

SQL Handbooks



SQL Server Execution Plans

Second Edition

By Grant Fritchey



SQL Server Execution Plans

Second Edition

By Grant Fritchey

Published by Simple Talk Publishing September 2012

First Edition 2008

Copyright Grant Fritchey 2012

ISBN: 978-1-906434-92-2

The right of Grant Fritchey to be identified as the author of this work has been asserted by him in accordance with the Copyright, Designs and Patents Act 1988

All rights reserved. No part of this publication may be reproduced, stored or introduced into a retrieval system, or transmitted, in any form, or by any means (electronic, mechanical, photocopying, recording or otherwise) without the prior written consent of the publisher. Any person who does any unauthorized act in relation to this publication may be liable to criminal prosecution and civil claims for damages.

This book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, re-sold, hired out, or otherwise circulated without the publisher's prior consent in any form other than which it is published and without a similar condition including this condition being imposed on the subsequent publisher.

Technical Reviewer: Brad McGehee

Editors: Tony Davis and Brad McGehee

Cover Image by Andy Martin

Typeset by Peter Woodhouse & Gower Associates

Table of Contents

Introduction	13
Changes in This Second Edition	15
Code Examples	16
Chapter 1: Execution Plan Basics	18
What Happens When a Query is Submitted?	19
Query parsing	19
Algebrizer	20
The query optimizer	21
Query execution	24
Estimated and Actual Execution Plans	25
Execution Plan Reuse	26
Clearing Plans from the Plan Cache	28
Execution Plan Formats	29
Graphical plans	29
Text plans	29
XML plans	30
Getting Started	31
Permissions required to view execution plans	31
Working with graphical execution plans	32
Working with text execution plans	42
Working with XML execution plans	46
Interpreting XML plans	47
Retrieving Plans from the Cache Using Dynamic Management Objects	51
Automating Plan Capture Using SQL Server Trace Events	53
Execution plan events	54
Capturing a Showplan XML trace	56
Why the actual and estimated execution plans might differ	59
Summary	61

Chapter 2: Graphical Execution Plans for Basic Queries _____ 62

The Language of Graphical Execution Plans _____ 62

Some Single Table Queries _____ 65

Clustered Index Scan _____ 65

Clustered Index Seek _____ 68

NonClustered Index Seek _____ 70

Key Lookup _____ 73

Table Scan _____ 79

RID Lookup _____ 80

Table Joins _____ 83

Hash Match join _____ 86

Nested Loops join _____ 89

Compute Scalar _____ 92

Merge Join _____ 93

Filtering Data _____ 96

Execution Plans with GROUP BY and ORDER BY _____ 99

Sort _____ 99

Hash Match (aggregate) _____ 103

Filter _____ 104

A brief aside on rebinds and rewinds _____ 105

Execution Plans for INSERT, UPDATE and DELETE Statements _____ 108

INSERT statements _____ 109

UPDATE statements _____ 112

DELETE statements _____ 114

Summary _____ 114

Chapter 3: Text and XML Execution Plans for Basic Queries __ 116

Text Execution Plans _____ 117

A text plan for a simple query _____ 117

A text plan for a slightly more complex query _____ 121

XML Execution Plans _____ 126

An estimated XML plan _____ 127

An actual XML plan _____ 134

Querying the XML	135
Summary	137

Chapter 4: Understanding More Complex Query Plans 138

Stored procedures	138
Using a sub-select	141
Derived tables using APPLY	145
Common table expressions	149
MERGE	154
Views	159
Indexes	164
Summary	176

Chapter 5: Controlling Execution Plans with Hints 177

Query Hints 178

HASH ORDER GROUP	178
MERGE HASH CONCAT UNION	182
LOOP MERGE HASH JOIN	185
FAST n	190
FORCE ORDER	191
MAXDOP	196
OPTIMIZE FOR	199
PARAMETERIZATION SIMPLE FORCED	205
RECOMPILE	205
ROBUST PLAN	208
KEEP PLAN	209
KEEPFIXED PLAN	210
EXPAND VIEWS	210
MAXRECURSION	212
USE PLAN	212

Join Hints 212

LOOP	213
MERGE	216

Table Hints	218
Table hint syntax	218
NOEXPAND	219
INDEX()	221
FASTFIRSTROW	223
Summary	226

Chapter 6: Cursor Operations **227**

Simple cursors	227
Logical operators	229
Physical operators	237
More cursor operations	238
Static cursor	238
Keyset cursor	243
READ_ONLY cursor	246
Cursors and performance	247
Summary	254

Chapter 7: Special Datatypes and Execution Plans **255**

XML	255
FOR XML	257
OPENXML	266
XQuery	271
Hierarchical Data	279
Spatial Data	282
Summary	286

Chapter 8: Advanced Topics **287**

Reading Large-scale Execution Plans	288
Parallelism in Execution Plans	295
Max degree of parallelism	296
Cost threshold for parallelism	297
Are parallel plans good or bad?	298
Examining a parallel execution plan	299

How Forced Parameterization Affects Execution Plans	305
Using Plan Guides to Modify Execution Plans	310
Object plan guides	311
SQL plan guides	314
Template plan guides	315
Plan guide administration	316
Plan forcing	317
Summary	321

About the Author

Grant Fritchey is a SQL Server MVP with over 20 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 Red Gate 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 two books: the one you're reading now and *SQL Server 2012 Query Performance Tuning Distilled*, 3rd Edition (Apress, 2012).

In the past, people have called him intimidating and scary. To which his usual reply is "Good."

You can contact him through *grant -at- scarydba dot kom* (de-obfuscate as necessary).

About the Technical Reviewer

Brad M. McGehee is a MCTS, MCSE+I, MCSD, and MCT (former) with a Bachelor's degree in Economics and a Master's in Business Administration. Currently a DBA with a Top 10 accounting firm, Brad is an accomplished Microsoft SQL Server MVP with over 17 years' SQL Server experience, and over 8 years' training experience; he has been involved in IT since 1982.

Brad is a frequent speaker at SQL PASS, European PASS, SQL Server Connections, SQLTeach, devLINK, SQLBits, SQL Saturdays, TechFests, Code Camps, SQL in the City, SQL Server user groups, webinars, and other industry seminars, where he shares his 17 years of cumulative knowledge and experience.

In 2009, 2010, and 2011, Brad has made 86 different public presentations to a total of 6,750 attendees in six different countries.

Brad was the founder of the popular community site, [www.SQL-SERVER-PERFORMANCE.COM](http://www.sql-server-performance.com), and operated it from 2000 through 2006, where he wrote over one million words on SQL Server topics.

A well-respected and trusted name in SQL Server literature, Brad is the author or co-author of more than 15 technical books and over 300 published articles. His most recent books include *How to Become an Exceptional DBA* (2nd Edition), *Brad's Sure Guide to SQL Server 2008: The Top Ten New Features for DBAs*, *Mastering SQL Server Profiler*, and *Brad's Sure Guide to SQL Server Maintenance Plans*. These books are available, free, in PDF format at: [HTTP://WWW.SQLSERVERCENTRAL.COM/BOOKS/](http://www.sqlservercentral.com/books/). His blog is at [WWW.BRADMCGEHEE.COM](http://www.bradmcgehee.com).

Foreword

I have attended many SQL Server conferences since 2000, and I have spoken with hundreds of people attending them. One of the most significant trends I have noticed over the past 12 years is the huge number of people who have made the transition from IT Professional or Developer, to SQL Server Database Administrator. In some cases, the transition has been planned and well thought-out. In other cases, it was an accidental transition, when an organization desperately needed a DBA, and the closest warm body was chosen for the job.

No matter the route you took to get there, all DBAs have one thing in common: we have had to learn how to become DBAs through self-training, hard work, and trial and error. In other words, there is no school you can attend to become a DBA; it is something you have to learn on your own. Some of us are fortunate to attend a class or two, or to have a great mentor to help us. However, in most cases, we DBAs become DBAs the hard way: we are thrown into the water and we either sink or swim.

One of the biggest components of a DBA's self-learning process is reading. Fortunately, there are many good books on the basics of being a DBA that make a good starting point for your learning journey. Once you have read the basic books and have gotten some experience under your belt, you will soon want to know more of the details of how SQL Server works. While there are a few good books on the advanced use of SQL Server, there are still many areas that aren't well covered. One of those areas of missing knowledge is a dedicated book on SQL Server execution plans.

That's where *SQL Server Execution Plans* comes into play. It is the first book available anywhere that focuses entirely on what SQL Server execution plans are, how to read them, and how to apply the information you learn from them in order to boost the performance of your SQL Servers.

This was not an easy book to write because SQL Server execution plans are not well documented. Grant Fritchey spent a huge amount of time researching SQL Server execution plans, and conducting original research as necessary, in order to write the material in this book. Once you understand the fundamentals of SQL Server, this book should be on top of your reading list, because understanding SQL Server execution plans is a critical part of becoming an Exceptional DBA.

As you read the book, take what you have learned and apply it to your own unique set of circumstances. Only by applying what you have read will you be able to fully understand and grasp the power of what execution plans have to offer.

Brad M McGehee

Springfield, MO USA 2012

Introduction

Every day, out in the various discussion boards devoted to Microsoft SQL Server, the same types of questions come up repeatedly:

- Why is this query running slow?
- Is SQL Server using my index?
- Why isn't SQL Server using my index?
- Why does this query run faster than this query?
- And so on (and on).

The correct response is probably different in each case, but in order to arrive at the answer you have to ask the same return question every time: have you looked at the execution plan?

Execution plans show you what's going on behind the scenes in SQL Server. They can provide you with a wealth of information on how SQL Server is executing your queries, including the points below.

- Which indexes are getting used, and where no indexes are being used at all.
- How the data is being retrieved, and joined, from the tables defined in your query.
- How aggregations in `GROUP BY` queries are put together.
- The anticipated load, and the estimated cost, that all these operations place upon the system.

All this information makes the execution plan a fairly important tool in the tool belt of database administrator, database developers, report writers, developers, and pretty much anyone who writes T-SQL to access data in a SQL Server database.

Given the utility and importance of the tool, you'd think there'd be huge swathes of information devoted to this subject. To be sure, fantastic information is available from various sources, but there isn't one place to go for focused, practical information on how to use and interpret execution plans.

This is where my book comes in. My goal was to gather into a single location as much useful information on execution plans as possible. I've tried to organize this information in such a way that it provides a clear route through the subject, right from the basics of capturing plans, through their interpretation, and then on to how to use them to understand how you might optimize your SQL queries, improve your indexing strategy, and so on.

Specifically, I cover:

- How to capture execution plans in graphical, as well as text and XML formats.
- A documented method for interpreting execution plans, so that you can create these plans from your own code and make sense of them in your own environment.
- How SQL Server represents and interprets the common SQL Server objects – indexes, views, derived tables and so on, in execution plans.
- How to spot some common performance issues such as **Bookmark Lookups** or unused/missing indexes.
- How to control execution plans with hints, plan guides and so on, and why this is a double-edged sword.
- How XML code appears in execution plans.
- Advanced topics such as parallelism, forced parameterization and plan forcing.

Along the way, I tackle such topics as SQL Server internals, performance tuning, index optimization and so on. However, I focus always on the details of the execution plans, and how these issues are manifest in these plans.

If you are specifically looking for information on how to optimize SQL, or build efficient indexes, then you need a book dedicated to these topics. However, if you want to understand how to interpret these issues within an execution plan, then this is the place for you.

Changes in This Second Edition

This second edition is more evolution than revolution. I spent a lot of time clarifying the descriptions of all of the major plan operators, updating the examples that illustrated how these operators manifest in SQL Server's execution plans, and improving the descriptions of how to read and interpret these plans, in all their guises (graphical, text, and XML).

There is also plenty of new content in here, including coverage of topics such as:

- How to get the cached execution plan for a query, using the Dynamic Management Views (DMVs).
- Expanded coverage of reading XML plans, including how to use XQuery to query cached plans.
- Discussion of the **MERGE** statement and how it manifests in execution plans.
- Expanded coverage of complex data types, to include hierarchical and spatial data as well as XML.
- How to read large-scale plans using XQuery.
- Additional functionality added to SQL Server 2012.

From this point forward, I plan to embark on a program of "continuous improvement," gradually adding new content, and updating existing content for a more complete set of information on SQL Server 2012.

I'll push out updated versions of the eBook, at semi-regular intervals, for you to download and provide your feedback.

Be sure to check out the website for this book:

([HTTP://WWW.SIMPLE-TALK.COM/BOOKS/SQL-BOOKS/SQL-SERVER-EXECUTION-PLANS/](http://www.simple-talk.com/books/sql-books/sql-server-execution-plans/)),
where you can:

- **Find links to download** the latest versions of the eBook and buy the latest printed book.
- **Sign up to receive email notifications** when I release a new eBook version.
- **View the "Change Log,"** describing what has changed in each new version.
- **Most importantly, let me know what you think!** Seriously, hit me with whatever feedback you have. Be honest, brutal...scary even, if necessary. If you provide feedback that makes it into the next edition of the book, you'll receive an Amazon voucher to buy yourself a copy of the latest printed version.

Enjoy the book, and I look forward to hearing from you!

Code Examples

Throughout this book, I'll be supplying T-SQL code that you're encouraged to run for yourself, in order to generate the plans we'll be discussing. From the following URL, you can obtain all the code you need to try out the examples in this book:

[WWW.SIMPLE-TALK.COM/REDGATEBOOKS/GRANTFRITCHEY_SQLSERVEREXECUTIONPLANS_CODE.ZIP](http://www.simple-talk.com/redgatebooks/grantfritchey_sqlserverexecutionplans_code.zip)

I wrote and tested the examples on SQL 2008 Release 2 sample database, **Adventure-Works2008R2**. However, the majority of the code will run on all editions and versions of SQL Server, starting from SQL Server 2005.

Some of the code may not work within SQL Azure, but a large amount of it will. You can get hold of get a copy of **AdventureWorks** from CodePlex:

[HTTP://WWW.CODEPLEX.COM/MSFTDBProdSamples](http://www.codeplex.com/MSFTDBProdSamples).

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, in order 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: Execution Plan Basics

An execution plan, simply put, is the result of the query optimizer's attempt to calculate the most efficient way to implement the request represented by the T-SQL query you submitted.

Execution plans can tell you how SQL Server may execute a query, or how it did execute a query. They are, therefore, the primary means of troubleshooting a poorly performing query. Rather than guess at why a given query is performing thousands of scans, putting your I/O through the roof, you can use the execution plan to identify the exact piece of SQL code that is causing the problem. For example, your query may be reading an entire table-worth of data when, by removing a function in your `WHERE` clause, it could simply retrieve only the rows you need. The execution plan displays all this and more.

The aim of this chapter is to teach you to capture actual and estimated execution plans, in either graphical, text or XML format, and to understand the basics of how to interpret these plans. In order to do this, we'll cover the following topics:

- **A brief backgrounder 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.
- **Actual and estimated execution plans** – What they are and how they differ.
- **Capturing and interpreting the different visual execution plan formats** – We'll investigate graphical, text and XML execution plans.
- **Retrieve execution plans directly from the cache** – Accessing the plan cache through Dynamic Management Objects (DMOs).
- **Automating execution plan capture** – using SQL Server Trace Event.

What Happens When a Query is Submitted?

When you submit a query to SQL Server, a number of processes on the server go to work on that query. The purpose of all these processes is to manage the system such that it will `SELECT`, `INSERT`, `UPDATE` or `DELETE` the data.

