicator.

You'll see the warning there on the **SELECT** operator of the plan. While we can see the warning from the tooltip, it will only show the first warning. If there is more than one warning, it's best to use the properties. Figure 12-10 shows the **Warnings** section of the properties for the **SELECT**.

Warnings	The query memc
⊟ MemoryGrantWarning	
GrantedMemory	80840
GrantWarningKind	ExcessiveGrant
MaxUsedMemory	3192
RequestedMemory	80840

Figure 12-10: MemoryGrantWarning properties.

The full text of the warning is as follows:

```
The query memory grant detected "ExcessiveGrant," which may impact
the reliability. Grant size: Initial 80840 KB, Final 80840 KB, Used
3192 KB.
```

The initial estimate on rows from the columnstore index was 12.3 million, but the actual was only 10.8 million. From there, the **Hash Match** operator estimated that the aggregation, based on the statistics sampled, would return 25200 rows. The **Hash Match (Aggregate)**, in this example, uses a hash table optimized for aggregation, because it stores `GROUP BY` values and intermediate aggregation results, instead of storing all input rows unchanged, as other **Hash Match** operators do. This means that its memory grant is based on the estimated number of rows produced (25200), not read. However, it only produced 10 K rows. This over-estimation also affects the memory grant for the subsequent **Adaptive Join** which, until the end of its build phase, will require memory at the same time as the **Hash Match**, and the memory grant for the **Sort**.

In short, these over-estimated row counts meant that a larger amount of memory was requested, 80840, than was consumed, 3192. However, while SQL Server often allows a large margin for error in its memory allocations to prevent spills, these relatively modest over-estimations don't quite explain why the memory grant estimate is quite so big in this case.

Starting in SQL Server 2017 and in Azure SQL Database, the query engine can now adjust the memory grant for subsequent executions, either up or down, based on the values of the previous executions of the query. In short, if we re-execute the query, the memory allocation, during batch mode processing, will adjust itself on the fly. Let's take an example. Assuming I've just executed Listing 12-8, I'm going to execute the stored procedure again, supplying a different value for the `@Cost` parameter.

```sql
EXEC dbo.CostCheck @Cost = 15.035;
```

Listing 12-9

This query has a similar result set. Executing this will result in reusing the execution plan already in cache. However, because we're doing batch mode processing, the memory grant can be adjusted on subsequent executions based on similar processes that enable the adaptive join. The plan now looks as shown in Figure 12-11.

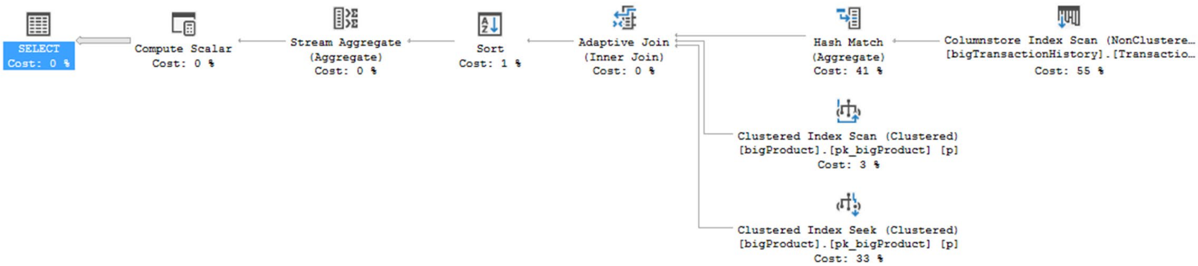


Figure 12-11: Execution plan without a warning.

The warning has been removed, even though the plan has not been recompiled, statistics haven't been adjusted, or any of the other processes that would normally result in a change to the memory allocation. The memory grant has been adjusted on the fly as we can see in the **SELECT** operator properties in Figure 12-12.I've also expanded the **Parameter List** property, to verify that the optimizer has reused the plan compiled for a `Cost` of zero.

Memory Grant	6296
⊞ MemoryGrantInfo	
⊞ MissingIndexes	
Optimization Level	FULL
⊞ OptimizerHardwareDependentProperties	
⊞ OptimizerStatsUsage	
⊟ Parameter List	@Cost
Column	@Cost
Parameter Compiled Value	($0.0000)
Parameter Data Type	money
Parameter Runtime Value	($15.0350)

Figure 12-12: Properties showing adjusted memory.

The adaptive memory can work in either direction, either under- or over-calculated memory allocations, to adjust the memory during subsequent executions of similar allocations. However, this could lead to thrashing if a query has lots of different types of allocations so, at some point, automatically, the adaptive memory will be turned off. This is tracked on a per-plan basis. It can be turned off for one query and still works for other queries, and it will be turned on again every time the plan for a query is recompiled.

You can't tell directly from a single plan whether adaptive memory has been turned off for that plan. You would have to set up monitoring through Extended Events to observe that behavior. If you suspect it's happened, you can compare the values of the memory allocation from one execution to the next. If they are not changing, even though the query experiences spills or large over-allocation, then the adaptive memory grant has been disabled.

While adaptive memory is only available currently with batch mode processing, Microsoft has stated that they will enable row mode adaptive memory processing at some point in the future.

Loss of Batch Mode Processing

SQL Server 2017, when dealing with columnstore indexes, has a very heavy bias towards using batch mode processing for all, or at least part, of any query executed against the columnstore index. If you're working on SQL Server 2014 or 2016, then you'll find that certain of the following operations will not run in batch mode:

- `UNION ALL`
- `OUTER JOIN`
- `IN/EXISTS` or `NOT IN/NOT EXISTS`
- `OR` in `WHERE`
- Aggregation without `GROUP BY`
- `OVER`

You will need to check the actual execution plan, because it's going to show whether the operators within the plan used batch mode or if they went to row mode. However, on testing all these in SQL Server 2017, the plan always went, in whole or in part, to batch mode processing.

Summary

The new batch mode execution mode means that the query engine can pass around large groups of rows at once, rather than moving data row by row. In addition, there are some specific performance optimizations that are only available in batch mode.

For now, batch mode comes with certain preconditions. It currently only works with queries on tables that have a columnstore index, but that is going to change in the future. In older SQL Server versions, batch mode is supported by a relatively limited set of operators.

Batch mode can offer huge performance benefits when processing large data sets, but queries that perform point lookups and limited range scans are still better off in row mode.

Chapter 13: The XML of Execution Plans

Behind each of the execution plans we've been examining up to this point in the book is XML. An "XML plan" is not in any way different from a graphical plan; it contains the same information you can find in the operators and properties of a graphical plan. XML is just a different format in which to view that same plan. If we save a plan, it will be saved in its native XML format, which makes it easy to share with others.

I would imagine that very few people would prefer to read execution plans in the raw XML format, rather than graphical. Also, the XML having received barely a mention in the previous twelve chapters of this book, it should be clear that you don't need to read XML to understand execution plans. However, there are a few cases where access to it will be useful, which I'll highlight, and then we'll discuss the one overriding reason why you may want to use the raw XML data: programmability. You can run XQuery T-SQL queries against XML files and XML plans. In effect, this gives us a direct means of querying the plans in the Plan Cache.

A Brief Tour of the XML Behind a Plan

The easiest way to view the XML for any given plan in SSMS is simply to right-click on any graphical plan and select **Show Execution Plan XML** from the context menu.

If required, you can capture the XML plan programmatically, by encapsulating the batch within `SET SHOWPLAN_XML ON/OFF` commands, for the estimated plan, or `SET STATISTICS XML ON/OFF` for the actual plan (more on this later).

The XML for an estimated plan

Display the estimated plan for the query in Listing 13-1, which retrieves some details for customers in the state of New York..

```
SELECT c.CustomerID, a.City, s.Name, st.Name
  FROM Sales.Customer AS c
    JOIN Sales.Store AS s
      ON c.StoreID = s.BusinessEntityID
    JOIN Sales.SalesTerritory AS st
```

```
        ON c.TerritoryID = st.TerritoryID
      JOIN Person.BusinessEntityAddress AS bea
        ON c.CustomerID = bea.BusinessEntityID
      JOIN Person.Address AS a
        ON bea.AddressID = a.AddressID
      JOIN Person.StateProvince AS sp
        ON a.StateProvinceID = sp.StateProvinceID
   WHERE st.Name = 'Northeast' AND sp.Name = 'New York';
 GO
```

Listing 13-1

Figure 13-1 shows the usual graphical plan.

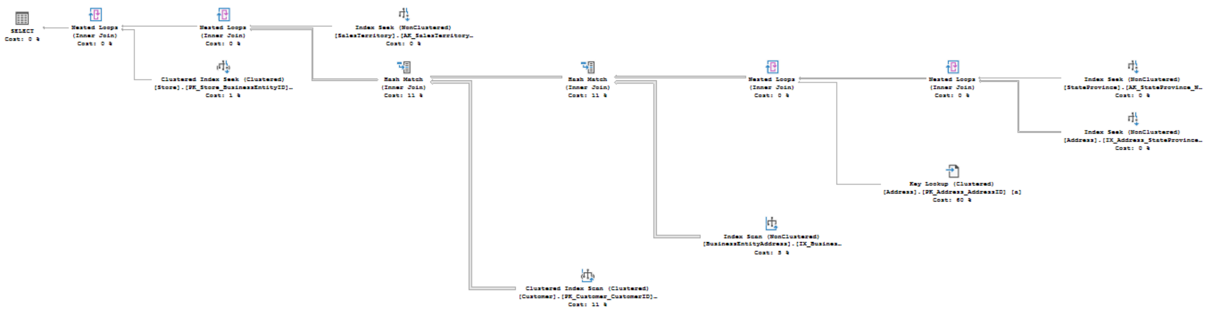


Figure 13-1: Execution plan for New York state customers query.

Right-click in any white space area of the plan and choose **Show Execution Plan XML** to get to the XML behind this estimated plan. The results, even for our simple query, are too large to output here, and Figure 13-2 just shows the opening section. Content is often added in new SQL Server versions, and the order of attributes, and sometimes elements, can differ between versions, so don't worry if it looks different on your system.

```xml
<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan" Version="1.2" Build="12.0.2000.8">
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementText="SELECT  c.CustomerID ,&#xD;&#xA;        a.City ,&#xD;&#xA;        s.Name ,&#xD;&#xA;        s
          <StatementSetOptions QUOTED_IDENTIFIER="true" ARITHABORT="true" CONCAT_NULL_YIELDS_NULL="true" ANSI_NULLS="true" ANSI_
          <QueryPlan NonParallelPlanReason="CouldNotGenerateValidParallelPlan" CachedPlanSize="88" CompileTime="15" CompileCPU="
            <MemoryGrantInfo SerialRequiredMemory="2048" SerialDesiredMemory="2632" />
            <OptimizerHardwareDependentProperties EstimatedAvailableMemoryGrant="260902" EstimatedPagesCached="65225" EstimatedA
            <RelOp NodeId="0" PhysicalOp="Nested Loops" LogicalOp="Inner Join" EstimateRows="1" EstimateIO="0" EstimateCPU="5.93
              <OutputList>
                <ColumnReference Database="[AdventureWorks2014]" Schema="[Sales]" Table="[Customer]" Alias="[c]" Column="Custome
                <ColumnReference Database="[AdventureWorks2014]" Schema="[Sales]" Table="[Store]" Alias="[s]" Column="Name" />
                <ColumnReference Database="[AdventureWorks2014]" Schema="[Sales]" Table="[SalesTerritory]" Alias="[st]" Column="
                <ColumnReference Database="[AdventureWorks2014]" Schema="[Person]" Table="[Address]" Alias="[a]" Column="City" /
              </OutputList>
              <NestedLoops Optimized="0">
                <OuterReferences>
                  <ColumnReference Database="[AdventureWorks2014]" Schema="[Sales]" Table="[Customer]" Alias="[c]" Column="Store
                </OuterReferences>
                <RelOp NodeId="1" PhysicalOp="Nested Loops" LogicalOp="Inner Join" EstimateRows="14.1868" EstimateIO="0" Estimat
                  <OutputList>
                    <ColumnReference Database="[AdventureWorks2014]" Schema="[Sales]" Table="[Customer]" Alias="[c]" Column="Cus
                    <ColumnReference Database="[AdventureWorks2014]" Schema="[Sales]" Table="[Customer]" Alias="[c]" Column="Sto
                    <ColumnReference Database="[AdventureWorks2014]" Schema="[Sales]" Table="[SalesTerritory]" Alias="[st]" Colu
                    <ColumnReference Database="[AdventureWorks2014]" Schema="[Person]" Table="[Address]" Alias="[a]" Column="Cit
                  </OutputList>
                  <NestedLoops Optimized="0">
                    <Predicate>
                      <ScalarOperator ScalarString="[AdventureWorks2014].[Sales].[SalesTerritory].[TerritoryID] as [st].[Territo
                        <Compare CompareOp="EQ">
                          <ScalarOperator>
                            <Identifier>
                              <ColumnReference Database="[AdventureWorks2014]" Schema="[Sales]" Table="[SalesTerritory]" Alias="
                            </Identifier>
                          </ScalarOperator>
                          <ScalarOperator>
                            <Identifier>
                              <ColumnReference Database="[AdventureWorks2014]" Schema="[Sales]" Table="[Customer]" Alias="[c]" C
                            </Identifier>
                          </ScalarOperator>
                        </Compare>
                      </ScalarOperator>
                    </Predicate>
                    <RelOp NodeId="2" PhysicalOp="Index Seek" LogicalOp="Index Seek" EstimateRows="1" EstimateIO="0.003125" Esti
                      <OutputList>
                        <ColumnReference Database="[AdventureWorks2014]" Schema="[Sales]" Table="[SalesTerritory]" Alias="[st]"
                        <ColumnReference Database="[AdventureWorks2014]" Schema="[Sales]" Table="[SalesTerritory]" Alias="[st]"
                      </OutputList>
                      <IndexScan Ordered="1" ScanDirection="FORWARD" ForcedIndex="0" ForceSeek="0" ForceScan="0" NoExpandHint="0
                        <DefinedValues>
```

Figure 13-2: The XML of an execution plan.

Right at the start, we have the schema definition. The XML has a standard structure, consisting of elements and attributes, as defined and published by Microsoft. A review of some of the common elements and attributes and the full schema is available at https://bit.ly/2BU9Yhf .

Listed first are the `BatchSequence`, `Batch`, and `Statements` elements. In this example, we're only looking at a single batch and a single statement, so nothing else is displayed.

Next, as part of the `StmtSimple` element, we see the text of the query followed by a list of attributes of the statement itself. After that, the `StatementSetOptions` element shows the database-level options that were in force. Listing 13-2 shows the `StmtSimple` and `StatementSetOptions` for the estimated plan.

```
<StmtSimple StatementText="SELECT c.CustomerID, a.City, s.Name,
                           st.Name FROM Sales.Customer AS c
                           JOIN Sales.Store AS s ON c.StoreID =
s.BusinessEntityID
                           JOIN Sales.SalesTerritory AS st
                            …etc…
                           WHERE st.Name = 'Northeast'AND sp.Name =
'New York'"
            StatementId="1" StatementCompId="1"
StatementType="SELECT"
            StatementSqlHandle="0x0900A7CAC098F11600D1596466
7F6395453000000 …"
            DatabaseContextSettingsId="3" ParentObjectId="0"
            StatementParameterizationType="0"
RetrievedFromCache="true"
            StatementSubTreeCost="1.04758" StatementEstRows="1"
            SecurityPolicyApplied="false" StatementOptmLevel="FULL"
            QueryHash="0x6F422E0A48C0E2DA" QueryPlanHash="0xBF47C49
83DC8361D"
             StatementOptmEarlyAbortReason="TimeOut"
             CardinalityEstimationModelVersion="140">
  <StatementSetOptions QUOTED_IDENTIFIER="true" ARITHABORT="true"
                       CONCAT_NULL_YIELDS_NULL="true" ANSI_
NULLS="true"
                       ANSI_PADDING="true" ANSI_WARNINGS="true"
                       NUMERIC_ROUNDABORT="false">
  </StatementSetOptions>
```

Listing 13-2

Next is the `QueryPlan` element, which shows some plan- and optimizer-level properties (the `OptimizerStatsUsage` element is collapsed).

```
<QueryPlan NonParallelPlanReason="CouldNotGenerateValidParallelPl
an"
            CachedPlanSize="104" CompileTime="10" CompileCPU="10"
            CompileMemory="1160">
  <MemoryGrantInfo SerialRequiredMemory="2048"
```

```
SerialDesiredMemory="2632" />
  <OptimizerHardwareDependentProperties EstimatedAvailableMemoryGrant="157286"
                                         EstimatedPagesCached="19660"
                                         EstimatedAvailableDegreeOfParallelism="2"
                                         MaxCompileMemory="1475656"
/>
  <OptimizerStatsUsage>…</OptimizerStatsUsage>
```

Listing 13-3

Collectively, Listings 13-2 and 13-3 show the same information available to us by looking at the properties of the first operator, in this case a **SELECT**, in the graphical plan. You can see information such as the `CompileTime`, the `CachedPlanSize` and the `Statement-OptmEarlyAbortReason`. These get translated to **Compile Time**, **Cached Plan Size**, and **Reason for Early Termination of Optimization** when you're looking at the graphical plan. As always, some of the values in your XML (for estimated costs and row counts, for example) may differ from those shown here.

Within the `QueryPlan` element is a nested hierarchy of `RelOp` elements, each one describing an operator in the plan and its properties. The `RelOp` elements are listed in the order in which they are called, akin to reading a graphical plan left to right, so in Figure 13-2 you can see that the very first operator called, with a `NodeId` of "0," is a **Nested Loops** operator, followed by another **Nested Loops**, with a `NodeId` of "1," and then an **Index Seek** on the `SalesTerritory` table, and so on.

XML data is more difficult to take in, all at once, than the graphical execution plans, but you can expand and collapse elements using the "+" and "–" nodules down the left-hand side, and in doing so, the hierarchy of the plan becomes somewhat clearer. Nevertheless, finding specific operators in the XML is not easy, especially for complex plans. If you know the `NodeId` of the operator (from the graphical plan) then you can do a **Ctrl-F** for **NodeID="xx."**

Listing 13-4 shows the properties of the first **Nested Loops** join (reformatted somewhat for legibility).

```
<RelOp NodeId="0" PhysicalOp="Nested Loops" LogicalOp="Inner Join"
EstimateRows="1"
       EstimateIO="0" EstimateCPU="5.9304e-005" AvgRowSize="149"
       EstimatedTotalSubtreeCost="1.04758" Parallel="0"
```

```
EstimateRebinds="0"
        EstimateRewinds="0" EstimatedExecutionMode="Row">
```

Listing 13-4

After that, we see a nested element, `OutputList`, showing the data returned by this operator (I've reformatted it, and reduced nesting levels, for readability). This operator, as you would expect, returns values for all the columns requested in the `SELECT` list of our query.

```
<OutputList>
  <ColumnReference Database="[AdventureWorks2016]" Schema="[Sales]"
                    Table="[Customer]" Alias="[c]"
Column="CustomerID" />
  <ColumnReference Database="[AdventureWorks2016]" Schema="[Sales]
                   "  Table="[Store]" Alias="[s]" Column="Name" />
  <ColumnReference Database="[AdventureWorks2016]" Schema="[Sales]"
                    Table="[SalesTerritory]" Alias="[st]"
Column="Name" />
  <ColumnReference Database="[AdventureWorks2016]"
Schema="[Person]"
                    Table="[Address]" Alias="[a]" Column="City" />
</OutputList>
```

Listing 13-5

For complex plans, I find this a relatively easily digestible way to see all the columns and their attributes returned.

After that we see the `NestedLoops` element, which contains elements for specific properties of this operator, as shown in Listing 13-6. In this case, we can see that this operator resolves the join condition using `OuterReferences` (see Chapter 4 for a full description). Below that, I've included the collapsed version for the two inputs to the first operator, the outer input being another `NestedLoops` (with `NodeId="1"`), and the inner input a **Clustered Index Seek** (`NodeId="14"`), which is the last operator called in this plan.

The `StoreID` column values returned by the outer input are pushed down to the inner input, where they are used to perform a Seek operation on the `Store` table to return the `Name` column for matching rows.

```
  <NestedLoops Optimized="0">
    <OuterReferences>
      <ColumnReference Database="[AdventureWorks2016]"
Schema="[Sales]"
                        Table="[Customer]" Alias="[c]"
Column="StoreID">
      </ColumnReference>
    </OuterReferences>
    <RelOp AvgRowSize="101" EstimateCPU="0.00059304" EstimateIO="0"
           EstimateRebinds="0" EstimateRewinds="0"
EstimatedExecutionMode="Row"
           EstimateRows="14.1876" LogicalOp="Inner Join" NodeId="1"
Parallel="false"
           PhysicalOp="Nested Loops" EstimatedTotalSubtreeCo
st="1.03974">…</RelOp>
    <RelOp AvgRowSize="61" EstimateCPU="0.0001581"
EstimateIO="0.003125"
           EstimateRebinds="1.78078" EstimateRewinds="11.4068"
           EstimatedExecutionMode="Row" EstimateRows="1"
EstimatedRowsRead="1"
           LogicalOp="Clustered Index Seek" NodeId="14"
Parallel="false"
           PhysicalOp="Clustered Index Seek" EstimatedTotalSubtreeC
ost="0.00778646"
            TableCardinality="701">…</RelOp>
```

Listing 13-6

By contrast, within the equivalent element for the second **Nested Loops** (`NodeId="1"`), when you expand the first input again you will see that this operator resolves the join condition to the `Store` table, using a `Predicate` property. I've not shown the whole predicate but, in short, this operator receives `TerritoryID` and `Name` values from the **Index Seek** on the `SalesTerritory` table, and will join this data with that from the bottom input, only returning rows that have matching values for `TerritoryID` in the `Customer` table.

```
<NestedLoops Optimized="0">
  <Predicate>
    <ScalarOperator
         ScalarString="[AdventureWorks2016].[Sales].
[SalesTerritory].[TerritoryID]
                        as [st].[TerritoryID]=
                      [AdventureWorks2016].[Sales].[Customer].
[TerritoryID]
```

```
                           as [c].[TerritoryID]">
…Etc…
  </Predicate>
```

Listing 13-7

The XML for an actual plan

If you return to Listing 13-1 and this time execute it and capture the actual plan, you'll see all the same information as in the XML for the estimated plan, plus some new elements, namely those that are only populated at execution time, rather than compile time.

Listing 13-8 compares the content of the `QueryPlan` element for the estimated plan (shown first) and then the actual plan. You can see that the latter contains additional information, including the `DegreeOfParallelism` (more on parallelism in Chapter 11), the `MemoryGrant` (which is the amount of memory needed for the execution of the query), and some additional properties within the `MemoryGrantInfo` element.

```
<QueryPlan
  NonParallelPlanReason="CouldNotGenerateValidParallelPlan"
  CachedPlanSize="104" CompileTime="9" CompileCPU="9"
CompileMemory="1160">
<MemoryGrantInfo SerialRequiredMemory="2048"
SerialDesiredMemory="2632">
<QueryPlan
  DegreeOfParallelism="0"
  NonParallelPlanReason="CouldNotGenerateValidParallelPlan"
  MemoryGrant="2632"
  CachedPlanSize="104" CompileTime="9" CompileCPU="9"
CompileMemory="1160">
<MemoryGrantInfo SerialRequiredMemory="2048"
SerialDesiredMemory="2632"
  RequiredMemory="2048" DesiredMemory="2632" RequestedMemory="2632"
  GrantWaitTime="0" GrantedMemory="2632" MaxUsedMemory="640"
MaxQueryMemory="576112" />
```

Listing 13-8

Another major difference is that, in the XML for an actual plan, each operator has a `RunTimeInformation` element, showing the thread, actual rows, and the number of executions for that operator along with additional information.

```
<RelOp AvgRowSize="149" EstimateCPU="5.9304E-05" EstimateIO="0"
       EstimateRebinds="0" EstimateRewinds="0"
EstimatedExecutionMode="Row"
       EstimateRows="1" LogicalOp="Inner Join" NodeId="0"
Parallel="false"
       PhysicalOp="Nested Loops" EstimatedTotalSubtreeCo
st="1.04758">
  <OutputList>
  …Etc…
  </OutputList>
  <RunTimeInformation>
    <RunTimeCountersPerThread Thread="0" ActualRows="1" Batches="0"
                              ActualEndOfScans="1"
ActualExecutions="1"
                              ActualExecutionMode="Row"
                              ActualElapsedms="6" ActualCPUms="6"
/>
  </RunTimeInformation>
```

Listing 13-9

Safely Saving and Sharing Execution Plans

Though you can output an execution plan directly in its native XML format, you can only save it from the graphical representation. If we attempt to save to XML directly from the result window we only get what is on display in the result window. Another option is to use a PowerShell script, or similar, to output from XML to a **.sqlplan** file.

Simply right-click on the graphical plan and select **Save Execution Plan As…** to save it as a **.sqlplan** file. This XML file, as we've seen, provides all the information in the plan, including all properties. This can be a very useful feature. For example, we might collect multiple plans in XML format, save them to file and then open them in easy-to-view (and to compare) graphical format. This is useful to third-party applications, too (covered briefly in Chapter 17).

A word of caution, though; as we saw earlier, the XML of the execution plan stores both the query and parameter values. That information could include proprietary or personally identifying information. Exercise caution when sharing an execution plan publicly.

When You'll Really Need the XML Plan

As you can see, while all the information is in there, reading plans directly through the XML is just not as easy as reading the graphical plan and the property sheets for each operator. However, there are a few specific cases where you'll need the XML, and I'll review those briefly here (there may be others!).

Use the XML plan for plan forcing

In Chapter 9, we discussed plan forcing, by using a plan guide to apply the `USE PLAN` query hint. Here, you need to supply the plan's XML to the `@hints` parameter of the `sp_create_plan_guide` system stored procedure, when creating the plan guide.

To do this, you'll first need to capture the plan programmatically. This query pulls some information from the `Purchasing.PurchaseOrderHeader` table and filters the data on the `ShipDate`.

```
SET STATISTICS XML ON;
SELECT  poh.PurchaseOrderID,
        poh.ShipDate,
        poh.ShipMethodID
FROM    Purchasing.PurchaseOrderHeader AS poh
WHERE   poh.ShipDate BETWEEN '20140103' AND '20140303';
GO
SET STATISTICS XML OFF;
```

Listing 13-10

Figure 13-3 shows the result, in the default grid mode.

Figure 13-3: A clickable link to the XML plan in SSMS.

If you have your query results outputting to text mode, you'll see some of the XML string, but it won't be clickable and, depending on the settings within SSMS, it may not be complete.

In grid mode, clicking on this link opens the execution plan as a graphical plan. However, instead, if you're doing plan forcing, just right-click on the link, copy it and paste it into the `@hints` parameter when creating the plan guide.

First operator properties when capturing plans using Extended Events

When you capture a plan using Extended Events, you won't see the first operator, the **SELECT**, **INSERT**, **UPDATE**, or **DELETE** in the graphical plan, so you won't have access to all the useful metadata it hides, except by switching to the XML representation, where some of it is still stored. This is because the XML for the plans captured using Extended Events (and Trace Events, for that matter) differs from every other source of execution plans (SSMS, plan cache, and the Query Store).

Listing 13-11 shows the relevant section of the XML, for a plan captured using Extended Events (you'll see how to do this in Chapter 15), between the `Statement` element and the first `RelOp` element.

```
<StmtSimple StatementSubTreeCost="1.04758" StatementEstRows="1"
            SecurityPolicyApplied="false" StatementOptmLevel="FULL"
            QueryHash="0x6F422E0A48C0E2DA" QueryPlanHash="0xBF47C49
83DC8361D"
            StatementOptmEarlyAbortReason="TimeOut"
            CardinalityEstimationModelVersion="140">
  <QueryPlan DegreeOfParallelism="0" MemoryGrant="2632"
             NonParallelPlanReason="CouldNotGenerateValidParallelPl
an"
             CachedPlanSize="104" CompileTime="10" CompileCPU="10"
             CompileMemory="1160">
    <MemoryGrantInfo SerialRequiredMemory="2048"
SerialDesiredMemory="2632"
                     RequiredMemory="2048" DesiredMemory="2632"
                     RequestedMemory="2632" GrantWaitTime="0"
GrantedMemory="2632"
                     MaxUsedMemory="640" MaxQueryMemory="573840">
    </MemoryGrantInfo>
```

```
    <OptimizerHardwareDependentProperties EstimatedAvailableMemoryG
rant="157286"

EstimatedPagesCached="19660"
                                          EstimatedAvailableDegreeO
fParallelism="2"

MaxCompileMemory="1475656">
    </OptimizerHardwareDependentProperties>
  <OptimizerStatsUsage>…</OptimizerStatsUsage>
```

Listing 13-11

It is a reduced set of information and I don't have a complete story from Microsoft on why this is so. The code for capturing the plans seems to have come originally from Trace Events and was duplicated in Extended Events. Nevertheless, what remains is still useful and it's only available in the XML.

Pre-SQL Server 2012: full "missing index" details

As we've seen previously in the book, often, you'll see a message at the top of a plan saying that there is a missing index that will "reduce the cost" of an operator by some percentage. Prior to SQL Server 2012, if there was more than one missing index, only one would be visible in the missing index hint in the graphical plan. So, if you're still working on earlier SQL Server versions, the XML is the only place you'll find the full list.

Also, using the execution plan directly ties the missing index information to the query itself. Using only the Microsoft-supplied DMVs, you won't see which query will benefit from the suggested index.

If you open the XML for the actual execution plan for Listing 13-10, you'll notice an element near the top labeled `MissingIndexes`, which lists tables and columns where the optimizer recognizes that, potentially, if it had an index it could result in a better execution plan and improved performance.

```
<MissingIndexes>
  <MissingIndexGroup Impact="83.5833">
    <MissingIndex Database="[AdventureWorks2016]"
Schema="[Purchasing]" Table="[PurchaseOrderHeader]">
      <ColumnGroup Usage="INEQUALITY">
        <Column Name="[ShipDate]" ColumnId="8" />
      </ColumnGroup>
      <ColumnGroup Usage="INCLUDE">
        <Column Name="[ShipMethodID]" ColumnId="6" />
      </ColumnGroup>
    </MissingIndex>
  </MissingIndexGroup>
</MissingIndexes>
```

Listing 13-12

While the information about missing indexes can sometimes be useful, it is only as good as the available statistics, and can sometimes be very unreliable. It also does not consider the added cost of maintaining the index. Always put appropriate testing in place before acting on these suggestions.

Querying the Plan Cache

For the remainder of this chapter, we'll focus on the one overriding reason why it's very useful to have the raw XML behind a plan: namely for querying it, using XQuery. We can run XQuery queries against the `.sqlplan` file, or against execution plans stored in XML columns in tables, or directly against the XML that exists in the plan cache or Query Store in SQL Server.

This section introduces only a few of the core concepts for writing XQuery and some useful examples to start you off, because an in-depth tutorial is far beyond the scope of this book. For that, I recommend *XML and JSON Recipes for SQL Server* by Alex Grinberg.

Why query the XML of plans?

As discussed in Chapter 9, several DMOs, such as `sys.dm_exec_query_stats` and `sys.dm_exec_cached_plans` store the `plan_handle` for a plan, which we can supply to the `sys.dm_exec_query_plan` function, to return the execution plan in XML format, as well as to the `sys.dm_exec_sql_text` function to return the SQL text. All the queries stored in the Query Store are also in that same XML format, although stored by default as text, which you must `CAST` to `XML`.

We can then use XQuery to return the elements, properties, and value within the plan XML, many of which we discussed earlier in the chapter. Why is this useful?

Firstly, let's suppose we have a lot of plans that we need to examine. Thousands, or more. Rather than attempt to walk through these plans, one at a time, looking for some common pattern, we can write queries that search on specific elements or terms within the plan XML, such as "**Reason For Early Termination**," and so track down recurring issues within the entire set of plans.

Secondly, as we know, the DMOs and the Query Store contain a lot of other useful information, such as execution statistics for the queries that used the cached plans. This means, for example, we could query the plan cache or the Query Store, for all plans with missing index recommendations, and the associated SQL statements, along with appropriate execution statistics, so we can choose the right index strategy for the workload, rather than query by query. The XML is the only place you can retrieve certain information, such as missing index information correlated to its query, so the ability to retrieve information from the XML may make using XQuery helpful.

Finally, sometimes a plan is very large, and it does become slightly easier to search the plan XML for certain values and properties, rather than scroll through looking at individual operator properties in the graphical plans. We'll cover this idea more in Chapter 14.

Before we start, though, a note of caution: XML querying is inherently costly, and queries against XML might seriously affect performance on the server, primarily due to the memory that XQuery consumes. Always apply due diligence when running these types of queries, and try to minimize the overhead caused by XQuery, by applying some filtering criteria to your queries, for example restricting the results to a single database, to limit the amount of data accessed.

Better still, we could export the XML plans, and potentially also the runtime stats, to a table on a different server and then run the XQuery against that, in order to avoid placing too much of a load directly against a production machine.

Query the plan XML for specific operators

Listing 13-13, given purely as an example of what's possible, returns the top three operators from the most frequently called query in the plan cache, assuming that this query has a cached plan, based on the total estimated cost of each operator.

It illustrates how we can construct queries against the plan cache, but I would hesitate before running this query on a production system if that system was already under stress.

```
WITH Top1Query
AS (SELECT TOP (1)
           dest.text,
           deqp.query_plan
    FROM sys.dm_exec_query_stats AS deqs
        CROSS APPLY sys.dm_exec_sql_text(deqs.sql_handle) AS dest
        CROSS APPLY sys.dm_exec_query_plan(deqs.plan_handle) AS
deqp
    ORDER BY deqs.execution_count DESC)
SELECT TOP 3
       tq.text,
       RelOp.op.value('@PhysicalOp', 'varchar(50)') AS PhysicalOp,
       RelOp.op.value('@EstimateCPU', 'float') + RelOp.op.value('@
EstimateIO', 'float') AS EstimatedCost
FROM Top1Query AS tq
    CROSS APPLY tq.query_plan.nodes('declare default element
namespace "http://schemas.microsoft.com/sqlserver/2004/07/
showplan";
    //RelOp') RelOp(op)
ORDER BY EstimatedCost DESC;
```

Listing 13-13

The basic logic is easy enough to follow. First, I define a common table expression (CTE), `Top1Query`, which returns the SQL text and the plan for the most frequently executed query currently in cache, as defined by the `execution_count`.

Next, skip down to the `FROM` clause of the second query, the "recursive member," which references the CTE. For every row in our `Top1Query` CTE, (in this case there is only one row), the `CROSS APPLY` will evaluate the subquery, which in this case uses the `.nodes` method to "shred" the XML for the plan, stored in the `query_plan` column of `sys.dm_exec_query_plan`, exposing the XML as if it were a table. Worth noting is that the query uses the sum of the `EstimatedCPU` and `EstimatedIO` to arrive at an `EstimatedCost` value for each operator. Normally, but not always, this will match exactly the value displayed for the **Estimated Operator Cost** in the graphical plan properties. For some operators, other factors (such as memory grants) are considered as part of the **Estimated Operator Cost** value.

This done, the `SELECT` list of the second query takes advantage of the methods available within XQuery, in this instance `.value`. We define the path to the location within the XML from which we wish to retrieve information, such as the `@PhysicalOp` property.

The results from my system look as shown in Figure 13-4.

Editor | Results | Messages

	text	PhysicalOp	EstimatedCost
1	SELECT c.CustomerID, a.City, ...	Hash Match	0.115106
2	SELECT c.CustomerID, a.City, ...	Hash Match	0.114258
3	SELECT c.CustomerID, a.City, ...	Clustered Index Scan	0.1139729

Figure 13-4: The three operators with highest estimated cost for the most frequently executed query.

Querying the XML for missing index information

Let's look at one more example. You've already seen the Missing Index information that was present in the execution plan (see Listing 13-12). There are Missing Index Dynamic Management Views that show you all the suggested possible missing indexes found by the optimizer. However, those DMVs do not have any mechanism for correlating the information back to the queries involved. If we want to see both the missing index information and also which queries they might be related to, we can use the query in Listing 13-14.

```
WITH XMLNAMESPACES
(
    DEFAULT 'http://schemas.microsoft.com/sqlserver/2004/07/
showplan'
)
SELECT deqp.query_plan.value(N'(//MissingIndex/@Database)[1]',
'NVARCHAR(256)')
           AS DatabaseName,
       dest.text AS QueryText,
       deqs.total_elapsed_time,
       deqs.last_execution_time,
```

```
        deqs.execution_count,
        deqs.total_logical_writes,
        deqs.total_logical_reads,
        deqs.min_elapsed_time,
        deqs.max_elapsed_time,
        deqp.query_plan,
        deqp.query_plan.value(N'(//MissingIndex/@Table)[1]',
'NVARCHAR(256)')
            AS TableName,
        deqp.query_plan.value(N'(//MissingIndex/@Schema)[1]',
'NVARCHAR(256)')
            AS SchemaName,
        deqp.query_plan.value(N'(//MissingIndexGroup/@Impact)[1]',
'DECIMAL(6,4)')
            AS ProjectedImpact,
        ColumnGroup.value('./@Usage', 'NVARCHAR(256)') AS
ColumnGroupUsage,
        ColumnGroupColumn.value('./@Name', 'NVARCHAR(256)') AS
ColumnName
FROM sys.dm_exec_query_stats AS deqs
    CROSS APPLY sys.dm_exec_query_plan(deqs.plan_handle) AS deqp
    CROSS APPLY sys.dm_exec_sql_text(deqs.sql_handle) AS dest
    CROSS APPLY deqp.query_plan.nodes('//MissingIndexes/
MissingIndexGroup/MissingIndex/ColumnGroup') AS t1(ColumnGroup)
    CROSS APPLY t1.ColumnGroup.nodes('./Column') AS
t2(ColumnGroupColumn);
```

Listing 13-14

In the results shown in Figure 13-5, I ran a slightly modified version of Listing 13-14 to filter the results to only show information regarding the `AdventureWorks2014` database and limit the number of columns, for readability.

	TableName	ColumnName	ColumnGroupUsage	QueryText
1	[Address]	[City]	EQUALITY	CREATE PROC dbo.spAddressByCity @City NVARCHAR(3...
2	[PurchaseOrderHeader]	[ShipDate]	INEQUALITY	SELECT poh.PurchaseOrderID, poh.ShipDate, ...
3	[PurchaseOrderHeader]	[PurchaseOrderID]	INCLUDE	SELECT poh.PurchaseOrderID, poh.ShipDate, ...
4	[PurchaseOrderHeader]	[ShipMethodID]	INCLUDE	SELECT poh.PurchaseOrderID, poh.ShipDate, ...

Figure 13-5: Missing Index suggestions for AdventureWorks 2014.

The query in its current form returns multiple rows for the same missing index suggestions, so in rows 2 to 4 you see the single missing index suggestion for the query from Listing 13-13. Row 1 shows an additional suggestion for a stored procedure that may need an index created on it.

The `TableName` and `ColumnName` information is self-explanatory. The `ColumnGroup-Usage` is suggesting where the column should be added to the index. An `EQUALITY` or `INEQUALITY` value is suggesting that the column in question be added to the key of the index. An `INCLUDE` value is suggesting adding that column to the `INCLUDE` clause of the index creation statement. Each suggested index in this query is associated with the relevant `QueryText`.

The query uses the `.nodes` method, to which we supply the path to the `ColumnGroup` element in the XML plan stored in cache:

```
'//MissingIndexes/MissingIndexGroup/MissingIndex/ColumnGroup'
```

The values passed to `.node` here ensure that only information from this full path is used to run the rest of the `.value` functions, which return information about the index, specifically the `TableName`, `ColumnName`, and `ColumnGroupUsage` information. With that you can just refer to the path `//MissingIndexGroup/` and then supply a property value such as `@Schema` to arrive at data.

This is a useful way to filter or sort for queries currently in cache that have missing index suggestions, to find queries that need tuning quickly. However, do bear in mind that not all problem queries have missing indexes and not all queries with missing indexes are problem queries. Finally, not all problem queries are guaranteed to be in cache when you run Listing 13-14.

Very few people will sit down and write their own XQuery queries to retrieve data from execution plans. Instead, you can take a query like Listing 13-14 and then adjust for your own purposes. The only hard part is figuring out how to get the path correct. That's best done by simply looking at the XML and stepping through the tree to arrive at the correct values.

Summary

The data provided in XML plans is complete, and the XML file is easy to share with others. However, reading an XML plan is not an easy task and, unless you are the sort of data professional who needs to know every internal detail (the majority of which are available through the properties of a graphical plan), it is not one you will spend time mastering.

Much better to read the plans in graphical form and, if necessary, spend time learning how to use XQuery to access the data in these plans programmatically, and so begin automating access to your plans in some instances, such as the Missing Index query shown in this chapter.

Chapter 14: Plans for Special Data Types and Cursors

Some of the data types introduced to SQL Server over the years have quite different functionality from the standard set of numbers, strings, and dates that account for most of the data with which we work. These data types have special functionality and indexing that affect how they work, and when our queries, procedures, and functions work with these data types, the differences can show up in execution plans.

We'll spend a large part of the chapter looking at plans for queries that use XML, since this is the "special" data type most of us have encountered at some point. We'll examine the plans that convert data from XML to relational (`OPENXML`), from relational to XML (`FOR XML`), and ones that query XML data using XQuery. We won't dive into any tuning details, but I will let you know where in the plan you might look for clues, if a query that uses XML is performing poorly.

SQL Server 2016 added support for JavaScript Object Notation (JSON). It provides no JSON-specific data type (it stores JSON data in an `NVARVAR` type) and consequently none of the kinds of methods available to the XML data type. However, it does provide several important T-SQL language elements for querying JSON, and we'll look at how that affects execution plans.

We'll also look briefly at plans for queries that use the `HIERARCHYID` data type. We'll then examine plans for queries that access **spatial** data, though only their basic characteristics, because even rather simple spatial queries can have impressively complex plans.

The final part of the chapter examines plans for cursors. These don't fit neatly into the special data type category; you can't store a cursor in a column and so it is not, strictly, a data type, although, Microsoft does use "cursor" as the data type for a variable or output parameter that references a cursor. In any event, cursors are certainly special in that they are a programming construct that allows us to process query results one row at a time, rather than in the normal and expected, set-based fashion. This will, of course, affect the execution plan, and not often in a good way.

XML

XML is a standard data type in many applications, and sometimes leads to storage of XML within SQL Server databases, using the XML data type. However, if our database simply accepts XML input and stores it in an XML column or variable, or reads an XML column or variable, and returns it in XML form, then at the execution plan level, this is no different from storing and retrieving data of any other type.

XML becomes relevant to execution plans if we query the XML data using XQuery, or if a query uses the FOR XML clause to convert relational data to XML, or the OPENXML rowset provider to go from XML to relational.

These methods of accessing and manipulating XML are very useful, but come at a cost. Manipulating XML uses a combination of T-SQL and XQuery expressions, and problems both in the T-SQL and in the XQuery parts can affect performance. Also, the XML parser, which is required to manipulate XML, uses memory and CPU cycles that you would normally have available only for T-SQL.

Overall, there are reasons to be judicious in your use and application of XML in SQL Server databases.

Plans for queries that convert relational data to XML (FOR XML)

By using the FOR XML clause in our T-SQL queries, we can transform relational data into XML format, usually for outputting to a client, but sometimes so that we can store it in an XML variable or column. We can use the FOR XML clause in any of the following four modes, AUTO, RAW, PATH, or EXPLICIT. The first three can be used in the same way, will create a different format of XML output from the same query, and the execution plan will be the same in each case. The fourth mode allows us to define explicitly, in the query itself, the shape of the resulting XML tree. This requires a query rewrite and so results in a different execution plan.

Plans for basic FOR XML queries

Listing 14-1 shows a standard query that produces a list of stores and the contact person for that store.

```
SELECT s.Name AS StoreName,
       bec.PersonID,
       bec.ContactTypeID
FROM Sales.Store AS s
    JOIN Person.BusinessEntityContact AS bec
        ON s.BusinessEntityID = bec.BusinessEntityID
ORDER BY s.Name;
```

Listing 14-1

The resulting plan is very straightforward and needs no explanation at this stage of the book.

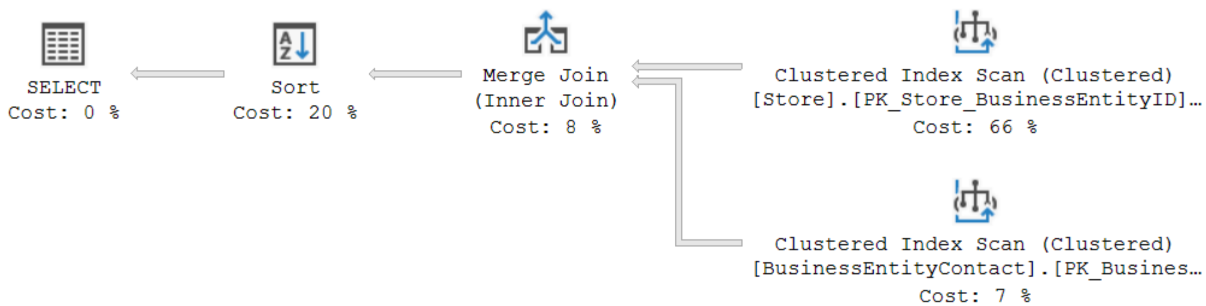


Figure 14-1: Traditional execution plan like elsewhere in the book.

To see the impact on the plan of converting the relational output to an XML format, we simply add the FOR XML clause to Listing 14-1.

```
SELECT s.Name AS StoreName,
       bec.PersonID,
       bec.ContactTypeID
FROM Sales.Store AS s
    JOIN Person.BusinessEntityContact AS bec
        ON s.BusinessEntityID = bec.BusinessEntityID
ORDER BY s.Name
FOR XML AUTO;
```

Listing 14-2

In this case, I've used the AUTO mode but, regardless of whether I use that, or RAW, or PATH, the plan in each case is as shown in Figure 14-2.

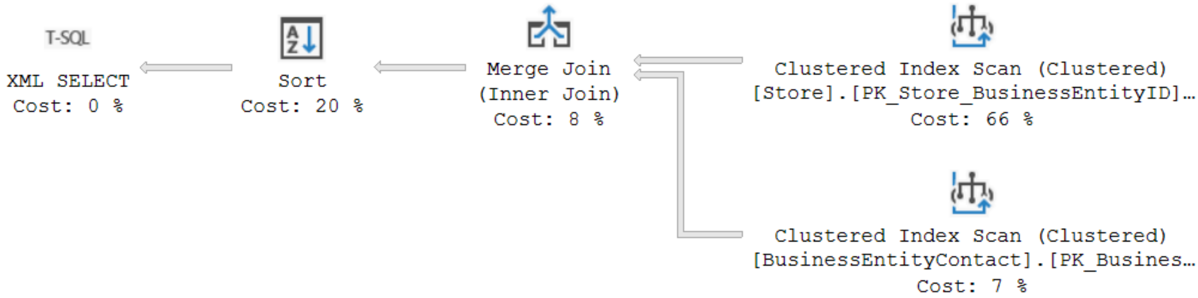


Figure 14-2: An execution plan showing output to XML through the XML SELECT operator.

The only visible difference is that the SELECT operator is replaced by an XML SELECT operator, and in fact this really is the only difference. The plans for the query with relational output, and those for FOR XML queries with AUTO, RAW, or PATH seem to be identical in all respects. However, each of the three FOR XML modes produces a different XML output from the same query, as shown below, for the first row of the result set, in each case.

```
-- AUTO:
<s StoreName="A Bicycle Association">
  <bec PersonID="2050" ContactTypeID="11" />
</s>
-- RAW:
<row StoreName="A Bicycle Association" PersonID="2050"
ContactTypeID="11" />
-- PATH
<row>
  <StoreName>A Bicycle Association</StoreName>
  <PersonID>2050</PersonID>
  <ContactTypeID>11</ContactTypeID>
</row>
```

Each of these basic modes of FOR XML return text that is formatted like XML. If we want the data to be returned in native XML format (as an XML data type), then we need to use the TYPE directive. If you don't use the TYPE directive then, while it may look like XML to you and me, to SQL Server and SSMS, it's just a string.

Returning XML as XML data type

An extension of the `XML AUTO` mode allows you to specify the `TYPE` directive, to output the results of the query as the `XML` data type, not simply as text in XML format. The `TYPE` directive is mainly relevant if you use subqueries with `FOR XML`. The query in Listing 14-3 returns the same data as the previous one, but in a different structure. We're using the subquery to make XML using `TYPE`, and then combining that with data from the outer query, which is then output as XML-formatted text.

```
SELECT s.Name AS StoreName,
       (   SELECT bec.BusinessEntityID,
                  bec.ContactTypeID
           FROM Person.BusinessEntityContact AS bec
           WHERE bec.BusinessEntityID = s.BusinessEntityID
           FOR XML AUTO, TYPE, ELEMENTS) AS contact
FROM Sales.Store AS s
ORDER BY s.Name
FOR XML AUTO;
```

Listing 14-3

Figure 14-3 shows two result sets, the first for the query as written in Listing 14-3, and the second for the same query but without the `TYPE` directive.

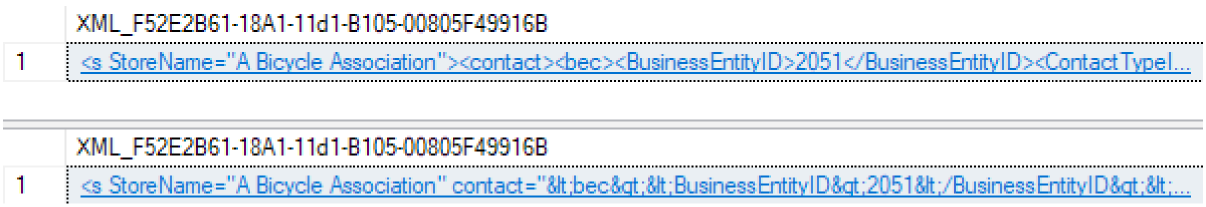


Figure 14-3: Output of FOR XML AUTO, both with and without the TYPE directive.

Notice that in the latter case the angle brackets in the subquery are converted to `>` and `<` because the subquery is considered text to be converted to XML. In the former case, it's formatted as XML.

Figure 14-4 shows the resulting execution plan for Listing 14-3 (with the `TYPE` directive).

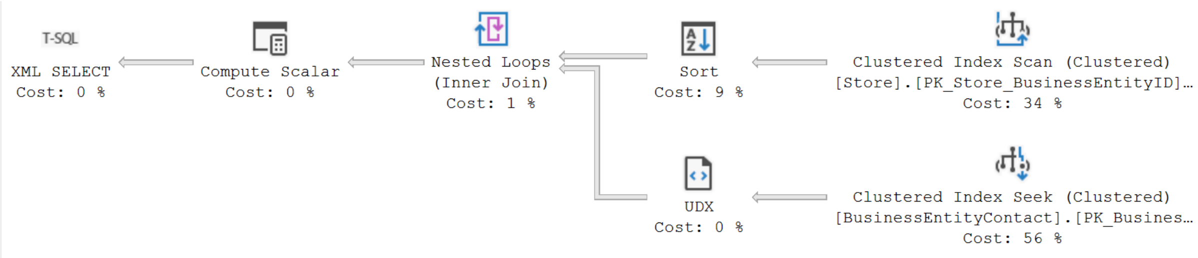
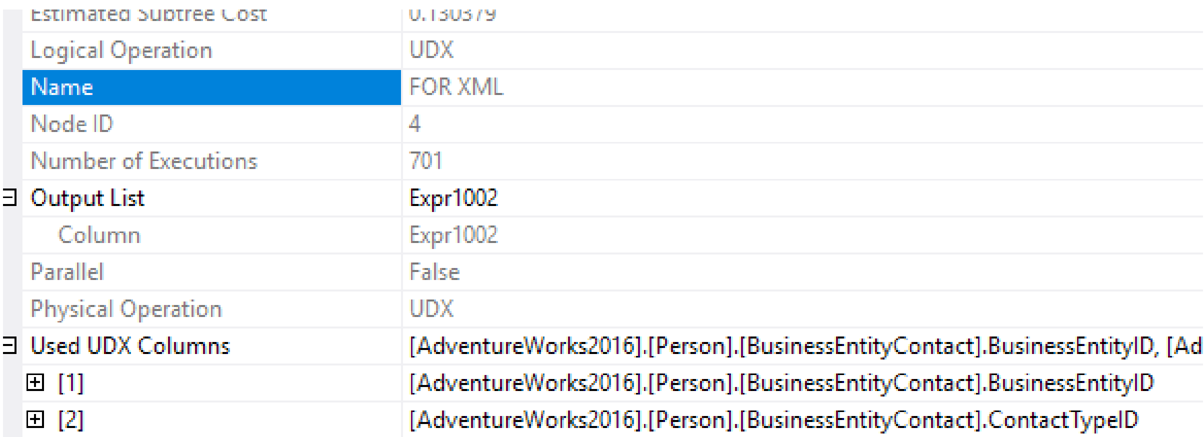


Figure 14-4: Execution plan for XML AUTO.

First, it's worth noting that this query now causes 1515 logical reads, about 10 times more than the query in Listing 14-2. This is because the optimizer uses a **Nested Loops** join to data from the query and subquery, and since the outer input produces 701 rows.

The outer input returns the `BusinessEntityID` and `Name` columns, sorted by `Name`. The `BusinessEntityID` values are pushed down to the inner input, and we see 701 seeks, for the matching rows, returning the `BusinessEntityID` and `ContactTypeID` columns. We then see the **UDX** operator, which in this case converts each row emerging from the **Index Seek** into `XML` format.

Figure 14-5 shows the Properties window for the **UDX** operator. The **Name** property has the value **FOR XML**, which tells us that it's converting relational data into XML. The **Used UDX Columns** property shows which input data it processes. And the **Output List** contains the internal name of the created XML data, in this case `Expr1002`, which consists of the two `BusinessEntityID` and `ContactTypeID` columns from the `Business-EntityContact` table.



Estimated Subtree Cost	0.130379
Logical Operation	UDX
Name	FOR XML
Node ID	4
Number of Executions	701
⊟ Output List	Expr1002
Column	Expr1002
Parallel	False
Physical Operation	UDX
⊟ Used UDX Columns	[AdventureWorks2016].[Person].[BusinessEntityContact].BusinessEntityID, [Ad
⊞ [1]	[AdventureWorks2016].[Person].[BusinessEntityContact].BusinessEntityID
⊞ [2]	[AdventureWorks2016].[Person].[BusinessEntityContact].ContactTypeID

Figure 14-5: Properties of the UDX operator.

The **UDX** operator is often seen in plans that perform XPath and XQuery operations, and so we'll see it again later in the chapter.

Finally, we see the **Compute Scalar** operator, which for some ill-defined reason assigns the value of **Expr1002** to **Expr1004**, then passes **Expr1004** to its parent.

Plans for Explicit mode FOR XML queries

XML `EXPLICIT` mode is there for the occasions when we need to exert very precise control over the format of the XML generated by the query. The downside is that the rowset our query produces must obey certain formatting rules. If you try to run Listing 14-2 using the `EXPLICIT` mode of `FOR XML`, you'll see an error to the effect that the format of your result set is wrong.

```
Msg 6803, Level 16, State 1, Line 46
          FOR XML EXPLICIT requires the first column to hold
          positive integers that represent XML tag IDs.
```

So, it's up to us to write the query so that the rowset is in the right format, depending on the required structure of the XML output. `EXPLICIT` mode is used to create very specific XML, mixing and matching properties and elements in any way you choose based on what you define within the query. Listing 14-4 shows a simple example.

```
SELECT  1 AS Tag,
        NULL AS Parent,
        s.Name AS [Store!1!StoreName],
        NULL AS [BECContact!2!PersonID],
        NULL AS [BECContact!2!ContactTypeID]
FROM    Sales.Store s
JOIN    Person.BusinessEntityContact AS bec
        ON s.BusinessEntityID = bec.BusinessEntityID
UNION ALL
SELECT  2 AS Tag,
        1 AS Parent,
        s.Name AS StoreName,
        bec.PersonID,
        bec.ContactTypeID
```

```
FROM     Sales.Store s
JOIN     Person.BusinessEntityContact AS bec
         ON s.BusinessEntityID = bec.BusinessEntityID
ORDER BY [Store!1!StoreName],
         [BECContact!2!PersonID]
FOR      XML EXPLICIT;
```

Listing 14-4

Figure 14-6 shows the actual execution plan for this query, which is somewhat more complex.

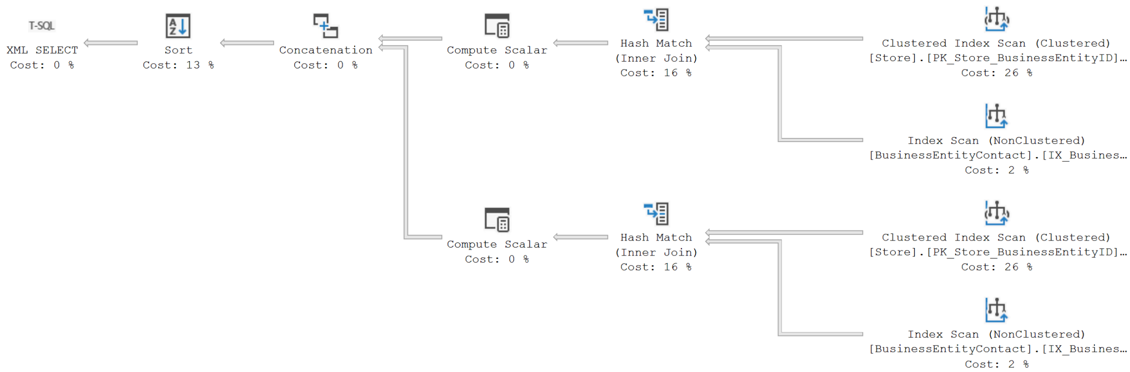


Figure 14-6: Execution plan showing how XML EXPLICIT works.

To build the hierarchy of XML, we had to use the `UNION ALL` clause in T-SQL, between two almost identical copies of the same query. The double execution of this branch makes it about twice as expensive as the plan for the query in Listing 14-2. This is not as a direct result of using `FOR XML EXPLICIT`, but is an indirect result of the requirements that option puts on how we write the query.

So, while you get more control over the XML output, it comes at the cost of added overhead, due to the need for the `UNION ALL` clause and the explicit formatting rules. This leads to decreased performance due to the increased number of queries required to put the data together.

Again, if you simply rerun the query without the `FOR XML EXPLICIT` clause, the only difference in the plan will be an **XML Select** operator instead of a **Select**. Only the format of the results is different. With `FOR XML EXPLICIT` you get XML; without it, you get an oddly-formatted result set, since the structure you defined in the `UNION` query is not naturally nested, as the XML makes it.

Plans for queries that convert XML to relational data (OPENXML)

We can use OPENXML in our T-SQL queries to "shred" XML into a relational format, most often to take data from the XML format and change it into structured storage within a normalized database.

OPENXML takes an XML document, stored in an nvarchar variable, and converts it into a "rowset view" of that document, which can be treated as if it were a normal table. By rowset, we mean a traditional view of the data in a tabular format, as if it were being queried from a table. We can use OPENXML as a data source in any query. It can take the place of a table or view in a SELECT statement, or in the FROM clause of modification statements, but it cannot be the target of INSERT, UPDATE, DELETE, or MERGE.

To demonstrate this, we need an XML document. I've had to break elements across lines in order to present the document in a readable form.

```
<ROOT>
  <Currency CurrencyCode="UTE"
     CurrencyName="Universal Transactional Exchange">
    <CurrencyRate FromCurrencyCode="USD" ToCurrencyCode="UTE"
                   CurrencyRateDate="2007/1/1" AverageRate=."553"
                   EndOfDateRate= ."558" />
    <CurrencyRate FromCurrencyCode="USD" ToCurrencyCode="UTE"
                   CurrencyRateDate="2017/6/1/" AverageRate=."928"
                   EndOfDateRate= "1.057" />
  </Currency>
</ROOT>
```

Listing 14-5

In this example, we're creating a new currency, the Universal Transactional Exchange, otherwise known as the UTE. We need exchange rates for converting the UTE to USD. We're going to take all this data and insert it, in a batch, into our database, straight from XML. Listing 14-6 shows the script.

```
BEGIN TRAN;
DECLARE @iDoc AS INTEGER;
DECLARE @Xml AS NVARCHAR(MAX);
SET @Xml = '<ROOT>
<Currency CurrencyCode="UTE" CurrencyName="Universal
  Transactional Exchange">
   <CurrencyRate FromCurrencyCode="USD" ToCurrencyCode="UTE"
     CurrencyRateDate="2007/1/1" AverageRate=."553"
     EndOfDayRate= ."558" />
   <CurrencyRate FromCurrencyCode="USD" ToCurrencyCode="UTE"
     CurrencyRateDate="2007/6/1" AverageRate=."928"
     EndOfDayRate= "1.057" />
</Currency>
</ROOT>';
EXEC sys.sp_xml_preparedocument
    @iDoc OUTPUT,
    @Xml;
INSERT  INTO Sales.Currency
        (CurrencyCode,
         Name,
         ModifiedDate
        )
SELECT  CurrencyCode,
        CurrencyName,
        GETDATE()
FROM    OPENXML (@iDoc, 'ROOT/Currency',1)
           WITH (CurrencyCode NCHAR(3), CurrencyName NVARCHAR(50));
INSERT  INTO Sales.CurrencyRate
        (CurrencyRateDate,
         FromCurrencyCode,
         ToCurrencyCode,
         AverageRate,
         EndOfDayRate,
         ModifiedDate
        )
SELECT  CurrencyRateDate,
        FromCurrencyCode,
        ToCurrencyCode,
        AverageRate,
        EndOfDayRate,
        GETDATE()
FROM    OPENXML(@iDoc , 'ROOT/Currency/CurrencyRate',2)
          WITH (CurrencyRateDate DATETIME '@CurrencyRateDate',
                FromCurrencyCode NCHAR(3) '@FromCurrencyCode',
```

```
                    ToCurrencyCode NCHAR(3) '@ToCurrencyCode',
                    AverageRate MONEY '@AverageRate',
                    EndOfDayRate MONEY '@EndOfDayRate');
EXEC sys.sp_xml_removedocument
    @iDoc;
ROLLBACK TRAN;
```

Listing 14-6

From this query, we get two actual execution plans, one for each `INSERT`. The first `INSERT` is against the `Currency` table, as shown in Figure 14-7.

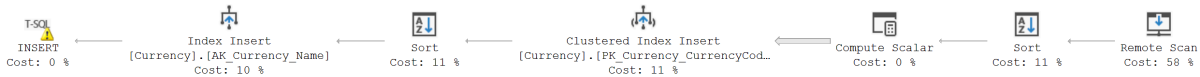


Figure 14-7: Execution plan for the INSERT against the Currency table.

A quick scan of the plan reveals a single new operator, **Remote Scan**. All the `OPENXML` statement processing is handled within that **Remote Scan** operator. This operator represents the opening of a remote object, meaning a DLL or some external process such as a CLR object, within SQL Server, which will take the XML and convert it into a format within memory that looks to the query engine like normal rows of data. Since the **Remote Scan** is not actually part of the query engine itself, the optimizer represents the call, in the plan, as a single icon.

The only place where we can really see the evidence of the XML is in the **Output List** for the **Remote Scan**. In Figure 14-8, we can see the `OPENXML` statement referred to as a table, and the properties selected from the XML data listed as columns.

Output List	[OpenXML].CurrencyCode, [OpenXM
◢ [1]	[OpenXML].CurrencyCode
Column	CurrencyCode
Table	[OpenXML]
◢ [2]	[OpenXML].CurrencyName
Column	CurrencyName
Table	[OpenXML]

Figure 14-8: Properties of the OPENXML operator.

From there, it's a straightforward query with the data first being sorted for insertion into the clustered index, and then sorted a second time for addition to the other index on the table.

The main point to note is that the Optimizer uses a fixed estimate of 10,000 rows returned for the **Remote Scan**, which explains why it decides to **Sort** the rows first, to make inserting into the indexes more efficient, though in this case that's unnecessary as we only actually return 1 row. This fixed estimate affects other operator choices that the optimizer makes, and so can affect performance.

Also worth noting are the different arrow sizes coming in and out of **Compute Scalar**, which are the result of a bad estimate. A **Compute Scalar** never actually does its own work, so it only presents estimated row counts even in an actual plan. The size of the incoming arrow reflects actual row counts (1 row), and the outgoing arrow reflects estimated (10,000 rows).

The second execution plan describes the `INSERT` against the `CurrencyRate` table.

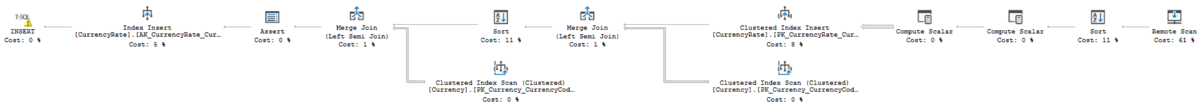


Figure 14-9: Execution plan for CurrencyRate table.

This query is the more complicated of the two because of the extra steps required for the maintenance of referential integrity (see Chapter 6) between the `Currency` and `CurrencyRate` tables. There are two checks done for this because of the `FromCurrency` and `ToCurrency` columns Yet still we see no XML-specific icons, since all the XML work is hidden behind the **Remote Scan** operation. In this case, we see two comparisons against the parent table, through the **Merge Join** operations. The data is sorted, first by `FromCurrencyCode` and then by `ToCurrencyCode`, in order for the data to be used in a **Merge Join**, the operator picked by the optimizer because it estimated 10,000 rows would be returned by the **Remote Scan**.

As you can see, it's easy to bring XML data into the database for use within our queries, or for inclusion within our database. However, a lot of work goes on behind the scenes to do this, and not much of that work is visible in the execution plan. First, SQL Server has to call the `sp_xml_preparedocument` function, which parses the XML text using the MSXML parser. However, we see none of this work in the plan. Next, it needs to transform the parsed document into a rowset, but this work is "hidden" and represented by the **Remote Scan** operator. However, we do see that the estimated row count for OPENXML is fixed at 10,000 rows, which may affect query performance. If this is causing performance problems for you,

you should focus on other mechanisms of data manipulation, such as loading to a temporary table first in order to get statistics for a better-performing execution plan.

One caveat worth mentioning is that parsing XML uses a lot of memory. You should plan on opening the XML, getting the data out, and then closing and de-allocating the XML as soon as possible. This will reduce the amount of time that the memory is allocated within your system.

Plans for querying XML using XQuery

The true strength of querying XML within SQL Server is through XQuery. We'll examine the execution plans for a few simple XQuery examples, so that you can start to see how incorporating XQuery expressions in our T-SQL queries can affect those plans. However, we can't cover the full breadth and depth of execution plan patterns you can see with XQuery (and nor can I teach you XQuery; that would require an entire book of its own). For a thorough introduction, read this white paper offered from Microsoft at http://bit.ly/1UH6KfP.

The purpose of seeing how to query XML, specifically the XML within execution plans, is to be able to search for values in lots of plans rather than browsing the plans themselves. This can be used against plans in cache, plans in the Query Store, and plans that are files. There are examples of how to do this in the *Querying the Plan Cache* section of Chapter 13.

Effectively, using XQuery means a completely new query language to learn in addition to T-SQL. The XML data type is the mechanism used to provide the XQuery functionality through the SQL Server system. When you want to query from the XML data type, there are five basic methods:

- `.query()` – used to query the XML data type and return the XML data type
- `.value()` – used to query the XML data type and return a non-XML scalar value
- `.nodes()` – a method for pivoting XML data into rows
- `.exist()` – queries the XML data type and returns a `bit` to indicate whether or not the result set is empty, just like the `EXISTS` keyword in T-SQL
- `.modify()` – a method for inserting, updating, and deleting XML snippets within the XML data set.

Generally, the optimizer seems to implement these methods using two specific operators, **Table-Valued Function (XML Reader)**, with or without an XPath filter, and **UDX**, combined in different patterns.

The various options for running a query against XML, including the use of FLWOR (For, Let, Where, Order By and Return) statements within the queries, all affect the execution plans. I'm going to cover just two examples, to acquaint you with the concepts and introduce you to the sort of execution plans you can expect to see. It's outside the scope of this book to cover this topic in the depth that would be required to demonstrate all aspects of the plans this language generates.

Plans for queries that use the .exist method

The Resume column of the JobCandidate table in AdventureWorks is an XML data type. If we need to query the résumés of all employees to find out which of the people hired were once sales managers, we'll need to use the .exist method in our XQuery expression, so that our query only returns a row if the JobTitle element of the document contains the text "Sales Manager."

```
SELECT p.LastName, p.FirstName, e.HireDate, e.JobTitle
  FROM Person.Person AS p
    INNER JOIN HumanResources.Employee AS e
      ON p.BusinessEntityID = e.BusinessEntityID
    INNER JOIN HumanResources.JobCandidate AS jc
      ON e.BusinessEntityID = jc.BusinessEntityID
     AND jc.Resume.exist(
           ' declare namespace
        res="http://schemas.microsoft.com/sqlserver/2004/07/
adventure-works/Resume";
        /res:Resume/res:Employment/res:Emp.JobTitle[contains
             (.,"Sales Manager")]') = 1;
```

Listing 14-7

Figure 14-10 shows the actual execution plan for this query.

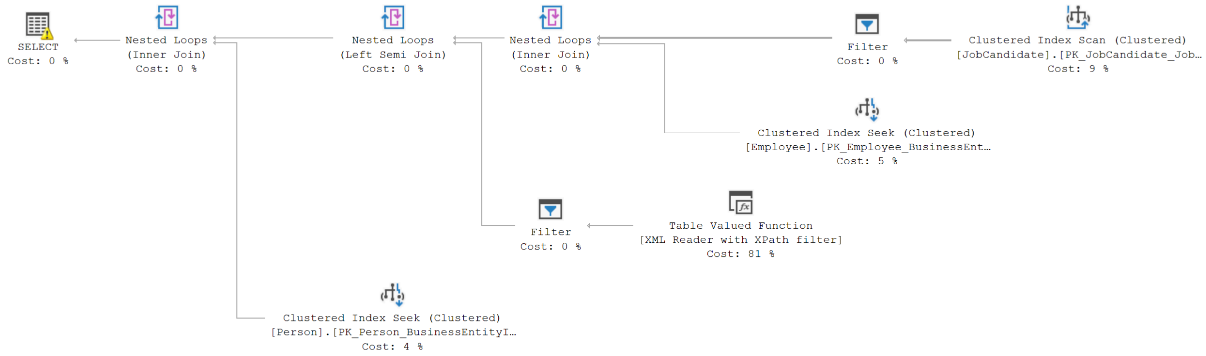


Figure 14-10: Execution plan for the .exist XQuery method.

Following the data flow, from right to left, we see a normal execution plan. A **Clustered Index Scan** against the `JobCandidate` table followed by a **Filter** that ensures that the `Resume` field is not null. A **Nested Loops** join combines this data from the filtered `JobCandidate` table with data returned from the `Employee` table, filtering us down to two rows.

Then, another **Nested Loops** operator is used to combine data from a new operator, a **Table Valued Function** operator, subtitled "XML Reader with XPath filter," which represents as relational data the output from the XQuery. The role it plays is not dissimilar to that of the **Remote Scan** operation from the `OPENXML` query. However, the **Table Valued Function**, unlike the **Remote Scan** in the earlier example, is part of the query engine and is represented by a distinct icon. Unlike a multi-statement table-valued function, the table-valued functions used by XQuery do not have a plan we can access through the cache or the query store, or by capturing an estimated plan. Its execution is purely internal.

The properties for the **Table Valued Function** show that the operator was executed two times and four rows were returned.

Actual Execution Mode	Row
Actual Number of Batches	0
Actual Number of Rows	4
Actual Rebinds	2
Actual Rewinds	0
Defined Values	[XML Reader with XPath
Description	Table valued function.

Figure 14-11: Properties of the Table Valued Function showing XML operation.

These rows are passed to a **Filter** operator. Two values are defined by the **Table Valued Function**, **value** and **lvalue**. It's not completely clear how this works, but the **Filter** operator determines if the XPath query we defined equals 1 and is `NOT NULL` (and the `NOT NULL` check isn't necessary, but it's there). This results in a single row for output to the **Nested Loops** operator. From there, it's a typical execution plan, retrieving data from the `Contact` table and combining it with the rest of the data already put together.

Plans for queries that use the .query method

The `.query` method returns XML. We use this if, rather than simply filter based on the XML, we want to return some or all of the XML we are querying against. In our example, we'll query demographics data to find stores that are greater than 20,000 square feet in size. We have to define the XML structure to be returned and, to this end, the query uses XQuery's **FLWOR** expressions: **For**, **Let Where**, **Order**, **Return**.