These processes kick into action every time we submit a query to the system. While there are many different actions occurring simultaneously within SQL Server, we're going to focus on the processes around queries. The processes for meeting the requirements of queries break down roughly into two stages:

1. Processes that occur in the relational engine.
2. Processes that occur in the storage engine.

In the relational engine, the query is parsed and then processed by the query optimizer, which generates an execution plan. The plan is sent (in a binary format) to the storage engine, which then uses that plan as a basis to retrieve or modify the underlying data. The storage engine is where processes such as locking, index maintenance, and transactions occur. Since **execution plans** are created in the relational engine, that's where we'll be focusing the majority of our attention.

Query parsing

When we pass a T-SQL query to the SQL Server system, the first place it goes to is the relational engine.¹ As the T-SQL arrives, 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

¹ A T-SQL query can be an ad hoc query from a command line or a call to request data from a stored procedure, any T-SQL within a single batch or a stored procedure, or between `GO` statements.

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.

If the T-SQL string is not a data manipulation language (DML) statement, but instead is a data definition language (DDL) query, it will not be optimized because, for example, there is only one "right way" for the SQL Server system to create a table; therefore, there are no opportunities for improving the performance of that type of statement.

Algebrizer

If the T-SQL string is a DML statement and it has parsed correctly, the parse tree is passed to a process called the **algebrizer**. The algebrizer resolves all the names of the various objects, tables and columns, referred to within the query string. The algebrizer 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 `GROUP BY`, 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. As an example, the algebrizer would quickly find the table `Person.Person` in the `AdventureWorks2008R2` database. However, the `Product.Person` table, which doesn't exist, would cause an error and the whole optimization process would stop.

The algebrizer outputs a binary called the **query processor tree**, which is then passed on to the **query optimizer**. The algebrizer's output includes a hash, a coded value representing the query. The optimizer uses the hash to determine whether there is already a plan generated and stored in the plan cache. If there is a plan there, the process stops here and that plan is used. This reduces all the overhead required by the query optimizer to generate a new plan.

The query optimizer

The query optimizer is essentially a piece of software that "models" the way in which the database relational engine works. The most important pieces of data used by the optimizer are statistics, which SQL Server generates and maintains against indexes and columns, explicitly for use by the optimizer. Using the **query processor tree** and the **statistics** it has about the data, the optimizer applies the model in order to work out what it thinks will be the optimal way to execute the query – that is, it generates an execution plan.

In other words, the optimizer figures out how best to implement the request represented by the T-SQL query you submitted. It decides if it can access the data through indexes, what types of joins to use and much more. The decisions made by the optimizer are based on what it calculates to be the cost of a given execution plan, in terms of the required CPU processing and I/O. Hence, this is a **cost-based** plan.

The optimizer will generate and evaluate many plans (unless there is already a cached plan) and, generally speaking, will choose the lowest-cost plan, that is, the plan it thinks will execute the query as fast as possible and use the least amount of resources, CPU and I/O. The calculation of the execution cost is the most important calculation, and the optimizer will use a process that is more CPU-intensive if it returns results that much faster. Sometimes, the optimizer will settle for a less efficient plan if it thinks it will take more time to evaluate many plans than to run a less efficient plan. The optimizer doesn't find the best possible plan. The optimizer finds the plan with the least cost in the shortest possible number of iterations, meaning the least amount of time within the processor.

If you submit a very simple query – for example, a **SELECT** statement against a single table with no aggregates or calculations within the query – then, rather than spend time trying to calculate the absolute optimal plan, the optimizer will simply apply a **trivial plan** to these types of queries. For example, a query like the one in Listing 1.1 would create 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. If the query is non-trivial, the optimizer will perform a cost-based calculation to select a plan. In order to do this, it relies on the statistics that by SQL Server maintains.

Statistics are collected on columns and indexes within the database, and describe the data distribution and the uniqueness, or selectivity, of the data. 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, so it relies on statistics, a sample of the data that provides a mathematical construct of the data used by the optimizer to represent the entire collection of data. The reliance the optimizer has on statistics means that these things need to be as accurate as possible or the optimizer could make poor choices for the execution plans it creates.

The information that makes up statistics is represented by a **histogram**, a tabulation of counts of the occurrence of a particular value, taken from 200 data points evenly distributed across the data. It's this "data about the data" that provides the information necessary for the optimizer to make its calculations.

If statistics exist for a relevant column or index, then the optimizer will use them in its calculations. The optimizer will examine the statistics to determine if the index supplies a sufficient level of selectivity to act as assistance for the query in question. Selectivity is how unique the data is across the whole set of the data. The level of selectivity required to be of assistance for an index is quite high, usually with x% of unique values required in most instances.

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 ON clause. Table variables do not ever have statistics generated on them, so the optimizer always assumes they contain a single row, regardless of their actual size.

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 use these statistics.

The optimizer takes these statistics, along with the query processor tree, and heuristically determines the best plan. This means that it works through a series of plans, testing different methods of accessing data, attempting different types of join, rearranging the join order, trying different indexes, and so on, until it arrives at what it thinks will be the least cost plan. 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.

It's important to note that the estimated cost is just that – an estimate. Given an infinite amount of time and complete, up-to-date statistics, the optimizer would find the perfect plan for executing the query. However, it attempts to calculate the best plan it can in the least amount of time possible, and is limited by the quality of the statistics it has available. Therefore, these cost estimations are very useful as measures, but are unlikely to reflect reality precisely.

Once the optimizer arrives at an execution plan, the estimated plan is created and stored in a memory space known as the **plan cache** – although this is all different if a plan already exists in cache (more on this shortly, in the section on *Execution Plan Reuse*). As stated earlier, if the optimizer finds a plan in the cache that matches the currently executing query, this whole process is short-circuited.

Query execution

Once the optimizer has generated an execution plan, or retrieved one from cache, the action switches to the storage engine, which usually executes the query according to the plan.

We will not go into detail here, except to note that the carefully generated execution plan may be subject to *change* during the actual execution process. For example, this might happen if:

- SQL Server determines that the plan exceeds the threshold for a parallel execution (an execution that takes advantage of multiple processors on the machine – more on parallel execution in Chapter 3)
- the statistics used to generate the plan were out of date, or have changed since the original execution plan was created
- processes or objects within the query, such as data inserts to a temporary table, result in a recompilation of the execution plan.

Any one of these could change the estimated execution plan.

SQL Server returns the results of the query after the relational engine changes the format to match that requested in the submitted T-SQL statement, assuming it was a `SELECT`.

Estimated and Actual Execution Plans

As discussed previously, there are two distinct types of execution plan. First, there is the plan that represents the output from the optimizer. This is an **estimated execution plan**. The operators, or steps, within the plan are logical steps, because they're representative of the optimizer's view of the plan and don't represent what physically occurs when the query runs.

Next is the plan that represents the output from the actual query execution. This type of plan is, funnily enough, the **actual execution plan**. It shows data representing what actually happened when the query executed.

These plans represent distinctly different sets of data, but can look largely the same. Most of the time, the same operators with the same costs will be present in both plans. There are occasions where, usually due to recompiles, SQL Server will drop a plan from the plan cache and recreate it, and these versions can differ greatly. The cause is usually changes in statistics, or other changes that occur as the storage engine processes the queries. We'll discuss this issue in more detail a little later in the chapter.

Estimated plans are the types of plans stored in the plan cache, so this means that we can access the data available in actual execution plans only by capturing the execution of a query. Since estimated plans never access data, they are very useful for large, complex queries that could take a long time to run. Actual execution plans are preferred because they show important execution statistics such as the number of rows accessed by a given operator. In general, this additional information makes actual execution plans the one you use the most, but estimated plans are extremely important, especially because that's what you get from the plan cache.

Execution Plan Reuse

It is expensive for the server to go through all the processes described above that are required to generate execution plans. While SQL Server can do all this in less than a millisecond, depending on the query it can take seconds or even minutes to create an execution plan, so SQL Server will keep and reuse plans wherever possible in order to reduce that overhead. As they are created, plans are stored in a section of memory called the **plan cache** (prior to SQL Server 2005 this was called the **procedure cache**).

When we submit a query to the server, the algebraizer process creates a hash, like a coded signature, of the query. The hash is a unique identifier; its nickname is the *query fingerprint*. With an identifier that is unique for any given query, including all the text that defines the query, including spaces and carriage returns, the optimizer compares the hash to queries in the cache. If a query exists in the cache that matches the query coming into the engine, the entire cost of the optimization process is skipped and the execution plan in the plan cache is reused.

This is one of the strengths of SQL Server, since it reduces the expense of creating plans. It is a major best practice to write queries in such a way that SQL Server can reuse their plans. To ensure this reuse, it's best to use either stored procedures or parameterized queries. Parameterized queries are queries where the variables within the query are identified with parameters, similar to a stored procedure, and these parameters are fed values, again, similar to a stored procedure.

If, instead, variables are hard coded, then the smallest change to the string that defines the query can cause a cache miss, meaning that SQL Server did not find a plan in the cache (even though, with parameterization, there may have existed a perfectly suitable one) and so the optimization process is fired and a new plan created. It is possible to get a look at the query hash and use it for some investigation of performance (more on this in the section on DMOs).

SQL Server does not keep execution plans in memory forever. They are slowly aged out of the system using an "age" formula that multiplies the estimated cost of the plan by the number of times it has been used (e.g. a plan with an estimated cost of 10 that has been referenced 5 times has an "age" value of 50). 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.

If the following criteria are met, the plan is removed from memory:

- more memory is required by the system
- the "age" of the plan has reached zero
- the plan isn't currently being referenced by an existing connection.

Execution plans are not sacrosanct. Certain events and actions can cause a plan to be recompiled. It is important to remember this, because recompiling execution plans can be a very expensive operation. The following actions can lead to recompilation of an execution plan:

- changing the structure or schema of a table referenced by the query
- changing an index used by the query
- dropping an index used by the query
- updating the statistics used by the query
- calling the function, `sp_recompile`
- subjecting the keys in tables referenced by the query to a large number of Inserts or Deletes (which leads to statistics changes)
- for tables with triggers, significant growth of the **inserted** or **deleted** tables
- mixing DDL and DML within a single query, often called a deferred compile
- changing the SET options within the execution of the query

- changing the structure or schema of temporary tables used by the query
- changes to dynamic views used by the query
- changes to cursor options within the query
- changes to a remote rowset, like in a distributed partitioned view
- when using client-side cursors, if the FOR BROWSE options are changed.

Clearing Plans from the Plan Cache

Since the cache plays such an important role in how execution plans operate, you need a few tools for querying and working with the plan cache. First off, while testing, you may want to see how long a plan takes to compile, or to investigate how minor adjustments might create slightly different plans. To clear the cache, run this:

```
DBCC FREEPROCCACHE
```

Listing 1.2

WARNING: Clearing the cache in a production environment

Running Listing 1.2 in a production environment will clear the cache for all databases on the server. That can result in a significant performance hit because SQL Server must then recreate every single plan stored in the plan cache, as if the plans were never there and the queries were being run for the first time ever.

While working with an individual query, it's usually better to target that query to remove just it from the plan cache. You can use `DBCC FREEPROCCACHE` and pass either the `sql_handle` or `plan_handle` to remove just the referenced plan. The `plan_handle` and `sql_handle` are available from various DMO objects (see the section on DMOs).

Execution Plan Formats

While SQL Server produces a single execution plan for a given query, we can view it in three different ways:

- as graphical plans
- as text plans
- as XML plans.

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

Graphical plans

Graphical plans are the most commonly used type 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.

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.

There are three text plan formats:

- **SHOWPLAN_ALL** – A reasonably complete set of data showing the estimated execution plan for the query.
- **SHOWPLAN_TEXT** – Provides a very limited set of data for use with tools like **osql.exe**. It, too, only shows the estimated execution plan
- **STATISTICS PROFILE** – Similar to **SHOWPLAN_ALL** except it represents the data for the actual execution plan.

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 co-workers. Using XQuery, we can also query the XML data directly. Every graphical execution plan is actually XML under the covers. XML is very hard to read, so, useful though these types of plan are, you're more likely to use the text or graphical plans for simply browsing the execution plan.

There are two varieties of XML plan:

- **SHOWPLAN_XML** – The plan generated by the optimizer prior to execution.
- **STATISTICS_XML** – The XML format of the actual execution plan.

Getting Started

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.

Please note that occasionally, especially when we move on to more complex plans, the plan that you see, if you follow along by executing the relevant script (all scripts are available in the code download for this book) may differ slightly from the one presented in the book. This might be because we are using different versions of SQL Server (different SP levels and hot fixes), or we are using slightly different versions of the AdventureWorks database, or because of how the AdventureWorks database has been altered over time as each of us has played around in it. So, while most of the plans you get should be very similar to what we display here, don't be too surprised if you try the code and see something different.

Permissions required to view execution plans

In order to generate 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 in be 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 DMOs, either `VIEW SERVER STATE` or `VIEW DATABASE STATE`, depending on the DMO in question, will be required.

Working with graphical execution plans

In order to focus on the basics of capturing estimated and actual execution plans, the first query will be one of the simplest possible queries, and we'll build from there. Open up Management Studio and in a query window, type the following:

```
SELECT *  
FROM dbo.DatabaseLog;
```

Listing 1.4

Getting the estimated plan

We'll start by viewing the graphical **estimated execution plan** that the query optimizer generated, so there's no need to run the query yet.

We can find out what the optimizer estimates to be the least costly plan in one of following ways:

- Click on the **Display Estimated Execution Plan** icon on the tool bar.
- Right-click the query window and select the same option from the menu.
- Click on the **Query** option in the menu bar and select the same choice.
- Hit CTRL+L on the keyboard.

I tend to click the icon more often than not but, either way, we see our very first **estimated execution plan**, as in Figure 1.1.

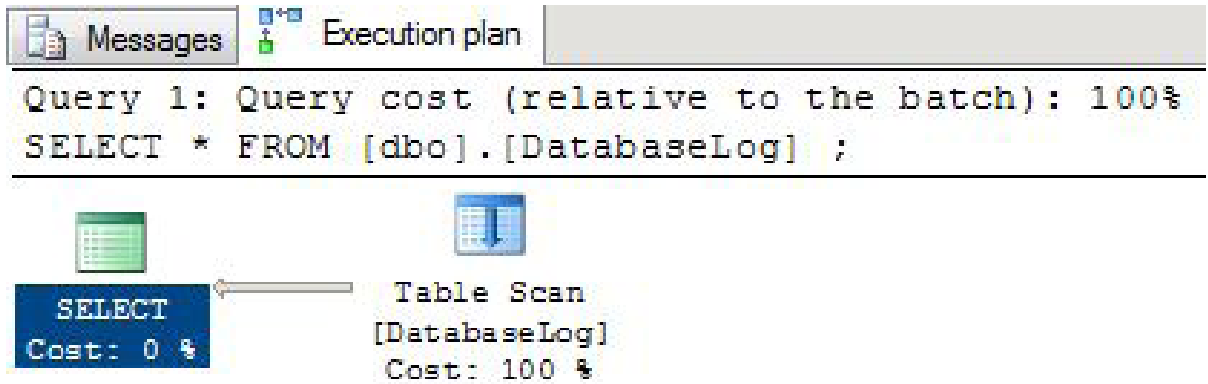


Figure 1.1

Visually, there's no easy way to tell the difference between an estimated plan and an actual plan. The differences are in the underlying data, which we'll be covering in some detail throughout the book.

We'll explain what this plan represents shortly, but first, let's capture the actual execution plan.

Getting the actual plan

Actual execution plans, unlike estimated execution plans, do not represent the calculations of the optimizer. Instead, this execution plan shows exactly how SQL Server executed the query. The two will often be identical but will sometimes differ, due to changes to the execution plan made by the storage engine, as we discussed earlier in the chapter.