In this example, we need to generate a list of stores managed by a particular salesperson. Specifically, we want to look at any of the demographics for stores managed by this salesperson that have more than 20,000 square feet, where those stores have recorded any demographic information. We'll also list the stores that don't have it. The demographics information is semi-structured data, so it is stored within XML in the database. To filter the XML directly, we'll be using the `.query` method. Listing 14-8 shows our example query and execution plan.

```
SELECT s.Name,
  s.Demographics.query
('
   declare namespace ss="http://schemas.microsoft.com/
sqlserver/2004/07/adventure-works/StoreSurvey";
   for $s in /ss:StoreSurvey
   where ss:StoreSurvey/ss:SquareFeet > 20000
   return $s
') AS Demographics
  FROM Sales.Store AS s
  WHERE s.SalesPersonID = 279;
```

Listing 14-8

Figure 14-12 shows the plan.

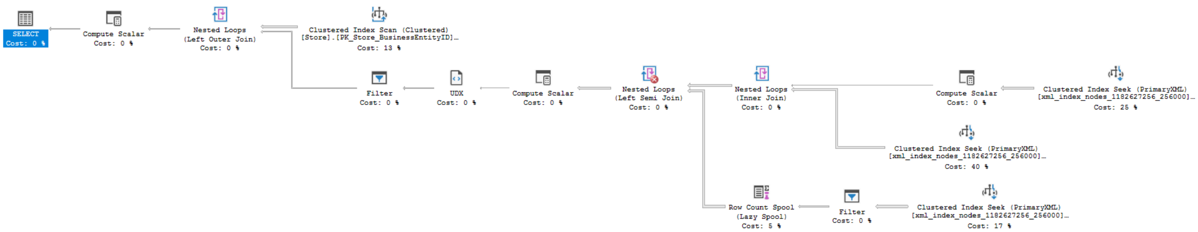


Figure 14-12: Full execution plan for .query XQuery method.

The T-SQL consists of two queries:

- a regular T-SQL query against the `Store` table to return the rows where the `SalesPersonId = 279`
- an XQuery expression that uses the `.query` method to return the data where the store's square footage was over 20,000.

Stated that way, it sounds simple, but a lot more work is necessary around those two queries to arrive at a result set.

Let's break this execution plan down into three parts, each of which has separate responsibilities: one for the relational part of the query, the second to read and filter the XML data according to the XQuery expression, and the third to take the data and convert it back into proper XML.

First, Figure 14-13 shows the top-left part of the plan, which contains the standard parts of the query that is retrieving information from the `Store` table.

Figure 14-13: Blow-up of plan showing traditional data access.

The data flow starts with a **Clustered Index Scan** against the `Sales` table, filtered by the `SalesPersonId`. The data returned is fed into the top half of a **Nested Loops**, left outer join. This **Nested Loops** operator then calls its lower input (the section of the plan in Figure 14-13) for each row, pushing the data (values from the `BusinessEntityID` and `Demographics` columns) from the top input into the lower input, as seen in the **Outer References** property. The result of that lower input is then combined with the data read from the `Stores` table and returned to the client.

Going over to the right to find the second stream of data for the join, we find three familiar **Clustered Index Seek** operators, but this time though, they're accessing an XML clustered index. Figure 14-14 shows a blow-up of that part of the plan.

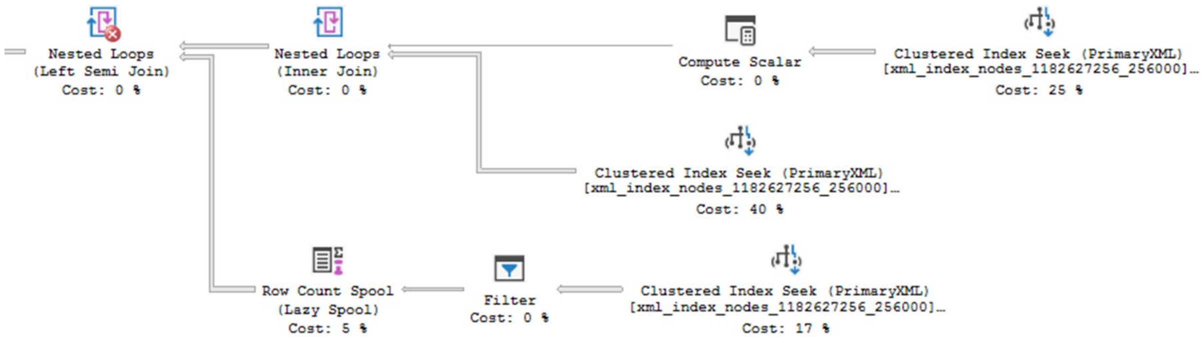


Figure 14-14: Blow-up of plan showing XML index use for XQuery statement.

The data in the `XML` data type is stored separately from the rest of the table, and there is an XML index available. The three seeks and the way they are combined are an artefact of how XML data is encoded in XML indexes, and I won't delve into this in detail. The **Clustered Index Seek** operator at the top right retrieves that data, using the pushed-down values from the **Nested Loops** discussed previously.

Number of Executions	80
Object	**[AdventureWorks2014].[sys].[xml_index_n**
Alias	[StoreSurvey:1]
Database	[AdventureWorks2014]
Index	[PXML_Store_Demographics]
Index Kind	PrimaryXML
Schema	[sys]
Storage	RowStore
Table	[xml_index_nodes_526624919_256000]
Ordered	True
Output List	**[AdventureWorks2014].[sys].[xml_index_n**
◢ [1]	[AdventureWorks2014].[sys].[xml_index_n
Alias	[StoreSurvey:1]
Column	id
Database	[AdventureWorks2014]
Schema	[sys]
Table	[xml_index_nodes_526624919_256000]
▷ [2]	[AdventureWorks2014].[sys].[xml_index_n
▷ [3]	[AdventureWorks2014].[sys].[xml_index_n

Figure 14-15: Properties of the Index Seek showing XML data access.

You can see in Figure 14-15 that the seek is occurring on `PXML_Store_Demographics`, returning the 80 rows from the index that match the `BusinessEntityId` column from the `store` table. You can also see the output of the columns from the XML index nodes. This information allows you to understand better how SQL Server is retrieving the XML from the index in question.

The **Filter** operator implements the `WHERE` part of the FLWOR expression in the XQuery expression. Its predicate shows that it tests a column named "value," extracted from the XML, against the value 20,000, since we're only returning stores with a square footage of greater than this value. This illustrates that not all FLWOR logic is pushed into a special XML-related operator, as we saw in earlier examples. Parts of the XQuery expression are evaluated as if they were relational expressions. Here, the engine extracts data out of the XML, making it relational, operates on it using the normal operators, and will later put it back into XML format.

The result of this fragment of the plan is data extracted from the XML column and manipulated according to the XQuery expression, but presented as a rowset, i.e. in relational format.

The third part of the plan does the conversion to XML. You can see this section blown up in Figure 14-16.

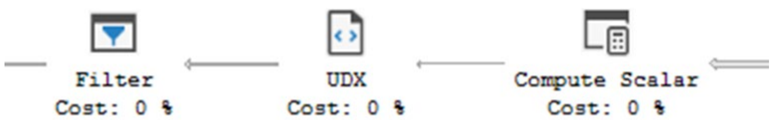


Figure 14-16: Blow-up of the plan showing conversion to XML.

The **Compute Scalar** does some prep work for the **UDX** operator, which converts the data information retrieved through the operations defined above back into XML format. That, in fact, is the final part of the XML-related portion of the plan. The Filter operator uses a **Startup Expression Predicate** property to suppress execution of the entire subtree of this plan for any rows with a NULL value in the XML column (i.e. for the Demographics data), preventing needless loss of performance.

All of this is combined with the original rows returned from the `Store` table through the **Nested Loops** operator in Figure 14-13.

When to use XQuery

These examples show that all the familiar operators and a few new operators are combined to implement XQuery, but that a full coverage is beyond the scope of this book. XQuery can take the place of `FOR XML`, but you might see some performance degradation.

You can also use XQuery in place of `OPENXML`. The functionality provided by XQuery goes beyond what's possible within `OPENXML`. Combining that with T-SQL will make for a powerful combination when you have to manipulate XML data within SQL Server. As with everything else, please test the solution with all possible tools to ensure that you're using the optimal one for your situation.

JavaScript Object Notation

JavaScript Object Notation (JSON), an open-standard file format using human-readable text, is supported by SQL Server and Azure SQL Database starting with SQL Server 2016. There are mechanisms around storage and retrieval built into SQL Server to deal with JSON data. We won't explore all that information here. We are going to look at one example of a JSON query, because it results in differences to the execution plans generated, so you need to know what to look for. For a more detailed examination of the complete JSON functionality within SQL Server please refer to the Microsoft documentation: https://bit.ly/2qCP8Mx.

Unfortunately, the current version of AdventureWorks does not have any JSON data, so we must first build some.

```
SELECT p.BusinessEntityID,
       p.Title,
       p.FirstName,
       p.LastName,
       (   SELECT p2.FirstName AS "person.name,"
                  p2.LastName AS "person.surname,"
                  p2.Title,
                  p2.BusinessEntityID
           FROM Person.Person AS p2
           WHERE p.BusinessEntityID = p2.BusinessEntityID
           FOR JSON PATH) AS JsonData
INTO dbo.PersonJson
FROM Person.Person AS p;
```

Listing 14-9

This query moves data into a table called `dbo.PersonJson`. I've included both the regular data and the JSON data, just so you can see the conversion if you run queries against it. This is using the `JSON PATH` command to arrive at defined JSON data, similar to how we'd use the `XML PATH` command.

Not only will this load data into the table and convert some of it into JSON, but we can look at the execution plan for this query to see the JSON formatting in action.

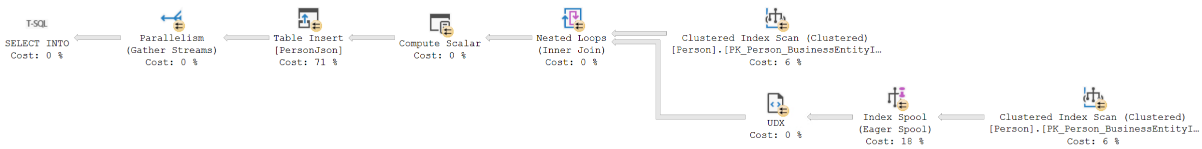


Figure 14-17: Execution plan showing the UDX operator for JSON PATH.

This query processes 19,000 rows, as well as converting them into JSON data, so it's quite a high-cost plan, which explains why the optimizer parallelized it.

There are only two main points of note. First, an **Index Spool** was used to ensure that the **Clustered Index Scan** wasn't used over and over again. You can verify this looking at the **Execution Count** values in both the **Clustered Index Scan** operators, which have a value of 1. The **Index Spool** itself has a value of 19,972, once for each row. Next, the **UDX** operator.

In this case the **UDX** operator is satisfying the needs of the `JSON PATH` operation. We can validate this by looking at the properties. The **Name** value is **FOR JSON**. That's the only indicator we have of what's occurring within this operator. It outputs an expression, **Expr1005**, but there are no other definitions given. You can see all this in Figure 14-18.

⊟ Defined Values	Expr1005
Expr1005	
Description	UDX.
Estimated CPU Cost	0.000001
Estimated Execution Mode	Row
Estimated I/O Cost	0
Estimated Number of Executions	19972
Estimated Number of Rows	1
Estimated Operator Cost	0.02 (0%)
Estimated Rebinds	19971
Estimated Rewinds	0
Estimated Row Size	4035 B
Estimated Subtree Cost	11.3939
Logical Operation	UDX
Name	FOR JSON
Node ID	5
Number of Executions	19972
⊟ Output List	Expr1005
Column	Expr1005

Figure 14-18: FOR JSON expressed within the UDX operator.

The **Compute Scalar** operator performs some type of conversion on **Expr1005** to create **Expr1007**, and then the **Table Insert** operator inserts the rows into the `JsonData` column.

Defined Values	[Expr1007] = Scalar Operator([Exp
⊟ Expr1007	Scalar Operator([Expr1005])
⊟ Identifier	
⊟ Column Reference	Expr1005
Column	Expr1005
ScalarString	[Expr1005]
Description	Compute new values from existir

Figure 14-19: Scalar Operator performing a final operation to create JSON data.

We can't see any of the JSON operations at work based on the properties of the operators. We just know that one operator is `FOR JSON` and the other does some type of conversion. Nothing else is clear.

We can also see evidence of JSON queries at work. Listing 14-10 shows how we can retrieve the JSON data from the table.

```
SELECT oj.FirstName,
       oj.LastName,
       oj.Title
FROM dbo.PersonJson AS pj
    CROSS APPLY
    OPENJSON(pj.JsonData,
             N'$')
    WITH (FirstName VARCHAR(50) N'$.person.name',
          LastName VARCHAR(50) N'$.person.surname',
          Title VARCHAR(8) N'$.Title',
          BusinessEntityID INT N'$.BusinessEntityID') AS oj
WHERE oj.BusinessEntityID = 42;
```

Listing 14-10

We can simply query this data from the columns within the table, but the purpose here is to show `OPENJSON` at work, so we used that instead. The resulting execution plan is quite interesting.

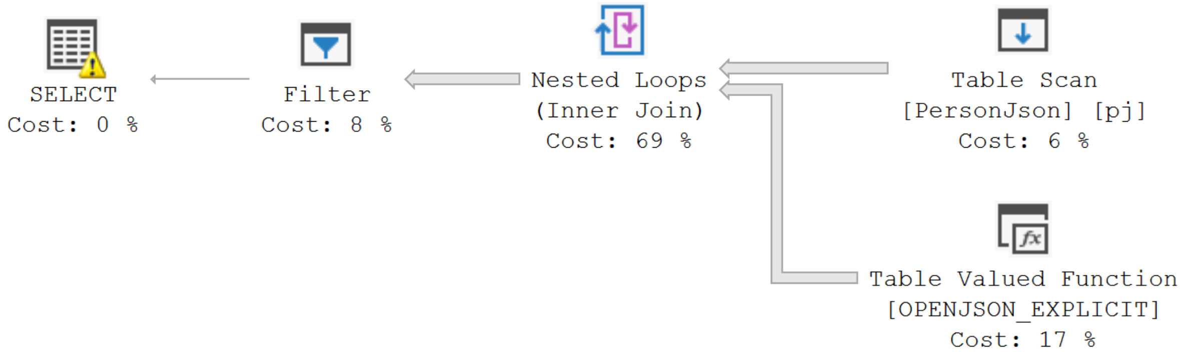


Figure 14-20: Execution for OPEN JSON query.

The **Table Scan** operator is used because the table in question, `dbo.PersonJson`, has no index, so there isn't any other way to retrieve the data. A **Nested Loops** join joins the data from the table to data produced by calls to a function, **Table Valued Function (OPENJSON_EXPLICIT)**. The choice of the **Nested Loops** join might seem surprising given that the **Table Scan** returns an estimated (and actual) 19972 rows, meaning 19972 executions of a relatively expensive table-valued function. The reason is simply because the optimizer has no choice in this case. This query uses a `CROSS APPLY` and the inner input produces different rows for each row from the outer input. The only way for the optimizer to implement this in current versions is with a **Nested Loops**.

The optimizer uses a fixed estimate of 50 rows returned, per execution of the JSON table-valued function. In fact, it returns one row per execution. These rows do not represent a table row, but a new row of data, with the selected JSON data extracted as a relational column in the rowset. The **Filter** operator eliminates all rows other than those that match our `WHERE` clause value on the `BusinessEntityID` column of 42. In other words, it shreds all the JSON in all the rows before applying the filter when, of course, what we'd much rather it did was push down the predicate and only shred the required rows!

If we open the properties of the **Table Valued Function**, we can see some of the JSON activity at work. First, at the bottom of the properties, we see the **Parameter List** values as shown in Figure 14-21.

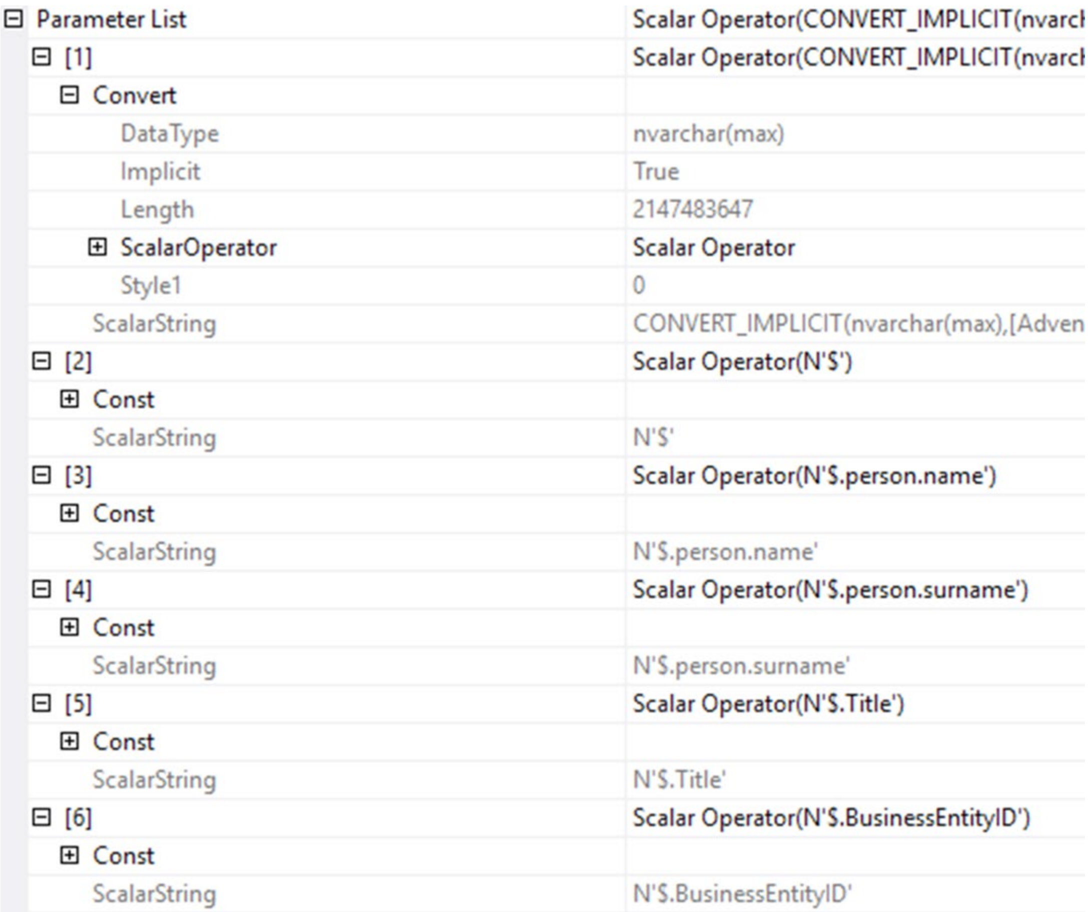


Figure 14-21: Parameter values for the OPENJSON Table Valued Function.

At the top, we're passing in the full JSON string. Then, we pass the path operation and each of the values we're retrieving. We can't see how these parameter values are used internally, but we can see the definitions are very clear.

You also get to see the function's **Defined Values**, as shown in Figure 14-22.

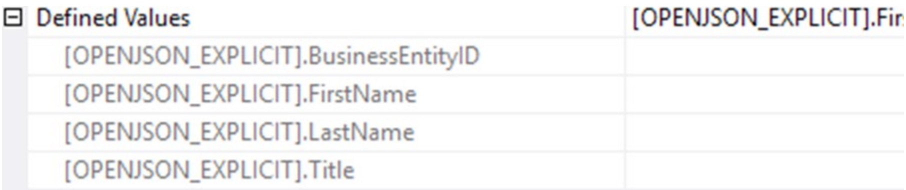


Figure 14-22: Defined values within the OPEN JSON Table Valued Function.

These are the defined aliases within the `WITH` clause of the `OPEN JSON` command in Listing 14-10. These are also the names of the columns used in the **Output List** of the operation.

There are no other indications of exactly how JSON data is converted within SQL Server beyond these hints that you can see within the execution plan. With this information, you can observe the effects of JSON on your queries. In this case, we've discussed several causes for concern: the need to use an inefficient **Nested Loops** join for the `CROSS APPLY`, the fixed estimate of 50 rows returned by the `OPEN JSON` table-valued function, and the need to shred the JSON for every row, *before* filtering. To help with the latter you might consider using a persisted computed column and indexing it for the JSON data that is most often used in filters.

Hierarchical Data

SQL Server can store hierarchical data using `HIERARCHYID`, a data type introduced in SQL Server 2008 (implemented as a CLR data type). It doesn't automatically store hierarchical data; you must define that storage from your applications and T-SQL code, as you make use of the data type. As a CLR data type, it comes with multiple functions for retrieving and manipulating the data. Again, this section simply demonstrates how hierarchical data operations appear in an execution plan; it is not an exhaustive overview of the data type.

Listing 14-11 shows a simple listing of employees that are assigned to a given manager. I've intentionally kept the query simple so that we can concentrate on the activity of the `HIERARCHYID` within the execution plan and not have to worry about other issues surrounding the query.

```
DECLARE @ManagerID HIERARCHYID;
SELECT @ManagerID = e.OrganizationNode
  FROM HumanResources.Employee AS e
  WHERE e.JobTitle = 'Vice President of Engineering';
SELECT e.BusinessEntityID, p.LastName
```

```
FROM HumanResources.Employee AS e
  JOIN Person.Person AS p
    ON e.BusinessEntityID = p.BusinessEntityID
WHERE e.OrganizationNode.IsDescendantOf(@ManagerID) = 1;
```

Listing 14-11

Figure 14-23 shows the execution plan.

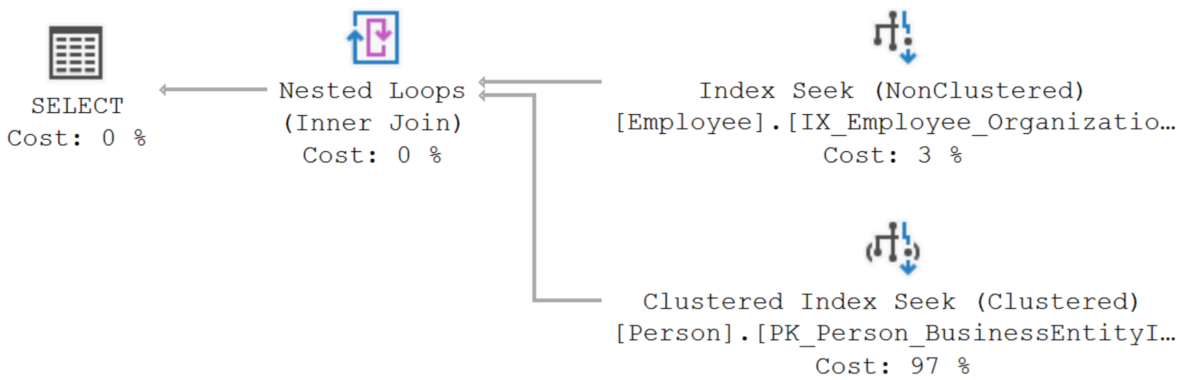


Figure 14-23: Execution plan for hierarchy data.

As you can see, it's a very simple and clean plan. The optimizer is able to make use of an index on the `HIERARCHYID` column, `OrganizationNode`, in order to perform an **Index Seek**. The data then flows out to the **Nested Loops** operator, which retrieves data as needed through a series of **Clustered Index Seek** operations on the `Person.Person` table, to retrieve the additional data requested. The interesting aspect of this plan is the **Seek Predicate** of the **Index Seek** operator, as shown in Figure 14-24.

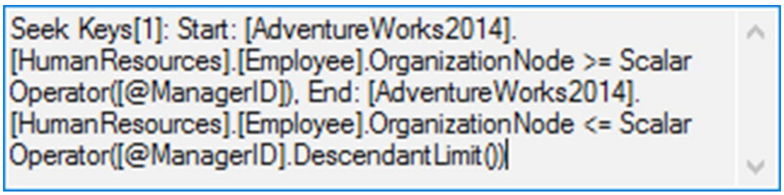


Figure 14-24: Index Seek properties showing hierarchy filtering at work.

Now you can see some of the internal operations performed by the CLR data type. The predicate supplies `Start` and `End` parameters, both working from mechanisms within the `HIERARCHYID` operation. The index is just a normal index, and the `HIERARCHYID` column, `OrganizationNode`, is just a `varbinary` column as far as the Index Seek is concerned. The work is done by internal functions, such as the `DescendantLimit` we see in the Index Seek properties in Figure 14-24, which finds the appropriate `varbinary` value.

If I had run the query and added an extra column to the `SELECT` list, such as `JobTitle` from the `HumanResources.Employee` table, the query would have changed to a **Clustered Index Scan**, or to an **Index Seek** and **Key Lookup**, depending on cost estimates, since the index on `OrganizationNode` would no longer be a covering index.

We could explore a few other functions with the `HIERARCHYID` data type, but this gives a reasonable idea of how it manifests in execution plans, so let's move on to a discussion about another one of the CLR data types, spatial data.

Spatial Data

The spatial data type introduces two different types of information storage. The first is the concept of geometric shapes, and the second is data mapped to a projection of the surface of the Earth. There are a huge number of functions and methods associated with spatial data types and we simply don't have the room to cover all this in detail in this book. For a detailed introduction to spatial data, I recommend *Pro Spatial with SQL Server 2012* (Apress) by Alastair Aitchison.

Like the `HIERARCHYID` data type, there are indexes associated with spatial data, but these indexes are extremely complex in nature. Unlike a clustered or nonclustered index in SQL Server, these indexes can (and do), work with functions, but not all functions. Listing 14-12 shows a query that could result in the use of a spatial index, if one existed, on a SQL Server database.

```
DECLARE @MyLocation GEOGRAPHY = geography::STPointFromText('POI
NT(-122.33383 47.610870)', 4326);
SELECT a.AddressLine1,
       a.City,
       a.PostalCode,
       a.SpatialLocation
FROM Person.Address AS a
WHERE @MyLocation.STDistance(a.SpatialLocation) < 1000;
```

Listing 14-12

This query creates a `GEOGRAPHY` variable and populates it with a specific point on the globe, which coincides with the Seattle Sheraton, near where, most years, PASS hosts its annual Summit. It then uses the `STDistance` calculation on that variable to find all addresses in the database that are within a kilometer (1,000 meters) of that location.

Figure 14-25 shows the plan which, in the absence of a useful index, is just a **Clustered Index Scan**, and then a **Filter**. If we were to review the properties of the **SELECT** operator, we'd see that the **Estimated Subtree Cost** for the plan is 19.9.

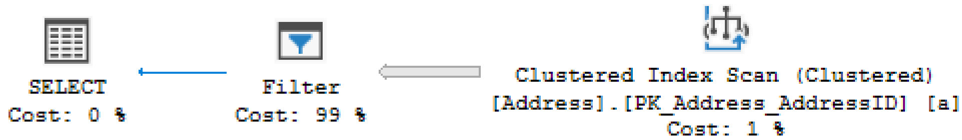


Figure 14-25: Plan for a spatial query with no spatial index.

Let's now create a spatial index on the `Address` table for our spatial query to use, as shown in Listing 14-13.

```
CREATE SPATIAL INDEX TestSpatial
ON Person.Address (SpatialLocation)
USING GEOGRAPHY_GRID
WITH (GRIDS = (LEVEL_1 = MEDIUM, LEVEL_2 = MEDIUM, LEVEL_3 =
MEDIUM, LEVEL_4 = MEDIUM),
      CELLS_PER_OBJECT = 16,
      PAD_INDEX = OFF,
      SORT_IN_TEMPDB = OFF,
      DROP_EXISTING = OFF,
      ALLOW_ROW_LOCKS = ON,
      ALLOW_PAGE_LOCKS = ON)
ON [PRIMARY];
GO
```

Listing 14-13

Rerun Listing 14-12 and you'll see an execution plan that is rather large and involved, when you consider that we're querying a single table, although the estimated cost of the plan is much lower, down from 19.9 to 0.67.

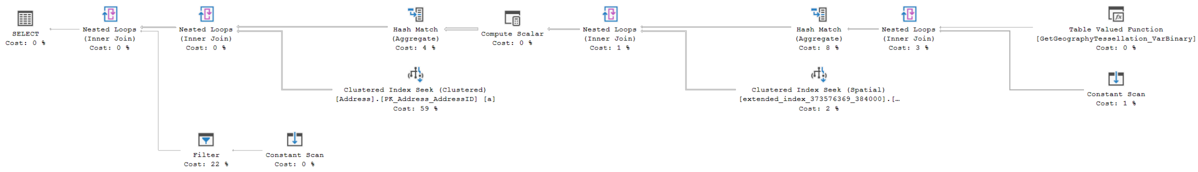


Figure 14-26: Complex execution plan using a spatial index to retrieve data.

To say that spatial indexes are complicated doesn't begin to describe what's going on. You can see that, despite a simple query, a ton of activity is occurring. We'll have to break this down into smaller pieces to understand it. Figure 14-27 focuses on the operators retrieving the initial set of data from the disk.

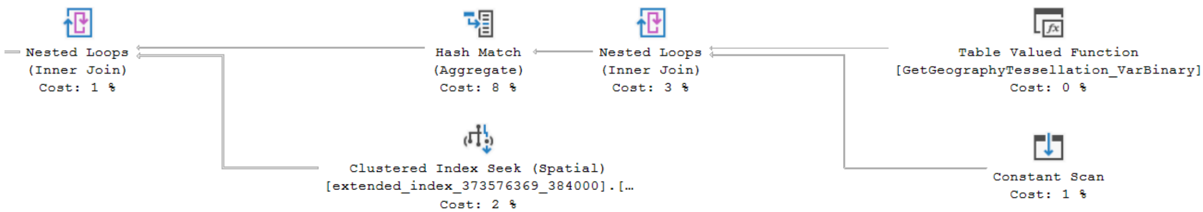


Figure 14-27: Blow-up of plan showing data access.

The **Table Valued Function**, which is named, **GetGeographyTessellation_VarBinary**, is retrieving information using a process called tessellation. It consists of tiles of information defined by our 1000-meter radius around a single point. You can see the parameter values passed in by looking at the properties as shown in Figure 14-28.

Parameter List	Scalar Operator([@MyLocation]), Scalar Operato
⊞ [1]	Scalar Operator([@MyLocation])
⊞ [2]	Scalar Operator((3))
⊞ [3]	Scalar Operator((3))
⊞ [4]	Scalar Operator((3))
⊞ [5]	Scalar Operator((3))
⊞ [6]	Scalar Operator((1024))
⊞ [7]	Scalar Operator((1))
⊞ [8]	Scalar Operator((1.0000000000000000e+003))

Figure 14-28: Parameters for tessellation within Table Valued Function.

Without getting into the details of exactly how geographical data is stored and retrieved, this function reflects the settings of the index we created earlier and shows, in the final parameter value, how the 1000-meter limit is being supplied to the function that retrieves an initial set of data. You get some idea of the complexity of accessing spatial indexes because of this. We can even go further. On the left side of the plan in Figure 14-27 we can see that the values generated are used to perform the **Clustered Index Seek (Spatial)** against the additional storage created as part of the spatial index. This seek isn't the same as others we've seen before, which usually consist of a simple comparison operator, as you can tell by looking at the **Seek Predicates** in Figure 14-29.

Seek Predicates	Seek Keys[1]: Start: [Adventure
⊟ [1]	Seek Keys[1]: Start: [Adventure
⊟ [1]	Start: [AdventureWorks2017].[s
⊟ End	[AdventureWorks2017].[sys].[e
⊞ Range Columns	[AdventureWorks2017].[sys].[e
⊞ Range Expressions	Scalar Operator([Expr1067])
Scan Type	LE
⊟ Start	[AdventureWorks2017].[sys].[e
⊞ Range Columns	[AdventureWorks2017].[sys].[e
⊞ Range Expressions	Scalar Operator([Expr1066])
Scan Type	GE

Figure 14-29: The Seek Predicates against the Spatial Index.

The number of operators involved does make this plan more complicated. It reflects all the work necessary to satisfy a different data type, spatial data.

To clean up, you can drop the index created earlier.

```
DROP INDEX TestSpatial ON Person.Address;
```

Listing 14-14

While these spatial functions are complex and require a lot more knowledge to use, you can see that the execution plans still use the same tools to understand these operations, although in very complex configurations, making troubleshooting these queries harder.

Cursors

Cursors, despite how they are defined within T-SQL, are not data types. They represent a different type of processing behavior. Most operations within a SQL Server database should be set based, rather than using the procedural, row-by-row processing embodied by cursors. However, there will still be occasions when a cursor is the more appropriate or more expedient way to resolve a problem, and there may be times when you do not have time to replace the cursor with a set-based solution, but you need to investigate issues with this code.

While there are some operators that are cursor specific, mainly the optimizer uses the same operators doing the same things we've already seen throughout the rest of the book. However, the operators display differently between estimated and actual plans.

Static cursor

We'll start with the simplest type of cursor, a static cursor. This is the easiest to understand because the data within the cursor can't change, so it simplifies the processing rather radically. Listing 14-15 defines the first cursor.

```
DECLARE CurrencyList CURSOR STATIC FOR
SELECT c.CurrencyCode, cr.Name
  FROM Sales.Currency AS c
    JOIN Sales.CountryRegionCurrency AS crc
      ON crc.CurrencyCode = c.CurrencyCode
    JOIN Person.CountryRegion AS cr
      ON cr.CountryRegionCode = crc.CountryRegionCode
  WHERE c.Name LIKE '%Dollar%';
OPEN CurrencyList;
FETCH NEXT FROM CurrencyList;
WHILE @@Fetch_Status = 0
  BEGIN
    -- Normally there would be operations here using data from
cursor
    FETCH NEXT FROM CurrencyList;
  END;
CLOSE CurrencyList;
DEALLOCATE CurrencyList;
GO
```

Listing 14-15

In Listing 14-15, I don't do anything with the cursor. It doesn't process data or perform other actions commonly associated with cursors. This is simply so we can focus only on the actions of the cursor itself, within execution plans.

Capture the estimated plan for Listing 14-15.

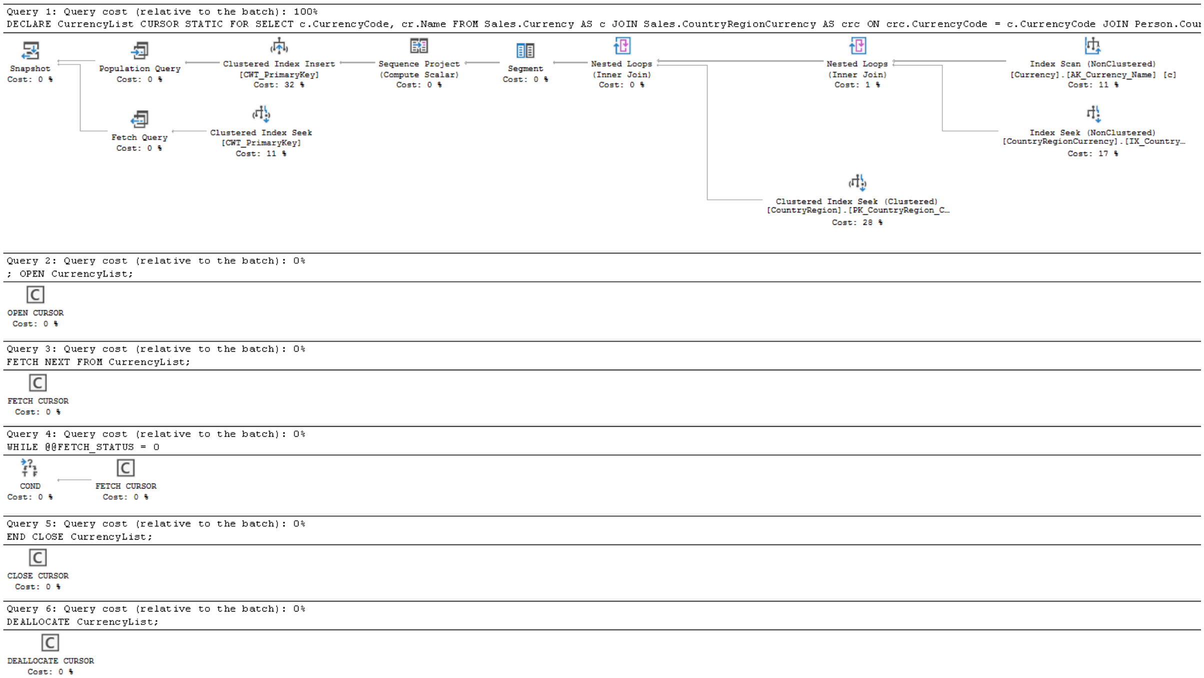


Figure 14-30: Estimated plan for all the statements defining cursor use.

In the estimated plan, most cursor operators are represented using a placeholder icon. The `declare` statement shows the plan that will be used; this is the first execution plan you see at the top of Figure 14-30, and it shows how the cursor will be satisfied, as defined in Listing 14-15.

The plan for the `DECLARE CURSOR` statement shows how the cursor will be populated and accessed based on the other statements from Listing 14-15. We'll focus only on the top plan to start with. Figure 14-31 shows a small part of the plan.

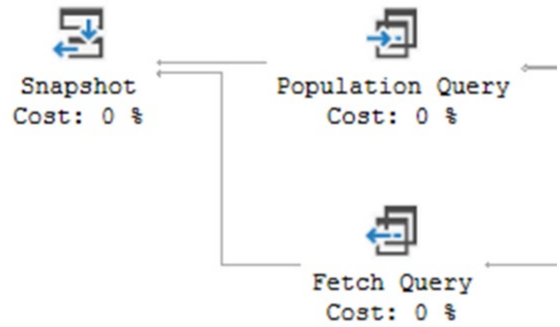


Figure 14-31: Cursor definition showing the Population and Fetch queries.

As you can see, we have an initial operator showing what kind of cursor we have, **Snapshot** in this case. This operator is a lot like the **SELECT** operator; it contains information about the cursor we're defining. Figure 14-32 shows the properties of this operation, providing a full definition of the cursor.

Cursor Actual Type	SnapShot
Cursor Concurrency	ReadOnly
Cursor Name	CurrencyList
Cursor Requested Type	SnapShot
Description	A cursor that does not see changes made by others.
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0309524
Forward Only	False
RetrievedFromCache	true
Statement	DECLARE CurrencyList CURSOR STATIC FORSELECT c.Cu

Figure 14-32: Properties of the Snapshot cursor operator.

The real magic for cursors is in the next two operators shown in Figure 14-31, **Population Query** and the **Fetch Query** operators.

The **Population Query** represents the optimizer's plan to execute the query that will collect the data set to be processed by the cursor. This runs when we `OPEN` the cursor, and then the **Fetch Query** represents the optimizer's plan to fetch each of the rows, and this runs once for every `FETCH` statement.

In this case, because a static cursor should not show any changes made later, the `OPEN` statement simply executes the query and stores the results in a temporary table, and `FETCH` then retrieves rows from that temporary table. Other cursor types use the same basic idea of combining a **Population Query** and a **Fetch Query**, but modified to accommodate the requested cursor type, as we'll see later.

Each of these operators has properties defining the query, again, similar to how the **SELECT** operator would work. Figure 14-33 shows the properties for the `Population Query` operator.

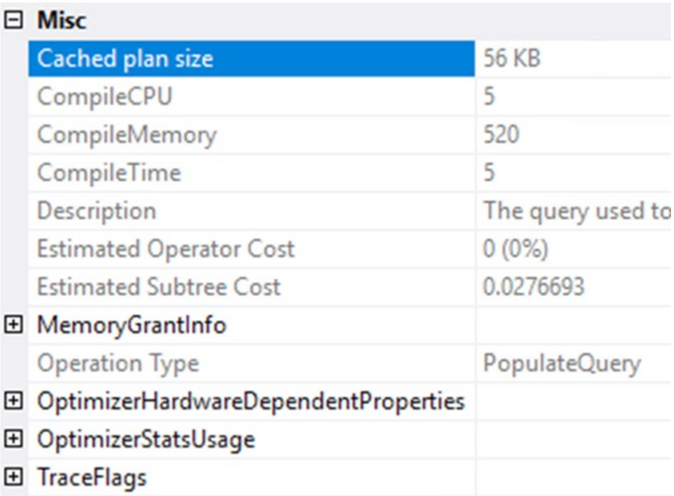


Figure 14-33: Properties of the Population Query operator.

You would use this data in the same way as you would the information in the **SELECT** operator. It provides you the information you need to understand some of the choices made by the optimizer, just as with other plans.

With the understanding that there are two queries at work, let's look at the definitions of those queries as expressed by the execution plans that define this cursor, starting with the first, the **Population Query**. In this case, it's performing two actions. First, it's retrieving data from the disk, as shown in Figure 14-34.

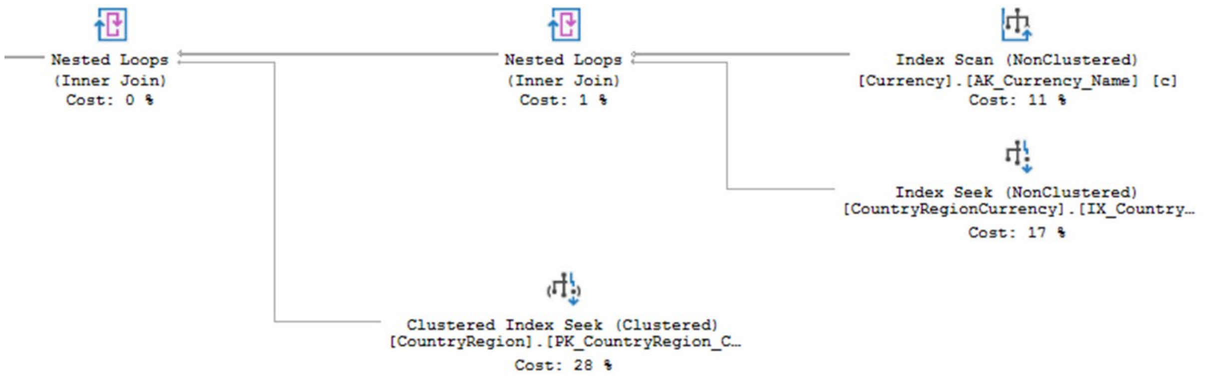


Figure 14-34: Data retrieval for the Population Query of the Static Cursor.

It's a very straightforward execution plan that resolves the query in Listing 14-15. The interesting parts of the execution plan comes after the data set has been defined, as shown in Figure 14-35.

Figure 14-35: Creation of temporary storage for the Static Cursor.

I've included the **Population Query** operator and the **Nested Loops** operator as bookends to the interesting part of the operations, so that it's clearer exactly where these are taking place.

After the data is retrieved and joined, we see a **Segment** and **Sequence Project (Compute Scalar)** operators, which we saw in Chapter 5 when discussing the plans for Window functions. In this case, the **Group By** property of **Segment** is empty, so the entire input is considered a single segment.

The **Sequence Project (Compute Scalar)** operator, which is used by ranking functions, works off an ordered set of data, with segment marks added by the **Segment** operator. In this case, it's adding a row number based on the segmented values, counting from zero each time the segment changes. Here, though, there is only a single segment. Once again, we can see this in the properties as shown in Figure 14-36.

⊞ Defined Values	[Expr1005] = Scalar Operator(i4_row_number)
Description	Adds columns to perform computations over an ordered set.

Figure 14-36: Adding a row_number column to the data set.

What this has done is to create an artificial primary key on the result set of our data for the cursor in question. All the data is then added to a temporary clustered index, `CWT_PrimaryKey`. All this happened in `tempdb`, as we can see this in the properties as shown in Figure 14-37.

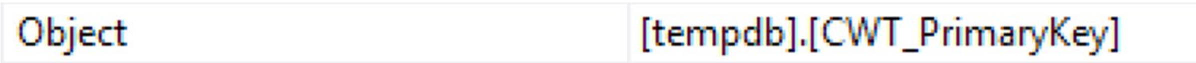


Figure 14-37: The location of the CWT_PrimaryKey object is tempdb.

As noted earlier, the `FETCH` command then simply retrieves rows from this clustered index. Its purpose is to prevent the need to keep going back to the data repeatedly, through a standard query, working much as we've seen spool operators in other plans.

Figure 14-38: Fetch Query execution plan defined.

The rest of the operators in the estimated plan, back in Figure 14-30, are various processes within cursor operations; `OPEN`, `FETCH`, `CLOSE`, and `DEALLOCATE`. Each of these is represented by the Cursor catch-all operator, shown in Figure 14-39.

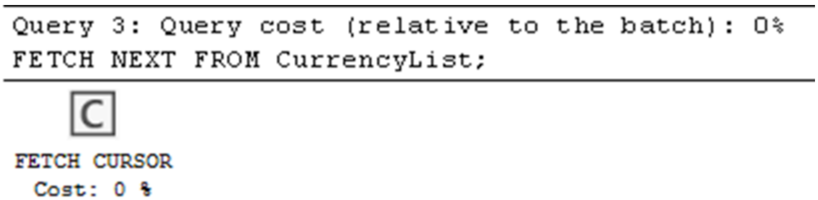


Figure 14-39: Cursor catch-all operator shown in FETCH NEXT command.

These operators will only be visible in the estimated plan. The properties of the operator don't reveal any useful information in most cases since they simply represent the cursor command in question, such as the `FETCH NEXT` command in Figure 14-39.

We can also capture actual plans for a cursor. If you do this, though, be ready to deal with the fact that you will get multiple plans. In this case, one for the **Population Query** and then one each for every row of data for the **Fetch Query**. It will look something like Figure 14-40.

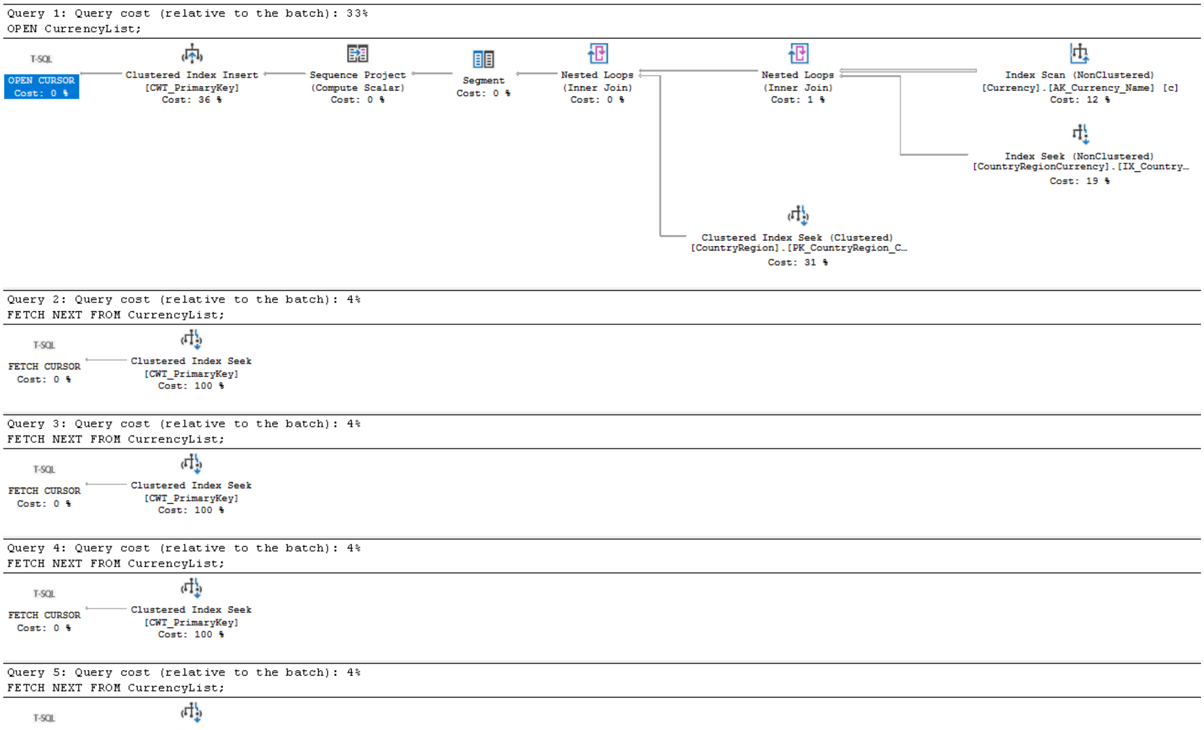


Figure 14-40: Actual plans for a static cursor.

As expected, based on what we saw in the estimated plans, the data is retrieved and put into a clustered index, and then that clustered index is used again and again as we `FETCH` data. The only other point of interest for the actual plan is how the **SELECT** operator has again been replaced, first by an **OPEN CURSOR** operator, and then by multiple **FETCH CURSOR** operators. However, the information within each of these is the same as that found within the **SELECT** operator, including such interesting bits of information as the **Compile Time**, **Query Hash**, and **Set** options.

Capturing actual plans for cursors is an expensive operation and probably shouldn't be done in most circumstances. Instead, use Extended Events to capture a single execution of one of the queries, or use `SET STATISTICS XML ON` for a single statement.

Let's see how the behavior of the plans change as we use different types of cursors.

Keyset cursor

A keyset cursor retrieves a set of keys for the data in question. This is very different than what we saw with the Static cursor above. Keyset cursors should not show new rows, but they should show new data if concurrent updates modify existing rows. To achieve this, the **Population Query** will store the key values in the temporary table, and the **Fetch Query** uses these key values to retrieve the current values in those rows. Our query will now look like Listing 14-16.

```
DECLARE CurrencyList CURSOR KEYSET
FOR
SELECT c.CurrencyCode,
       cr.Name
FROM Sales.Currency AS c
    JOIN Sales.CountryRegionCurrency AS crc
        ON crc.CurrencyCode = c.CurrencyCode
    JOIN Person.CountryRegion AS cr
        ON cr.CountryRegionCode = crc.CountryRegionCode
WHERE c.Name LIKE '%Dollar%';
OPEN CurrencyList;
FETCH NEXT FROM CurrencyList;
WHILE @@FETCH_STATUS = 0
    BEGIN
 -- Normally there would be operations here using data from cursor
        FETCH NEXT FROM CurrencyList;
    END
CLOSE CurrencyList;
DEALLOCATE CurrencyList;
GO
```

Listing 14-16

If we capture an estimated plan for this set of queries, we'll again see a plan that defines the cursor, and a series of catch-all plans for the rest of the supporting statements for the cursor operations. We'll focus here on just the definition of the cursors. The full plan is shown in Figure 14-41.

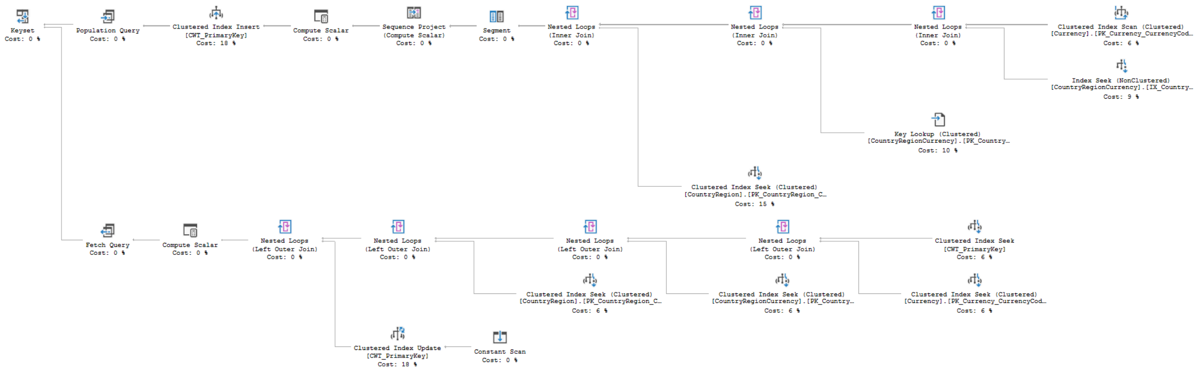


Figure 14-41: Plan to define a Keyset cursor.

Once again, the plan for the `DECLARE CURSOR` statement shows the **Population Query** and the **Fetch Query**. The differences are in the fundamental behavior. We'll start with the part of the plan that retrieves the data for the **Population Query**.

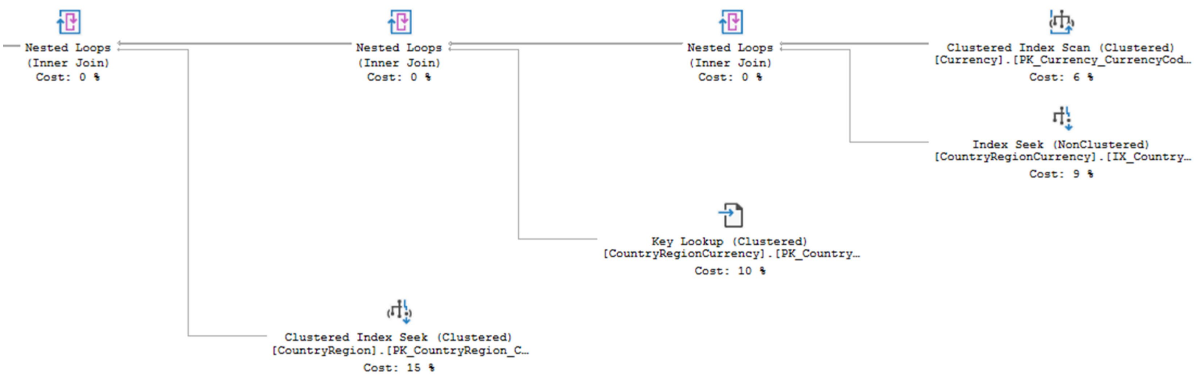


Figure 14-42: Data retrieval for the Population Query of the Keyset cursor.

Again, this execution plan doesn't introduce anything we haven't seen elsewhere in the book. The one very important thing to note, though, is that this plan for data retrieval is different than the earlier plan for data retrieval with the static cursor (Figure 14-34). The **Key Lookup** operator has been added because, to support the Keyset cursor, it must retrieve all key values. So while, before, the **Nonclustered Index Seek** satisfied the plan, now we have to get a new value, a key check value that can only come from the clustered index key. You can see this in the output of each of the **Clustered Index Seek** and **Clustered Index Scan** operators, in Figure 14-43.

⊟ Output List	Chk1002, [AdventureWorks2014].[Sales].[Currency].CurrencyCode
⊟ [1]	Chk1002
Column	Chk1002
⊟ [2]	[AdventureWorks2014].[Sales].[Currency].CurrencyCode
Alias	[c]
Column	CurrencyCode
Database	[AdventureWorks2014]
Schema	[Sales]
Table	[Currency]

Figure 14-43: Check columns added from the clustered indexes.

This value will be used later, as we'll see. The next part of the **Population Query** is much the same as before.

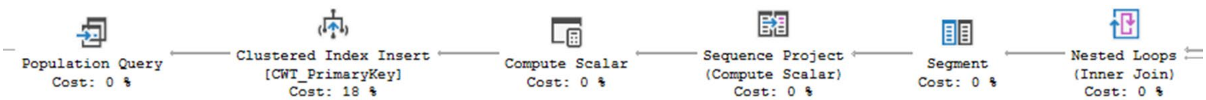


Figure 14-44: Loading information into a temporary index for later use.

A temporary index is created for use by the **Fetch Query**, the plan for which is shown in Figure 14-45.

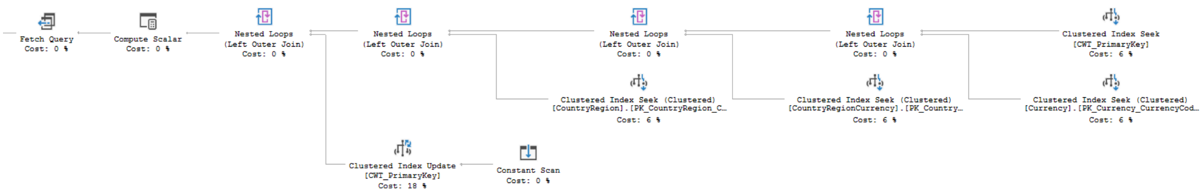


Figure 14-45: Fetch Query for the Keyset operator.

This is much more complicated than the previous cursor. This is because, with the Keyset cursor, the data can change. So, to retrieve the correct data set, instead of simply looking at everything stored within the temporary index, it has read the key values from the clustered index scan on `CWT_PrimaryKey`, then used them to do **Clustered Index Seek**s on the other tables. Also note that those are all using a **Left Outer Join**, because it is possible that the referenced row has been deleted since.

Then, we're going to each of those tables to retrieve the data based on the key values stored.

Output List	Chk1006, IsBaseRow1007, [AdventureWorks2014].[Person].[CountryRegion
⊞ [1]	Chk1006
⊞ [2]	IsBaseRow1007
⊞ [3]	[AdventureWorks2014].[Person].[CountryRegion].Name
Parallel	False
Physical Operation	Clustered Index Seek
Scan Direction	FORWARD
Seek Predicates	Seek Keys[1]: Prefix: [AdventureWorks2014].[Person].[CountryRegion].Cou
⊟ [1]	Seek Keys[1]: Prefix: [AdventureWorks2014].[Person].[CountryRegion].Cou
⊟ [1]	Prefix: [AdventureWorks2014].[Person].[CountryRegion].CountryRegionCc
⊟ Prefix	[AdventureWorks2014].[Person].[CountryRegion].CountryRegionCode = S
⊞ Range Columns	[AdventureWorks2014].[Person].[CountryRegion].CountryRegionCode
⊞ Range Expressions	Scalar Operator([CWT].[COLUMN5])
Scan Type	EQ

Figure 14-46: Retrieving the data from the tables based on the key values.

There is also a check to see if data has been deleted, which explains the final **Compute Scalar** operator. The **Nested Loops** (**Left Outer Join**) operator, immediately to the right of the **Compute Scalar**, is there to put together data in preparation for the check.

The actual plans are much the same as before. You'll see one instance of the execution plan for the **Population Query** and then a series of plans for the **Fetch Query**.

Dynamic cursor

Finally, we'll look at a dynamic cursor. Here, any of the data can be changed in any way, at any point where we access the cursor. The actual code change is small so, instead of repeating the entire code list, I'll just show the change in Listing 14-17.

```
DECLARE CurrencyList CURSOR DYNAMIC
```

Listing 14-17

Capturing an estimated plan for this new cursor results in yet another variation on the execution plans we've already seen. I'll focus on the details of the execution plan for the cursor definition, since all the catch-all behaviors are the same.

Figure 14-47 shows the estimated plan.

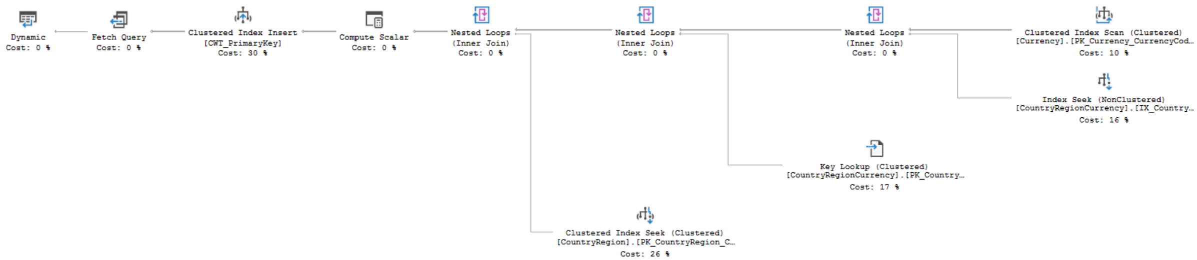


Figure 14-47: Estimated execution plan for a Dynamic cursor.

The biggest point to note here is that we only have a **Fetch Query**. There is no **Populate Query** for dynamic cursors. The data and the order of the data can change, so all we can do is run the full query, every time. There is a **Compute Scalar** operator to add an **ID** value, and we store the information retrieved into a temporary clustered index. This enables us to move in multiple directions within the cursor, not just forward, but the data is fetched repeatedly as the cursor runs, which is why this is the least efficient of the various cursor types.

Interestingly enough, somewhere in the internals, there are checks that somehow keep the engine from executing the query over and over, every time. The details are not known to me but, effectively, you need to think about this approach as if it did execute the query 15 times. Capturing the actual plans for this cursor will only show the same execution plan over and over.

There are several other options that can affect cursor behavior, but that won't reflect in any novel ways within the execution plan. The behaviors you can expect are reflected in the examples provided.

Summary

The introduction of these different data types, XML, Hierarchical, Spatial and JSON, radically expands the sort of data that we can store in SQL Server and the sort of operations that we can perform on this data. Each of these types is reflected differently within execution plans. Cursors also add new wrinkles to what we're going to see within execution plans. Neither the complex data types, nor cursors, fundamentally change what's needed to understand the execution plans. Many of the same operators are in use, even though these special data types and cursors have added values. You still have to drill down to the properties and walk through the details in order to come to an understanding of how execution plans display the behaviors defined within your T-SQL, even if it's for a cursor or a special data type.

Chapter 15: Automating Plan Capture

Throughout this book, we've been executing ad hoc queries and code modules from within SSMS, and capturing their execution plans. In Chapter 9, we also explored how to retrieve the plans currently in the plan cache, by querying a set of execution-related Dynamic Management Objects (DMOs). These DMO queries allowed us to return interesting properties for each plan, such as its size and the number of times it had been used, as well as runtime metrics. Many of the columns that store these metrics are counters and return a row for each query statement in each plan. For example, each time a cached plan is executed, the time taken to execute each query is added to `total_elapsed_time` counter value for that row. In other words, the metrics are aggregated over the time the plan has been in cache. If you're using the Query Store, you can capture plans and track aggregated runtime metrics over even longer periods (as explored in detail in Chapter 16).

While this information is useful, there are times when the history of aggregated metrics obscures the cause of a recent problem with a query. If a query is performing erratically, or a SQL instance is experiencing performance problems only at specific times, then you'll want to capture the plans and associated execution metrics for each of the queries in a workload, over that period. If that period happens to be at around 2 a.m., then you'd probably rather have a tool to capture the information for you automatically.

We're going to look at how to use two tools, Extended Events and SQL Trace, to capture automatically the execution plans for each query in the workload or, perhaps more specifically, for the most resource-intensive and long-running queries in that workload.

Why Automate Plan Capture?

Situations of all kinds can arise where capturing a plan using SSMS, using the information in the Query Store, or querying the plan cache, won't give you the data you need, or won't give it to you easily and accurately. For example, if your applications submit a very high number of ad hoc, unparameterized queries, this will essentially flood the cache with single-use plans, and older plans will quickly age out of the cache, as discussed in Chapter 9. In that situation, you'll probably have Query Store configured so that it's not capturing all the plans either. Therefore, the plans for a given query you want to investigate may no longer be cached or within the Query Store.

During development work, you can capture the plans for your test workload simply by adding `SET STATISTICS XML` commands to the code. However, this requires code changes that are not always possible or easy when tackling a production server workload.

It's under these circumstances and others that we're going to go to other tools to retrieve execution plans.

Tools for Automating Plan Capture

First, I'm going to show you how to use Extended Events to capture actual plans, and then the tool that it replaced, SQL Trace. Starting in Azure SQL Database, and in SQL Server 2016 or better, you also have access to the Query Store as a means of investigating execution plans, and we'll cover that topic in the next chapter. However, one thing that Query Store does not give you that Extended Events and SQL Trace do, is detailed runtime metrics.

My basic assumption is that you're working on SQL Server 2012 or higher, or on Azure SQL Database. In either case, you really should be using Extended Events rather than SQL Trace, as it's a far superior tool for collecting diagnostic data for all the different types of events that occur within our SQL Server instances and databases.

All new functionality in SQL Server uses Extended Events as its internal monitoring mechanism. The GUI built into Management Studio is updated regularly and has a lot of functionality to make it quite attractive, especially when tuning queries and looking at execution plans. Diagnostic data collection with Extended Events adds a much lower overhead than with SQL Trace, and so has a much lower impact on the server under observation, since the events are captured at a lower level within the SQL Server system. SQL Trace, broadly speaking, works on the principle of collecting all the event data that could possibly be required, and then discarding that which individual traces don't need. Extended Events works on the opposite principle; it collects as little data as possible and allows us to define precisely the circumstances under which to collect event data. Finally, SQL Trace events are on the deprecation path within SQL Server so, at some point, they won't be available.

All this said, if you're still working on SQL Server 2005, then you'll have to use SQL Trace, since Extended Events were only introduced in SQL Server 2008. If you're working on SQL Server 2008 or SQL Server 2008R2, then Extended Events are available, but these early releases of it offered a far less complete set of events, and one of the missing events in 2008 and 2008R2 is the ability to capture execution plans. There are other weaknesses too, such as the absence of an SSMS-integrated GUI, meaning we must parse the XML event data.

CAUTION! Automating plan capture on production servers
With either of these tools, you are capturing the cached plan on the same thread that executes the query; in other words, it is an in-process operation. Further, execution plans can be big, so capturing them using these tools adds considerable in-process memory and I/O overhead. As such, do exercise caution when running Extended Events sessions or server-side traces that capture the plan, on a production server. Be sure to add very granular filters to these execution plan events, so that you are capturing the plan for as few event instances as possible.

Automating plan capture using Extended Events

With Extended Events we can collect and analyze diagnostic data for many types of events occurring within SQL Server instances and databases. For example, we can collect data for events relating to T-SQL statement or stored procedure execution, locked processes, deadlocks, and many more. We create an **event session**, loosely the equivalent of a trace in SQL Trace, to which we add the required events, specify any additional data required that we wish to collect as an **action**, add the **predicates** of filters that will limit when data is collected and how much, and finally specify the **targets** that will consume the event data.

For example, if we have long-running queries that consume a lot of CPU and I/O resources, then we may want to capture the plans for these queries, along with one or two other useful events, to find out why. You can capture wait statistics for a given query or stored procedure. You can use extended events to observe the statistics being queried and consumed by the optimizer as it compiles a plan. You can see compile and recompile events and you can correlate each of these to others so that you can achieve a complete picture of the behavior of queries within the system, far beyond anything possible before the introduction of Extended Events, but all of which goes beyond the scope of this book.

Extended Events provides three events that capture execution plans. Each one captures the plan at a different stage in the optimization process. The `query_post_compilation_showplan` event fires only on plan compilation. The first time you call a stored procedure, or execute a batch or ad hoc query, you'll see this event fired. If you execute them again, and their plans are reused from cache, the event won't fire. This event will also fire when you request an estimated plan, assuming there is no cached plan for that query.

The `query_pre_execution_showplan` event fires right before a query executes. It shows the plan that was either compiled or retrieved from cache. This is a very useful event when you're dealing with lots of recompiles and you want to see plans before and after the recompile.

Since both the above events fire before query execution, neither contains runtime statistics. If you want those, you'll need to capture the `query_post_execution_showplan` event. Since it's capturing the plan, plus runtime metrics, for all queries that meet the filter criteria of your event session, it's also more expensive than capturing the equivalent pre-execution events. While I advocate its use, remember my earlier caution: please be careful with this event, and the other two. Carefully test any event session that captures them, prior to running it in a production environment.

Create an event session using the SSMS GUI

Event sessions are stored on each server, and you can find them in SSMS Object Explorer, as shown in Figure 15-1. The `AlwaysOn_health`, `system_health` and `telemetry_xevents` are built-in event sessions, and all the rest are sessions I've created. Green arrows define the event sessions that are currently running and red squares those currently stopped.

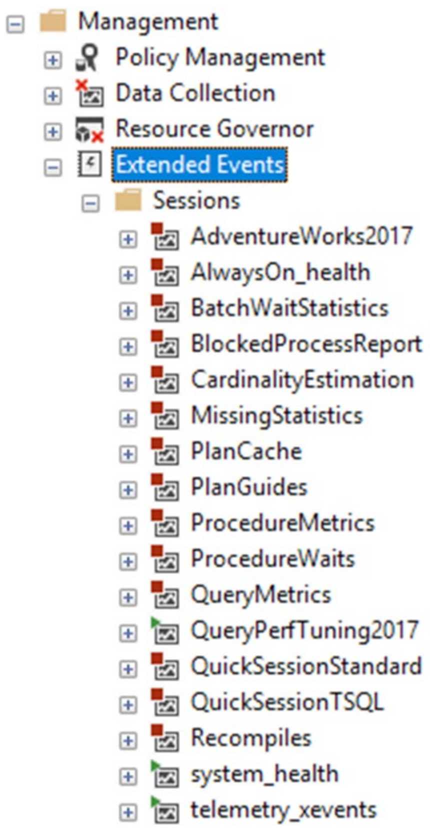


Figure 15-1: A list of Extended Events sessions.

My preferred way to create new event session is using T-SQL, but it's sometimes useful to use the GUI to create a new session quickly, and then script it out and tweak it, as required. Therefore, let's see how to create an event session that captures our execution plan-related events, using the **New Session** dialog.

I won't cover every detail and every available option when creating event sessions. For that, please go to the Microsoft documentation (https://bit.ly/2Ee8cok). I also won't cover the **New Session Wizard**, because it has various limitations, and can only be used to create new event sessions. If you want to alter an existing event, then the dialog for that uses the same layout and options as the **New Session** dialog.

Right-click on the **Sessions** folder and select **New session…** from the context menu to open the **New Session** dialog.

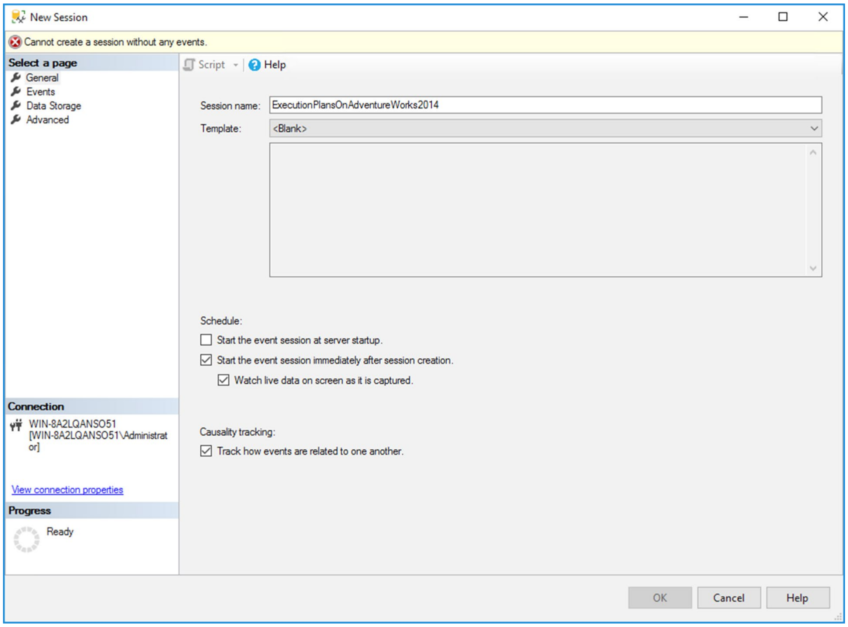


Figure 15-2: New Session window for Extended Events.

This figure shows the **General** page of the dialog, where we give the session a name, specify when we want the session to start running and a few other options. I've given the new event session a name, **ExecutionPlansOnAdventureWorks2014**, and specified that the session should start running as soon as we create it, and that I want to watch live event data, on the screen. I have also turned on **Causality tracking** for this session and I'll explain what that does, briefly, later in the chapter.

Now click over to the next page, **Events**, where we can select the events for the session.