As with estimated execution plans, there are several ways to generate our first graphical actual execution plan:

- click on the icon on the tool bar called **Include Actual Execution Plan**
- right-click within the query window and choose the **Include Actual Execution Plan** menu item
- choose the same option in the **Query** menu choice
- type CTRL+M.

Each of these methods acts as an "On" switch and SQL Server will then create an execution plan for all queries run from that query window, until you turn it off again.

So, turn on actual execution plans by your preferred method and execute the query. You should see an execution plan like the one in Figure 1.2.

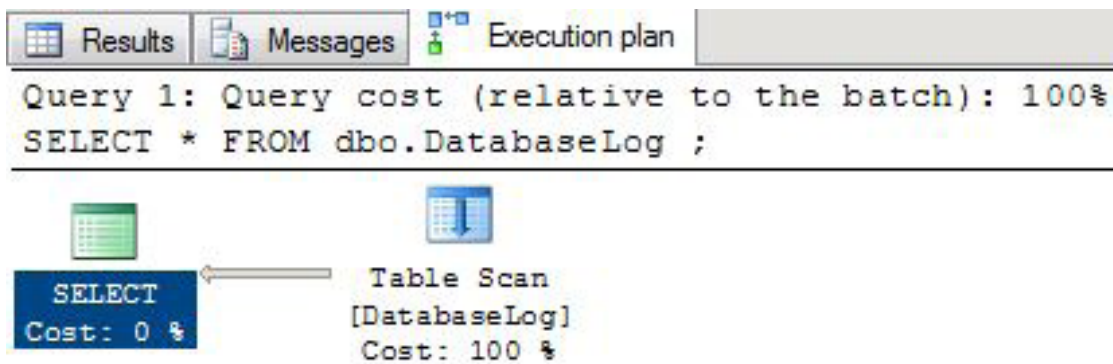


Figure 1.2

In this simple case, the actual plan is visually identical to the estimated plan.

Interpreting graphical execution plans

The icons you see in Figures 1.1 and 1.2 represent the first two of approximately 79 operators that represent various actions and decisions that potentially make up an execution plan. On the left side of the plan is the **Select** operator, an operator that you'll see quite often. Not only will you see this operator, but you'll also frequently reference it for the important data it contains. It's the final result, and formatting, from the relational engine. The icon on the right represents a **Table Scan**.² This is one of the easiest operators to look for when trying to track down possible causes of performance problems.

Each operator has a logical and a physical component. They are frequently the same, but when looking at an estimated plan, you are only seeing logical operators. When looking at an actual plan, you are only seeing physical operators, laid out in the logical processing order in which SQL Server will retrieve the information defined by the query. This means that, logically, we read the plans from the left to the right. In the example above, the logical order is the definition of the **SELECT** criteria followed by the **Scan** operator.

However, you're going to find that you will generally read the execution plan the other direction, going from right to left. This is not because the execution plans are laid out "incorrectly." It's just because the physical order of operations is frequently easier to understand than the logical order of operations. The basic process is to pull data, not to push it, so the execution plan represents this pulling of the data.

You'll note that there is an arrow pointing between the two icons. This arrow represents the data passed between the operators, as represented by the icons. In this case, if we read the execution plan in the direction of the data flow, the physical direction, from right to left, we have a **Table Scan** operator producing the result set, which passes to the **Select** operator. The direction of the arrow emphasizes further the direction of data flow.

² A **Table Scan** occurs when a query forces the storage engine to walk through a heap (a table without a clustered index), row by row, either returning everything, or searching everything to identify the appropriate rows to return to the user. In our case, the scan returns everything because we're not using a **WHERE** clause and we're not hitting a covering index (an index that includes all the columns referred to in the query for a given table). As you might imagine, as the number of rows in the table grows, this operation gets more and more expensive.

The thickness of the arrow reflects the amount of data passed, a thicker arrow meaning more rows. This is another visual clue as to where performance issues may lie. You can hover with the mouse pointer over these arrows and it will show the number of rows that it represents in a ToolTip.

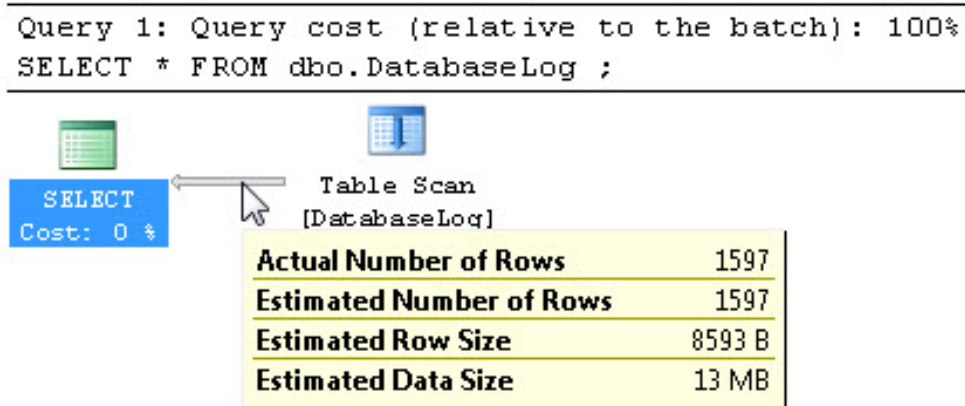


Figure 1.3

For example, if your query returns two rows, but the execution plan shows a big thick arrow between some of the operators early in the plan indicating many rows being processed, only to narrow to a thin arrow at the end, right before the **Select** operator, then that's something to possibly investigate.

Below each icon is displayed a number as a percentage. This number represents the estimated **relative cost** to the query for that operator. That cost, returned from the optimizer, is the estimated execution time for that operation in seconds. Understand, though, that execution time is not a representation of your system or any other actual system. The story related to me is that the developer tasked with creating execution plans in SQL Server 2000 used his workstation as the basis for these numbers, and they have never been updated. Just think about them as units of cost, only of significance to the optimizer, rather than any type of real measure. Through the rest of the book, I will refer to these numbers as costs because most documentation refers to them in this way. In our

case, all the estimated cost is associated with the Table Scan. While a cost may be represented as 0% or 100%, remember that, as these are percentages, not actual numbers, even an operator displaying 0% will have a small associated cost.

Above the icons is displayed as much of the query string as will fit into the window, and a "cost (relative to batch)" of 100%. You can see this in Figure 1.3. 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. They will then show up as different costs as a part of the whole. These costs are also based on estimates and therefore must be interpreted with an eye towards what's actually happening within the plans represented, not simply assuming the number as valid.

ToolTips

Associated with it, each of the icons and the arrows has a pop-up window called a **ToolTip**, which you can access by hovering your mouse pointer over the icon.

Using the query above, pull up the estimated execution plan as outlined previously. Hover over the **Select** operator, and you should see the ToolTip window shown in Figure 1.4.

SELECT	
Cached plan size	16 B
Degree of Parallelism	1
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.554668
Estimated Number of Rows	1597
Statement	
SELECT *	
FROM dbo.DatabaseLog ;	

Figure 1.4

Here we get the numbers generated by the optimizer on the following:

- **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 used multiple processors. This plan uses a single processor as shown by the value of 1.
- **Estimated Operator Cost** – We've already seen this as the percentage cost in Figure 1.1.
- **Estimated Subtree Cost** – Tells us the accumulated optimizer cost assigned to this step and all previous steps, but remember to read from right to left. This number is meaningless in the real world, but is a mathematical evaluation used by the query optimizer to determine the cost of the operator in question; it represents the estimated cost that the optimizer thinks this operator will take.
- **Estimated Number of Rows** – Calculated based on the statistics available to the optimizer for the table or index in question.

In Figure 1.4, we also see the **Statement** that represents the entire query that SQL Server is processing.

If we look at the ToolTip information for the next operator in the plan, the **Table Scan**, we see the information in Figure 1.5. Each of the different operators will have a distinct set of data. The operator in Figure 1.5 is performing work of a different nature than the **Select** operator in Figure 1.4, and so we get a different set of details.

First are listed the **Physical Operation** and **Logical Operation**. The logical operators are the results of the optimizer's calculations for what should happen when the query executes. The physical operators represent what actually occurred. The logical and physical operators are usually the same, but not always – more on that in Chapter 2.

Table Scan	
Scan rows from a table.	
Physical Operation	Table Scan
Logical Operation	Table Scan
Estimated I/O Cost	0.552755
Estimated CPU Cost	0.0019137
Estimated Number of Executions	1
Estimated Operator Cost	0.554668 (100%)
Estimated Subtree Cost	0.554668
Estimated Number of Rows	1597
Estimated Row Size	8569 B
Ordered	False
Node ID	0
Object	
[AdventureWorks2008R2].[dbo].[DatabaseLog]	
Output List	
[AdventureWorks2008R2].[dbo].	
[DatabaseLog].DatabaseLogID,	
[AdventureWorks2008R2].[dbo].	
[DatabaseLog].PostTime, [AdventureWorks2008R2].	
[dbo].[DatabaseLog].DatabaseUser,	
[AdventureWorks2008R2].[dbo].[DatabaseLog].Event,	
[AdventureWorks2008R2].[dbo].[DatabaseLog].Schema,	
[AdventureWorks2008R2].[dbo].[DatabaseLog].Object,	
[AdventureWorks2008R2].[dbo].[DatabaseLog].TSQL,	
[AdventureWorks2008R2].[dbo].	
[DatabaseLog].XmlEvent	

Figure 1.5

After that, we see the estimated costs for I/O, CPU, operator and subtree. Just like in the previous ToolTip, the subtree cost is simply the section of the execution tree that we have looked at so far, working again from right to left, and top to bottom. SQL Server bases all estimations on the statistics available on the columns and indexes in any table.

The I/O cost and CPU cost are not actual values, but rather the estimated cost numbers assigned by the query optimizer during its calculations. These numbers can be useful when considering whether most of the estimated cost is I/O-based (as in this case), or if we're potentially putting a load on the CPU. A bigger number means that SQL Server might use more resources for this operation. Again, these are not hard and absolute measures, but rather estimated values that help to suggest where the actual cost in a given operation may lie.

You'll note that, in this case, the operator cost and the subtree cost are the same, since the **Table Scan** is the only significant operator, in terms of the work done to execute the query. For more complex trees, with more operators, you'll see that the subtree cost accumulates as the individual cost for each operator is added to the total. You get the full cost of the plan from the final operation in the query plan, in this case, the **Select** operator.

Again, we see the estimated number of rows. The plan displays this number for each operation because each operation is dealing with different sets of data. When we get to execution plans that are more complicated, we'll see the number of rows change as various operators perform their work on the data as it passes between each operator. Knowing how each operator adds rows, or filters out rows, will help us to understand the execution process for that query.

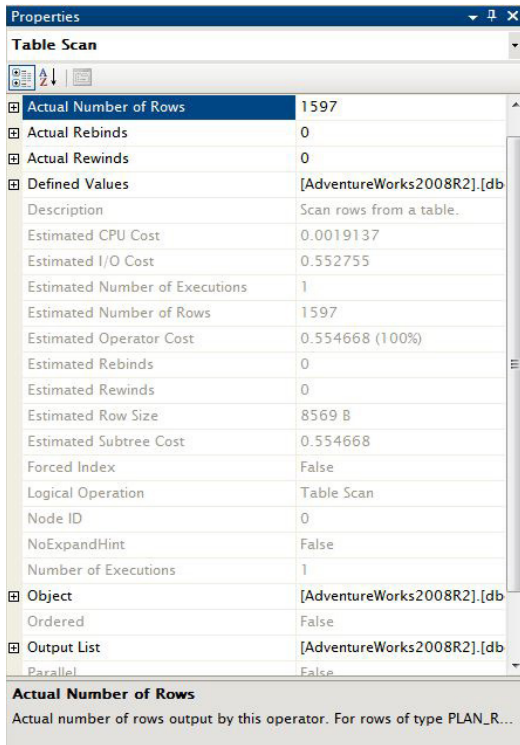
Another important piece of information, when attempting to troubleshoot performance issues, is the Boolean value displayed for **Ordered**; in this case, as shown in Figure 1.5, this is **False**. This tells us whether the data that this operator is working with is in an ordered state. Certain operations, for example, an **ORDER BY** clause in a **Select** statement, may require data to be in a particular order, sorted by a particular value or set of values. Knowing whether the data is in an **Ordered** state helps show where extra processing may be occurring to get the data into that state.

Finally, **Node ID** is the ordinal, which simply means numbered in order, of the node itself. When the optimizer generates a plan, it numbers the operations in the *logical* order of operations.

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 using excessive amounts of CPU or tables that need more indexes, or identify other performance issues.

Operator properties

Even more information is available than that presented in the ToolTips. 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. Figure 1.6 shows the details from the original **Table Scan**.



Properties	
Table Scan	
Actual Number of Rows	1597
Actual Rebinds	0
Actual Rewinds	0
Defined Values	[AdventureWorks2008R2].[db
Description	Scan rows from a table.
Estimated CPU Cost	0.0019137
Estimated I/O Cost	0.552755
Estimated Number of Executions	1
Estimated Number of Rows	1597
Estimated Operator Cost	0.554668 (100%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	8569 B
Estimated Subtree Cost	0.554668
Forced Index	False
Logical Operation	Table Scan
Node ID	0
NoExpandHint	False
Number of Executions	1
Object	[AdventureWorks2008R2].[db
Ordered	False
Output List	[AdventureWorks2008R2].[db
Parallel	False
Actual Number of Rows	
Actual number of rows output by this operator. For rows of type PLAN_R...	

Figure 1.6

Some of this information should be familiar from the ToolTip, but most of it is new. I'm not going to detail all the properties and their meanings. However, I do want to point out how this window works, so that you can use it to investigate your own execution plans later. You'll note that some of the properties, such as the **Object** property, have a plus sign on the left. This means that we can expand them to display further information. Many, but not all of the properties will have a description on the bottom of the screen, as you can see in Figure 1.6 for the **Actual Number of Rows** property. Other properties, not shown in this example, have an ellipsis, which allows us to open a new window with more details on the given property.

Working with text execution plans

The graphical execution plans are very useful because they're so easy to read. However, much of the data about the operators is not immediately visible; we can see some of it, in a limited form, in the ToolTip windows, and the complete set is available in the **Properties** window. Wouldn't it be great if there were a way to see all that information at once?

In the case of large queries, with incredibly complex plans or large number of batch statements, wouldn't it be handy to be able to search through for particular bits of information, **Table Scans** or the highest operator cost or something? Well, you can. Two methods exist: Text Execution Plans and XML Execution Plans.

Getting the estimated text plan

Please note, before we start, that text plans are on the list for deprecation at some release of SQL Server ([HTTP://MSDN.MICROSOFT.COM/EN-US/LIBRARY/MS190233\(V=SQL.105\).ASPX](http://msdn.microsoft.com/en-us/library/ms190233(v=SQL.105).aspx)). This section of the book is included for backwards compatibility only. Ideally, you'll use the XML execution plans instead.

To turn on the text version of the estimated text execution plan, simply issue the following command in Listing 1.5 at the start of a query.

```
SET SHOWPLAN_ALL ON;
```

Listing 1.5

It's important to remember that, with `SHOWPLAN_ALL` set to `ON`, SQL Server collects execution information for all subsequent T-SQL statements, but those statements are not actually executed. Hence, we get the estimated plan. It's very important to remember to turn `SHOWPLAN_ALL` off after you have captured the information you require. If you forget and execute any DML or DDL statements, they will not execute.