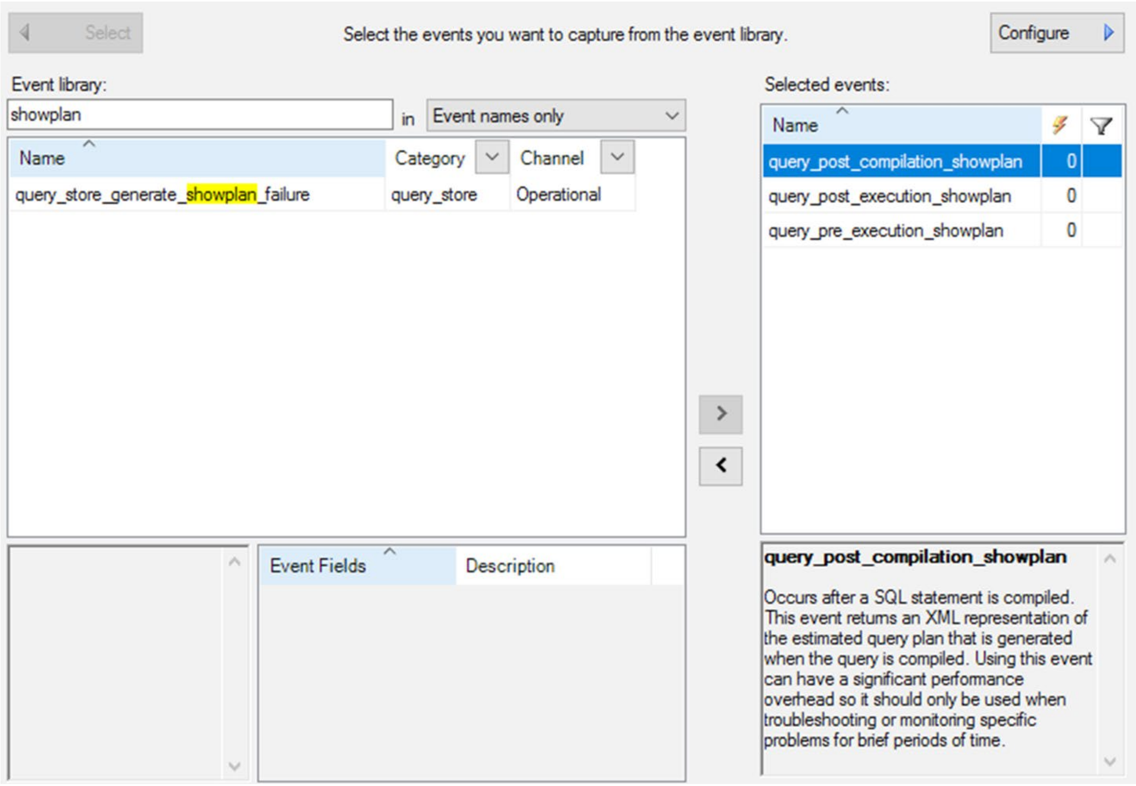


Figure 15-3: The Events page of a new Extended Events Session.

In the left-hand pane, we identify the events we want to capture. I've used the **Event library** textbox to filter for event names that contain the word **showplan**. There are four of them, and in Figure 15-3, I've already used the '>' arrow to select the three events I want.

I've highlighted the `query_post_compilation_showplan` event, and it shows a description for that event in the panel below, along with a warning that you could see performance issues by capturing this event.

I also want to capture one other event, not directly related to execution plans, `sql_batch_completed`, which fires when a T-SQL batch has finished executing, and provides useful performance metrics of that query. Often, it's also useful to add the `sql_statement_recompile` event, which fires when a statement-level recompilation occurs, for any kind of batch, and provides useful event fields that reveal the cause of the recompile, and the identity of database and object on which it occurred.

Having selected all four events, click on the **Configure** button at the top right. This changes our view but we're still on the same **Events** page, and it's here that we can control the behavior of our event sessions.

On the **Filter (predicate)** tab, we can define predicates for our event session that will define the circumstances in which we wish the event to fire fully, and to collect that data. In this example, we only want to collect the event data if the event fires on the `AdventureWorks2014` database *and* only for a query that accesses the `Person.Person` table, as shown in Figure 15-4.

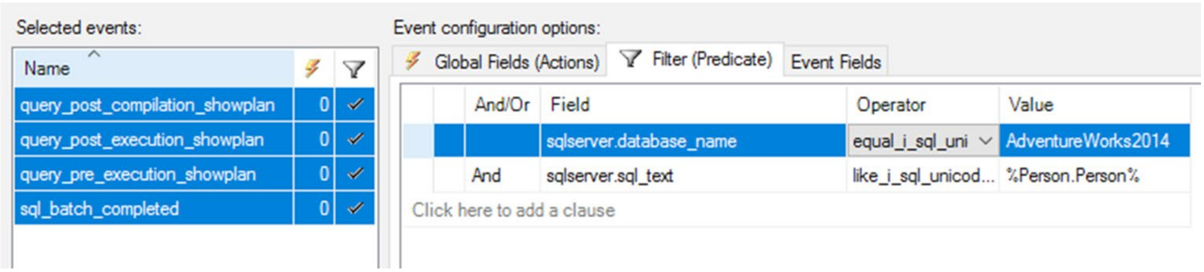


Figure 15-4: Configuring the selected events in a new Extended Events session.

I've used the mouse and the **Shift** key to select all four events and then added the two filters to all four events. To limit event data collection to the `AdventureWorks2014` database, we need to create a predicate on the `sqlserver.database_name` global field. The required operator is the `equal_i_sql_unicode_string` textual comparator, in order to compare the `database_name` for the event raised with the string '`AdventureWorks2014`'. The event engine will only fire the event fully and collect the data if they match. To restrict data collection still further, I add the `And` operator and a second predicate on the `sqlserver.sql_text` global field, selecting the `like_i_sql_unicode_string` comparator, to use the `LIKE` command, and the value `%Person.Person%`.

In this way, despite the query plan Extended Events being expensive, I've ensured that I'm only capturing a very limited set of those events.

While we won't do this here, we can use the other two tabs to control the data we want the event session to collect. In the **Event Fields** tab, we can see the event data columns that define the **base payload** for the event, i.e. they will always be captured when the event fires, plus any event data columns that are configurable.

In the **Global Fields (Actions)** tab, we can specify any additional data we want to add to the event payload, as an "action." No global fields are collected by default, in stark contrast to SQL Trace, where every event collects this data when the event fires, even if it is not a part of the trace file definition. For example, if we wanted to collect the exact SQL text on an event that doesn't already collect that information, then we'd add the `sql_text` global field to the event session, explicitly, as an action. Actions add some additional overhead, so choose when and how to use them with some caution.

Next, click on the **Data Storage** page on the left, where we can specify one or more targets in which to collect the event data.

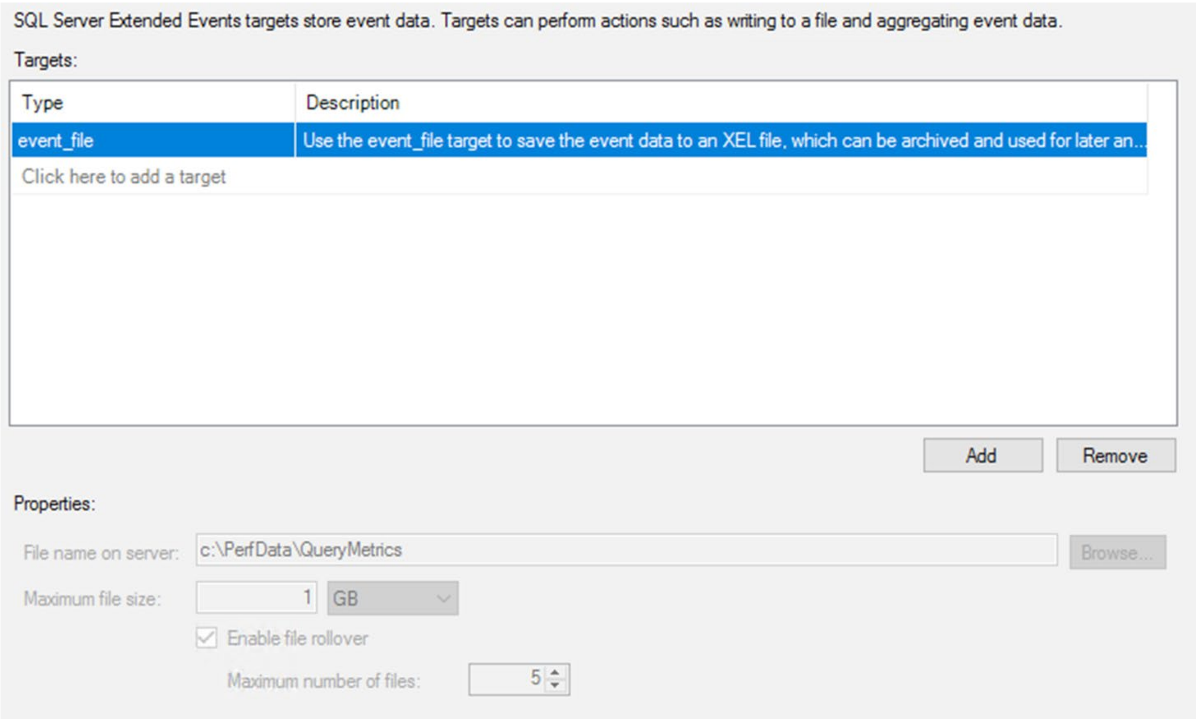


Figure 15-5: The Data Storage page of a new Extended Events session.

I've used the `event_file` target, which is simply flat file storage, similar to the server-side trace file. It's the most commonly used target for standard event sessions, and usually performs better than the other options. You can define the file properties in the lower window. Except for defining the precise location of the file on the server, I've accepted all defaults in this instance.

There is a final **Advanced** page, where we can set a range of advanced session options, which relate to configuring the memory buffer for events, dispatch frequency to the target, and event retention in the target. We won't be covering that here.

With this, you can click on the **OK** button to create the new event session. If you have done as I have, specifying that the session should start and to show the **Live Data** window, you'll not only see a new session, but a new window will open in SSMS. We'll get to that in just a minute.

Create an event session in T-SQL

As I stated earlier, I generally prefer to create event sessions in T-SQL. It's simple and clear and makes it easier for you to migrate sessions between different servers. In this case, I'll simply show the T-SQL for the **ExecutionPlansOnAdventureWorks2014** event session that we just created (simply right-click on the event session in SSMS Object Explorer and use **Script Session As…**).

```
CREATE EVENT SESSION ExecutionPlansOnAdventureWorks2014
ON SERVER
    ADD EVENT sqlserver.query_post_compilation_showplan
    (WHERE (    sqlserver.database_name = N'AdventureWorks2014'
                AND sqlserver.like_i_sql_unicode_string(sqlserver.
sql_text, N'%Person.Person%'))),
    ADD EVENT sqlserver.query_post_execution_showplan
    (WHERE (    sqlserver.database_name = N'AdventureWorks2014'
                AND sqlserver.like_i_sql_unicode_string(sqlserver.
sql_text, N'%Person.Person%'))),
    ADD EVENT sqlserver.query_pre_execution_showplan
    (WHERE (    sqlserver.database_name = N'AdventureWorks2014'
                AND sqlserver.like_i_sql_unicode_string(sqlserver.
sql_text, N'%Person.Person%'))),
    ADD EVENT sqlserver.sql_batch_completed
    (WHERE (    sqlserver.database_name = N'AdventureWorks2014'
                AND sqlserver.like_i_sql_unicode_string(sqlserver.
sql_text, N'%Person.Person%')))
    ADD TARGET package0.event_file
    (SET filename = N'C:\PerfData\ExecutionPlansOnAdventureWor
ks2014.xel')
```

```
WITH (MAX_MEMORY = 4096KB,
      EVENT_RETENTION_MODE = ALLOW_SINGLE_EVENT_LOSS,
      MAX_DISPATCH_LATENCY = 30 SECONDS,
      MAX_EVENT_SIZE = 0KB,
      MEMORY_PARTITION_MODE = NONE,
      TRACK_CAUSALITY = ON,
      STARTUP_STATE = OFF)
GO
```

Listing 15-1

Each of the execution plan events uses the same predicate or filter definitions, as we can see in the WHERE clause for each event. The code is straightforward and you can see every choice we made in the GUI reflected in the T-SQL statements.

Viewing the event data

If you followed exactly, you have a session running and the **Live Data Viewer** window open. If not, you'll need to right-click on a session and select **Start** from the menu choice, then right-click again and select **Watch Live Data**.

Now, execute the query shown in Listing 15-2. To make sure that we capture all three execution plan events, the opening section of the code grabs the plan_handle for a cached plan for a query that contains the text %Person.Person%, and then uses it to remove those plans from cache. That done, we run the query that will cause the events to fire.

```
USE AdventureWorks2014;
GO
DECLARE @PlanHandle VARBINARY(64);
SELECT  @PlanHandle = deqs.plan_handle
FROM    sys.dm_exec_query_stats AS deqs
        CROSS APPLY sys.dm_exec_sql_text(deqs.sql_handle) AS dest
WHERE   dest.text LIKE '%Person.Person%';
IF @PlanHandle IS NOT NULL
    BEGIN
        DBCC FREEPROCCACHE(@PlanHandle);
    END;
GO
```

```
SELECT  p.LastName + ', ' + p.FirstName ,
        p.Title ,
        pp.PhoneNumber
FROM    Person.Person AS p
        JOIN Person.PersonPhone AS pp
        ON pp.BusinessEntityID = p.BusinessEntityID
        JOIN Person.PhoneNumberType AS pnt
        ON pnt.PhoneNumberTypeID = pp.PhoneNumberTypeID
WHERE   pnt.Name = 'Cell'
        AND p.LastName = 'Dempsey';
GO
```

Listing 15-2

I've run both the DMO query in one batch, and the actual query in a second batch, and both in the context of `AdventureWorks`, so you'll see all four events fire twice. Figure 15-6 shows only the four relating to the execution of the second batch.

	query_post_compilation_showplan	2018-04-02 12:35:34.4988601
	query_pre_execution_showplan	2018-04-02 12:35:34.4996299
	query_post_execution_showplan	2018-04-02 12:35:34.5015638
	sql_batch_completed	2018-04-02 12:35:34.5016588

Figure 15-6: Events in the Live Data viewer showing the events we captured.

If you rerun just the query batch, you'll only see three events; you won't see a `post_compilation` event since the query won't be compiling again. Click on any of the ***_showplan** event instances in the upper grid to see the associated query plan displayed graphically in the **Query Plan** tab.

We're not going to explore this plan in detail, except to note that the first operator is not the **SELECT** operator, as we've seen throughout the book. Instead, the first operator for plans captured using Extended Events is the first operator of the plan as defined by the `NodeID` value. For some reason known only to Microsoft, some of the properties normally displayed for the first operator are not displayed in Extended Events. As explained in Chapter 13, you can still find this information in the XML for the plan simply by right-clicking on the graphical plan, selecting **Show Execution Plan XML**, and looking in the `StmtSimple` element for the plan.

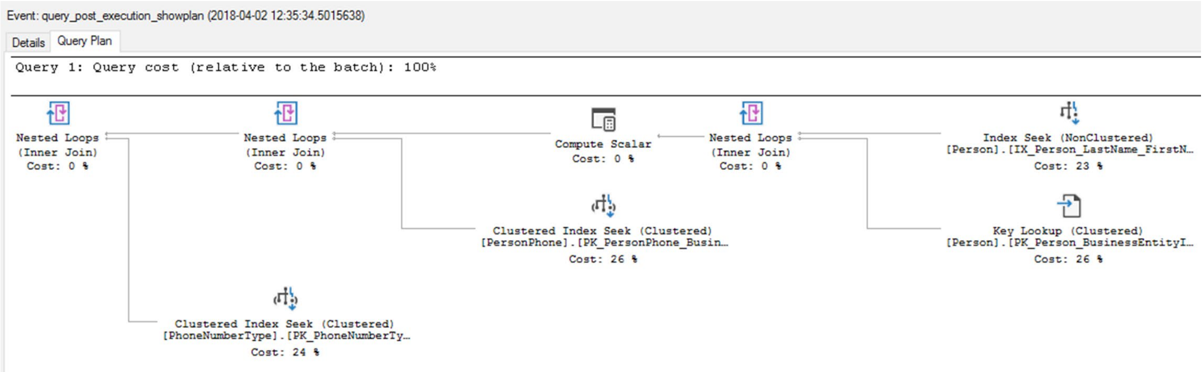


Figure 15-7: An execution plan captured by Extended Events.

The **Details** tab for each event reveals some information that can be useful to your query-tuning efforts. For example, Figure 15-8 shows the **Details** pane for our first event, `query_post_compilation_showplan`.

Event: query_post_compilation_showplan (2018-04-02 12:35:49.6156056)

Details | Query Plan

Field	Value
attach_activity_id.guid	7D38C6DB-E960-4ADB-84A5-2E6B8B3CA4DE
attach_activity_id.seq	1
attach_activity_id_xfer.guid	ED12089B-9305-4C89-8EAE-2292673D3BD5
attach_activity_id_xfer.seq	0
begin_offset	4
cpu_time	9
database_name	
duration	9328
end_offset	738
estimated_cost	0
estimated_rows	1
nest_level	0
object_id	464886589
object_name	Dynamic SQL
object_type	ADHOC
plan_handle	0x060005003D9BB51B00BC852479010000010000000000000...
recompile_count	1
serial_ideal_memory_kb	0
showplan_xml	<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver...
source_database_id	5
sql_handle	0x020000003D9BB51B693714ED340CE1DF4861449EA499C4...

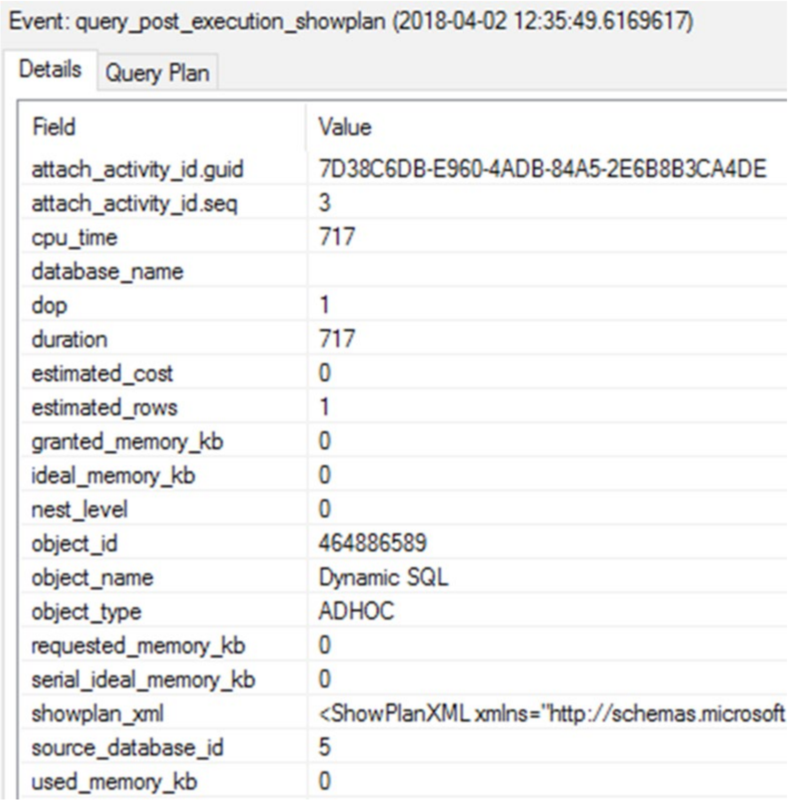
Figure 15-8: query_post_compilation_showplan details.

The first set of fields, with names starting with `attach_`, is added to event sessions for which `TRACK_CAUSALITY` is set to `ON`, as it is for this session. This means that a set of events that are linked will have a common ID and a sequence. You can see that, in our sequence, this is the first event. This is useful if you want to group all the activities together for any given set of events, defined by the `attach_activity_id.guid` value, and order these events in the precise order in which they occurred within SQL Server, as shown by the `attach_activity_id.seq` value. On a test system such as mine, this may not matter because I'm the only one running queries. However, capturing events like this on a production system, even well-filtered events, you may see additional queries and event sets in which you've no interest. Alternatively, you may see multiple interesting events, but interlaced because they were executed at the same time, and in these cases the `activity_id` values can help you find out which ones belong together.

The interesting information is further down. For example, the `duration` field shows the time it took to compile this plan, which was 4192 microseconds on my machine. You can also see that the estimated number of rows returned was 1. We also have the `plan_handle` and `sql_handle` which can be used to retrieve this plan and the T-SQL code from cache, if required. The `showplan_xml` column has the plan as XML. The `object_name` column describes this query as Dynamic SQL. This is accurate for the kind of query I'm running in this case, which is just a T-SQL statement, not a prepared statement or stored procedure. When pulling plans for stored procedures or other objects, you'll be able to see their object names as well as the `object_type`.

The next event is `query_pre_execution_showplan` which shows similar information, but the base payload for this event doesn't include a few of the event fields that we saw for the previous event, such as the `plan_handle` and `sql_handle`.

The third event in the sequence is `query_post_execution_showplan` with the details shown in Figure 15-9.



Event: query_post_execution_showplan (2018-04-02 12:35:49.6169617)

Details | Query Plan

Field	Value
attach_activity_id.guid	7D38C6DB-E960-4ADB-84A5-2E6B8B3CA4DE
attach_activity_id.seq	3
cpu_time	717
database_name	
dop	1
duration	717
estimated_cost	0
estimated_rows	1
granted_memory_kb	0
ideal_memory_kb	0
nest_level	0
object_id	464886589
object_name	Dynamic SQL
object_type	ADHOC
requested_memory_kb	0
serial_ideal_memory_kb	0
showplan_xml	<ShowPlanXML xmlns="http://schemas.microsoft
source_database_id	5
used_memory_kb	0

Figure 15-9: query_post_execution_showplan details.

As you can see there's not much additional information about the plan here. The detail is in the plan itself. Importantly, the plan captured by this event has runtime information. Click on the **Query Plan** tab and examine the **Properties** for the **Nested Loops** operator, and you'll see that we have actual runtime counters for the number of rows and number of executions, as well as estimated values.

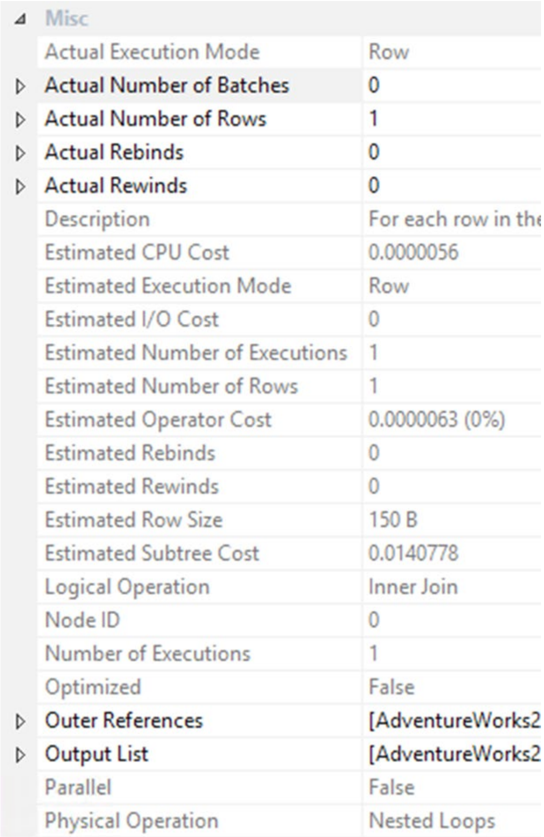


Figure 15-10: Properties for the Nested Loops operator showing runtime metrics.

Ensuring "lightweight" event sessions when capturing the plan

The most important aspect of all this is that you have an execution plan and that you captured it in an automated fashion. Just remember that capturing plans using extended events is a high-cost operation. You should only run the event session for a limited time. It should only capture exactly the data you need and no more. You'd very rarely want to run an event session that captured all three **showplan** events, as I did in Listing 15-1. Instead, just pick one; I generally use the `query_post_execution_showplan` event. Also, define filters, as I did, to control strictly the circumstances in which the event fires fully, which will limit the number of events for which the event session collects the event data.

Listing 15-3 offers a more realistic example of the sort of event session you might use for capturing specific plans, when query tuning.

```
CREATE EVENT SESSION ExecPlansAndWaits
ON SERVER
    ADD EVENT sqlos.wait_completed
    (WHERE (    (sqlserver.database_name = N'AdventureWorks2014')
               AND (sqlserver.like_i_sql_unicode_string(sqlserver.
sql_text, N'%ProductTransferByReference%')))),
    ADD EVENT sqlserver.query_post_execution_showplan
    (WHERE (    (sqlserver.database_name = N'AdventureWorks2014')
               AND (sqlserver.like_i_sql_unicode_string(sqlserver.
sql_text, N'%ProductTransferByReference%')))),
    ADD EVENT sqlserver.rpc_completed
    (WHERE (    (sqlserver.database_name = N'AdventureWorks2014')
               AND (sqlserver.like_i_sql_unicode_string(sqlserver.
sql_text, N'%ProductTransferByReference%')))),
    ADD EVENT sqlserver.rpc_starting
    (WHERE (    (sqlserver.database_name = N'AdventureWorks2014')
               AND (sqlserver.like_i_sql_unicode_string(sqlserver.
sql_text, N'%ProductTransferByReference%'))))
    ADD TARGET package0.event_file
    (SET filename = N'C:\PerfData\ExecPlansAndWaits.xel')
WITH (TRACK_CAUSALITY = ON)
GO
```

Listing 15-3

It captures the `rpc_starting` and `rpc_completed` events, which fire when a stored procedure starts and completes execution, respectively; `wait_completed`, which fires for any waits that occurred while it executed; and `query_post_execution_showplan`, to capture the plan, once the query has executed.

I've filtered these events by database and by procedure name and added causality tracking. With this, I could see when the procedure started to execute, including parameter values, each wait as it completes, and the order in which they completed, and the completion of the procedure along with the execution plan. That would be just about everything you need to troubleshoot performance on one specific query.

Start this on a production system, capture a few minutes' worth of executions, or whatever is appropriate to your system, and then turn it back off. The load will be as minimal as you can make it while still capturing useful data that will help drive your query-tuning choices.

Automating plan capture using SQL Trace

As discussed at the start of the chapter, if you are running SQL Server 2008/R2 or lower, you may have to use Trace Events instead.

We can use SQL Profiler to define a server-side trace to capture XML execution plans, as the queries are executing. We can then examine the collected plans, starting with the queries with the highest costs, and look for potential optimization possibilities, such as indexes that may enable the optimizer to perform index seek rather than scan operations for frequent queries that access large tables, or by investigating the accompanying SQL to find the cause of specific warnings in the plans, such as sorts that spill to disk.

> ***CAUTION! Never use the Profiler GUI to Capture Event Data***
>
> I'm going to show how to set up a server-side trace; never use the Profiler to capture event data directly. The Profiler GUI uses a different caching mechanism that can have a profoundly negative impact on the server that is the target of event collection. You can use the GUI to generate a trace script, but then you should run it independently as a server-side trace, saving the data to a file.

The basic principle of SQL Trace is to capture data about events as they occur within the SQL Server engine, such as the execution of T-SQL or a stored procedure. However, capturing trace events is very expensive, especially when compared to Extended Events. Many of the events have a much heavier default payload, any data that is not actually required simply being discarded. Also, the mechanisms of filtering in trace events are highly inefficient. As discussed earlier, I strongly advise against using SQL Trace events if you can instead use Extended Events.

Trace events for execution plans

There are many trace events that will capture an execution plan. The most commonly used ones are as follows:

- **Showplan XML** – the event fires with each execution of a query and captures the compile-time execution plan, in the same way as the `query_pre_execution_showplan` event in Extended Events. This is probably the preferable event if you need to minimize impact on the system. The others should be avoided because of the load they place on the system or because they don't return data that is usable for our purposes.

- **Showplan XML for Query Compile** – like **Showplan XML** above, but it only fires on a compilation of a query, like the `query_post_compilation_showplan` event in Extended Events.
- **Performance Statistics** – can be used to trace when execution plans are added to or removed from cache.
- **Showplan XML Statistics Profile** – this event will generate the actual execution plan for each query, after it has executed. While this is the one you'll probably want to use the most, it's also the most expensive one to capture.

You must be extremely cautious when running traces that capture any of these events on a production machine, as it can cause a significant performance hit. SQL Trace's filtering mechanism is far less efficient than for Extended Events. Even if we filter on database and SQL text, as we did earlier for our events sessions, SQL trace still fires the event fully for every database and for any SQL text, and only applies the filter at the point the individual trace consumes the event. Aside from collecting the execution plans, these events will also collect several global fields by default, whether you want them or not.

Run traces for as short a time as possible. If you can, you absolutely should replace SQL Trace with Extended Events.

Creating a Showplan XML trace using Profiler

The SQL Server Profiler **Showplan XML** event captures the XML execution plan created by the query optimizer and so doesn't include runtime metrics. To capture a basic Profiler trace, showing estimated execution plans, start Profiler from the **Tools** menu in SSMS, create a new trace and connect to your SQL Server instance. By default, only a person logged in as `sa`, or a member of the `SYSADMIN` group can create and run a Profiler trace. For other users to create a trace, they must be granted the `ALTER TRACE` permission.

On the **General** tab, change the template to **blank**, give the trace a name and then switch to the **Events Selection** tab and make sure that the **Show all events** and **Show All columns** checkboxes are selected. The **Showplan XML** event is located within the **Performance** section, so click on the plus (+) sign to expand that selection. Click on the checkbox for the **Showplan XML** event.

While you can capture the **Showplan XML** event by itself in Profiler, it is generally more useful if, as I did with the extended events session, you capture it along with some other basic events, such as `RPC:Completed` (in **Stored Procedures** event class) and `SQL:BatchCompleted` (**TSQL** event class).

These extra events provide additional information to help put the XML plan into context. For example, we can see which parameters were passed to a stored procedure in which we are interested.

I won't go into the details of which data fields to choose for each event but, if you're running the trace in a shared environment, you may want to add the `database_name` field and then filter on it (using **Column Filters…**) so you see only the events in which you're interested.

Deselect "**Show All Events**" and "**Show All Columns**" once you're done. The event selection screen should look like Figure 15-11.

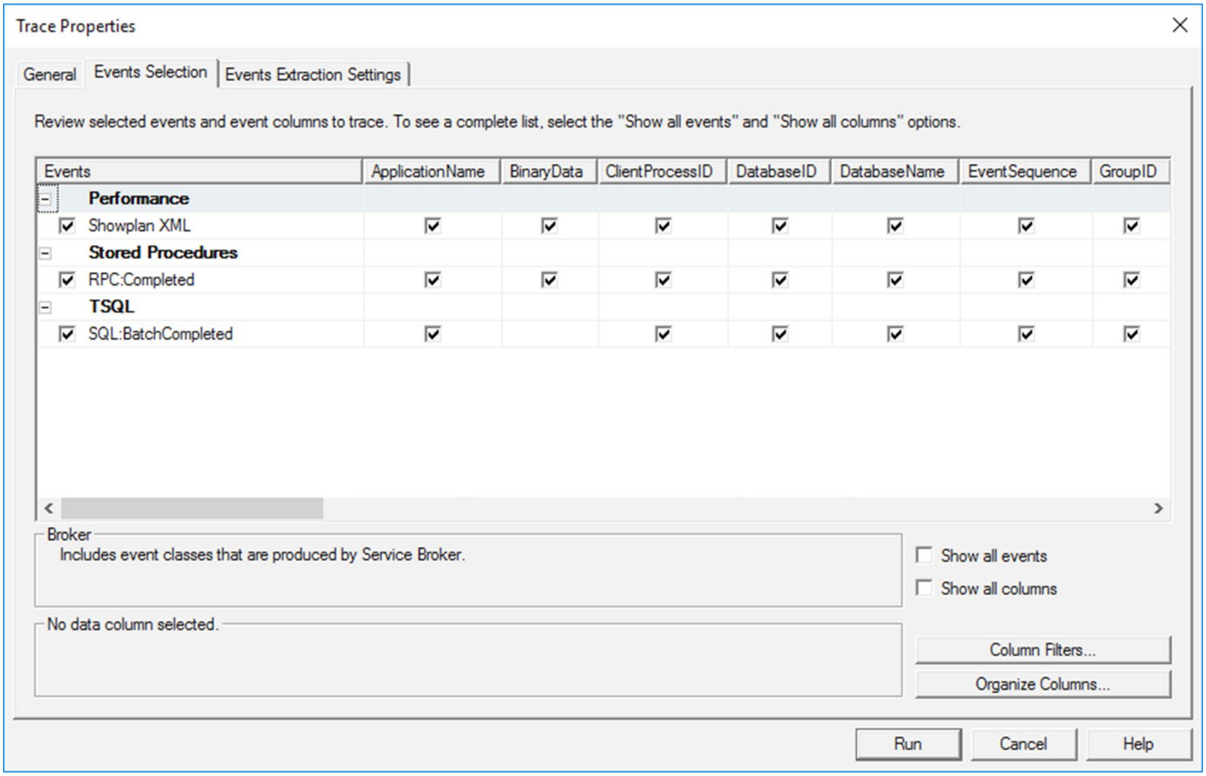


Figure 15-11: Trace defined within Profiler.

With **Showplan XML** or any of the other XML events selected, a third tab appears, called **Events Extraction Settings**. On this tab, we can choose to output, to a separate file for later use, a copy of the XML as it's captured. Not only can we define the file, we can also determine whether all the XML will go into a single file or a series of files, unique to each execution plan.

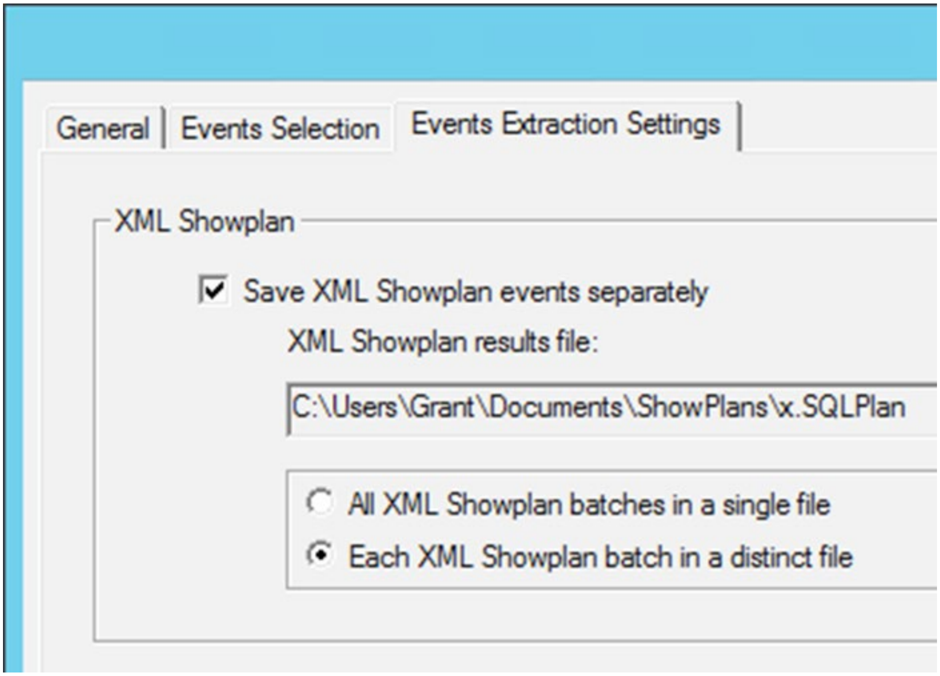


Figure 15-12: Setting up the execution plan extraction.

For test purposes only, to prove the trace works correctly, and never on a production system, click on the **Run** button to start the trace. Rerun the code from Listing 15-2 and you should see the events captured, as shown in Figure 15-13.

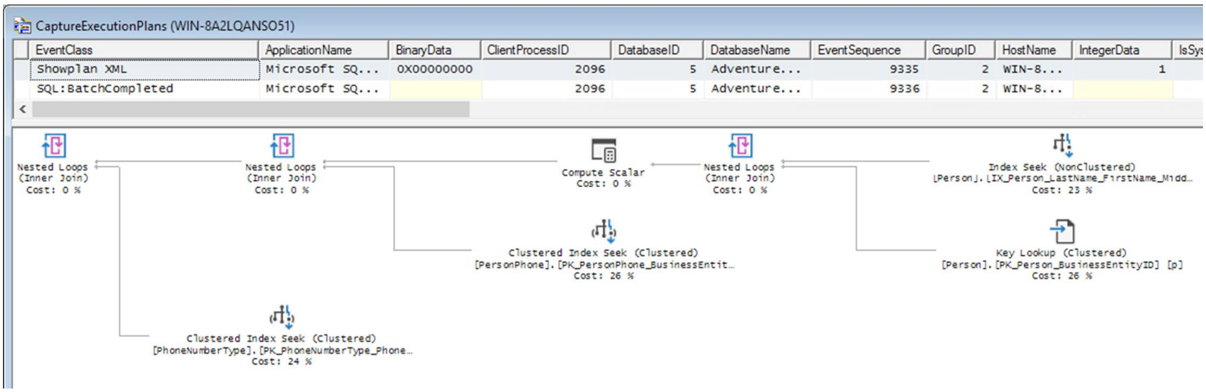


Figure 15-13: Output from Trace Event with an execution plan on display.

Stop the trace running. In the collected event data, I have clicked on the **Showplan XML** event. In the lower pane, you can see the graphical execution plan. Note that the captured plan again does not have the **SELECT** operator.

You cannot access the operator properties from this window; you'll need to browse the plan's XML, available under the **TextData** column, or export it to a file by right-clicking on the row and selecting **Extract Event Data**. However, in this case, we already have the plans in files because of the **Events Extraction Settings**, shown in Figure 15-12.

Creating a server-side trace

As noted earlier, if we are using SQL Trace, we want to run server-side traces, saving the results to a file. One quick way to script out a trace file definition is to start and immediately stop the trace running, in Profiler, and then click on **File** | **Export** | **Script Trace Definition** | **For SQL Servers 2005–2017….**

Listing 15-4 shows a truncated extract of the saved trace file.

```
EXEC @rc = sp_trace_create @TraceID OUTPUT,
                           0,
                           N'InsertFileNameHere',
                           @maxfilesize,
                           NULL;
IF (@rc != 0)
    GOTO error;
-- Client side File and Table cannot be scripted
-- Set the events
DECLARE @on BIT;
SET @on = 1;
EXEC sp_trace_setevent @TraceID, 122, 1, @on;
EXEC sp_trace_setevent @TraceID, 122, 9, @on;
EXEC sp_trace_setevent @TraceID, 122, 2, @on;
…
-- Set the Filters
DECLARE @intfilter INT;
DECLARE @bigintfilter BIGINT;
-- Set the trace status to start
EXEC sp_trace_setstatus @TraceID, 1;
```

```
-- display trace id for future references
SELECT @TraceID AS TraceID;
GOTO finish;
error:
SELECT @rc AS ErrorCode;
finish:
GO
```

Listing 15-4

Yes, this lengthy script is roughly equivalent to that in Listings 15-1 or 15-3, just much less clear and much longer-winded. Follow the instructions in the comments to use this on your own servers.

Summary

Automating plan capture will allow you to target queries or plans that you might not be able to get through more traditional means. This will come in extremely handy when you want the execution plan and a correlated number of other events, such as wait statistics or recompile events. Try not to use trace events for doing this, because they place a very high load on the system. Instead, where possible, use Extended Events. Just remember that Extended Events, though very low cost in terms of their overhead on the system, especially compared to Trace Events, are not free, so you should carefully filter the events captured.

Chapter 16: The Query Store

Introduced to Azure SQL Database in 2015, and to the boxed version with SQL Server 2016, the Query Store is a new mechanism for monitoring query performance metrics at the database level. In addition to capturing query performance, the Query Store also retains execution plans, including multiple versions of plans for a given query if the statistics or settings for that query can result in different execution plans. This chapter will cover the Query Store as it relates directly to execution plans and execution plan control; it is not a thorough documentation on all the behavior surrounding the Query Store.

Behavior of the Query Store

The aim of the Query Store is to capture the information without interfering with normal operations of your database and server. With this intent, then, the information that the Query Store captures is initially written in an asynchronous fashion to memory. The Query Store then has a secondary process that will flush the information from memory to disk, again asynchronously. The Query Store does not directly interfere with the query optimization process. Instead, once an execution plan has been generated by the optimization process, the Query Store will capture that plan at the same time as it gets written to cache.

Some plans are not written to cache. For example, an ad hoc query with a `RECOMPILE` hint will generate a plan, but that plan is not stored in cache. However, all plans, by default, are captured by the Query Store at the time they would have been written to cache.

After a query executes, another asynchronous process captures runtime information about the behavior of that query, how long it ran, how much memory it used, etc., and stores aggregated data about the query behavior, first to memory, then flushed to disk in an asynchronous process, just like the plans.

All this information is stored within system tables for each database on which you enable the Query Store. By default, the Query Store is not enabled in SQL Server 2016, but it is enabled by default in Azure SQL Database. You can control whether the Query Store is enabled or disabled, but you have no ability to change where the information it gathered is placed, because it is within system tables, so it will always be in the Primary file group.

The organizing principle of the Query Store is the query. Not stored procedures and not batches, but individual queries. For each query, one or more execution plans will also be stored. There are several options regarding the behavior of the Query Store and the queries it captures, length of retention, etc. None of that is directly applicable to the behavior of the execution plans within the Query Store, so I won't be addressing them here.

The information about execution plans is stored in one table within the Query Store as shown in Figure 16-1.

sys.query_store_plan

Script | Summary | Copy

Column Name	Data Type	Nullability
plan_id	bigint	not null
query_id	bigint	not null
plan_group_id	bigint	null
engine_version	nvarchar(32)	null
compatibility_level	smallint	not null
query_plan_hash	binary(8)	not null
query_plan	nvarchar(max)	null
is_online_index_plan	bit	not null
is_trivial_plan	bit	not null
is_parallel_plan	bit	not null
is_forced_plan	bit	not null
is_natively_compiled	bit	not null
force_failure_count	bigint	not null
last_force_failure_reason	int	not null
last_force_failure_reason_desc	nvarchar(128)	null
count_compiles	bigint	null
initial_compile_start_time	datetimeoffset(7)	not null
last_compile_start_time	datetimeoffset(7)	null
last_execution_time	datetimeoffset(7)	null
avg_compile_duration	float	null
last_compile_duration	bigint	null
plan_forcing_type	int	not null
plan_forcing_type_desc	nvarchar(60)	null

Figure 16-1: Execution plans within the Query Store.

The plan itself is stored in the `query_plan` column as an `NVARCHAR(MAX)` data type. Additionally, there is a large amount of metadata about the plan stored as various other columns within the catalog view. The data is stored as text, `NVARCHAR`, even though it is an XML execution plan, because there is a limit on the nesting levels of XML within SQL Server. Storing the plan as text avoids that issue. If you want to retrieve the plan from the catalog view and view it graphically, you must either `CAST` as XML (assuming it will be below the XML nesting-depth limit), or export to a `.showplan` file.

Since there are a few options that affect plan retention and capture within the Query Store, I want to talk about those, so that you can be sure you capture, or don't capture, the correct plans for your queries.

Query Store Options

By default, you can capture up to 200 different plans for each query. That should be enough for almost any query I've heard of. It is possible, although I have yet to see it, that this value could be too high for a system and you may want to adjust it down. It's also possible for a given system that this value is too low and may need to go up. The method for adjusting Query Store settings is to use the `ALTER DATABASE` command as shown in Listing 16-1.

```
ALTER DATABASE AdventureWorks2014 SET QUERY_STORE (MAX_PLANS_PER_
QUERY = 20);
```

Listing 16-1

In that example I change the plans for each query from the default of 200 down to 20. Let me repeat, I'm not recommending this change. It's just an example. The default values should work fine in most cases. There are a few defaults that you may want to consider adjusting.

The first Query Store option that is going to be significant for execution plans and plan capture is the `Query Capture Mode`. By default, this is set to `ALL` in SQL Server 2016–2017 and `AUTO` in Azure SQL Database. There are three settings:

ALL	Captures all plans for all queries on the database for which you have enabled Query Store.
AUTO	Captures plans based on two criteria. Either queries with a significant compile time and execution duration, in tests, greater than one second execution time, but this is controlled by Microsoft. Alternatively, a query must be called at least three times before the plan will be captured.
NONE	Leaves Query Store enabled on the database, but stops capturing information on new queries, while continuing to capture runtime metrics on existing queries.

If you have a database where you have enabled **Optimize For Ad Hoc Workloads**, a setting that ensures a query must be executed twice before the plan is loaded into cache, it might be a good idea to change your capture mode to `AUTO`. This will help to reduce wasted space in the Query Store data set. To make this change, you use the `ALTER DATABASE` command again.

```
ALTER DATABASE AdventureWorks2014 SET QUERY_STORE (QUERY_CAPTURE_
MODE = AUTO);
```

Listing 16-2

Having the Query Store set to `NONE` means that no additional plans for any query will be captured (as noted above). However, it will continue to capture the execution runtime metrics for the plans and queries that it has already captured. This may be useful under some circumstances where you only care about a limited set of queries.

Another setting that you may want to control is the automatic clean-up of the information in the Query Store. By default, it keeps 367 days' worth of data, leap year plus one day. This may be too much, or not enough. You can adjust it using the same functions as above. By default, Query Store will also clean up the data once this limit is reached. You may want to turn this off, depending on your circumstances.

In addition to using T-SQL to control the Query Store, you can use the Management Studio GUI. I prefer T-SQL because it allows for automation of the processing. To get to the GUI settings, right-click on a database and select **Properties** from the context menu. There will be a new page listed, Query Store, and it contains the basic information about the Query Store on the database in question, as shown in Figure 16-2.

You can't control all the settings from this GUI, so you will need to use the `ALTER DATABASE` command for some settings. For example, the maximum number of plans per query which we demonstrated in Listing 16-1 can't be adjusted from the GUI. The GUI report on disk usage is handy, but if you really need to monitor it, you'll, again, want to set up queries to retrieve that information.

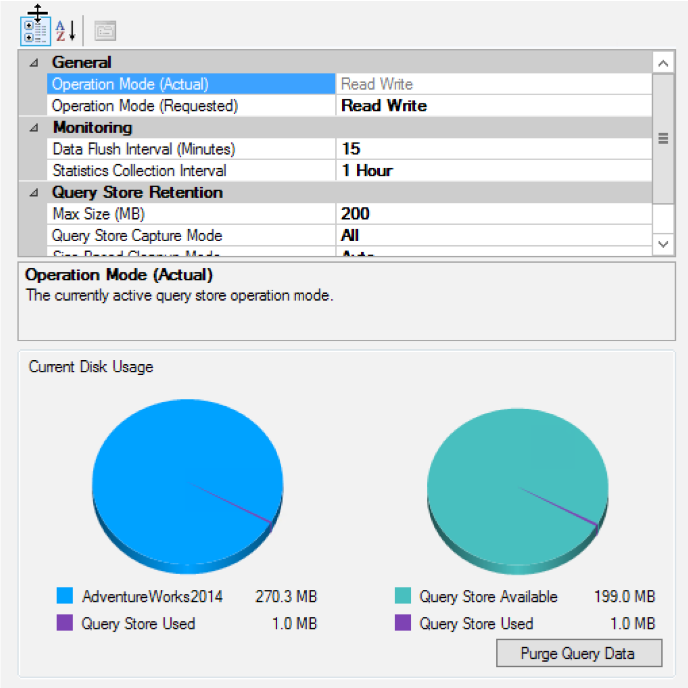


Figure 16-2: SSMS GUI for managing the Query Store.

Retrieving Plans from the Query Store

Retrieving execution plans is straightforward. There are canned reports built in to Management Studio and available within the database. You can also use T-SQL to retrieve the execution plans from the catalog views exposed for the Query Store information. We'll start off with the basic view of a report from the Query Store and then we'll focus on using the catalog views to retrieve execution plans using T-SQL.

SSMS reports

SSMS provides several built-in reports, a couple of which can help you find problem queries and their plans. I can't cover these reports in any detail, but I'll describe the basics of what they offer, and then focus on using one of the reports available for the Query Store.

Overview of Query Store reports

If you expand your database within Object Explorer, you'll see a folder marked **Query Store**. Expand that folder, and you should see the reports shown in Figure 16-3.

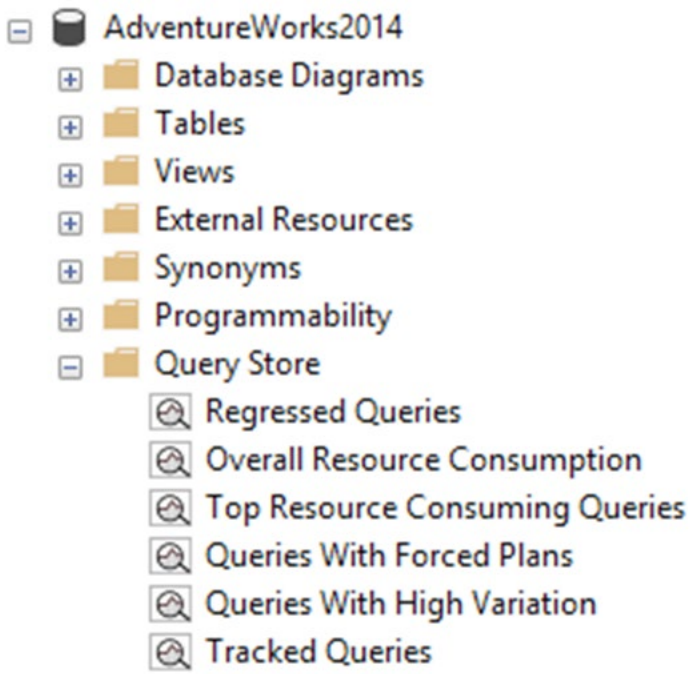


Figure 16-3: Query Store reports within SSMS.

Each of these reports brings back different information based on their structure. Most of the reports have a very similar layout. The exception is the **Overall Resource Consumption** report, which shows a very different set of data from the others. Opening that report shows queries sorted by resource consumption over time, based on the execution runtime data within the Query Store.

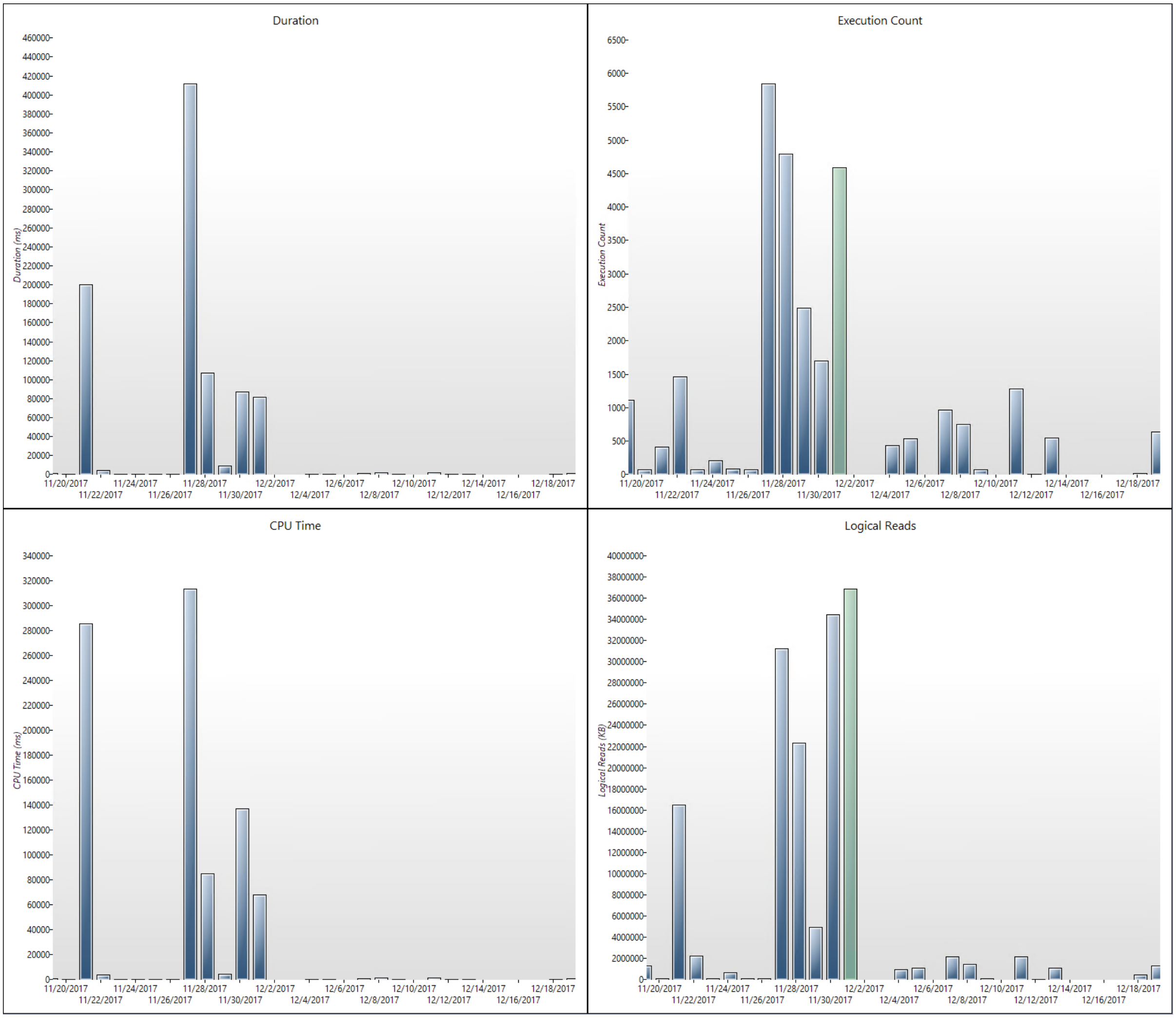


Figure 16-4: Overall Resource Consumption report from the Query Store.

This report is useful for identifying queries that are using more resources. Clicking on any one query opens the **Top Resource Consuming Queries** window, which we're going to go over in detail below.

The other reports are structured like the **Top Resource Consuming Queries** report, so we won't go through all their functions. However, let's outline where each report can be used.

Report	Usefulness
Regressed Queries	When the runtime behavior of a query changes at the same time as the execution plan changes, the query can be said to have regressed due to the execution plan change. This may come from bad parameter sniffing, a change in the optimization process, or others. This report will help you identify queries to focus on.
Overall Resource Consumption	Displayed above in Figure 16-4, this report breaks down queries by the resources they consume over time. It's useful when working on identifying which query is causing a particular problem with memory, I/O or CPU.
Top Resource Consuming Queries	This will be covered in detail below. It's simply a focused version of the Overall Resource Consumption report with a single metric being displayed.
Queries With Forced Plans	When you choose to force a plan, detailed below, this report will show which queries currently have plans that are being forced.
Queries With High Variation	These are queries that, based on a given metric, are experiencing more changes in behavior than other plans. This could be used in conjunction with the Regressed Queries report.
Tracked Queries	You can mark a query for tracking through the Query Store. The tracked queries will then be exposed in this report.

Each report displays unique sets of data based on the information captured by the Query Store but, except for the **Overall Resource Consumption** report, they all behave in roughly the same fashion.

The Top Resource Consuming Queries report

We'll focus on the **Top Resource Consuming Queries** report because it's one that is likely to be used regularly on most systems. If you've just enabled the Query Store, then you should run a few queries, to see some data in the report. Double-clicking on the report will open it up.

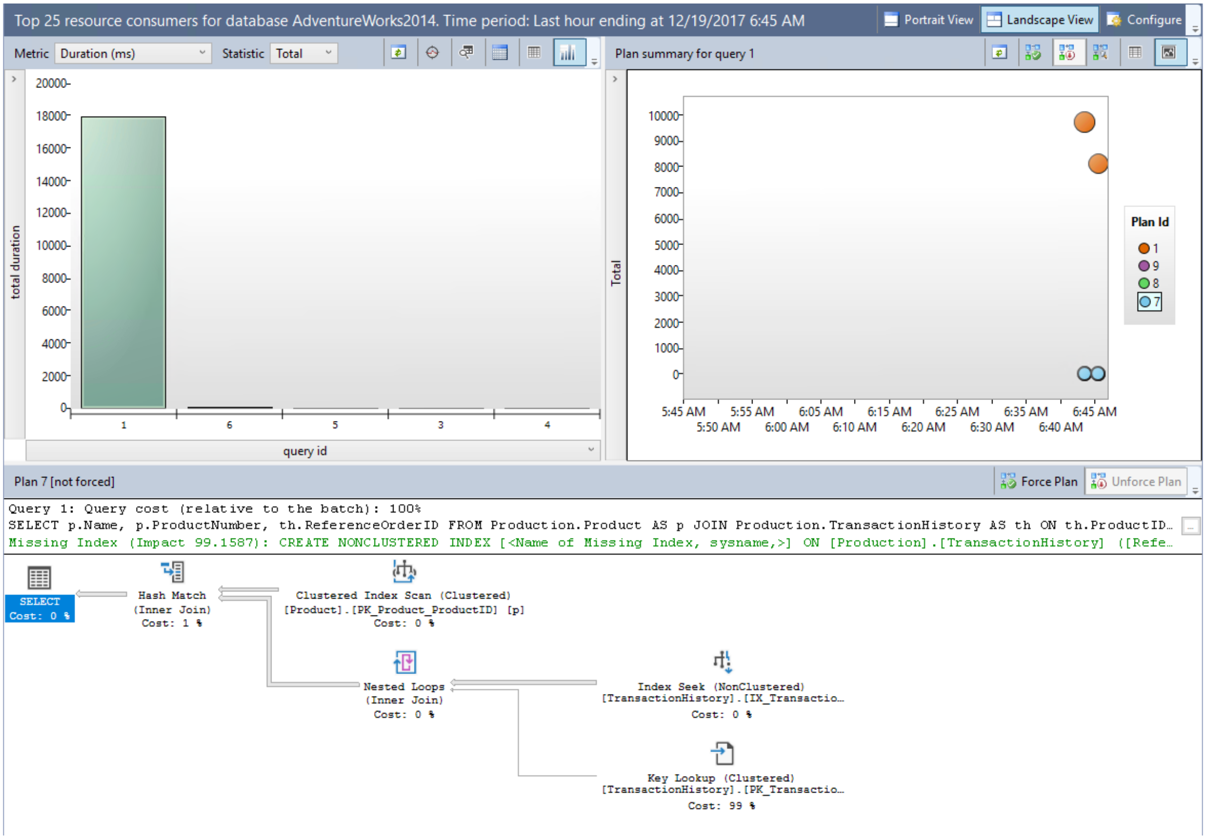


Figure 16-5: Top Resource Consuming Queries report for the Query Store.

The report is divided into three sections. On the top left is a listing of queries sorted by various metrics. The default is **Duration**. You can use the drop-down to choose amongst CPU and other measures provided by the Query Store. You can also choose the **Statistic** to measure. The default here is **Total**. These will populate the graph, showing you the queries that are most problematic, when considering **Total Duration**. To the right is a second section showing various execution times. Each circle represents, not an individual execution, but aggregated execution times over a time interval. There may be more than one plan. Selecting any of those plans changes the third pane of the report, on the bottom, to a graphical representation of the execution plan in question. That graphical plan functions exactly as any other graphical plan we've worked with throughout the book.

In short, this report ties together the query, an aggregation of its performance metrics, and the execution plan associated with those metrics. You can adjust the reports and modify them from being graphical to showing grids of data. Simply click the buttons on the upper-right of the first window of the report.

One additional piece of functionality is especially interesting from an Execution Plan standpoint. When you have more than one plan available, as in the example in Figure 16-5, you can select two of those plans, using the **SHIFT** key to select a second plan. With two plans selected, one of the buttons in the tool bar, shown in Figure 16-6, allows you to compare the plans.

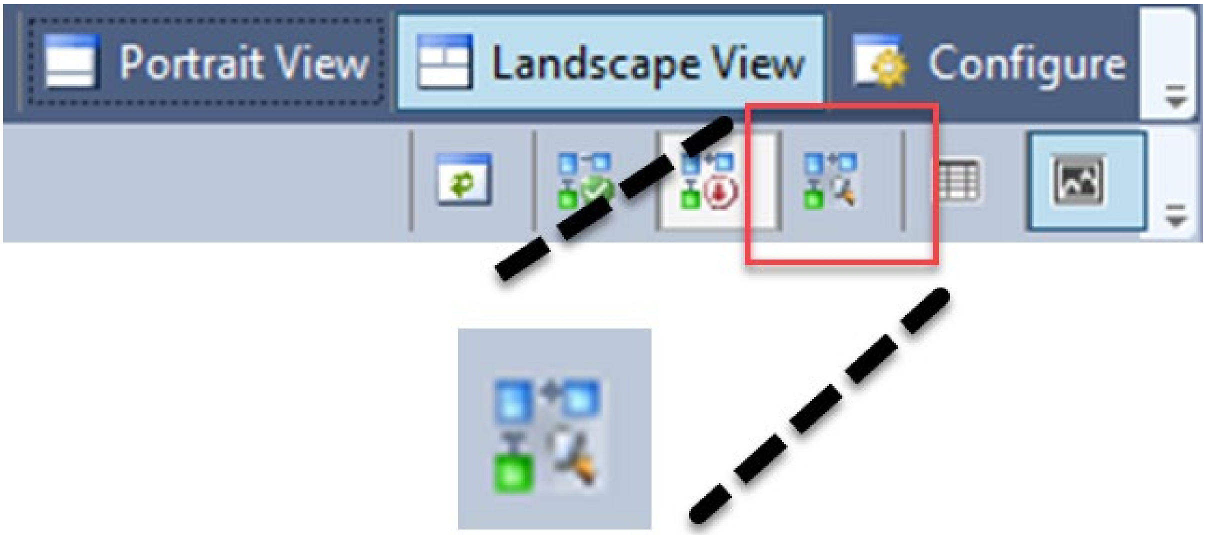


Figure 16-6: Compare Execution Plans button.

Clicking that button, opens the **Compare Execution Plan** window (covered in more detail in Chapter 17). You can see the two plans from the above example in Figure 16-7.

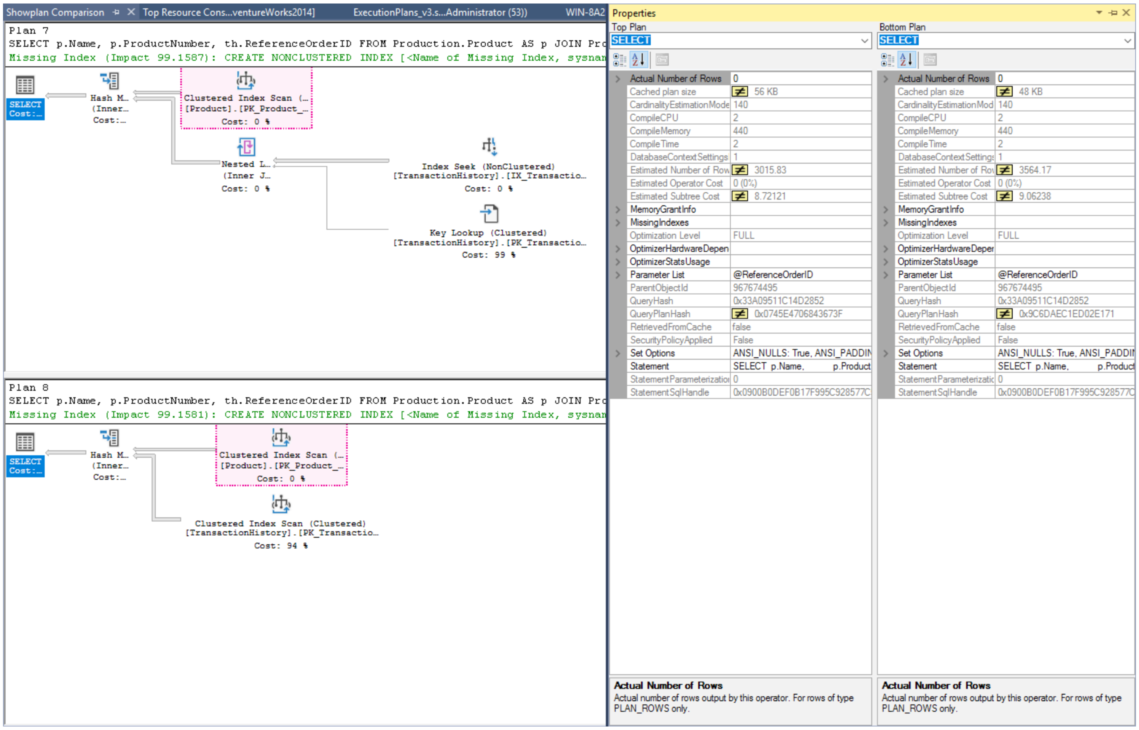


Figure 16-7: Execution plans compared from Query Store report.

The functionality is described in detail in Chapter 17. Common parts of the plan are highlighted in varying shades of color (in this case pink). Differences in the properties are displayed using the "not equals" symbol. You can explore and expose information about the differences and similarities between the plans.

Other than that, there's only one other piece of functionality directly applicable to execution plans and we'll cover it a little later in this chapter.

Retrieve Query Store plans using T-SQL

Getting information about the query plan from the Query Store system tables is quite straightforward. There are only a few catalog views (how you read a system table) providing the information, that are directly applicable to plans themselves:

- **`query_store_plan`** – the view that contains the execution plan itself along with information about the plan such as the `query_plan_hash`, compatibility level, and whether a plan is trivial (all as shown in Figure 16-1).

- **query_store_query** – the view that identifies each query, but not the query text, which is stored separately, and includes information such as the last compile time, the type of parameterization, the query hash, and more. Although the text and context are stored separately, they are how a query is identified.
- **query_context_settings** – this defines metadata about the query such as ANSI settings, whether a query is for replication, and its language.
- **query_store_query_text** – this view defines the actual text of the query.

While there are three other Query Store catalog views, they are very focused on query performance so I won't be directly addressing them in this book.

Querying to retrieve the plan is basically a matter of joining together the appropriate catalog views to retrieve the information you are most interested in. You can simply query the `sys.query_store_plan` table, but you won't have any context for that plan such as the text of the query or the stored procedure that it comes from. Listing 16-3 demonstrates a good use of the tables to retrieve an execution plan.

```
SELECT qsq.query_id,
       qsqt.query_sql_text,
       CAST(qsp.query_plan AS XML),
       qcs.set_options
FROM sys.query_store_query AS qsq
    JOIN sys.query_store_query_text AS qsqt
       ON qsqt.query_text_id = qsq.query_text_id
    JOIN sys.query_store_plan AS qsp
       ON qsp.query_id = qsq.query_id
    JOIN sys.query_context_settings AS qcs
         ON qcs.context_settings_id = qsq.context_settings_id
WHERE qsq.object_id = OBJECT_ID('dbo.AddressByCity');
```

Listing 16-3

Assuming you have at least once executed a stored procedure named `dbo.AddressByCity`, you'll get information back out. I've included the `query_context_settings` under the assumption that if a query is executed using different settings, you may see it more than one time. To make the results contain a clickable execution plan, I've opted to `CAST` the plan as XML. The results of this query would look like Figure 16-8.

	query_id	query_sql_text	(No column name)	set_options
1	19	(@City nvarchar(30))SELECT a.AddressID, ...	<ShowPlanXML xmlns="http://schemas.microsoft.com...	0x000010FB

Figure 16-8: Results from query against Query Store system tables.

This query returns the execution plan as a clickable column and shows the `query_id`. Retrieving additional information about the plan is just a question of adding columns to this query. One point worth noting is the text of the query as shown here. Listing 16-4 shows the full text from that column.

```
(@City nvarchar(30))SELECT a.AddressID  ,a.AddressLine1
,a.AddressLine2  ,a.City  ,sp.[Name] AS StateProvinceName
,a.PostalCode  FROM Person.Address AS a  JOIN Person.StateProvince
AS sp  ON a.StateProvinceID = sp.StateProvinceID  WHERE a.City = @
City
```

Listing 16-4

This is a query that contains a parameter as defined by the stored procedure that the query comes from:

```
CREATE OR ALTER PROC dbo.AddressByCity @City NVARCHAR(30)
AS
SELECT a.AddressID,
       a.AddressLine1,
       a.AddressLine2,
       a.City,
       sp.Name AS StateProvinceName,
       a.PostalCode
FROM Person.Address AS a
    JOIN Person.StateProvince AS sp
        ON a.StateProvinceID = sp.StateProvinceID
WHERE a.City = @City;
```

Listing 16-5

Note the change in the text of the query. In the Query Store, the definition of the parameter, `@City`, is included with the query text at the front of the statement, `(@City nvarchar(30))`. That same text is not included with the text of the query from the stored procedure as shown in Listing 16-5. This vagary in how Query Store works can make it difficult to track down individual queries within the catalog views.

There is a function, sys.fn_stmt_sql_handle_from_sql_stmt, that will help you resolve a simple, or forced parameterized query from the Query Store. This function doesn't work with stored procedures, though. There, you would be forced to use the LIKE operator to retrieve the information. You can use the object_id, but you'll have to deal with however many statements are contained within the procedure. To find individual statements, you'll be forced to use the functions listed below.

Let's look at an example of this in action, taking a very simple query like Listing 16-6.

```
SELECT bom.BillOfMaterialsID,
       bom.StartDate,
       bom.EndDate
FROM Production.BillOfMaterials AS bom
WHERE bom.BillOfMaterialsID = 2363;
```

Listing 16-6

The query in Listing 16-6 will result in a query plan that uses simple parameterization to ensure the potential of plan reuse. This means that the value, 2363, is replaced by a parameter, @1, within the plan stored in cache. If we ran a query like Listing 16-7, we wouldn't see any data.

```
SELECT qsqt.query_text_id
FROM sys.query_store_query_text AS qsqt
WHERE qsqt.query_sql_text = 'SELECT bom.BillOfMaterialsID,
       bom.StartDate,
       bom.EndDate
FROM Production.BillOfMaterials AS bom
WHERE bom.BillOfMaterialsID = 2363;';''
```

Listing 16-7

The results are a complete empty set because the Query Store doesn't have the original T-SQL we passed in. Instead, it has the new text that defines the parameter. This is where the sys.fn_stmt_sql_handle_from_sql_stmt function comes into play. We'll modify our query against the Query Store catalog views, to filter for the query in question.

```
SELECT qsqt.query_text_id
FROM sys.query_store_query_text AS qsqt
    JOIN sys.query_store_query AS qsq
        ON qsq.query_text_id = qsqt.query_text_id
    CROSS APPLY sys.fn_stmt_sql_handle_from_sql_stmt(
                    'SELECT bom.BillOfMaterialsID,
       bom.StartDate,
       bom.EndDate
FROM Production.BillOfMaterials AS bom
WHERE bom.BillOfMaterialsID = 2363;',
                    qsq.query_parameterization_type) AS fsshfss
WHERE fsshfss.statement_sql_handle = qsqt.statement_sql_handle;''
```

Listing 16-8

To work with `sys.fn_stmt_sql_handle_from_sql_stmt` you must supply two values. The first is the query in which you are interested. In our case that's the query from Listing 16-6. The second contains the type of parameterization. Luckily, this information is stored directly in the `sys.query_store_query` table, so we can go there to retrieve it. With these values supplied, we'll get the query we need in the result set.

Control Plans Using Plan Forcing

One of the most important aspects of Query Store, regarding execution plans, is the ability to pick an execution plan for a given query, and then use plan forcing in Query Store to force the optimizer to use this plan. It is much easier to use plan forcing within Query Store than it is to implement a plan guide (see Chapter 9). If you have an existing plan guide for a query, and then also force a plan, perhaps a different plan, using Query Store, then the Query Store plan forcing will take precedence. If you are in Azure SQL Database or using SQL Server 2016 or greater, and you need to force the optimizer to use an execution plan, the preferred method is to use plan forcing through the Query Store rather than plan guides.

Query Store is designed to collect data using an asynchronous process. Plan forcing is the one exception to that process. In this one case, when you define a plan as a forced plan, regardless of what happens with the plan in cache, compiles or recompiles, reboots of the server, even backup and restore of the database, that plan will be forced. To force a plan, the plan must be valid for the query and structure as currently defined; changes in indexing, for example, could mean that a plan is no longer valid for a query.

The information that a plan is forced is written into the system tables of the Query Store and stored with the database. With Query Store enabled, and if the plan is a valid plan, if it's forced, that's the execution plan that will be used. There is a relatively obscure situation where a "morally-equivalent" plan, a plan that is identical in all the core essentials, but not necessarily perfectly identical, can be used instead of the precise plan you define. However, this isn't common.

Plan forcing is a double-edged sword that can help or hurt depending on how it is implemented and maintained. I recommend extremely judicious use of plan forcing and I advise you to figure out a schedule for reviewing plans that have been forced. This is not something you set once and forget about.

That said, there are several situations where you may consider using plan forcing, one of which is the classic "parameter sniffing gone wrong" situation, which we've encountered several times previously in the book. However, another good use case is to fix "plan regression" problems, where some system change means that the optimizer generates a new plan, which does not perform as well as the old plan. Plan regression can occur after, for example, upgrading from a version of SQL Server prior to 2014 which used the old cardinality estimation engine, or applying Cumulative Updates or hot fixes that introduce changes to the query optimizer. There is a specific report available for regressed queries. During upgrades or while applying a CU, it's a very good idea to run Query Store prior to changing the compatibility level during an upgrade, or applying the CU in that situation.

How to force a plan

I'll demonstrate the basics of how to force a plan, using the "bad parameter sniffing" case as an example.

Execute the stored procedure `dbo.AddressByCity`, passing it a value of `'London'`.

```
EXEC dbo.AddressByCity @City = N'London';
```

Listing 16-9

Let's look at the execution plan.