To turn `SHOWPLAN_ALL` off, simply issue:

```
SET SHOWPLAN_ALL OFF;
```

Listing 1.6

We can also use the equivalent commands for `SHOWPLAN_TEXT`. The text-only show plan is meant for use with tools like **osql.exe**, where the result sets can be readily parsed and stored by a tool dealing with text values, as opposed to actual result sets, as the `SHOWPLAN_ALL` function does.

Getting the actual text plan

In order to turn the text version of the actual execution plan on or off, use the code in Listings 1.7 and 1.8, respectively.

```
SET STATISTICS PROFILE ON;
```

Listing 1.7

```
SET STATISTICS PROFILE OFF;
```

Listing 1.8

Interpreting text plans

This section provides only a brief overview of reading text plans; we'll return to this topic in much more detail in Chapter 3. We'll stick with the same basic query we used when discussing graphical plans, so execute the code in Listing 1.9.

```
SET STATISTICS PROFILE ON;
GO
SELECT *
FROM [dbo].[DatabaseLog];
GO
SET SHOWPLAN_ALL OFF;
GO
```

Listing 1.9

When you execute this query, the results pane shows actual plan. Figure 1.7 shows the first columns of the results.

	Rows	Executes	StmtText
1	1597	1	SELECT * FROM [dbo].[DatabaseLog];
2	1597	1	[-Table Scan(OBJECT:([AdventureWorks2008R2].[dbo].[DatabaseLog]))]

Figure 1.7

Figure 1.7 shows just the first three columns of the information returned, partly for readability, and partly to show that the basic information portrayed in a graphical plan is readily available in a text plan. The **StmtText** column shows the operators and, through indentation, shows how the data logically flows up, towards the top, similar to how the data flows to the left in a graphical plan. The first row is the submitted **SELECT** statement. The rows following are the physical operations occurring within the query plan. In our case, that means one row, i.e. the **Table Scan**.

While reading a text plan, we can very easily see many of the operations, almost a glance. This can make it simple to look through highly complex plans for **Scan** operations or other operations that are flags for performance problems. As each row is visible and we can scroll between the columns, it's very easy to see an operator and then look at some of the details for that operator. However, because of how the tabs and lines work, it's much more difficult, especially in a complex plan, to understand the neat flow of the data from one operator to the next.

In addition to the first column, the text plan displays, in a series of columns, the details hidden in the **ToolTip** or in the **Properties** window, for graphical plans. Most of the information that you're used to seeing is here, plus a little more. So, while the **NodeId** was available in the graphical plan, because of the nature of the graphical plan, nothing was required to identify the parent of a given node. In the `SHOWPLAN_ALL` we get a column showing the Parent **NodeId**. As you scan right on the full plan, assuming you're trying these out as we go, you'll see many other familiar columns, such as the `TotalSubTreeCost`, `EstimateRows` and so on. Some of the columns are harder to read, such as the **Defined List** (the values or columns introduced by this operation to the data stream, the output of the operator), which is displayed as just a comma-separated list in a column in the results, but overall the information is readily available.

The advantage of working with a text plan is the ability to see an operator and then scroll to the right to see the details for that operator. You also get to see the details for the operators before and after that one, which can make understanding what a plan is doing easier in some cases. In addition, with text plans, instead of outputting the data to a grid, you can output to text using the options available in SSMS (SQL Server Management Studio). Then, once it's text data, you can perform searches on the data as you would any other set of text on your computer, even saving it out to a file and opening it in an editor or even Word.

I prefer to work primarily with graphical plans and XML plans, but the text plan has its place and uses too, so don't dismiss it from your toolbox.

Working with XML execution plans

As discussed earlier, all graphical plans are XML under the covers. In addition, the plans stored in cache are also stored as XML. Storing plans in XML opens up several capabilities. First, it makes it very easy to make a copy of a plan in order to share it with someone who may be helping you solve any issues evidenced within the plan. Further, and this is the real strength of having XML available to us for execution plans, we can use the XQuery language to run queries directly against the execution plan and against plans in cache.

Getting the actual and estimated XML plans

In order to turn XML estimated execution plans on and off, use:

```
SET SHOWPLAN_XML ON
...
SET SHOWPLAN_XML OFF
```

Listing 1.10

As for `SHOWPLAN_ALL`, the `SHOWPLAN_XML` command is essentially an instruction not to execute any T-SQL statements that follow, but instead to collect execution plan information for those statements, in the form of an XML document. Again, it's important to turn `SHOWPLAN_XML` off as soon as you have finished collecting plan information, so that subsequent T-SQL executes as intended.

For the XML version of the actual plan, use:

```
SET STATISTICS XML ON
...
SET STATISTICS XML OFF
```

Listing 1.11

Interpreting XML plans

Once again, let's look at the same execution plan as we evaluated with the text plan.

```
SET SHOWPLAN_XML ON;  
GO  
SELECT *  
FROM [dbo].[DatabaseLog];  
SET SHOWPLAN_XML OFF;  
GO
```

Listing 1.12

Figure 1.8 shows the result, in the default grid mode.

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

Figure 1.8

The link is a pointer to an XML file located here:

```
\Microsoft SQL Server\90\Tools\Binn\schemas\sqlserver\2003\03\  
showplan\showplanxml.xsd
```

Clicking on this link opens the execution plan as a graphical plan. In order to view the XML directly, you must right-click on the graphical plan and select **Show Execution Plan XML** from the context menu. This opens the XML format in a browser window within the SSMS. You can view the output from SHOWPLAN_XML in text, grid or file (default is grid). You can change the output format from the **Query | Results To** menu option.

A lot of information is put at your fingertips with XML plans – much of which we won't encounter here with our simple example, but will get to in later, more complex, plans.

Nevertheless, even this simple plan will give you a good feel for the XML format. You will not need to read XML directly to learn how to read execution plans. It's just an additional tool available to you.

The results, even for our simple query, are too large to output here. I'll go over them by reviewing various elements and attributes. The full definition schema as defined by Microsoft is available at: [HTTP://SCHEMAS.MICROSOFT.COM/SQLSERVER/2004/07/.SHOWPLAN/](http://schemas.microsoft.com/sqlserver/2004/07/showplan/) (this location has not changed since SQL Server 2005).

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, like all the other execution plans we've reviewed so far, we see the query in question as part of the `StmtSimple` element. Within that, we receive a list of attributes of the statement itself, and some physical attributes of the `QueryPlan`.

```
<StmtSimple StatementText="SELECT *&#xD;&#xA; FROM [dbo].[DatabaseLog];  
  " StatementId="1" StatementCompId="1" StatementType="SELECT"  
  StatementSubTreeCost="0.108154" StatementEstRows="389"  
  StatementOptmLevel="TRIVIAL">  
  <StatementSetOptions QUOTED_IDENTIFIER="false" ARITHABORT="true"  
    CONCAT_NULL_YIELDS_NULL="false" ANSI_NULLS="false"  
    ANSI_PADDING="false" ANSI_WARNINGS="false"  
    NUMERIC_ROUNDABORT="false" />  
  <QueryPlan CachedPlanSize="9" CompileTime="31" CompileCPU="19"  
    CompileMemory="88">
```

Listing 1.13

Notice that the optimizer has chosen a trivial execution plan, as we might expect given the simplicity of this query. Information such as the `CachedPlanSize` will help you to determine the size of your query plans and can help determine which plans are causing the most memory pressure on your system.

After that, we have the `RelOp` element, which provides the information we're familiar with, regarding a particular operation, in this case the **Table Scan**.

```
<RelOp NodeId="0" PhysicalOp="Table Scan" LogicalOp="Table Scan"
  EstimateRows="451" EstimateIO="0.126829"
  EstimateCPU="0.0006531" AvgRowSize="8593"
  EstimatedTotalSubtreeCost="0.127482" Parallel="0"
  EstimateRebinds="0" EstimateRewinds="0">
```

Listing 1.14

Not only is there more information than in the text plans, but it's also much more readily available and easier to read than in the text plans. It is possible to get at more information quickly than in the graphical plans (although the flow through the graphical plans is much easier to read). For example, a problematic column, like the Defined List mentioned earlier, that is difficult to read, becomes the `OutputList` element with a list of `ColumnReference` elements, each containing a set of attributes to describe that column (Listing 1.15).

```
<OutputList>
  <ColumnReference Database="[AdventureWorks]" Schema="[dbo]"
    Table="[DatabaseLog]" Column="DatabaseLogID" />
  <ColumnReference Database="[AdventureWorks]" Schema="[dbo]"
    Table="[DatabaseLog]" Column="PostTime" />
  <ColumnReference Database="[AdventureWorks]" Schema="[dbo]"
    Table="[DatabaseLog]" Column="DatabaseUser" />
  <ColumnReference Database="[AdventureWorks]" Schema="[dbo]"
    Table="[DatabaseLog]" Column="Event" />
  <ColumnReference Database="[AdventureWorks]" Schema="[dbo]"
    Table="[DatabaseLog]" Column="Schema" />
  <ColumnReference Database="[AdventureWorks]" Schema="[dbo]"
    Table="[DatabaseLog]" Column="Object" />
  <ColumnReference Database="[AdventureWorks]" Schema="[dbo]"
    Table="[DatabaseLog]" Column="TSQL" />
  <ColumnReference Database="[AdventureWorks]" Schema="[dbo]"
    Table="[DatabaseLog]" Column="XmlEvent" />
</OutputList>
```

Listing 1.15

This makes XML not only easier to read, but much more readily translated directly back to the original query. You know what columns you selected from which table, residing on which schema, because you defined it in the original query. Here, that information is now in exactly the same format.

Back to the plan, after the `RelOp` element referenced above we have the **Table Scan** element:

```
<TableScan Ordered="0" ForcedIndex="0" NoExpandHint="0">
```

Listing 1.16

This is followed by a list of defined values that lays out the columns referenced by the operation.

```
<DefinedValues>
  <DefinedValue>
    <ColumnReference Database="[AdventureWorks]" Schema="[dbo]"
      Table="[DatabaseLog]" Column="DatabaseLogID" />
  </DefinedValue>
  <DefinedValue>
    <ColumnReference Database="[AdventureWorks]"
... (Output Cropped)...
```

Listing 1.17

Simply sitting and reading XML can be extremely difficult, especially, as I already noted, trying to understand which operation comes next in the plan. However, you can clearly see that the information is stored in all execution plans (SQL Server 2005 and greater) and so it's available to you to read or to query against using XQuery. This makes understanding what's available to you in the XML extremely valuable.

Saving XML plans as graphical plans

We can save the execution plan generated with `SHOWPLAN_XML` without opening it, by right-clicking within the results and selecting **Save As**. In SQL Server 2005, we then have to change the filter to `"*.*)" and, when typing the name of the file we want to save, adding the extension, ".sqlplan."` SQL Server 2008, and later, automatically selects the **.sqlplan** file type. This is how the Books Online recommends saving an XML execution plan. In fact, what we get when we save it this way is actually a **graphical execution plan** file. 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.

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.

In order to save an XML plan as XML, first we need to open the results into the XML window. 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 go to the place where the plan is stored, as defined above, and copy it.

Retrieving Plans from the Cache Using Dynamic Management Objects

SQL Server 2005 introduced a new area of functionality called Dynamic Management Objects (DMOs). These are system views and functions that expose internal information describing what SQL Server is doing and how it's working. One of the most interesting DMOs for our purposes is `sys.dm_exec_query_plan`, which retrieves execution plans from the plan cache. You're going to want to see the objects within the cache in order to

see how the optimizer and storage engine created your plan. With DMVs and dynamic management functions, we can easily put together a query to get a very complete set of information about the execution plans on our system:

```
SELECT  [cp].[refcounts] ,
        [cp].[usecounts] ,
        [cp].[objtype] ,
        [st].[dbid] ,
        [st].[objectid] ,
        [st].[text] ,
        [qp].[query_plan]
FROM    sys.dm_exec_cached_plans cp
        CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) st
        CROSS APPLY sys.dm_exec_query_plan(cp.plan_handle) qp;
```

Listing 1.18

With this query, we can see the SQL and the XML plan generated by the execution of that SQL. There are a number of different DMOs available. I used one, `sys.dm_exec_cached_plans`, which shows properties about the object in the plan cache, but not the plan, as the basis for this query. I then used the `CROSS APPLY` statement to access the two dynamic management functions to retrieve the query from `sys.dm_exec_sql_text` and the plan from `sys.dm_exec_query_plan`. The `CROSS APPLY` applies a function once for every row in the result set. This query returns all queries from the cache as XML.

These plans that are stored in cache do not contain runtime information such as the actual number of rows or actual number of executions. They are the estimated plans. This makes them slightly less useful than the real plans (obtained from tools such as Profiler) when you're doing detailed performance tuning because you can't see those runtime values. However, the ability to immediately pull the plan straight out of cache is incredibly useful, so this method is going to serve you well.

We can use the XML directly or open it as a graphical execution plan. Furthermore, information available in the cache allows us to take more direct control over our execution plans. For example, as was mentioned earlier, we can retrieve the `plan_handle` from the

DMOs, as was used above, `cp.plan_handle`. We can then pass the value of a given plan handle to the DBCC command `FREEPROCCACHE` in order to remove only that procedure from the cache, like this:

```
DBCC FREEPROCCACHE(0x05000E007721DF00B8E0AF0B0000000000000000000000)
```

Listing 1.19

Throughout the book, we'll use DMOs to retrieve information from the cache, primarily focusing on execution plans. For even more understanding of all that is possible using DMOs, please refer to Louis Davidson and Tim Ford's excellent book, *Performance Tuning with SQL Server Dynamic Management Views* ([HTTP://WWW.SIMPLE-TALK.COM/BOOKS/SQL-BOOKS/PERFORMANCE-TUNING-WITH-SQL-SERVER-DYNAMIC-MANAGEMENT-VIEWS/](http://www.simple-talk.com/books/sql-books/performance-tuning-with-sql-server-dynamic-management-views/)).

Automating Plan Capture Using SQL Server Trace Events

During development, we can capture execution plans for targeted T-SQL statements, using one of the techniques described previously in this chapter. We can activate execution plan capture, run the query in question, observe the behavior in the execution plan, and then disable execution plan capture.

However, if we are troubleshooting on a test or live production server, the situation is different. A production system may be subject to tens or hundreds of sessions executing tens or hundreds of queries, each with varying parameter sets and varying plans. In this situation, we need a way to automate plan capture so that we can collect targeted plans automatically.

In SQL Server 2005 and 2008, we can use Profiler to define a server-side trace to capture XML execution plans, as the queries are executing. We can then examine the collected plans, looking for the queries with the highest costs, or simply searching the plans to find, for example, **Table Scan** operations, that we'd like to eliminate.

SQL Server trace events form a powerful tool, allowing us to capture data about events, such as the execution of T-SQL or a stored procedure, occurring within SQL Server. Trace events can be tracked manually, through the Profiler GUI interface, or traces can be defined through T-SQL and automated to run at certain times, and periods.

We can view these traces on the screen or send them to a file or a table in a database.³

Execution plan events

The various trace events that will generate an execution plan are listed below.

- **Showplan Text** – This event fires with each execution of a query and will generate the same type of estimated plan as the `SHOWPLAN_TEXT` T-SQL statement. Showplan Text will work on SQL Server 2005 and SQL Server 2008 databases, but it only shows a subset of the information available to Showplan XML. We've already discussed the shortcomings of the text execution plans, and this is on the list for deprecation in the future.
- **Showplan Text (unencoded)** – Same as above, but it shows the information as a string instead of binary. This is also on the list for deprecation in the future.
- **Showplan All** – This event fires as each query executes and will generate the same type of estimated execution plan as the `SHOWPLAN_ALL` statement in T-SQL. This has the same shortcomings as Showplan Text, and is on the list for future deprecation.
- **Showplan All for Query Compile** – This event generates the same data as the Showplan All event, but it only fires when a query compile event occurs. This is also on the list for deprecation in the future.