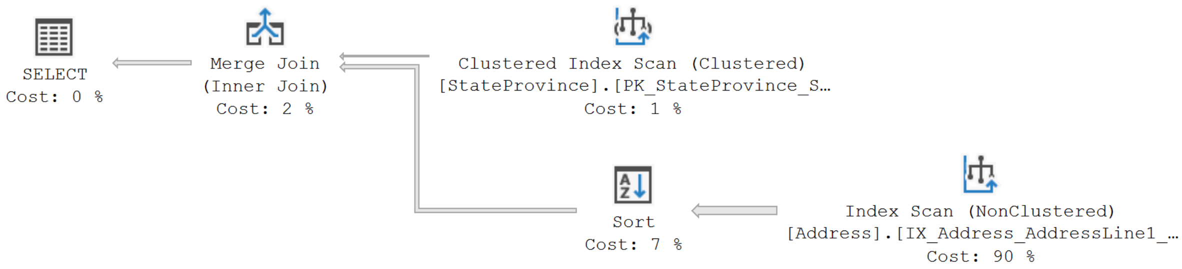


Figure 16-9: First execution plan from stored procedure.

Next, we should ensure that the execution plan for the `dbo.AddressByCity` stored procedure is removed from cache.

```
DECLARE @PlanHandle VARBINARY(64);
SELECT @PlanHandle = deqs.plan_handle
FROM sys.dm_exec_query_stats AS deqs
    CROSS APPLY sys.dm_exec_sql_text(deqs.sql_handle) AS dest
WHERE dest.objectid = OBJECT_ID('dbo.AddressByCity');
IF @PlanHandle IS NOT NULL
BEGIN;
    DBCC FREEPROCCACHE(@PlanHandle);
END;
GO
```

Listing 16-10

If we then execute the query again, but this time pass in the value of `'Mentor'`, we'll see a completely different execution plan.

```
EXEC dbo.AddressByCity @City = N'Mentor';
```

Listing 16-11

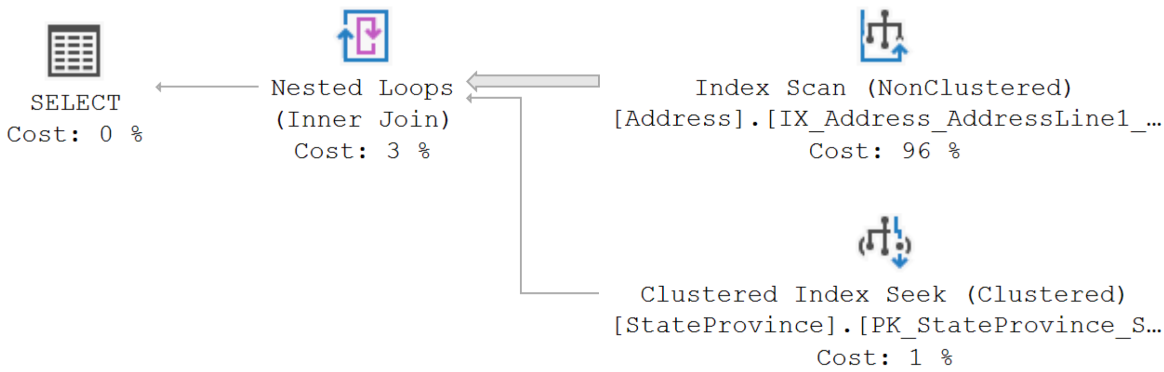


Figure 16-10: Second execution plan from stored procedure.

This is a classic case of parameter sniffing gone wrong. Each plan works very well for the estimated row counts, which are larger for `'London'` and smaller for `'Mentor'`, but problems arise when a query that returns many rows uses the plan that's optimized for returning smaller data sets. In some circumstances, this type of behavior leads to performance problems. Back in Chapter 10, we tackled this exact same problem by applying the `OPTIMIZE FOR` query hint.

Let's say that one of these plans leads to more consistent, predictable performance over a range of parameter values, than the other. We'd like to use the Query Store to force the optimizer to always use that plan.

The T-SQL to force a plan requires that we first get the `query_id` and the `plan_id`. This means we have to track down that information from the Query Store tables.

```
SELECT qsq.query_id,
       qsp.plan_id,
       CAST(qsp.query_plan AS XML)
FROM sys.query_store_query AS qsq
    JOIN sys.query_store_plan AS qsp
        ON qsp.query_id = qsq.query_id
WHERE qsq.object_id = OBJECT_ID('dbo.AddressByCity');
```

Listing 16-12

This will return the information we need along with the execution plan so that we can determine which plan we want. Look at the plans to determine the one you wish to force. Implementing the plan forcing is then extremely simple.

```
EXEC sys.sp_query_store_force_plan 214, 248;
```

Listing 16-13

Now, if I were to remove this plan from cache, using Listing 16-9 again, regardless of the value passed to the `dbo.AddressByCity` stored procedure, the plan generated will always be the plan I chose. The information within the plan and the behavior of the plan will be the same as any other execution plan within the system with a couple of exceptions. First, the plan defined will always be the plan returned (except when it is a morally equivalent plan or an invalid plan) until we stop forcing the plan or disable the Query Store. Second, one marker has been added to the execution plan properties so that we can see that it is a forced plan.

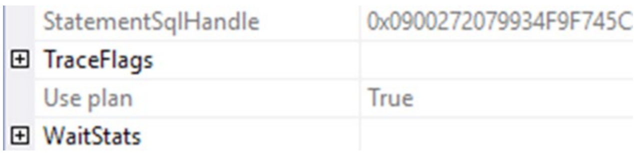


Figure 16-11: Use plan properties from SELECT operator.

In the first operator, in this case the **SELECT** operator, a new property will be added to any plans that are forced, **Use plan**. If that value is set to **True**, then that plan is a forced execution plan.

You can retrieve information about plans that are forced by querying the Query Store directly.

```
SELECT qsq.query_id,
       qsp.plan_id,
       CAST(qsp.query_plan AS XML)
FROM sys.query_store_query AS qsq
    JOIN sys.query_store_plan AS qsp
        ON qsp.query_id = qsq.query_id
WHERE qsp.is_forced_plan = 1;
```

Listing 16-14

With this information you can, if you choose, unforce a plan using another command.

```
EXEC sys.sp_query_store_unforce_plan 214, 248;
```

Listing 16-15

This will stop forcing the execution plan from the Query Store and all other behavior will return to normal.

You can also use the GUI to force and unforce plans. If you look at the report from Figure 16-4, shown again in Figure 16-11, you can see, on the right-hand side, two buttons, **Force Plan** and **Unforce Plan**.

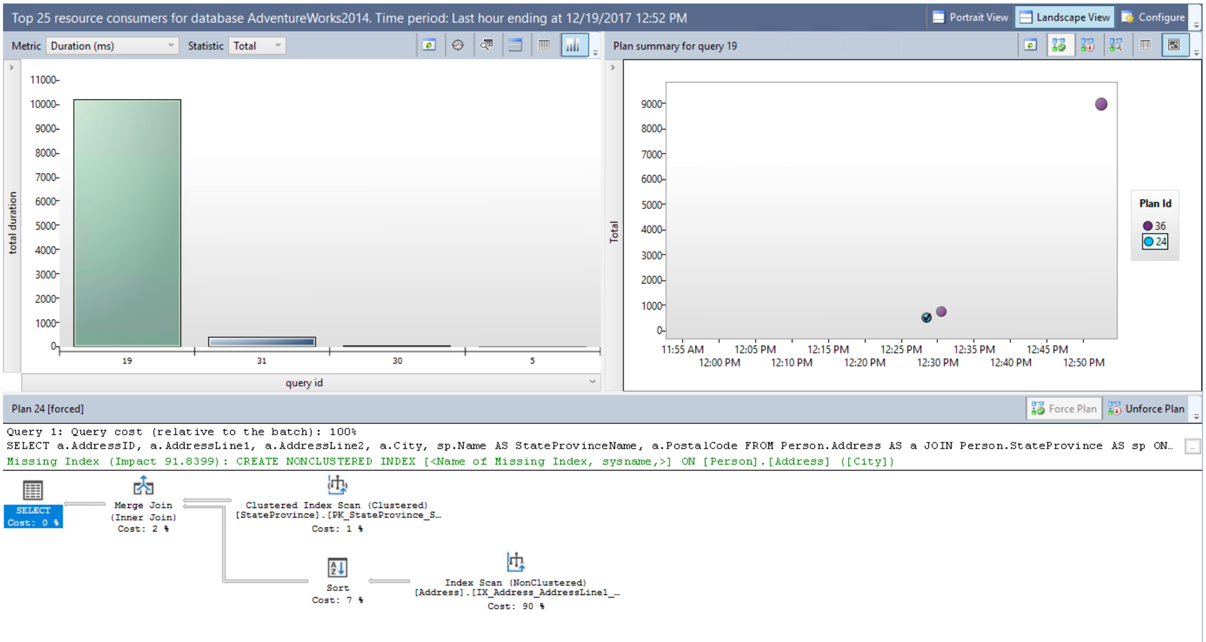


Figure 16-12: Forced plan in Query Store reports.

You can click on a plan in the upper-right pane, then select **Force Plan** to force the plan the same as if you used T-SQL to do it. Unforcing the plan is just as straightforward. If a plan is forced, you can see a check mark on it in the plan's listing to the right and anywhere that plan is visible. Choosing to force or unforce a plan from the report, you will be prompted to check whether you're sure.

Just remember that forcing a plan can be a good choice for dealing with plan regressions. However, that choice should be reviewed regularly to see if the situation has changed in some way that suggests removing the forced plan is a preferred choice.

Automated plan forcing

Introduced in SQL Server 2017, and the foundation of automatic tuning in the Azure SQL Database, the Query Store can be used to automatically identify and fix plan regression. It's referred to as automatic tuning, but understand, it's just using the most recent good plan that consistently runs better than other plans in the Query Store. It's not tuning the database in terms of updating statistics, adding, removing, or modifying indexes or, most importantly changing the code. However, for a lot of situations, this may be enough to automatically deal with performance problems.

The automatic tuning is disabled by default. To enable it, you first must have Query Store enabled and collecting data. Then, it's a simple command to enable the automated tuning.

```
ALTER DATABASE CURRENT SET AUTOMATIC_TUNING(FORCE_LAST_GOOD_PLAN =
ON);
```

Listing 16-16

The database engine will actually monitor the performance of queries using the information gathered in the Query Store. When a plan change clearly causes performance issues, a regression, the engine can automatically enable the last good plan. That may not be the best possible plan depending on the circumstances, but it will be a better plan than what is currently in use. However, the engine will also automatically check to see if performance improved or degraded. If it has degraded, the plan forcing will be revoked and the plan will recompile at the next call.

You can see immediately, even without enabling automatic tuning, if a potential automatic tuning opportunity is available. A new DMV, `sys.dm_db_tuning_recommendations`, is available to show these recommendations. Figure 16-13 shows all the columns returned from the DMV.

sys.dm_db_tuning_recommendations

Script | Summary | Copy

Column Name	Data Type	Nullability
name	nvarchar(4000)	null
type	nvarchar(4000)	null
reason	nvarchar(4000)	null
valid_since	datetime2(7)	null
last_refresh	datetime2(7)	null
state	nvarchar(4000)	null
is_executable_action	bit	null
is_revertable_action	bit	null
execute_action_start_time	datetime2(7)	null
execute_action_duration	time(7)	null
execute_action_initiated_by	nvarchar(4000)	null
execute_action_initiated_time	datetime2(7)	null
revert_action_start_time	datetime2(7)	null
revert_action_duration	time(7)	null
revert_action_initiated_by	nvarchar(4000)	null
revert_action_initiated_time	datetime2(7)	null
score	int	null
details	nvarchar(max)	null

Figure 16-13: DMV for automatic tuning recommendations.

While all the columns can be important depending on the situation, the most interesting ones are the type, reason, state, and details. The rest of the data is largely informational. However, we can't just query this data directly. The data in the `state` and `details` columns are stored as JSON. Listing 16-17 shows how to pull this information apart.

```
SELECT ddtr.reason,
       ddtr.score,
       pfd.query_id,
       JSON_VALUE(ddtr.state,
                  '$.currentValue') AS CurrentState
FROM sys.dm_db_tuning_recommendations AS ddtr
    CROSS APPLY
    OPENJSON(ddtr.details,
             '$.planForceDetails')
    WITH (query_id INT '$.queryId') AS pfd;
```

Listing 16-17

This query will pull together some of the interesting data from the DMV. However, to really put that data to work with the Query Store information to understand more fully what's going on, we'll have to expand the JSON queries quite a bit. Listing 16-18 combines the data from the `sys.dm_db_tuning_recommendations` DMV with the catalog views of the Query Store.

```
WITH DbTuneRec
AS (SELECT ddtr.reason,
           ddtr.score,
           pfd.query_id,
           pfd.regressedPlanId,
           pfd.recommendedPlanId,
           JSON_VALUE(ddtr.state,
                      '$.currentValue') AS CurrentState,
           JSON_VALUE(ddtr.state,
                      '$.reason') AS CurrentStateReason,
           JSON_VALUE(ddtr.details,
                      '$.implementationDetails.script') AS
ImplementationScript
    FROM sys.dm_db_tuning_recommendations AS ddtr
        CROSS APPLY
        OPENJSON(ddtr.details,
                 '$.planForceDetails')
        WITH (query_id INT '$.queryId',
              regressedPlanId INT '$.regressedPlanId',
              recommendedPlanId INT '$.recommendedPlanId') AS pfd)
SELECT qsq.query_id,
       dtr.reason,
       dtr.score,
       dtr.CurrentState,
       dtr.CurrentStateReason,
       qsqt.query_sql_text,
       CAST(rp.query_plan AS XML) AS RegressedPlan,
       CAST(sp.query_plan AS XML) AS SuggestedPlan,
       dtr.ImplementationScript
FROM DbTuneRec AS dtr
    JOIN sys.query_store_plan AS rp
        ON rp.query_id = dtr.query_id
           AND rp.plan_id = dtr.regressedPlanId
    JOIN sys.query_store_plan AS sp
        ON sp.query_id = dtr.query_id
           AND sp.plan_id = dtr.recommendedPlanId
    JOIN sys.query_store_query AS qsq
```

```
        ON qsq.query_id = rp.query_id
    JOIN sys.query_store_query_text AS qsqt
        ON qsqt.query_text_id = qsq.query_text_id;
```

Listing 16-18

This query will show the recommendation reason and the score (an estimated impact value from 0 to 100), the current state and reason for that, the query, the two plans in question, and finally, the script to implement the suggested change. You can use this query when Query Store is enabled (and you're on SQL Server 2017 and up) to find potential plan-forcing candidates; or you can enable automatic plan forcing and then this query will probably find queries that already have a plan forced by that feature.

You can see the output from my system in Figure 16-18.

	query_id	reason	score	CurrentState	CurrentStateReason	query_sql_text
1	4	Average query CPU time changed from 0.13ms to 0....	75	Reverted	VerificationForcedQueryRecompile	(@ReferenceOrderID int)SELECT p.Nam...

Figure 16-14: A suggested automatic tuning opportunity.

I have a simple stored procedure, `dbo.ProductTransactionHistoryByReference`, that generates five different execution plans when you run it against the entire list of `ReferenceID` values (I used a PowerShell script).

```
CREATE OR ALTER PROC dbo.ProductTransactionHistoryByReference (@
ReferenceOrderID INT)
AS
BEGIN
    SELECT p.Name,
           p.ProductNumber,
           th.ReferenceOrderID
    FROM Production.Product AS p
        JOIN Production.TransactionHistory AS th
            ON th.ProductID = p.ProductID
    WHERE th.ReferenceOrderID = @ReferenceOrderID;
END;
```

Listing 16-19

One of these plans is wildly slower than the others. With plans being recompiled regularly, it's inevitable that the slower plan will cause problems. At some point, the engine will identify these problems and create a forced plan. I can take advantage of the Forced Plans report to see the plan.

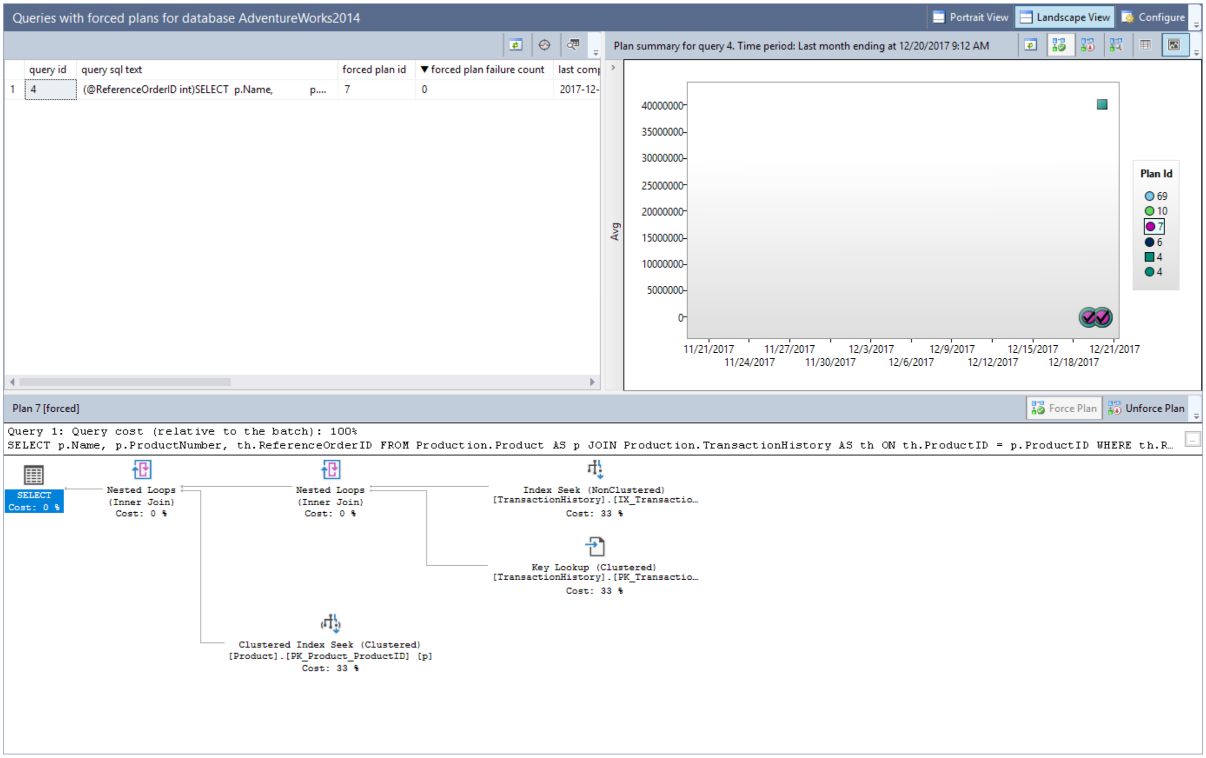


Figure 16-15: Forced Plans report from the Query Store showing automatic tuning.

You can see that there is a check mark on **Plan Id 7**, the plan that is highlighted and visible. That means that the system has forced this plan. I can verify this by going back to `sys.dm_db_tuning_recommendations` and looking at additional columns.

```
SELECT ddtr.reason,
       ddtr.valid_since,
       ddtr.last_refresh,
       ddtr.execute_action_initiated_by
FROM sys.dm_db_tuning_recommendations AS ddtr;
```

Listing 16-20

This will let us know, not only a suggested tuning process, but when it was initiated and by whom. The output from the system looks as shown in Figure 16-16.

	reason	valid_since	last_refresh	execute_action_initiated_by
1	Average query CPU time changed from 0.16ms to 6.95ms	2017-12-19 23:23:46.6533333	2017-12-20 16:19:03.0400000	System

Figure 16-16: Output from the `sys.dm_db_tuning_recommendations` DMV.

You can see that the action was taken by the system. This is direct evidence that the system has decided to force this execution plan.

Over time, the system continues to measure the performance of queries. In my example above it will occasionally, through measurements, decide that forcing this plan in fact hurts performance. In that case, you'll see the plan forcing will be removed, and if you look at `Revert_*` columns available in `sys.dm_db_tuning_recommendations`, you'll see they will be filled in. The fact that the plan was forced and, importantly, why, won't be removed from `sys.dm_db_tuning_recommendations` unless you remove the data from the Query Store (more on that in the next section).

Finally, you can decide to remove the plan forcing manually. You can either use the button on the report, visible in Figure 16-15 and other reports in this chapter, or using the T-SQL command shown in Listing 16-13. In this case, the `execute_action_initiated_by` column (Listing 16-20) will show `User` instead of `system`.

If you decide to override the automatic tuning, that query will not be automatically forced again, regardless of behavior. Your choices take precedence over the automation. The exception to this will arise if you remove the data from the Query Store. This will result in coming back around to the forced plan again because your override can't survive the loss of data. Any time you override the behavior of automatic tuning, it prevents any further automatic manipulation of the plans, on or off.

Remove Plans from the Query Store

If you disable the Query Store, it will leave all the information in place. If you want to remove every single bit of information from the Query Store, you could issue the command in Listing 16-21.

```
ALTER DATABASE AdventureWorks2014 SET QUERY_STORE CLEAR;
```

Listing 16-21

However, that is heavy handed unless your intention is to, for example, remove production data from a database prior to using that database in a development environment. If you wanted to only remove a particular query, and all its associated information including all execution plans, you could use Listing 16-22.

```
EXEC sys.sp_query_store_remove_query
    @query_id = 214;
```

Listing 16-22

If I had retrieved the `query_id` using another query, such as one from Listing 16-3, I could then use the value to run this query. It removes the query, all captured plans, and all recorded runtime stats. It would even stop plan forcing because the query has been removed and the information is no longer stored with the database.

You can also target just plans for removal. If we retrieved the `plan_id` using Listing 16-10, we could then remove a plan from the Query Store using Listing 16-23.

```
EXEC sys.sp_query_store_remove_plan @plan_id = 248;
```

Listing 16-23

This will leave the query intact as well as any other plans associated with that query. It will remove the execution plan defined by the `plan_id`. If that plan is associated with `plan_forcing`, then plan forcing will be stopped because the plan is no longer in the database.

An important thing to remember about the Query Store information is that it is stored with the database, within system tables. That means it gets backed up with the database. If you back up a production database, and then restore it to a non-production system, all the query store information will go with it. This includes any text stored with the query such as filtering criteria or compile-time parameter values. If you are working with data that has limited access, such as healthcare data, you need to take the Query Store into account when removing sensitive information from a database prior to giving it to unauthorized persons. Use the appropriate removal mechanism from above to ensure proper protection of your data.

Summary

The Query Store introduces a great deal of useful information for query performance tuning and execution plans. It persists this information with the database, which enables you to do all sorts of troubleshooting and performance tuning offline from your production system. Plan forcing means you don't have to worry about certain types of plan regressions in the future because you can easily undo them and prevent them from happening again. However, don't forget that data and statistics change over time, so the perfect plan to force today, may not be the perfect plan tomorrow.

Chapter 17: SSMS Tools for Exploring Execution Plans

Learning what makes up an execution plan and understanding how to read the properties and operators is a fundamental part of advancing your knowledge of writing efficient T-SQL queries, and improving your skills at tuning those that are causing problems.

However, as you've seen in some of the preceding chapters, certain plans are harder to navigate, and it takes time to piece together all the details of each operator, and their various properties, to work out exactly how SQL Server has chosen to execute a query, and why, and what help you can offer the optimizer to arrive at a better plan, if necessary. In such cases, it is not a bad idea to get a little extra help, and in this chapter I'll cover the SQL Server Management tools I use when I need a little extra guidance in reading and understanding a plan. I'll also mention briefly some of the third-party tools I've found useful when attempting to navigate more complex plans.

The Query

The real strength of these tools lies in the extra help they offer in reading and understanding more complex plans, often with hundreds of operators, rather than just a handful. However, it would be difficult to demonstrate those plans easily within the confines of a book.

Therefore, I've opted to use a relatively simple query, and straightforward plan, although with a few inherent problems. I'll use the same query throughout, shown in Listing 17-1.

```
SELECT soh.OrderDate,
       soh.Status,
       sod.CarrierTrackingNumber,
       sod.OrderQty,
       p.Name
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON sod.SalesOrderID = soh.SalesOrderID
    JOIN Production.Product AS p
        ON p.ProductID = sod.ProductID
WHERE sod.OrderQty * 2 > 60
      AND sod.ProductID = 867;
```

Listing 17-1

This query would benefit from a little tuning and a new index. First, the calculation on the column OrderQty is unnecessary. Next, there is no index to support the filter criteria in the WHERE clause. Figure 17-1 shows the resulting execution plan in SSMS.

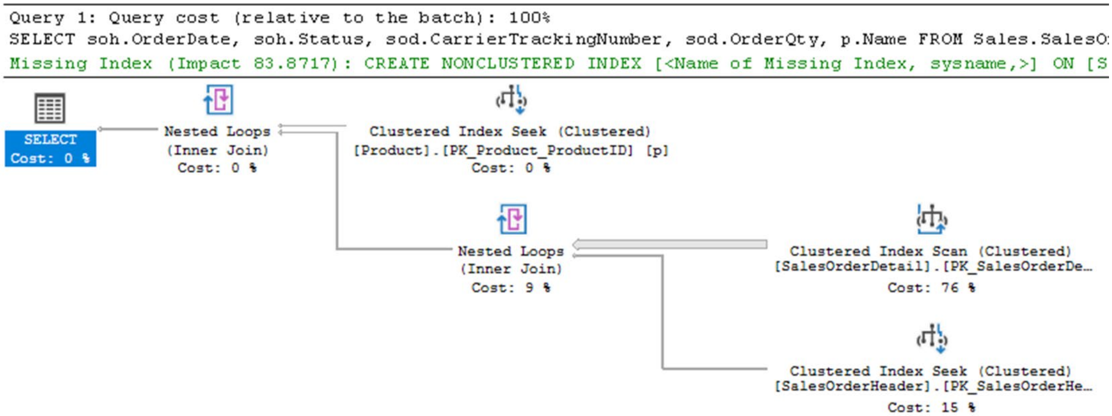


Figure 17-1: Execution plan in SSMS for the problematic query.

You can see that the scan against the primary key of the SalesOrderDetail table is estimated to be the most expensive operator. There's a suggestion for a possible index shown in the **Missing Index** information at the top of the screen:

```
CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]
ON [Sales].[SalesOrderDetail] ([ProductID])
INCLUDE ([SalesOrderID],[CarrierTrackingNumber],[OrderQty])
```

Given the simple nature of this query, we probably have enough information available to us already that we could begin to tune the query. However, let's now use it to explore the additional benefits of our tools.

The SQL Server Management Studio 17 Tools

After many years of relatively modest improvements to the information available with execution plans, the latest version, SSMS 17, has taken some bigger strides in increasing visibility of important information in the plans, allowing us to compare that information between plans, and more.

Before the release of SQL Server 2017, the announcement was made that SSMS would became a stand-alone piece of software, installed and maintained separately from the SQL Server engine. This divorced SSMS from the longer, slower, release cycle of Service Packs and Cumulative Updates and allowed the SSMS team to introduce enhancements at a faster pace than we'd become accustomed to, including several in support of execution plans.

It's still a free tool and you can download it from Microsoft (http://bit.ly/2kDEQrk). You can install it side by side with existing versions of SSMS. The current version (as of this writing) supports SQL Server 2008–2017, as well as Azure SQL Database, and has some limited support for Azure SQL Data Warehouse.

We've been exploring plans using SSMS throughout the book, so I'm only going to cover the new functionality that has been explicitly introduced to help you understand execution plans.

Right-click inside an execution plan in SSMS 17, and you'll see a context menu listing three newer pieces of functionality: **Compare Showplan**, **Analyze Actual Execution Plan**, and **Find Node**.

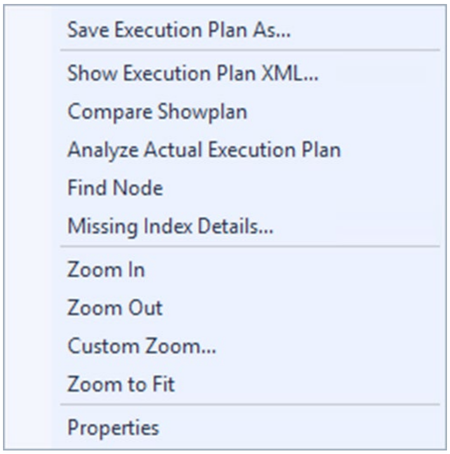


Figure 17-2: Context menu showing newer menu choices related to execution plans.

Analyze Actual Execution Plan

Select **Analyze Actual Execution Plan** from the menu, and it will open a new pane at the bottom of your query window, as shown in Figure 17-3.

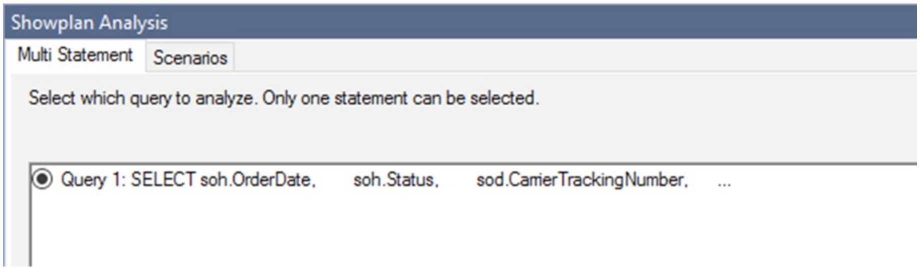


Figure 17-3: Showplan Analysis with a single query for the batch.

With a single statement batch, such as the example from Listing 17-1, you'll only see a single query. If you have multiple statements in your batch, you'll see multiple queries. To have one of the queries analyzed, just select that query using the radio buttons. You then click on the **Scenarios** tab, where each scenario shows *details on a category of potential issues found in the plans*.

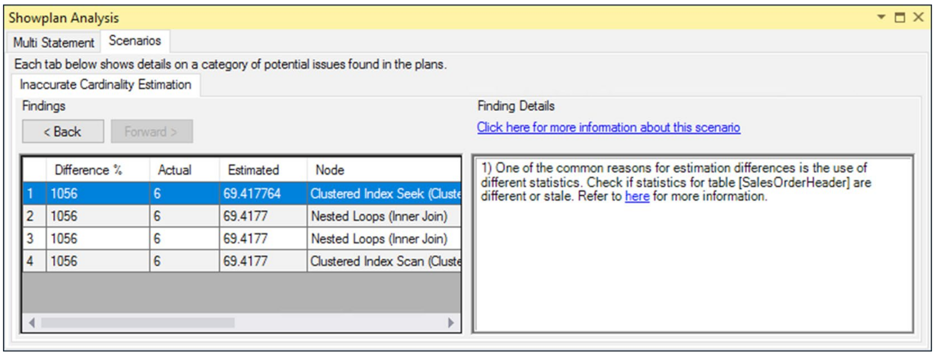


Figure 17-4: The Scenarios tab of the Showplan Analysis with suggested problems.

According to Microsoft the scenarios presented for a query will provide different analysis mechanisms to guide you through problematic plans. At time of writing, they've defined only one scenario, **Inaccurate Cardinality Estimation**. This is a good choice since it's a common problem in a stable environment and a very serious problem during upgrades, especially when moving from servers older than SQL Server 2014 to servers newer than SQL Server 2014 (where the new cardinality estimation engine was introduced).

For the **Inaccurate Cardinality Estimation** scenario, the information is broken into two parts. On the left is a list of operators where the cardinality estimations differ significantly between estimated and actual. You're provided with information about the differences, in a neat grid. This shows the **Actual** and **Estimated** values for each operator, the node involved, and the percentage difference. If you check the properties of the Clustered Index Seek (highlighted in Figure 17-4) operator in the graphical plan, you'll see that **Actual Number of Rows** is 6, and **Estimated Number of Rows** is 1, but the Showplan analysis accurately accounts for the fact that the **Estimated Number of Executions** is 69.4177, giving a total estimated number of rows returned of 69.4177.

On the right, you'll find an explanation of one or more possible reasons why the cardinality estimation may be different. This provides guidance on how to address the issue, and possibly improve the query performance, although never just assume that this guidance is 100% accurate. Always validate it on your system before implementing the advice.

Selecting any one of the nodes will also update which node is selected within the execution plan itself, and will update the guidance so that it reflects the selected node. In Figure 17-5, I've selected the third node in the list, one of the **Nested Loops** joins.

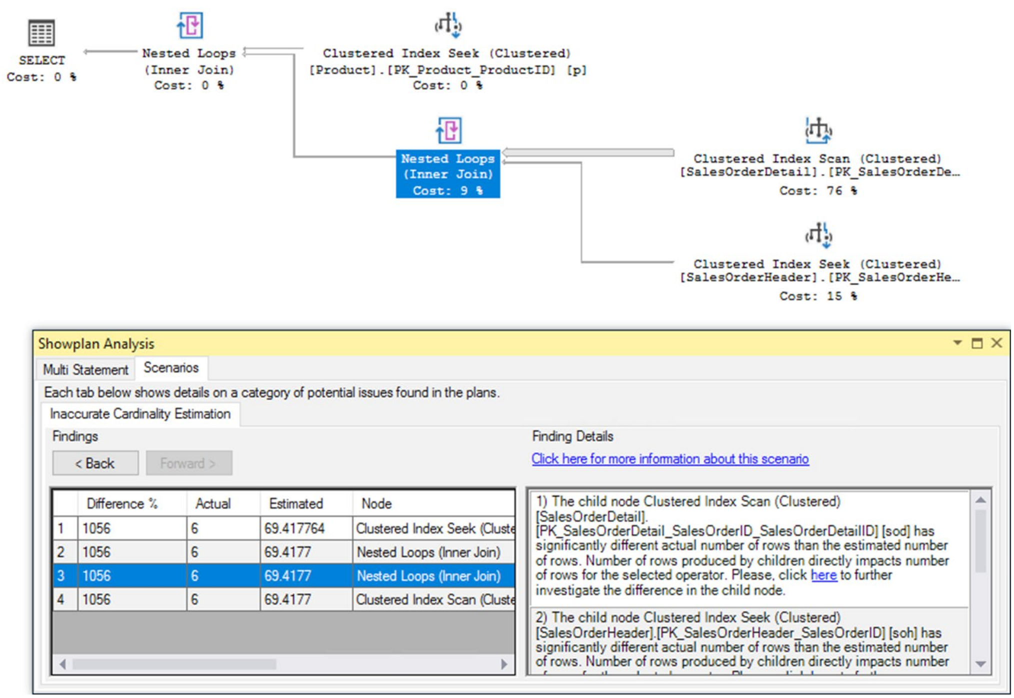


Figure 17-5: Selecting different suggestions also changes the nodes in the plan.

While this functionality is currently limited, I know there will be further enhancements, which should deepen your understanding of possible issues with your queries and data structures, as exposed through the execution plans in SSMS.

Compare Showplan

The **Compare Showplan** features allows us, perhaps unsurprisingly, to compare two different execution plans for similarities and differences. You can compare two actual plans, two estimated plans, or an actual plan to an estimated plan; any combination will work. You can also compare plans between different SQL Server versions, different patch levels, and so on. If you have two valid plans, and at least one of them stored as a file, you can compare them.

To test it out, we'll use the query in Listing 17-2, which is similar to Listing 17-1 in that it references the same tables and columns, but with a different `WHERE` clause.

```
SELECT soh.OrderDate,
       soh.Status,
       sod.CarrierTrackingNumber,
       sod.OrderQty,
       p.Name
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON sod.SalesOrderID = soh.SalesOrderID
    JOIN Production.Product AS p
        ON p.ProductID = sod.ProductID
WHERE sod.ProductID = 897;
```

Listing 17-2

Execute Listing 17-1, capture the actual plan, use **Save Execution Plan As…**, to save it as a **.sqlplan** file, and then capture the actual plan for Listing 17-2. Right-click on it, and select **Compare Showplans** from the context menu, which will open a File Explorer window. Locate and select your saved showplan file, and you should see a **Showplan Comparison** window that looks something like Figure 17-6.

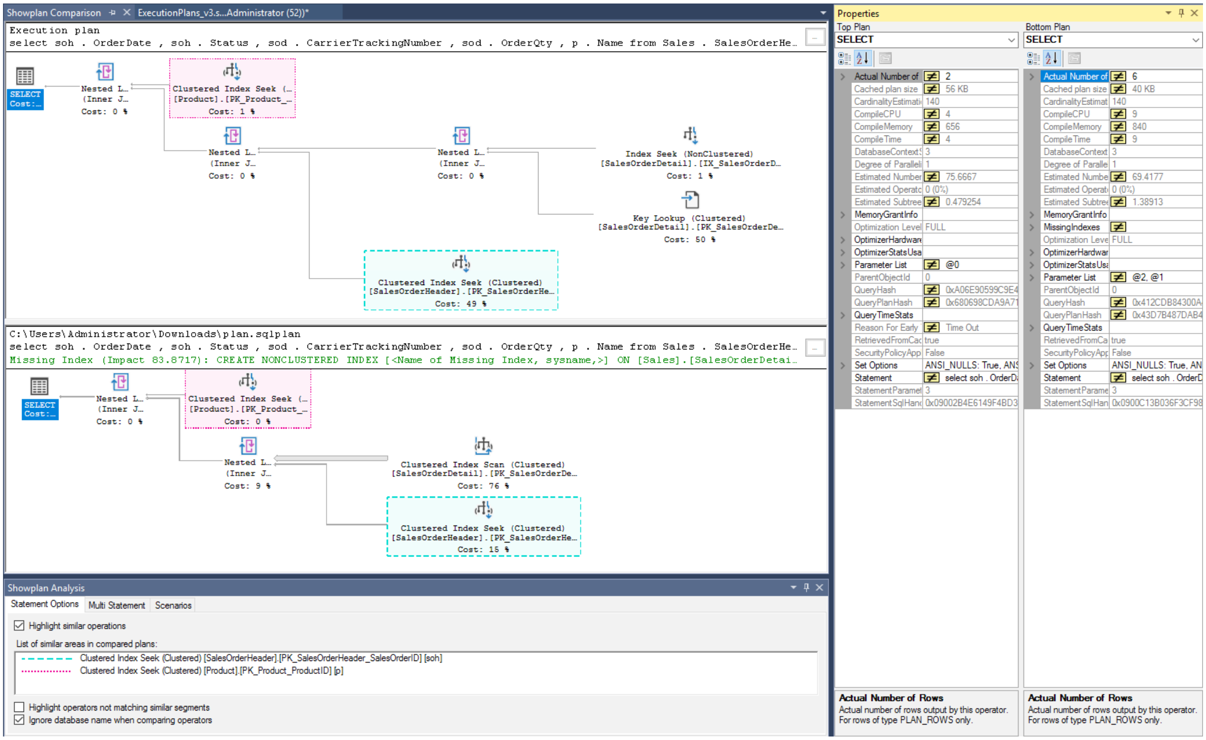


Figure 17-6: Showplan Comparison including the plans, Properties, and Statement Options.

The top plan is the one from which we initiated the comparison (Listing 17-2). Below the plans you'll see the **Showplan Analysis** tab, which we saw earlier, but now with an additional tab, **Statement Options**. Figure 17-7 shows a blow-up of this area.

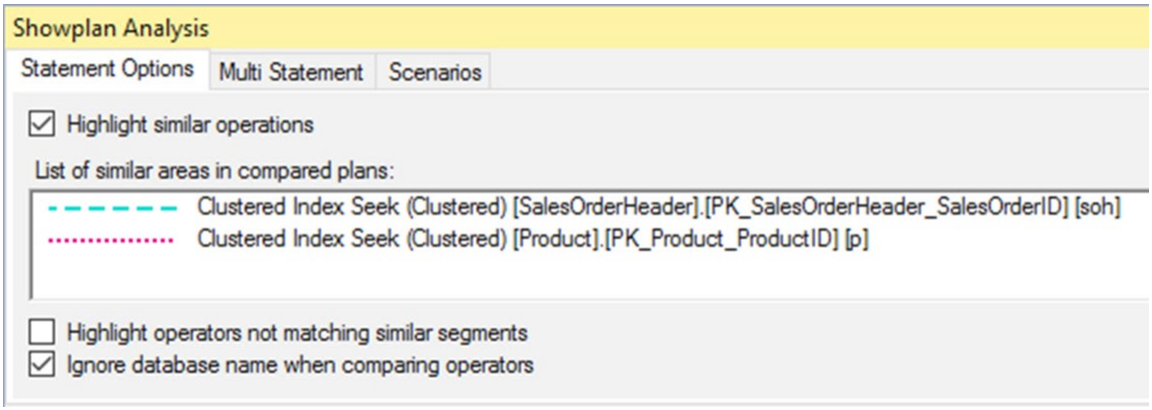


Figure 17-7: Statement Options tab of the Showplan Analysis window.

By default, the **Highlight similar operations** checkbox is activated, and the box below highlights areas of similar functionality within the plan. In this case, you can see two similar areas, highlighted in pink and green. If directly-connected operators are similar in each plan, they'll be grouped. In our case, two operators are similar, but in different parts of each plan. Also, by default, the plan comparison ignores database names. You may see no similarities at all, or you may see multiple sets of similarities, in which case each "similar area" will have a different color.

To the right of the graphical plans are the **Properties** windows for each plan, with the top plan on the left, which you can use to compare property values between the plans. In Figure 17-6, I've highlighted the **SELECT** operator in both plans, and **Compare Showplan** is highlighting with the "not-equals" sign those property values that don't match, as shown in Figure 17-8.

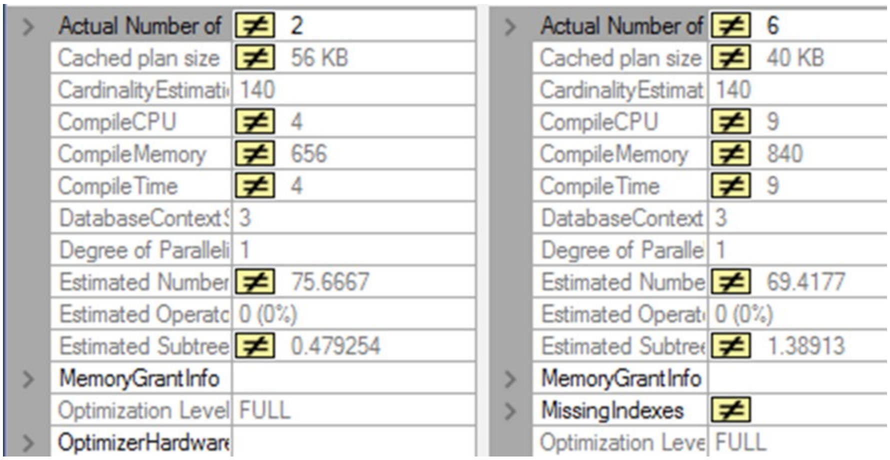


Figure 17-8: Properties in comparison between two plans.

Also, you can see that there are some properties visible in one plan that don't exist in the other. In this case, only the plan for Listing 17-2 shows a **MissingIndexes** property.

If you select the operator highlighted in pink in Figure 17-6, the **Clustered Index Seek** on the `Product` table, you can see that almost every property value between these two operators in two plans is identical.

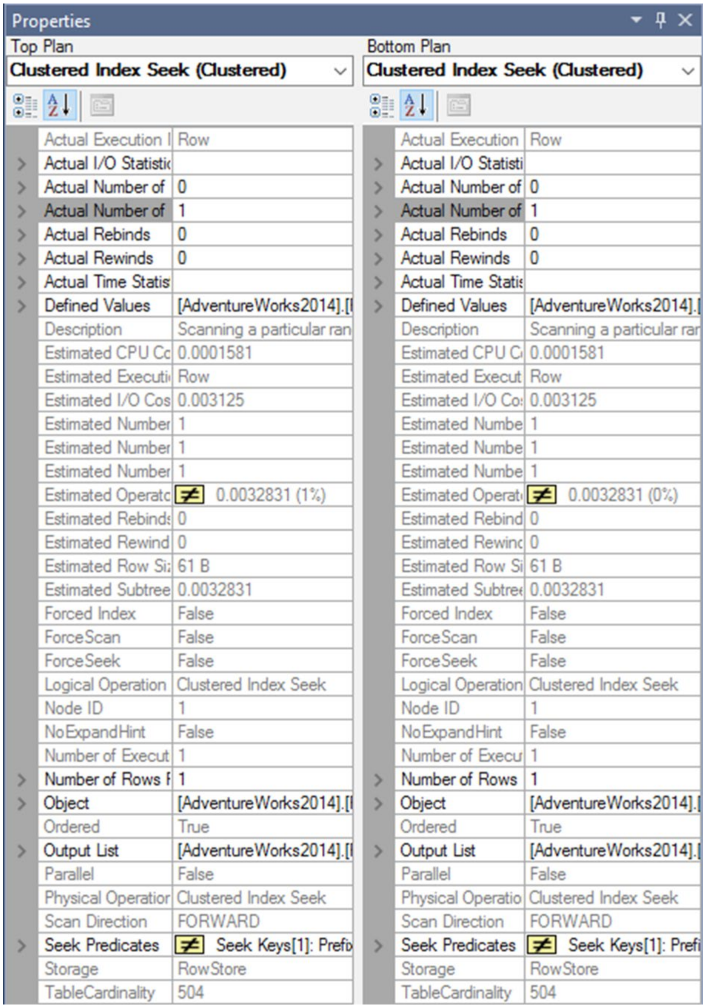


Figure 17-9: An operator that is very similar between the two plans.

Even the values for the Estimated Operator Cost are the same, but it's highlighted as different because the operator cost as a percentage of the whole plan is different in each case. The other highlighted difference is in the **Seek Predicates** property. In my case, this is simply because I have forced parameterization (see Chapter 9) in operation for this query, and the optimizer used different parameter names during the forced parameterization process. Without this, the differences will simply be the different literal values used, in each case.

We can change the comparison behavior of **Compare Showplan**, by activating the **Highlight operators not matching similar segments** checkbox shown in Figure 17-7, either instead of, or in addition to, the **Highlight similar operations** checkbox. I opted for the former, and Figure 17-10 shows that the non-matching operators are now highlighted in yellow.

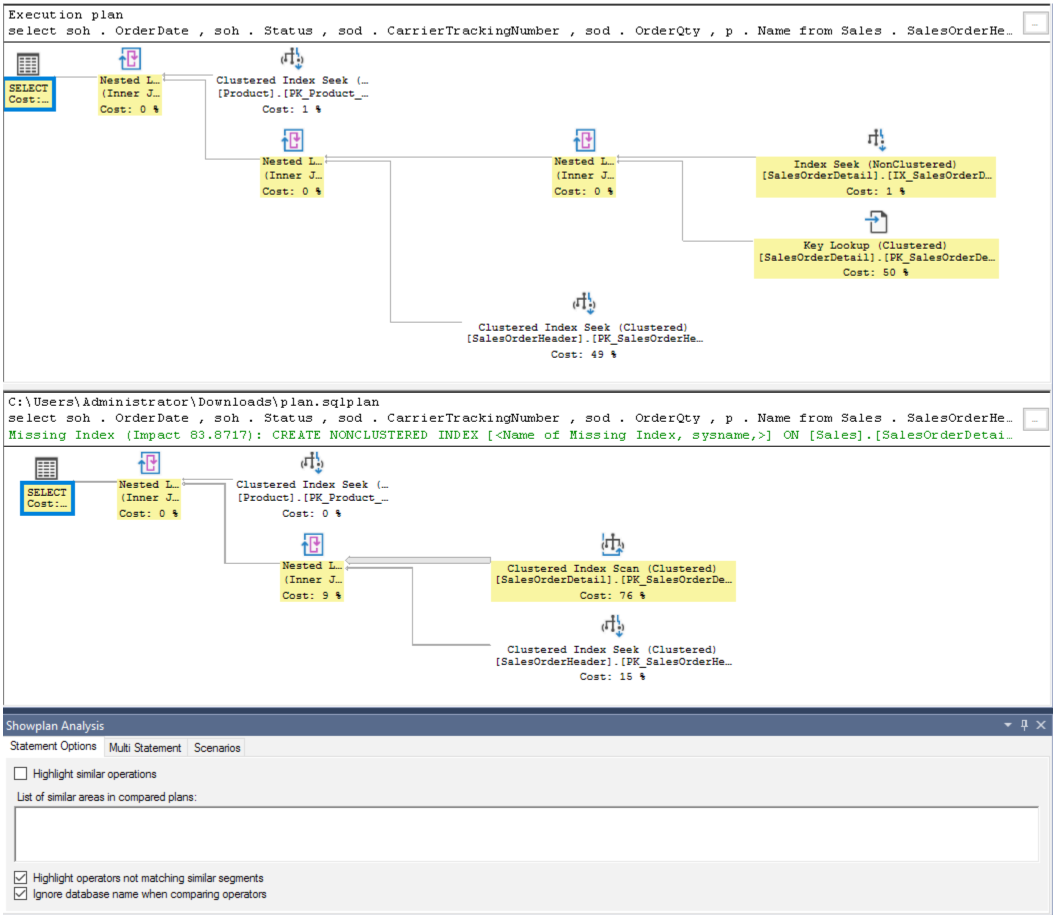


Figure 17-10: Non-matching operators are now highlighted.

I use this functionality all the time while tuning queries because, while sometimes there are glaring differences between plans, often they are much subtler, but with significant performance implications. This feature helps to be able to spot these small differences faster, especially when comparing two almost-identical, large-scale execution plans.

Find Node

Right-click on a graphical plan and choose **Find Node**, and a small window opens in the upper right of the execution plan. Listed in the left-hand drop-down is a big list of properties, as shown in Figure 17-11.

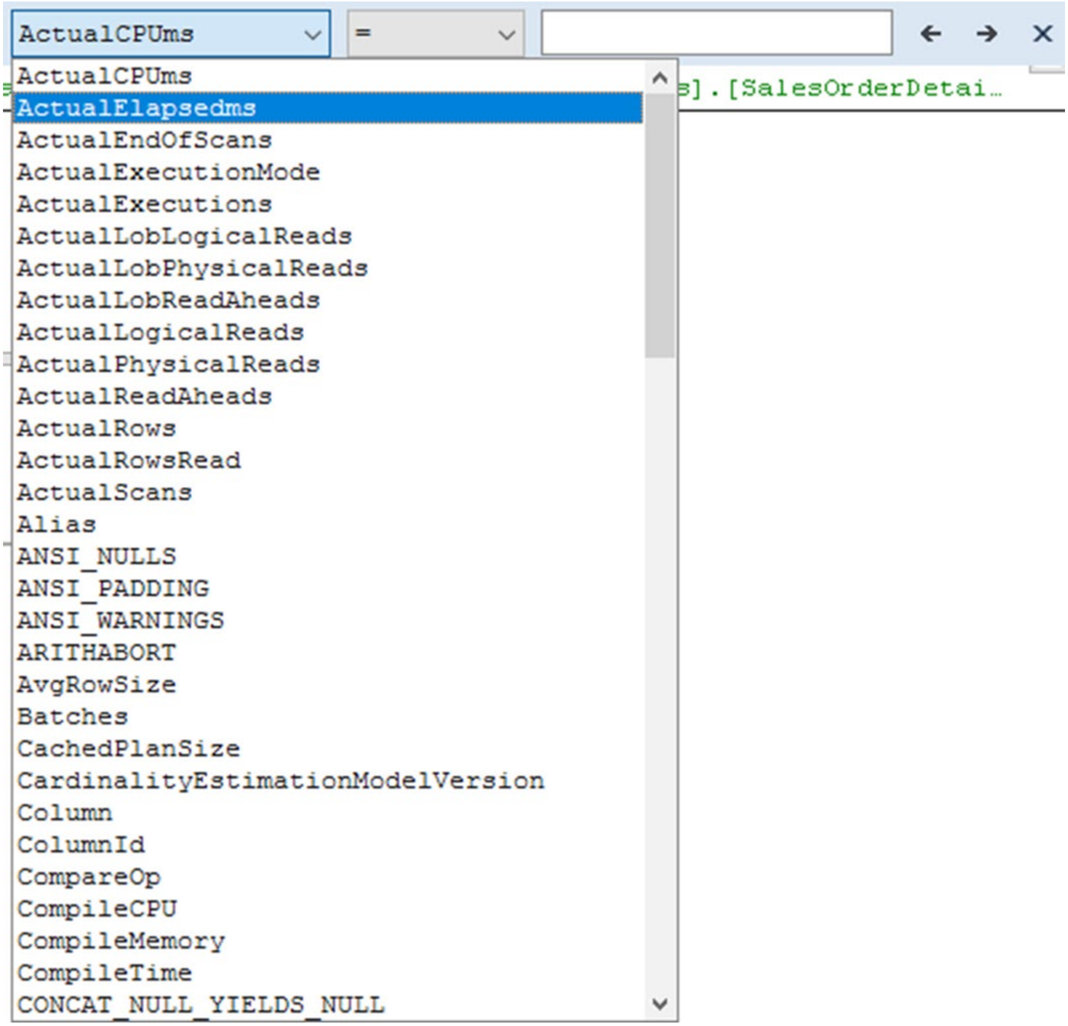


Figure 17-11: Drop-down of the Find Node feature, with all the properties of the plan.

Select a property, for example **ActualRows**, then select a comparison operator, "equals" for numeric searches or "contains" for text searches, and the value for which you want to search.

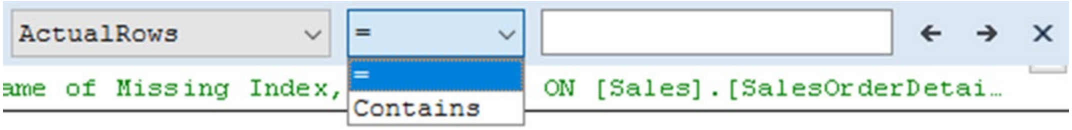


Figure 17-12: The comparison property list.

For text searches there is no need for wild cards; it assumes you'll want to see similar matches as well as exact. If you search on **ActualRows = 6**, and then click the left or right arrows, you can search through the plans, in `NodeId` order, for operators that return 6 rows.

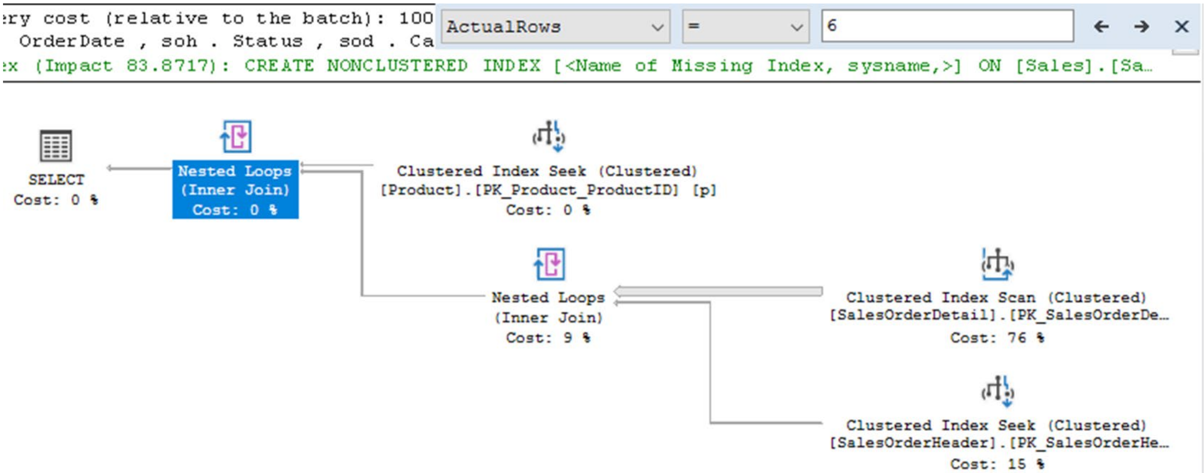


Figure 17-13: Finding the first operator matching the Find Node search criteria.

While you won't really need **Find Node** for small execution plans, it becomes a huge help when dealing with larger plans, making it much easier, for example, to find the operator with the **ParentNodeID** that matches the **NodeID** of a **Table Spool** operator, or to find every reference to a column name.

Live execution plans

A live execution plan is one that exposes per-operator runtime statistics, in real time, as the query executes. You'll get to see the query execution in action, and view the per-operator statistics, as the execution progresses and data flows from one operator to the next. This is useful if, for example, you need to understand how data moves through the plan for a very long-running query. A live execution plan will also show you the **estimated query progress**, which might be useful if you need to decide whether to kill the query.

SQL Server 2014 was the first version to introduce a way to track progress on a long-running query. You could query the `sys.dm_exec_query_profiles` Dynamic Management View (DMV) from another connection. However, it came with quite a high overhead, since the data was only captured if you executed the query with the option to include the actual execution plan enabled.

Subsequent SQL Server versions (and Service Pack 2 for SQL Server 2014) have introduced lower-overhead ways to view the in-progress runtime statistics, without the need to capture the actual plan, via a new extended event (`query_thread_profile`) or by enabling **Trace Flag 7412**. Enabling the trace flag allows us to use a new *lightweight query execution statistics profiling infrastructure*, which dramatically reduces the overhead of capturing the in-progress query execution statistics.

Using the trace flag is the lowest-cost method of the three, followed by using the extended event (which enables the trace flag automatically), and capturing the actual plan is the most expensive option. Caution, though: even if you're using the trace flag, low-cost doesn't mean no-cost. You should still test this carefully before enabling it on your production systems. There is overhead associated with capturing the runtime metrics.

Let's see all this in action. To do so, we'll introduce one new query, in Listing 17-3.

```
SELECT *
FROM sys.objects AS o,
     sys.columns AS c;
```

Listing 17-3

This query violates a bunch of rules, many of which we have maintained throughout this book. However, it takes about 40 seconds to run on my system, so it makes a good test bed for all the other functions we'll see within live execution plans.

Live per-operator statistics using sys.dm_exec_query_profiles

The `sys.dm_exec_query_profiles` DMV shows the number of rows processed by individual operators within a currently executing query, allowing you to see the status of the executing query, and compare estimated row-count values to actual values.

If you're testing this on SQL Server 2014, but pre-SQL Server 2014 SP2, you'll need to run Listing 17-3 using any of the options that include the actual execution plan, either in SSMS or by using one of the `SET` commands, or by capturing the `query_post_execution_showplan` event (see Chapter 15).

SQL Server 2016 introduced live execution statistics into SSMS, and added to Extended Events the new "debug" category event called `query_thread_profile`. SQL Server 2016 SP1 introduced the Trace Flag 7412. Both the extended event and the trace flag were retro-fitted into SQL Server 2014 SP2.

So, on SQL Server 2014 SP2, or on SQL Server 2016 SP1 and later, the best way is to first enable Trace Flag 7412, as shown in Listing 17-4.

```
DBCC TRACEON (7412, -1);
```

Listing 17-4

Now, start executing Listing 17-3, and from another session run the following query against the `sys.dm_exec_query_profiles` DMV. Note that I'm eliminating the current session from the query because otherwise it will show up in the results.

```
SELECT deqp.session_id,
       deqp.node_id,
       deqp.physical_operator_name,
       deqp.estimate_row_count,
       deqp.row_count
FROM sys.dm_exec_query_profiles AS deqp
WHERE deqp.session_id <> @@SPID
ORDER BY deqp.node_id ASC;
```

Listing 17-5

The DMV returns a lot more information than I've requested here (see the Microsoft documentation for a full description: https://bit.ly/2JKYe5s), and you can combine this DMV with others to return even more information. Figure 17-14 shows a subset of the results.

	session_id	node_id	physical_operator_name	estimate_row_count	row_count
1	58	1	Nested Loops	6264966	94465
2	58	2	Hash Match	2044	144
3	58	3	Clustered Index Seek	8	8
4	58	5	Hash Match	2034	144
5	58	6	Clustered Index Seek	2	2
6	58	8	Hash Match	2031	144
7	58	9	Clustered Index Seek	3	3

Figure 17-14: Results of query against sys.dm_exec_query_profiles.

You can see the nodes and their names along with the `estimated_row_count`, which shows the total estimated number of rows to be processed, which you can then compare to the actual number of rows currently processed, in the `row_count` column. You can see immediately that the node with an ID value of 1, the **Nested Loops** operator, has an estimated number of rows of 6,264,966, and has only actually processed 94,465. This lets us know that, without a doubt, the query is still processing, and has quite a way to go to get to the estimated number of rows. Of course, if the optimizer's row count estimates are inaccurate then the `row_count` and `estimated_row_count` may not match up. However, this provides one way to track the current execution status of a query and how much it has successfully processed.

If you query the DMV again while the query is still executing, you can see the changes to the data.

	session_id	node_id	physical_operator_name	estimate_row_count	row_count
1	58	1	Nested Loops	6264966	284022
2	58	2	Hash Match	2044	431
3	58	3	Clustered Index Seek	8	8
4	58	5	Hash Match	2034	431
5	58	6	Clustered Index Seek	2	2
6	58	8	Hash Match	2031	431
7	58	9	Clustered Index Seek	3	3

Figure 17-15: Changes to the information from sys.dm_exec_query_profiles.

As you can see, more rows have been processed by several of the operators, but the execution is not yet complete. When you run the query after the long running query has completed, you won't see a completed set of row counts. Instead, you'll see nothing at all, since there are no active sessions.

Using the query_thread_profile extended event

If you want to see just the completed information, you can capture the `query_thread_profile` extended event, which triggers *for each query plan operator and execution thread, at the end of query execution*. It's a "Debug" channel event, so you'll need to enable that channel in the SSMS GUI for Extended Events, to see the event.

Capturing the data from the event you'll see the execution statistics for each operator within a given execution plan. As stated earlier, this is a debug event, so caution should be exercised when using it. However, Microsoft has documented its use, so I have no problem sharing this with you. To add the event through T-SQL, just add the event. To add the event through the GUI, you will need to click on the drop-down for the Channel and select Debug. Figure 17-16 shows the information for the **Nested Loops** operator (**NodeId=1**) that we saw earlier.

Field	Value
actual_batches	0
actual_execution_mode	Row
actual_logical_reads	0
actual_physical_reads	0
actual_ra_reads	0
actual_rebinds	1
actual_rewinds	0
actual_rows	1273188
actual_writes	0
attach_activity_id.guid	3903C230-497C-4D90-8E0B-B9F38B476CFA
attach_activity_id.seq	1
attach_activity_id_xfer.guid	769DCC90-197B-4C56-88AC-8D2AE0AC0937
attach_activity_id_xfer.seq	0
cpu_time_us	1752251
estimated_rows	6264966
io_reported	False
node_id	1
thread_id	0
total_time_us	1752251

Figure 17-16: Output from query_thread_profile extended event.

You can see that the estimated number of rows is 6,264,966, as before. The actual number of rows shows the full execution-to-completion value of 1,273,188. So, in this case, the actual row count is significantly less than the estimated row count. You also get interesting additional information such as the `total_time_us` and `cpu_time_us`, which can be useful for performance tuning.

Live execution plans in SSMS

All the previous ways to see the "live" runtime information are useful. You could even build a tool that constantly queries these sources, to show a live view into an execution plan, as it is executing. However, we don't have to because, as of SQL Server 2016, this feature is already included in SSMS. Note again, this will only work on versions of SQL Server that can show the live query metrics we've been capturing in the sections above.

Figure 17-17 shows the **Include Live Query Statistics** icon in SSMS (the red arrow is all mine). This icon acts as a toggle, just like the **Include Actual Execution Plan** button to its left.

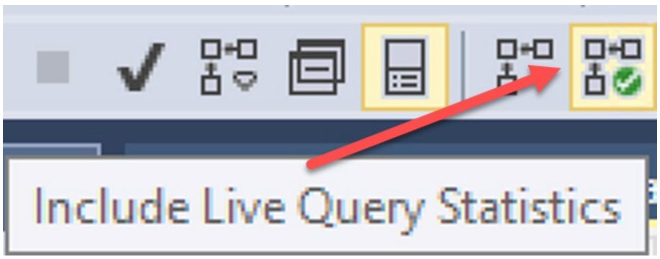


Figure 17-17: The tooltip and icon for Include Live Query Statistics.

If you enable **Include Live Query Statistics**, and then execute the query, you'll be able to capture a live execution plan, and view the execution statistics for the plan, while the query is still executing; turn it off, and you won't (unless you use Activity Monitor, as I'll demonstrate shortly). Since we're capturing the plan, we don't need to be running the `query_thread_profile` extended event or have Trace Flag 7412 enabled to use this feature. Note that enabling the trace flag doesn't make it more lightweight to use this SSMS feature; you're still paying the cost of capturing the plan.

Figure 17-18 shows the live execution plan for our long-running query.

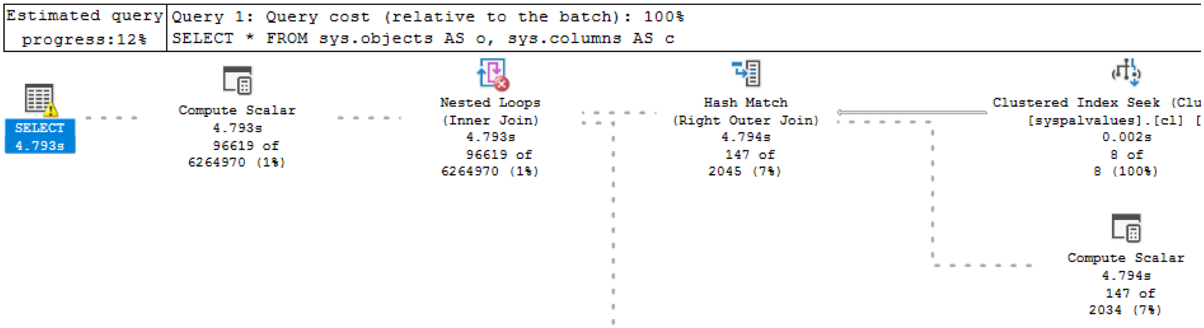


Figure 17-18: A subset of a live execution plan in action.

Of course, showing real-time, ever-changing output in a still frame, within a book, doesn't quite have the same impact. The only immediate indications that you're not just looking at another execution plan are the **Estimated query progress** in the upper left (currently at 12%), the dashed lines instead of solid lines between the operators, and the row counts with percentage complete beneath the operators. If you are viewing a live execution plan in SSMS, you will see the dashed lines moving, indicating data movement, and the row counts moving up as data is processed by an operator. This continues until the query completes execution, at which point you're just looking at a regular execution plan.

You can also look at the properties of any of the operators during the execution of the query. There you'll see a normal set of properties. However, the properties associated with an actual execution plan, such as the actual row count, will be changing in time with the plan, providing you indications as to the progress of the query, in real time.

Viewing the live execution plan in Activity Monitor

With Trace Flag 7412 enabled, or if you're capturing the `query_thread_profile` event, other tools can offer to display a live execution plan, any time while a query is executing, without the need to capture an actual plan.

So, we can use the Activity Monitor within SSMS to see queries that are actively consuming a large amount of resources as shown in Figure 17-19.

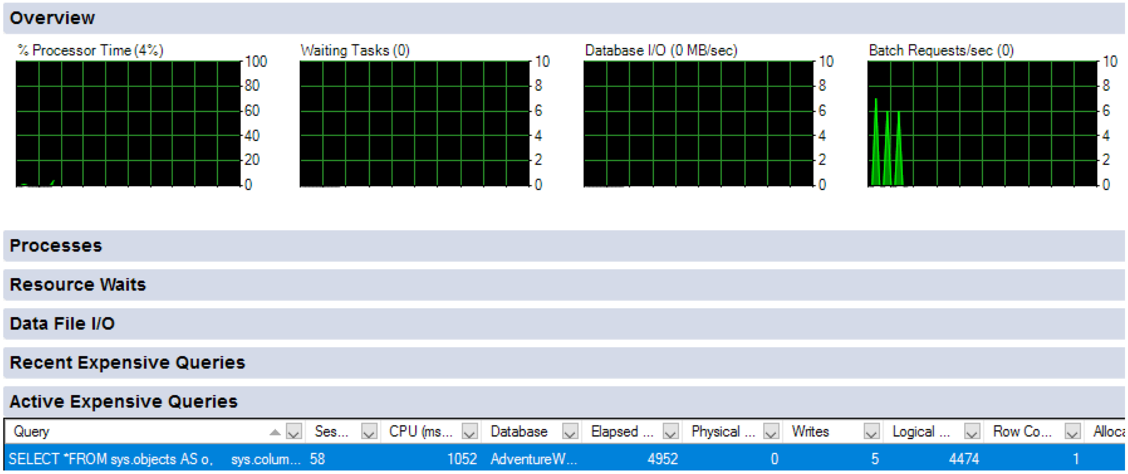


Figure 17-19: Activity Monitor showing an active expensive query.

The query shown in the **Active Expensive Queries** report is from Listing 17-3. If I right-click on that query while it's in the active state, I'll see a menu choice, as in Figure 17-20.

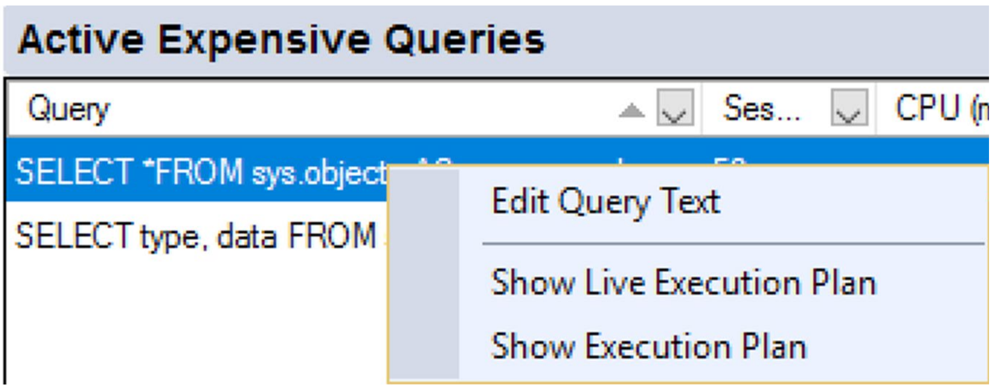


Figure 17-20: A context menus showing a choice for a live execution plan.

If I select **Show Live Execution Plan**, I will be brought to a window just like in Figure 17-18. The behavior from then on is the same.

Live execution plans are useful if you have very long-running queries, and wish to develop a more direct understanding of how the data moves within the operators. The information contained in live execution plans, as well as the associated DMVs and Extended Events, can help you decide when to roll back a transaction, or make other types of decisions, based on how far and how fast the processing has gone within a query.

They suffer from two issues. First, they are dependent on the estimated values. If those are off, so will the information be within the live execution plan. Second, capturing the information for a live execution plan, even the lightweight options of Trace Flag 7412 or the `query_thread_profile` event, may be too expensive for some systems. Exercise caution when implementing this fascinating and useful functionality.

Other Execution Plan Tools

While I decided that it was out of scope to cover third-party tools, I will mention here the ones that I've used, personally, and that not only display the plans, but also offer additional functionality that will help you understand them. This is not a complete list; they are just the ones I've used to date, and my apologies if I left out your favorite software.

Plan Explorer

Perhaps the best-known tool for navigating execution plans is Plan Explorer by SentryOne (sentryone.com). It is a full, stand-alone application that offers many different views and layouts of a plan. It also performs some intelligent analysis of the property values, index statistics, and runtime statistics, to help you read even large-scale plans, and spot possible causes of sub-optimal performance.

Supratimas

Supratimas is a web browser-based tool, available for free online at supratimas.com. You can simply "drag and drop" your query text, or **.sqlplan** file, and it will display the graphical plan, and visually highlight important property values, and the operators that are estimated to be the most expensive. It also has an SSMS plug-in that is free when supported by ads, or you can purchase it.

SSMS Tools Pack – Execution Plan Analyzer

SSMS Tools Pack (ssmstoolspack.com), written by Mladen Prajdić, is a collection of add-ons for SQL Server Management Studio that provide a whole slew of additional functionality to help make SSMS a friendlier place to work, including an **Execution Plan Analyzer**.

This tool works directly from your SSMS query window. It offers a range of different views of the "expensive" operators in the plan, and the analyzer will highlight potential problems, such as a large mismatch between estimated and actual row counts, and suggest possible courses of action.

SQL Server performance monitoring tools

I won't cover any of the third-party performance monitoring and tuning tools that capture the execution plan as part of their diagnostic data set, such as the one I use, Redgate SQL Monitor. These tools don't attempt to improve your understanding of plans, rather than just present them. That said, a tool like SQL Monitor is valuable precisely because it captures the plan for each query, within the context of all the other useful resource-usage data and performance metrics, collected *at the time the query executed*.

Summary

SSMS 17 has provided us with a lot more help than we ever had previously toward understanding execution plans, and the differences between plans. Also, there are some third-party tools that are useful, especially when trying to open and navigate around very large plans, to identify possible issues.

Each of these tools brings different strengths to the table, but none of them replaces your knowledge of how execution plans are generated through the query optimizer and how to read and understand them. Instead, they just add to your knowledge, ability and efficiency.

Index

P

S

T