³ Detailed coverage of Profiler is out of scope for this book, but you can find more information in *Mastering SQL Profiler* by Brad McGehee.

- **Showplan Statistics Profile** – This event generates the actual execution plan in the same way as the T-SQL command `STATISTICS PROFILE`. It still has all the shortcomings of the text output, including only supplying a subset of the data available to `STATISTICS XML` in T-SQL or the **Showplan XML Statistics Profile** event in SQL Server Profiler. The **Showplan Statistics Profile** event is on the list for deprecation.
- **Showplan XML** – The event fires with each execution of a query and generates an estimated execution plan in the same way as `SHOWPLAN_XML`. This is the event you want in most cases. 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 compile of a given query.
- **Performance Statistics** – Similar to the Showplan XML For Query Compile event, except this event captures performance metrics for the query as well as the plan. This only captures XML output for certain event subclasses, defined with the event. It fires the first time a plan is cached, compiled, recompiled, or removed from cache.
- **Showplan XML Statistics Profile** – This event will generate the actual execution plan for each query, as it runs.

Capturing all of the execution plans, using **Showplan XML** or **Showplan XML Statistics Profile**, inherently places a sizeable load on the server. These are not lightweight event capture scenarios. Even the use of the less frequent **Showplan XML for Query Compile** will cause a small performance hit. Use due diligence when running traces of this type against any production machine.

Capturing a Showplan XML trace

The SQL Server 2005 Profiler Showplan XML event captures the XML execution plan used by the query optimizer to execute a query. To capture a basic Profiler trace, showing estimated execution plans, start Profiler, create a new trace and connect to a server.⁴ Switch to the **Events Selection** tab and click on the **Show all events** check box. The **Showplan XML** event is located within the **Performance** section, so click on the plus (+) sign to expand that selection. Click on the **Showplan XML** event.

While you can capture the **Showplan XML** event by itself in Profiler, it is generally more useful if you capture it along with some other basic events, such as:

- **RPC:Completed**
- **SQL:BatchCompleted**

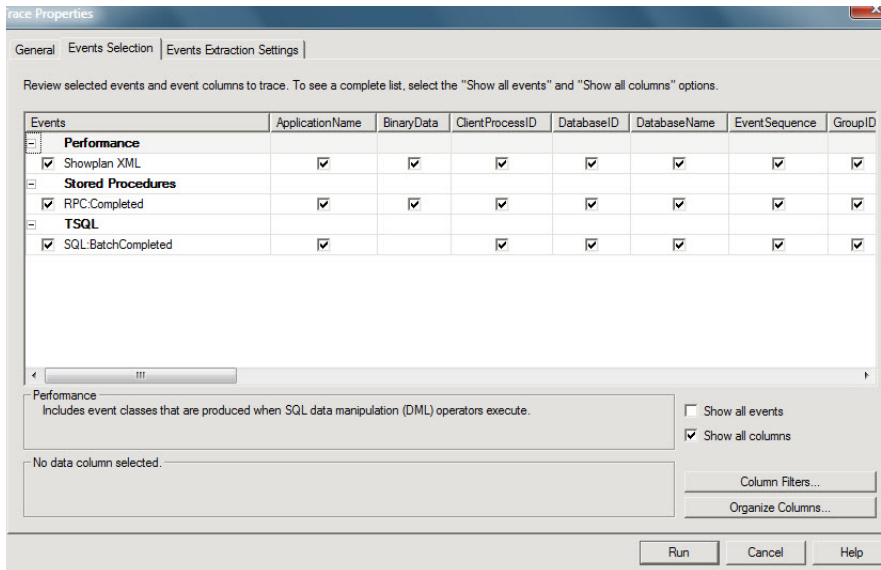


Figure 1.9

⁴ 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.

These extra events provide additional information to help put the XML plan into context. For example, we can see which parameters were passed for the event in which we are interested.

Once Showplan XML or any of the other XML events is selected, a third tab appears, called **Events Extraction Settings**. On this tab, we can choose to output the XML, as it's generated, to a separate file, for later use. 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.

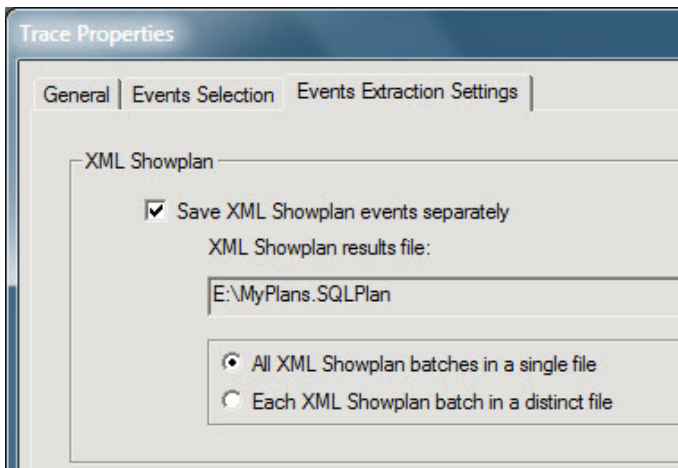


Figure 1.10

Click on the **Run** button in order to start the trace. When we capture the above events, we get a trace like the one shown in Figure 1.11.

Notice that I have clicked on the **Showplan XML** event. Under the **TextData** column, you see the actual XML plan code, and this plan can be saved to an individual file. In the second window, you can see the graphical execution plan, which is how most people prefer to read and analyze execution plans. So, in effect, the **Showplan XML** event available in Profiler not only shows you the XML plan code, but also the graphical execution plan.

EventClass	TextData	CPU	Reads	Writes	Duration	Appli
Trace Start						
SQL:BatchStarting	SELECT * FROM dbo.LogTest					Mici
Showplan XML	<ShowPlanXML xmlns="http://schemas....					Mici
SQL:BatchCompleted	SELECT * FROM dbo.LogTest	15	10	0	16	Mici
Trace Pause						




Table Scan
[LogTest]
Cost: 100 %

Trace is paused.

Ln 3, Col 2 Rows: 5

Figure 1.11

At this stage, we can also save the code for this particular **Showplan XML** event to a separate file. Simply right-click on the **Showplan XML** event you want to save, and then select **Extract Event Data**.

This brings up a dialog box where we can enter the path and filename of the XML code we want to store. Instead of storing the XML code with the typical XML extension, the extension used is **.SQLPlan**. By using this extension, when we double-click on the file from within Windows Explorer, the XML code will open up in Management Studio in the form of a graphical execution plan.

Whether capturing estimated execution plans or actual execution plans, the trace events operate in the same manner as when we run the T-SQL statements through the query window within Management Studio. The main difference is that this is automated across a large number of queries, from ad hoc to stored procedures, running against the server. Just remember, these do place a load on the system, so regulate their use carefully.

Why the actual and estimated execution plans might differ

Generally, the estimated and actual execution plans will be the same, in terms of the operations used. However, circumstances can arise that can cause differences between the estimated and actual execution plans.

When statistics are stale

The main cause of a difference between the estimated and actual plans is differences between the statistics and the actual data. This generally occurs over time, as data is added and deleted. This causes the key values that define the index to change, or their distribution (how many of what type) to change. 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. Not only can this cause differences between the plans, but you can get bad execution plans because the statistical data is not up to date.⁵

When the estimated plan is invalid

In some instances, the estimated plan won't work at all. For example, try generating an estimated plan for the simple bit of code in Listing 1.20, and you will get the error shown in Listing 1.21.

The optimizer, which is what generates estimated execution plans, doesn't execute T-SQL. It does run the statements through the algebrizer, the process outlined earlier that is responsible for verifying the names of database objects.

⁵ An example demonstrating how a drastic change in the data can affect the execution plan is given in the *Statistics and indexes* section of Chapter 4.

Since SQL Server has not yet executed the query, the temporary table does not yet exist. This is the cause of the error. Running this same bit of code through the actual execution plan will work perfectly.

```
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 1.20

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

Listing 1.21

When parallelism is requested

When a plan meets the threshold for parallelism (more about this in Chapter 8), the query engine may alter the plan as supplied by the optimizer, to include parallel operations, in an attempt to increase the performance of the query by making more use of the processors available. This parallel plan is only visible as an actual execution plan. The optimizer's plan does not include parallel execution, so none will be stored in cache.

Summary

In this chapter, we've approached how the optimizer and the storage engine work together to bring data back to your query. These operations are expressed in the estimated execution plan and the actual execution plan. I described a number of options for obtaining either of these plans: graphically, output as text, or as XML. Either the graphical plans or the XML plans will give us all the data we need, but it's going to be up to you to decide which to use and when, based on the needs you're addressing and how you hope to address them.

Chapter 2: Reading Graphical Execution Plans for Basic Queries

The aim of this chapter is to enable you to interpret *basic* graphical execution plans, in other words, execution plans for simple **SELECT**, **UPDATE**, **INSERT** or **DELETE** queries, with only a few joins and no advanced functions or hints. In order to do this, we'll cover the graphical execution plan topics below.

- **Operators** – Introduced in the last chapter – now you'll see more.
- **Joins** – What's a relational system without the joins between tables?
- **WHERE clause** – You need to filter your data and it does affect the execution plans.
- **Aggregates** – How grouping data changes execution plans.
- **Execution plans for data modifications** – **INSERT**, **UPDATE** and **DELETE** statements.

The Language of Graphical Execution Plans

In some ways, learning how to read graphical execution plans is similar to learning a new language, except that the language is icon-based, and the number of words (icons) we have to learn is minimal. Each icon represents a specific operator within the execution plan. We will be using the terms "icon" and "operator" interchangeably in this chapter.

In the previous chapter, we only saw two operators (**Select** and **Table Scan**). However, there are 78 available operators. Fortunately, for us, we don't have to memorize all 78 of them before we can read a graphical execution plan. Most queries use only a small subset of the operators, and we'll focus on those in this chapter. If you run across an icon not covered here, you can find more information about it on Books Online at [HTTP://MSDN2.MICROSOFT.COM/EN-US/LIBRARY/MS175913.ASPX](http://msdn2.microsoft.com/en-us/library/ms175913.aspx).

A graphical execution plan displays four distinct types of operator:

- **Logical and Physical Operators**, also called iterators, appear as blue icons and represent query execution or DML operations.
- **Parallelism Physical Operators** are also blue icons and represent parallelism operations. In a sense, they are a subset of logical and physical operators, but entail an entirely different level of execution plan analysis.
- **Cursor Operators** have yellow icons and represent Transact-SQL CURSOR operations.
- **Language Elements** are green icons and represent Transact-SQL language elements, such as ASSIGN, DECLARE, IF, SELECT (RESULT), WHILE, and so on.

In this chapter, we'll focus mostly on logical and physical operators, including the parallelism physical operators, with a few dives into some of the special information available in the language element operators. Books Online lists them in alphabetical order, but this isn't the easiest way to learn them, so we will forgo being "alphabetically correct" here. Instead, we will focus on the most used icons. Of course, what is considered most used and least used will vary from DBA to DBA, but the following are what I would consider the more common operators, listed roughly in the order of most common to least common.

Those highlighted in bold are covered in this chapter. We cover the rest in later chapters, when we move on to queries that are more complex.

We can learn a lot about how operators work by observing how they operate within execution plans. The key is to learn to use the properties of the operators and drill down on them. Each operator has a different set of characteristics. For example, they manage memory in different ways. Some operators – primarily **Sort**, **Hash Match (Aggregate)** and **Hash Join** – require a variable amount of memory in order to execute. Because of this, a query with one of these operators may have to wait for available memory prior to execution, possibly adversely affecting performance.

1.	Select (Result)	9.	Sort	17.	Spool
2.	Clustered Index Scan	10.	Key Lookup	18.	Eager Spool
3.	NonClustered Index Scan	11.	Compute Scalar	19.	Stream Aggregate
4.	Clustered Index Seek	12.	Constant Scan	20.	Distribute Streams
5.	NonClustered Index Seek	13.	Table Scan	21.	Repartition Streams
6.	Hash Match	14.	RID Lookup	22.	Gather Streams
7.	Nested Loops	15.	Filter	23.	Bitmap
8.	Merge Join	16.	Lazy Spool	24.	Split

Most operators behave in one of two ways, non-blocking or blocking. A non-blocking operator creates output data at the same time as it receives the input. A blocking operator has to get all the data prior to producing its output. A blocking operator might contribute to concurrency problems, hurting performance.

An example of a non-blocking operator would be the **Merge Join**, which produces data even as it is fed the data. We know this because the data in a **Merge Join** must be ordered for it to work properly, so it can produce its output as the input comes in.

An example of a blocking operator would be the **Hash Match** join. It has to gather all the data prior to performing its join operations and producing output. There are variations on some operators that may behave in other ways, but these are internals known only to Microsoft.

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.

Some Single Table Queries

Let's start by looking at some very simple plans, based on single table queries.

Clustered Index Scan



One of the more common operators is the **Clustered Index Scan**. This operation occurs when a Seek against the clustered index or some other index, can't satisfy the needs of the query. In that event, SQL Server must walk through, i.e. scan, the entire data set.

Consider the following simple (but inefficient!) query against the `Person.Contact` table in the AdventureWorks2008R2 database:

```
SELECT ct.*  
FROM Person.ContactType AS ct;
```

Listing 2.1

Figure 2.1 shows the actual execution plan:

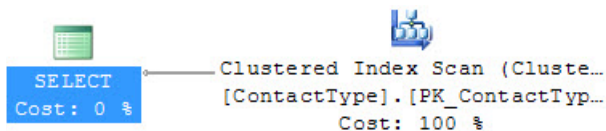


Figure 2.1

We can see that the engine used a **Clustered Index Scan** operation to retrieve the required data. Looking at only the graphic plan by itself doesn't tell us enough about the execution plan or the operators in it. More information is available with the ToolTips and **Property** sheets, as was mentioned in the Chapter 1. If you place the mouse pointer over the icon labeled **Clustered Index Scan** to bring up the ToolTip window, more information becomes immediately available, as you can see in Figure 2.2.

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Number of Rows	20
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.000179
Number of Executions	1
Estimated Number of Executions	1
Estimated Operator Cost	0.003304 (100%)
Estimated Subtree Cost	0.003304
Estimated Number of Rows	20
Estimated Row Size	73 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	0
Object	
[AdventureWorks2008R2].[Person].[ContactType].	
[PK_ContactType_ContactTypeID] [ct]	
Output List	
[AdventureWorks2008R2].[Person].	
[ContactType].ContactTypeID, [AdventureWorks2008R2].	
[Person].[ContactType].Name, [AdventureWorks2008R2].	
[Person].[ContactType].ModifiedDate	

Figure 2.2

Looking at Figure 2.2, near the bottom of the ToolTip, you find the label, **Object**. This indicates which object, if any, this operator references. In this case, the clustered index used was **PK_ContactType_ContactTypeID**. Listed above this are various other

properties about the index that can be useful in understanding how the operator works and what it is doing. Some of the properties are self-explanatory, but others might bear some information. The **Estimated I/O Cost** and **Estimated CPU Cost** are measures assigned by the optimizer, and each operator's cost contributes to the overall cost of the plan. I'll detail various points on these properties and others that are interesting for each execution plan as we hit that plan and that operator. Not all the operators will be useful in figuring out each execution plan. For example, rebinds and rewinds are only important when dealing with **Nested Loops** joins, but there are none of those in this plan, so those values are useless to you.

Indexes in SQL Server are stored in a balanced-tree, or a b-tree (a series of nodes that point to a parent). A clustered index not only stores the key structure, like a regular index, but also sorts and stores the data at the lowest level of the index, known as the leaf, which is the reason why there can be only one clustered index per table. This means that a **Clustered Index Scan** is very similar in concept to a **Table Scan**. The entire index, or a large percentage of it, is being traversed, row by row, in order to retrieve the data needed by the query.

An **Index Scan** often occurs, as in this case, when an index exists but the optimizer determines that there are so many rows to return that it is quicker to simply scan all the values in the index rather than use the keys provided by that index. In other situations, a scan is necessary because the index is not selective enough for the optimizer to be sure of finding the values it needs without scanning a large percentage of the index. That can also occur when the statistics for that index are out of date and showing incorrect information. You can also have situations where a query applies functions to columns, which means the optimizer can't determine what the value for that column could be so it has to scan the entire index to find it.

An obvious question to ask, if you see an **Index Scan** in your execution plan, is whether you are returning more rows than is necessary. If the number of rows returned is higher than you expect, that's a strong indication that you need to fine-tune the **WHERE** clause of your query so that only those rows that are actually needed are returned. Returning unnecessary rows wastes SQL Server resources and hurts overall performance.

Clustered Index Seek



A **Clustered Index Seek** operator occurs when a query uses the index to access only one row, or a few contiguous rows. It's one of the faster ways to retrieve data from the system. We can easily make the previous query more efficient by adding a **WHERE** clause. A **WHERE** clause limits the amount of data being returned, which makes it more likely that indexes will be used to return the appropriate data.

```
SELECT ct.*
FROM Person.ContactType AS ct
WHERE ct.ContactTypeID = 7
```

Listing 2.2

The plan now looks as shown in Figure 2.3.

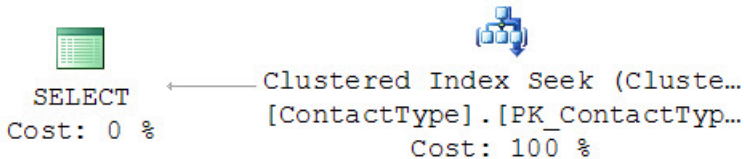


Figure 2.3

Index seeks are completely different from scans, where the engine walks through all the rows to find what it needs. **Clustered** and **NonClustered Index Seeks** occur when the optimizer is able to locate an index that it can use to retrieve the required records. Therefore, it tells the storage engine to look up the values based on the keys of the given index by assigning the **Seek** operation instead of a scan.

When an index is used in a **Seek** operation, the key values are used to look up and quickly identify the row or rows of data needed. This is similar to looking up a word in the index of a book to get the correct page number. The benefit of the **Clustered Index Seek** is that, not only is the **Index Seek** usually an inexpensive operation when compared to a scan, but no extra steps are required to get the data because it is stored in the index, at the leaf level.

In the above example, we have a **Clustered Index Seek** operation carried out against the `Person.ContactType` table. The ToolTip looks as shown in Figure 2.4.

Clustered Index Seek (Clustered)	
Scanning a particular range of rows from a clustered index.	
Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Number of Rows	1
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0.0032831 (100%)
Estimated Subtree Cost	0.0032831
Estimated Number of Rows	1
Estimated Row Size	73 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0
Object	
[AdventureWorks2008R2].[Person].[ContactType].	
[PK_ContactType_ContactTypeID] [ct]	
Output List	
[AdventureWorks2008R2].[Person].	
[ContactType].ContactTypeID, [AdventureWorks2008R2].	
[Person].[ContactType].Name, [AdventureWorks2008R2].	
[Person].[ContactType].ModifiedDate	
Seek Predicates	
Seek Keys[1]: Prefix: [AdventureWorks2008R2].[Person].	
[ContactType].ContactTypeID = Scalar Operator	
(CONVERT_IMPLICIT(int,[@1],0))	

Figure 2.4

The index used is the same as the previous example, specifically the `PK_ContactType_ContactTypeId`, which happens to be both the primary key and the clustered index for this table. This is again identified in the lower part of the ToolTip under **Object**. This time, however, the operation is a **Seek**. A seek has a predicate, or predicates, which are the filters by which the values will be retrieved. Here, the predicate is the value we passed in the query, 7, against the `Person.ContactType.ContactTypeID` column. However, look at the **Seek Predicates** section in Figure 2.4 and you'll notice that it's showing @1 instead of the value 7. This is because this is a simple query and it qualified for simple parameterization. Simple parameterization is SQL Server helping you by creating a parameterized query, which is very similar to a stored procedure, so that it can reuse the plan generated the next time this query is called with a different value. Otherwise, if this query is called again with the value 8, or something else, you'd get a whole new plan. Parameterization is discussed in detail later in the book.

By seeing that value change in the predicate, you can begin to understand how to use the information available through an execution plan to understand what is occurring within SQL Server.

Note, on the ToolTips window for the **Clustered Index Seek**, that the **Ordered** property is now **True**, indicating that the data was retrieved in order by the optimizer. 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 query more efficient.

NonClustered Index Seek



A **NonClustered Index Seek** operation is a seek against a non-clustered index. This operation is effectively no different than the **Clustered Index Seek** but the only data available is that which is stored in the index itself.

Let's run a slightly different query against the `Person.ContactType` table, one that uses a non-clustered index:

```
SELECT ct.ContactTypeId
FROM Person.ContactType AS ct
WHERE Name LIKE 'Own%'
```

Listing 2.3

We get an **Index Seek (NonClustered)** as shown in Figure 2.5.

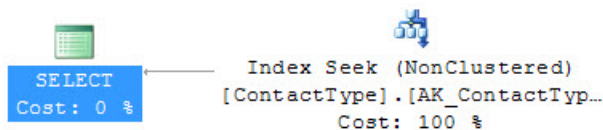


Figure 2.5

Like a **Clustered Index Seek**, a **NonClustered Index Seek** uses an index to look up the required rows. Unlike a **Clustered Index Seek**, a **NonClustered Index Seek** has to use a non-clustered index to perform the operation. Depending on the query and index, the query optimizer might be able to find all the data in the non-clustered index. However, a non-clustered index only stores the key values; it doesn't store the data. The optimizer might have to look up the data in the clustered index, slightly hurting performance due to the additional I/O required to perform the extra lookups – more on this in the next section.

In the example in Listing 2.3, the index satisfies all the needs of the query, meaning all the data needed is stored with the index, in the key, which makes this index, in this situation, a **covering index**. This is a covering index because we are only referring to two columns, the key column for the index itself, `Name`, and the key value for the clustered index, `ContactTypeId`. This is possible because the **Lookup** operator for non-clustered indexes to a clustered index is the key value of the clustered index.

To get an idea of the amount of information available, instead of looking at the ToolTip again, this time we'll right-click on the **Index Seek** operator and select **Properties** from the drop-down menu. Figure 2.6 shows the properties for the operator.

Properties	
Index Seek (NonClustered)	
<div> <div></div> <div></div> <div></div> </div>	
Misc	
Actual Number of Rows	2
Actual Rebinds	0
Actual Rewinds	0
Defined Values	[AdventureWorks2008R2].[Person].[ContactTy
Description	Scan a particular range of rows from a nonclu
Estimated CPU Cost	0.000161
Estimated I/O Cost	0.003125
Estimated Number of Executions	1
Estimated Number of Rows	3.63636
Estimated Operator Cost	0.003286 (100%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	54 B
Estimated Subtree Cost	0.003286
Forced Index	False
ForceSeek	False
Logical Operation	Index Seek
Node ID	0
NoExpandHint	False
Number of Executions	1
Object	[AdventureWorks2008R2].[Person].[ContactTy
Ordered	True
Output List	[AdventureWorks2008R2].[Person].[ContactTy
Parallel	False
Physical Operation	Index Seek
Predicate	[AdventureWorks2008R2].[Person].[ContactTy
Scan Direction	FORWARD
Seek Predicates	Seek Keys[1]: Start: [AdventureWorks2008R2].
TableCardinality	20
Actual Number of Rows	
Actual number of rows output by this operator. For rows of type PLAN_ROWS only.	

Figure 2.6

Reading this and comparing it to the ToolTips from Figures 2.4 and 2.2, you can see that there are many common values between the ToolTip and the **Property** sheet. The difference is that the property sheet carries even more information, as was mentioned in Chapter 1. Some values, such as **Seek Predicates**, have a plus sign to the left signifying that there is further information available. You can expand these properties to retrieve more information about the value. While the ToolTips are very handy, and I will continue to use them throughout the book, the best place to get the information you need is through the **Properties** sheet.

Key Lookup



A **Key Lookup** operator (there are two, **RID** and **Key**) is required to get data from the heap or the clustered index, respectively, when a non-clustered index is used, but is not a covering index. Listing 2.4 shows a query that returns a few columns from one table.

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

Listing 2.4

You should see a plan like that shown in Figure 2.7.

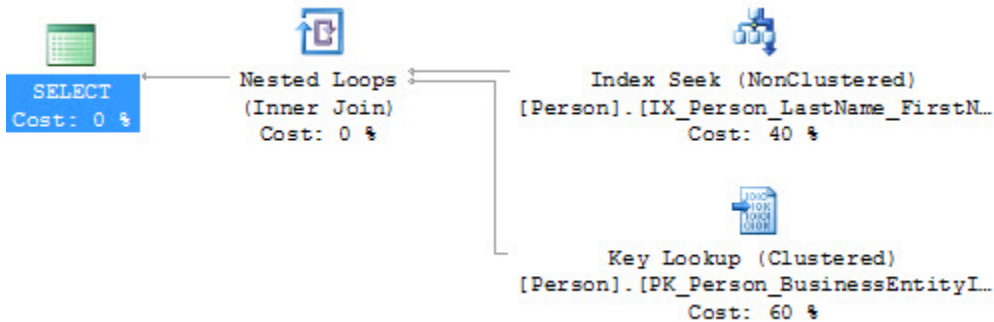


Figure 2.7

Finally, we get to see a plan that involves more than a single operation! Reading the plan in the physical order from right to left and top to bottom, the first operation we see is an **Index Seek** against the `IX_Person_LastName_FirstName_MiddleName` index. This is a non-unique, non-clustered index and, in the case of this query, it is *non-covering*. A *covering* index is a non-clustered index that contains all of the columns that need to be referenced by a query, including columns in the `SELECT` list, `JOIN` criteria and the `WHERE` clause.

In this case, the index on the `Lastname` and `FirstName` columns provides a quick way to retrieve information based on the filtering criteria `LIKE 'Jaf%'`. However, since only the name columns and the clustered index key are stored with the non-clustered index, the other column being referenced, `NameStyle`, has to be pulled in from somewhere else.

Since this index is not a covering index, the query optimizer is forced to not only read the non-clustered index, but also to read the clustered index to gather all the data required to process the query. This is a **Key Lookup** and, essentially, it means that the optimizer cannot retrieve the rows in a single operation, and has to use a clustered key (or a row ID from a heap table) to return the corresponding rows from a clustered index (or from the table itself).

We can understand the operations of the **Key Lookup** by using the information contained within the execution plan. If you first open the ToolTips window for the **Index Seek** operator by hovering with the mouse over that operator, you will see something similar to Figure 2.8. The pieces of information in which we're now interested are the **Output List** and the **Seek Predicates**, down near the bottom of the window.

Index Seek (NonClustered)	
Scan a particular range of rows from a nonclustered index.	
Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Number of Rows	1
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001592
Number of Executions	1
Estimated Number of Executions	1
Estimated Operator Cost	0.0032842 (40%)
Estimated Subtree Cost	0.0032842
Estimated Number of Rows	1.97157
Estimated Row Size	39 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	1
Predicate	
[AdventureWorks2008R2].[Person].[Person].[LastName] as [p].[LastName] like N'Jaf%'	
Object	
[AdventureWorks2008R2].[Person].[Person]. [IX_Person_LastName_FirstName_MiddleName] [p]	
Output List	
[AdventureWorks2008R2].[Person]. [Person].BusinessEntityID, [AdventureWorks2008R2]. [Person].[Person].FirstName, [AdventureWorks2008R2]. [Person].[Person].LastName	
Seek Predicates	
Seek Keys[1]: Start: [AdventureWorks2008R2].[Person]. [Person].LastName >= Scalar Operator(N'Jaf'), End: [AdventureWorks2008R2].[Person].[Person].LastName < Scalar Operator(N'JaG')	

Figure 2.8

Several of the columns are returned from the index, `Person.BusinessEntityID`, `Person.FirstName`, and `Person.LastName`. In addition, you can see how the optimizer can modify a statement. If you look at the **Seek Predicates**, instead of a `LIKE 'Jaf%'`, as was passed in the query, the optimizer has modified the statement so that the actual predicate used is `Person.LastName >= 'Jaf'` and `Person.LastName < 'JaG'` (minus a bit of operational code).

This information has nothing to do with the **Key Lookup** operation, but it's a good chance to see the optimizer in action. This is a fundamental part of the operations performed by the optimizer, as outlined in Chapter 1. In this case, the optimizer rearranged the `WHERE` clause. However, we still need to address the issue of getting the `NameStyle` column.

This means the optimizer now has to go and track down the additional information it needs from the clustered index. SQL Server uses the clustered index key as the method for looking up data that is stored in the clustered index from a non-clustered index, which only contains key information, the clustered key, and any lookup columns.

To find the appropriate rows of data, SQL Server uses the key values in a **Key Lookup** on the `PK_Person_BusinessEntityID` clustered index. Using the ToolTip window for the **Key Lookup** operator, you can see the **Output List** containing the final column needed by the query, `Person.NameStyle`, shown in Figure 2.9.

The presence of a **Key Lookup** is an indication that query performance might benefit from the presence of a covering index. A covering index is created by either having all the columns necessary as part of the key of the index, which has been explained several times already, 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.

Key Lookup (Clustered)	
Uses a supplied clustering key to lookup on a table that has a clustered index.	
Physical Operation	Key Lookup
Logical Operation	Key Lookup
Actual Number of Rows	1
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Number of Executions	1
Estimated Number of Executions	1.97157
Estimated Operator Cost	0.004946 (60%)
Estimated Subtree Cost	0.004946
Estimated Number of Rows	1
Estimated Row Size	9 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	3
Object	
[AdventureWorks2008R2].[Person].[Person].	
[PK_Person_BusinessEntityID] [p]	
Output List	
[AdventureWorks2008R2].[Person].[Person].NameStyle	
Seek Predicates	
Seek Keys[1]: Prefix: [AdventureWorks2008R2].	
[Person].[Person].BusinessEntityID = Scalar Operator	
([AdventureWorks2008R2].[Person].[Person].	
[BusinessEntityID] as [p].[BusinessEntityID])	

Figure 2.9

A join operation, which combines the results of the two operations, always accompanies a **Key Lookup**. In this instance, it was a Nested Loops join operation (Figure 2.10).

Nested Loops	
For each row in the top (outer) input, scan the bottom (inner) input, and output matching rows.	
Physical Operation	Nested Loops
Logical Operation	Inner Join
Actual Number of Rows	1
Estimated I/O Cost	0
Estimated CPU Cost	0.0000082
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0.00001 (0%)
Estimated Subtree Cost	0.0082401
Estimated Number of Rows	1.97157
Estimated Row Size	40 B
Actual Rebinds	0
Actual Rewinds	0
Node ID	0
Output List	
[AdventureWorks2008R2].[Person].	
[Person].BusinessEntityID, [AdventureWorks2008R2].	
[Person].[Person].NameStyle,	
[AdventureWorks2008R2].[Person].	
[Person].FirstName, [AdventureWorks2008R2].	
[Person].[Person].LastName	
Outer References	
[AdventureWorks2008R2].[Person].	
[Person].BusinessEntityID	

Figure 2.10

Typically, depending on the amount of data involved, a **Nested Loops** join by itself does not indicate any performance issues. In this case, because a **Key Lookup** operation is required, the **Nested Loops** join is needed to combine the rows of the **Index Seek** and **Key Lookup**. If the **Key Lookup** was not needed (because a covering index was available), then the **Nested Loops** operator would not be needed in this execution plan. However, because this operator was involved, along with the **Key Lookup** operator, at least two additional operations are required for every row returned from the non-clustered index. This is what can make a **Key Lookup** operation a very expensive process in terms of performance.

If this table had been a heap, a table without a clustered index, the operator would have been a **RID Lookup** operator. RID stands for row identifier, the means by which rows in a heap table are uniquely marked and stored within a table. The basics of the operation of a **RID Lookup** are the same as a **Key Lookup**.

Table Scan



This operator is self-explanatory and is one we encountered in Chapter 1. **Table Scans** only occur against heap tables, tables without clustered indexes. With a clustered index, we'd get a **Clustered Index Scan**, which is the equivalent of a **Table Scan** operation. You can see a **Table Scan** operation by executing the following query:

```
SELECT *  
FROM    dbo.DatabaseLog;
```

Listing 2.5

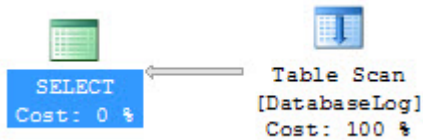


Figure 2.11

A **Table Scan** can occur for several reasons, but it's often because there are no useful 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 for the query optimizer to scan through each row and return them than look up each row in an index. This commonly occurs in tables with few rows.

Assuming that 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 rows are returned, then you might want to investigate ways to rewrite the query to return fewer rows, or add an appropriate index to speed performance.

RID Lookup



RID Lookup is the heap equivalent of the **Key Lookup** operation. As was mentioned before, non-clustered 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**.

If we specifically filter the results of our previous DatabaseLog query using the primary key column, we see a different plan that uses a combination of an **Index Seek** and a **RID Lookup**.

```
SELECT *  
FROM [dbo].[DatabaseLog]  
WHERE DatabaseLogID = 1
```

Listing 2.6

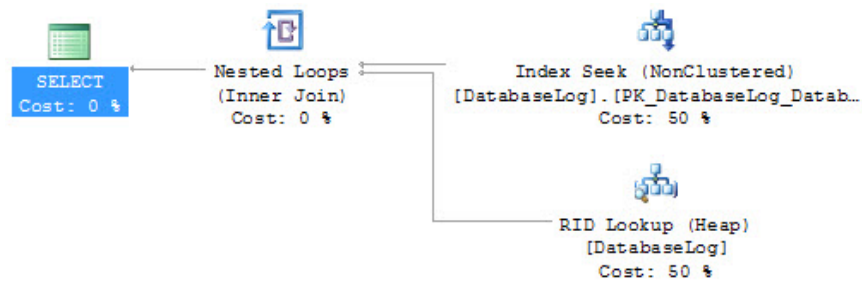


Figure 2.12

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?

Index Seek (NonClustered)	
Scan a particular range of rows from a nonclustered index.	
Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Number of Rows	1
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0.0032831 (50%)
Estimated Subtree Cost	0.0032831
Estimated Number of Rows	1
Estimated Row Size	19 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	1
Object	
[AdventureWorks2008R2].[dbo].[DatabaseLog]. [PK_DatabaseLog_DatabaseLogID]	
Output List	
Bmk1000, [AdventureWorks2008R2].[dbo]. [DatabaseLog].DatabaseLogID	
Seek Predicates	
Seek Keys[1]: Prefix: [AdventureWorks2008R2].[dbo]. [DatabaseLog].DatabaseLogID = Scalar Operator (CONVERT_IMPLICIT(int,[@1],0))	

Figure 2.13

If you look at the ToolTip in Figure 2.13 for the **Index Seek**, we see the value Bmk1000 in the **Output List**. This Bmk1000 is an additional column, not referenced in the query. It's the key value from the non-clustered index and it will be used in the **Nested Loops** operator to join with data from the **RID Lookup** operation.

Next, the query optimizer performs a **RID Lookup**, which is a type of Bookmark Lookup that occurs on a heap table (a table that doesn't have a clustered index), and uses a row identifier to find the rows to return. In other words, since the table doesn't have a clustered index (that includes all the rows), it must use a row identifier that links the index to the heap. This adds additional disk I/O because two different operations have to be performed instead of a single operation, which are then combined with a **Nested Loops** operation.

RID Lookup (Heap)	
RID Lookup	
Physical Operation	RID Lookup
Logical Operation	RID Lookup
Actual Number of Rows	1
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Number of Executions	1
Estimated Number of Executions	1
Estimated Operator Cost	0.0032831 (50%)
Estimated Subtree Cost	0.0032831
Estimated Number of Rows	1
Estimated Row Size	8589 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	3
Object	
[AdventureWorks2008R2].[dbo].[DatabaseLog]	
Output List	
[AdventureWorks2008R2].[dbo].	
[DatabaseLog].PostTime, [AdventureWorks2008R2].	
[dbo].[DatabaseLog].DatabaseUser,	
[AdventureWorks2008R2].[dbo].[DatabaseLog].Event,	
[AdventureWorks2008R2].[dbo].[DatabaseLog].Schema,	
[AdventureWorks2008R2].[dbo].[DatabaseLog].Object,	
[AdventureWorks2008R2].[dbo].[DatabaseLog].TSQL,	
[AdventureWorks2008R2].[dbo].	
[DatabaseLog].XmlEvent	
Seek Predicates	
Seek Keys[1]: Prefix: Bmk1000 = Scalar Operator	
((Bmk1000))	

Figure 2.14

In the ToolTip for the **RID Lookup**, notice that Bmk1000 is used again, but this time in the **Seek Predicates** section. This is the key value, which is a row identifier, or RID, from the non-clustered index.

In this particular 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.

Table Joins

Up to now, we have worked with single tables. Let's spice things up a bit and introduce joins into our query. 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 methods that the optimizer uses to combine data, which are, primarily, through the different **Join** operators, of which you have already seen a couple.

While this section is primarily concerned with **Join** operators, you'll see other operators at work. It's worth noting, that while in many (maybe even most) instances where a **JOIN** command is issued in T-SQL, it will be resolved through a **Join** operator, that's not necessarily what will happen. Further, again, as you've already seen, the optimizer can use **Join** operators to perform tasks other than T-SQL **JOIN** commands.

The query in Listing 2.7 retrieves employee information, 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  
        JOIN Person.BusinessEntityAddress AS bea  
        ON e.BusinessEntityID = bea.BusinessEntityID  
        JOIN Person.Address a  
        ON bea.AddressID = a.AddressID  
        JOIN Person.Person AS p  
        ON e.BusinessEntityID = p.BusinessEntityID ;
```

Listing 2.7

Figure 2.15 shows the execution plan for this query.

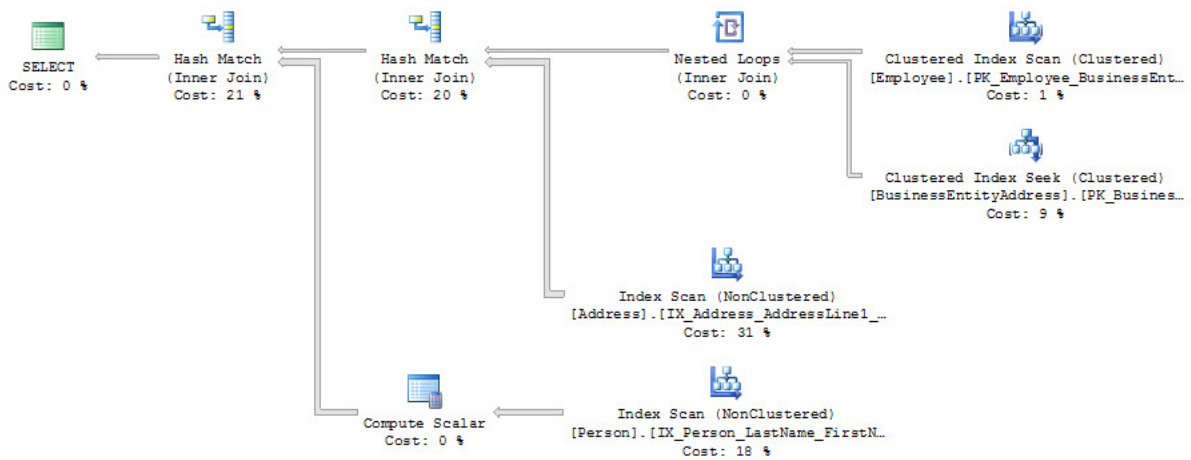


Figure 2.15

This plan has more operators than any we've seen so far. Just as with all previous plans, the logical flow of the operators is from the **Select** statement and then to the right through the various operators, while the data flows, and the accumulation of estimated costs occurs, as we move through the execution tree from right to left.

Associated with each operator icon, and displayed below it, is an estimated cost, assigned by the optimizer. From the relative estimated cost displayed, we can identify the three most costly operations in the plan, in descending order.

1. The **Index Scan** against the **Person.Address** table (31%).
2. The **Hash Match** join operation between the **Person.Person** table and the output from the first **Hash Match** (21%).
3. The other **Hash Match** join operator between the **Person.Address** table and the output from the **Nested Loop** operator (20%).

There are multiple problematic operators in this query since, in general, we're better off with **Seek** operations instead of **Scan** operations. Figure 2-16 shows the properties of the most expensive operator, as defined by the estimated costs.

Index Scan (NonClustered)	
Scan a nonclustered index, entirely or only a range.	
Physical Operation	Index Scan
Logical Operation	Index Scan
Actual Number of Rows	19614
Estimated I/O Cost	0.158681
Estimated CPU Cost	0.0217324
Number of Executions	1
Estimated Number of Executions	1
Estimated Operator Cost	0.180413 (31%)
Estimated Subtree Cost	0.180413
Estimated Number of Rows	19614
Estimated Row Size	45.8
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	6
Object	
[AdventureWorks2008R2].[Person].[Address].	
[IX_Address_AddressLine1_AddressLine2_City_StateProvinceID_PostalCode] [a]	
Output List	
[AdventureWorks2008R2].[Person].	
[Address].AddressID, [AdventureWorks2008R2].	
[Person].[Address].City	

Figure 2.16

The query optimizer needed to get at the `AddressId` and the `City` columns, as shown by the **Output List** at the bottom of the ToolTip. The optimizer calculated, based on index and column statistics, that the best way to arrive at that data was to scan the entire index, row by row. Walking through those 19,614 rows took an estimated 31% of the total query cost or an estimated operator cost of 0.158681. The estimated operator cost is the cost to the query optimizer for executing this specific operation, which is an internally calculated number used by the query optimizer to evaluate the relative costs of specific operations. As noted previously, this number doesn't reflect any real-world value, but the lower the estimated cost, the more efficient the operation, at least as determined by the optimizer. This does not mean that you can simply use these numbers to identify the most costly operator. However, these are the best values we have for making the determination for which operations are likely to be the most expensive. You still need to take into account other factors, such as the number of times a particular operator executes, to get closer to being able to model performance accurately, based on execution plans.

Hash Match join



A **Hash Match** operator appears in the plan when SQL Server puts two data sets into temporary tables, hash tables, and then uses these structures to compare data and arrive at the matching set.

Listing 2.7 had two different **Hash Match** join operations. Reading the plan logically, the first operation after the **Select** operator is a **Hash Match** join operation. This join is combining the output of one of the Index Scans with the combined output of the rest of the operations in the query. This **Hash Match** join operation is the second most expensive operation of this execution plan, so we would be interested in what we could do to improve its performance. Figure 2.17 shows the properties for this operator.

Hash Match	
Use each row from the top input to build a hash table, and each row from the bottom input to probe into the hash table, outputting all matching rows.	
Physical Operation	Hash Match
Logical Operation	Inner Join
Actual Number of Rows	290
Estimated I/O Cost	0
Estimated CPU Cost	0.123541
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0.123544 (21%)
Estimated Subtree Cost	0.587776
Estimated Number of Rows	274.988
Estimated Row Size	197 B
Actual Rebinds	0
Actual Rewinds	0
Node ID	0
Output List	
[AdventureWorks2008R2].[HumanResources].[Employee].JobTitle, [AdventureWorks2008R2].[Person].[Address].City, Expr1008	
Hash Keys Probe	
[AdventureWorks2008R2].[Person].[Person].BusinessEntityID	

Figure 2.17

Before we can talk about what a **Hash Match** join is, we need to understand two new concepts: **hashing** and a **hash table**. Hashing is a programmatic technique where data is converted into a symbolic form that makes searching for that data much more efficient. For example, SQL Server programmatically converts a row of data in a table into a unique value that represents the contents of the row. In many ways, it is like taking a row of data and encrypting it. Like encryption, a hashed value can be converted back to the original data.

A hash table, on the other hand, is a data structure that divides all of 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 row from a table, hash it into a hash value, and then store the hash value in a hash table, in tempdb.

Now that we understand these terms, we can discuss the **Hash Match** join operator. It occurs when SQL Server has to join two large data sets, and decides to do so by first *hashing* the rows from the smaller of the two data sets, and inserting them into a hash table. It then processes the larger data set, one row at a time, against the hash table, looking for matches, indicating the rows to be joined.

The smaller of the data sets provides the values in the hash table, so the table size is small, and because hashed values instead of real values are used, comparisons are quick. As long as the hash table is relatively small, this can be a quick process. On the other hand, if both tables are very large, a **Hash Match** join can be very inefficient as compared to other types of joins. All the data for the hash table is stored within `tempdb`, so excessive use of **Hash Joins** in your queries can lead to a heavier load on `tempdb`.

In this example, the data from `HumanResources.Employee` is matched with the `Person.Person` table.

Hash Match joins also work well for tables that are not sorted on `JOIN` columns; if they are, then **Merge Joins** tend to work better. **Hash Match** joins can be efficient in cases where there are no useable indexes.

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 Loop** or **Merge Join** operators. For example, seeing a **Hash Match** join in an execution plan sometimes indicates:

- a missing or unusable index
- a missing `WHERE` clause
- 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.

In other words, seeing a **Hash Match** join should be a cue for you to investigate whether you can tune the query, or add an index, to make the join operation more efficient. If so, then great but if not, then there is nothing else to do, and the **Hash Match** join might be the best overall way to perform the join.

Worth noting in this example is the slight discrepancy between the estimated number of rows returned, 274.988 (proving this is a calculation since you can't possibly return .988 rows), and the actual number of rows, 290. A difference this small is not worth worrying about, but a larger discrepancy can be an indication that your statistics need updating. Statistics being out of date or inaccurate can lead to a situation frequently referred to as "bad parameter sniffing." Chapter 5 discusses parameter sniffing in some detail.

Nested Loops join



Looking at the same query and execution plan in Figure 2-15, you can see that the second operation from the top right is a **Clustered Index Seek** against the `BusinessEntityAddress` table. This is a relatively inexpensive operation within this particular execution plan with an estimated cost of 9%. Figure 2.18 shows the ToolTip.

This seek is part of a join operation, and you can see the defined predicate, or search criteria, being used in the **Seek** operation in the **Seek Predicates** section at the bottom of the ToolTip. Let's explore that join in more detail.

A **Nested Loops** join functions by taking a set of data, referred to as the outer set, and comparing it, one row at a time to another set of data, called the inner set. This sounds like a cursor, and effectively, it is one but, with the appropriate data set, it can be a very efficient operation.

Chapter 2: Graphical Execution Plans for Basic Queries

Clustered Index Seek (Clustered)	
Scanning a particular range of rows from a clustered index.	
Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Number of Rows	290
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Estimated Number of Executions	290
Number of Executions	290
Estimated Operator Cost	0.0506686 (9%)
Estimated Subtree Cost	0.0506686
Estimated Number of Rows	1
Estimated Row Size	15 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	5
Object	
[AdventureWorks2008R2].[Person].[BusinessEntityAddress]. [PK_BusinessEntityAddress_BusinessEntityID_AddressID_Add ressTypeID] [bea]	
Output List	
[AdventureWorks2008R2].[Person]. [BusinessEntityAddress].BusinessEntityID, [AdventureWorks2008R2].[Person]. [BusinessEntityAddress].AddressID	
Seek Predicates	
Seek Keys[1]: Prefix: [AdventureWorks2008R2].[Person]. [BusinessEntityAddress].BusinessEntityID = Scalar Operator ([AdventureWorks2008R2].[HumanResources].[Employee]. [BusinessEntityID] as [e].[BusinessEntityID])	

Figure 2.18

The data scan against the Employee table and the seek against the BusinessEntityAddress tables is being driven by a **Nested Loops** join operation, whose ToolTip is displayed in Figure 2.19.

Nested Loops	
For each row in the top (outer) input, scan the bottom (inner) input, and output matching rows.	
Physical Operation	Nested Loops
Logical Operation	Inner Join
Actual Number of Rows	290
Estimated I/O Cost	0
Estimated CPU Cost	0.0012122
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0.0012122 (0%)
Estimated Subtree Cost	0.0599262
Estimated Number of Rows	288.821
Estimated Row Size	69 B
Actual Rebinds	0
Actual Rewinds	0
Node ID	2
Output List	
[AdventureWorks2008R2].[HumanResources].	
[Employee].JobTitle, [AdventureWorks2008R2].	
[Person].[BusinessEntityAddress].BusinessEntityID,	
[AdventureWorks2008R2].[Person].	
[BusinessEntityAddress].AddressID	
Outer References	
[AdventureWorks2008R2].[HumanResources].	
[Employee].BusinessEntityID, Expr1011	

Figure 2.19

Another name for the **Nested Loops** join is a **nested iteration**. This operation takes the input from two sets of data and joins them by scanning the outer data set (the bottom operator in a graphical execution plan) once for each row in the inner set. The number of rows in each of the two data sets was small, making this a very efficient operation. As long as the inner data set is small and the outer data set, small or not, is indexed, then this is an extremely efficient join mechanism. Except in cases of very large data sets, this is the best type of join to see in an execution plan.

Compute Scalar



This is not a type of join operation but since it appears in our plan, we'll cover it here. In the execution plan shown in Figure 2.15, right after the final **Index Scan** operator, we have a **Compute Scalar** operator. Figure 2.20 shows the **Properties** window for this operator.

Properties	
Compute Scalar	
<div> </div>	
Misc	
Defined Values	[Expr1008] = Scalar Operator(((AdventureWorks2008
Expr1008	Scalar Operator(((AdventureWorks2008R2).[Persd(...
Description	Compute new values from existing values in a row.
Estimated CPU Cost	0.0019972
Estimated I/O Cost	0
Estimated Number of Executions	1
Estimated Number of Rows	19972
Estimated Operator Cost	0.001997 (0%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	117 B
Estimated Subtree Cost	0.105767
Logical Operation	Compute Scalar
Node ID	8
Output List	[AdventureWorks2008R2].[Person].[Person].Busines
Parallel	False
Physical Operation	Compute Scalar

Figure 2.20

This is simply a representation of an operation to produce a scalar, a single defined value, 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 (0.001997), the cost is trivial enough in the context of the query to be essentially free. You can see what this operation is doing by

looking at the definition for the highlighted property, Expr1008 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 2-21.

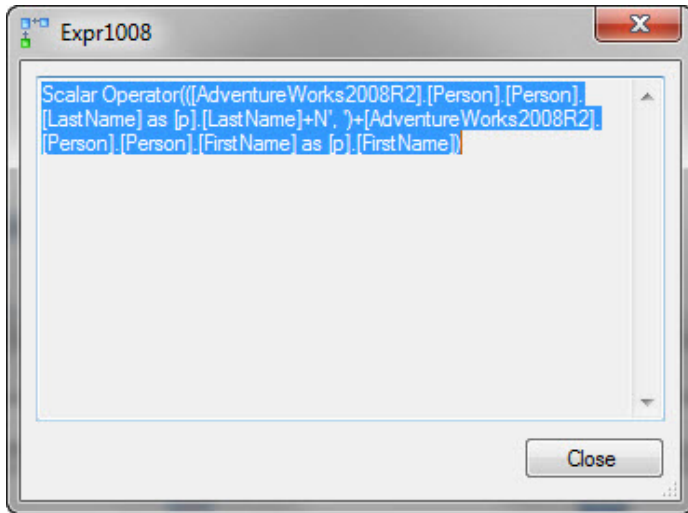


Figure 2-21

Merge Join



A **Merge Join** operator works from sorted data, and sorted data only. It takes the data from two different data sets and uses the fact that the data is sorted to simply merge it together, combining based on the matching values, which it can do very easily because the order of the values will be identical. If the data is sorted, this can be one of the most efficient join operations. However, the data is frequently not sorted, so sorting it for a **Merge Join** requires the addition of a **Sort** operator to ensure it works; this can make this join operation less efficient.

To see an example of a **Merge Join**, we can run the following code:

```
SELECT c.CustomerID
FROM Sales.SalesOrderDetail od
JOIN Sales.SalesOrderHeader oh
      ON od.SalesOrderID = oh.SalesOrderID
JOIN Sales.Customer c ON oh.CustomerID = c.CustomerID
```

Listing 2.8

Figure 2.22 shows the execution plan for this query.

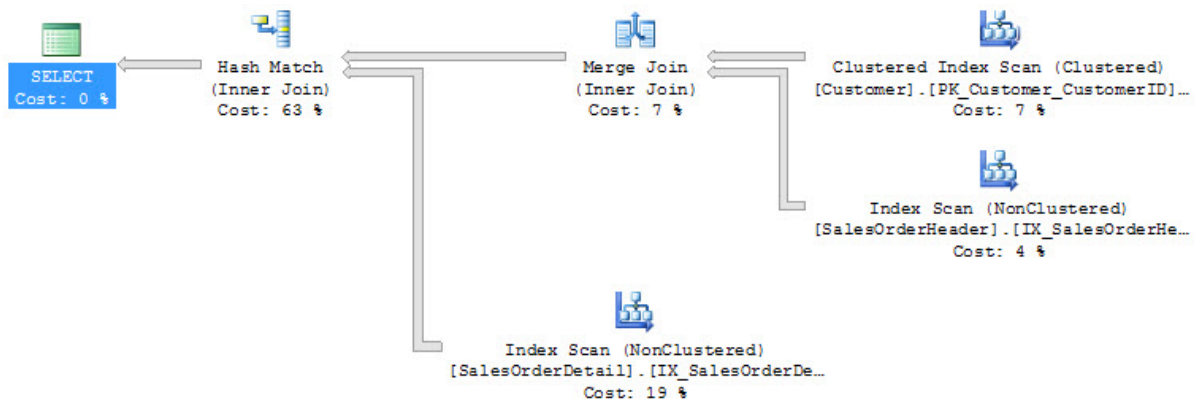


Figure 2.22

According to the execution plan, the query optimizer performs a **Clustered Index Scan** on the Customer table and a **NonClustered Index Scan** on the SalesOrderHeader table. Since the query did not specify a **WHERE** clause, a scan was performed on each table to return all the rows in each table.

Next, all the rows from both the Customer and SalesOrderHeader tables are joined using the **Merge Join** operator. A **Merge Join** occurs on tables where the join columns are sorted. For example, in the ToolTip window for the **Merge Join**, shown in Figure 2.23, we see that the join columns are Sales and CustomerID. In this case, the data in

the join columns, retrieved from both indexes, is ordered. A **Merge Join** is an efficient way to join two tables, when the join columns are sorted, but if the join columns are not sorted, the query optimizer has the option of a) sorting the join columns first, then performing a **Merge Join**, or b) performing a less efficient **Hash Match** join. The query optimizer considers its options and generally chooses the execution plan that uses the least resources, based on the statistics available.

Merge Join	
Match rows from two suitably sorted input tables exploiting their sort order.	
Physical Operation	Merge Join
Logical Operation	Inner Join
Actual Number of Rows	31465
Estimated I/O Cost	0
Estimated CPU Cost	0.116408
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0.1164114 (7%)
Estimated Subtree Cost	0.299389
Estimated Number of Rows	31098.7
Estimated Row Size	15 B
Actual Rebinds	0
Actual Rewinds	0
Many to Many	False
Node ID	1
Where (join columns)	
([AdventureWorks2008R2].[Sales].[SalesOrderHeader].CustomerID) = ([AdventureWorks2008R2].[Sales].[Customer].CustomerID)	
Output List	
[AdventureWorks2008R2].[Sales].[SalesOrderHeader].SalesOrderID, [AdventureWorks2008R2].[Sales].[Customer].CustomerID	

Figure 2.23

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.

The key to the performance of a **Merge Join** is that the joined columns are pre-sorted. If they are not, and the query optimizer chooses to sort the data in a separate operation before it performs a **Merge Join**, this might be an indication that a **Merge Join** is not an ideal way to join the tables, or it might indicate that you need to reconsider your indexing strategy.

Filtering Data

Only infrequently will queries run without some sort of conditional statements to limit the results set; in other words, a **WHERE** clause. We'll investigate two multi-table, conditional queries using graphical execution plans.

Run the following query against **AdventureWorks2008R2**, and look at the actual execution plan. This query is the same as the one we saw at the start of the *Table Joins* section, but now a **WHERE** clause has been added.

```
SELECT e.[Title],
       a.[City],
       c.[LastName] + ',' + c.[FirstName] AS EmployeeName
FROM   [HumanResources].[Employee] e
       JOIN [HumanResources].[EmployeeAddress] ed
           ON e.[EmployeeID] = ed.[EmployeeID]
       JOIN [Person].[Address] a
           ON [ed].[AddressID] = [a].[AddressID]
       JOIN [Person].[Contact] c
           ON e.[ContactID] = c.[ContactID]
WHERE  e.[Title] = 'Production Technician - WC20' ;
```

Listing 2.9

Figure 2.24 shows the actual execution plan for this query.

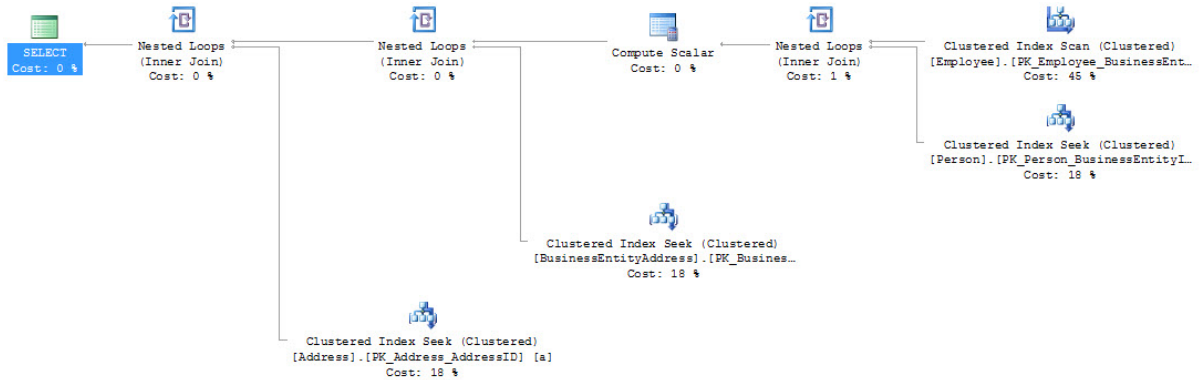


Figure 2.24

Starting from the right, we see that the optimizer has used the criteria from the **WHERE** clause to do a **Clustered Index Scan**, using the primary key. The **WHERE** clause limited the number of rows to 1, which you can see by hovering your mouse pointer over the arrow coming out of the **Clustered Index Scan** operator (see Figure 2.25).

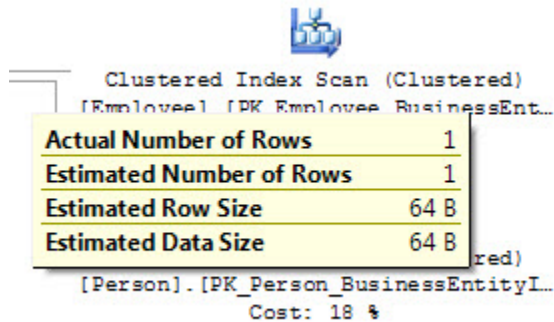


Figure 2.25

Using the available statistics, the optimizer was able to determine this up front, as we see by comparing the estimated and actual rows returned in the ToolTip.

Working with a smaller data set and a good index on the `Person.Person` table, the optimizer was able to use the more efficient **Nested Loop** join. Since the optimizer changed where that table was joined, it also moved the scalar calculation right next to the join. Since it's still only one row coming out of the **Scalar** operation, a **Clustered Index Seek** and another **Nested Loop** were used to join the data from the `Person.BusinessEntityAddress` table. This then leads to a final **Clustered Index Seek** and the final **Nested Loop**. All these more efficient joins are possible because we reduced the initial data set with the `WHERE` clause, as compared to the previous query which did not have a `WHERE` clause.

Frequently, developers who are not too comfortable with T-SQL will suggest that the "easiest" way to do things is to simply return all the rows to the application, either without joining the data between tables, or even without adding a `WHERE` clause. This was a very simple query with only a small set of data, but you can use this as an example, when confronted with this sort of argument.

The best way to compare one query to another is by examining the execution time and the amount of data accessed. Using `SET STATISTICS TIME ON` will cause SQL Server to return the amount of time it takes to compile and run a query. To get the amount of data returned, you can use `SET STATISTICS IO ON`, and SQL Server shows the number of reads, which represent SQL Server accessing a page of information.

If you look at the execution time, scans and reads for these queries, which is the best way to compare execution between two queries, the first query runs in about 202ms and the second runs on 140ms on my machine. As far as scans and reads go, the first query, without a `WHERE` clause...

```
Table 'Worktable'. Scan count 0, logical reads 0
Table 'Person'. Scan count 1, logical reads 109
Table 'Address'. Scan count 1, logical reads 216
Table 'BusinessEntityAddress'. Scan count 290
Table 'Employee'. Scan count 1, logical reads 9
```

...resulted in quite a high number of reads and only a few scans, while the other query...

```
Table 'Address'. Scan count 0, logical reads 2
Table 'BusinessEntityAddress'. Scan count 1
Table 'Person'. Scan count 0, logical reads 3
Table 'Employee'. Scan count 1, logical reads 9
```

...had only a very few reads, which means it uses fewer resources and takes less time. So it's clear, with a `WHERE` clause, you can limit the resources used by SQL Server.

Execution Plans with GROUP BY and ORDER BY

When we add other basic T-SQL clauses to a query, the resulting execution plans show different operators.

Sort



Take a simple `SELECT` with an `ORDER BY` clause as an example:

```
SELECT Shelf
FROM Production.ProductInventory
ORDER BY Shelf
```

Listing 2.10

The execution plan is shown in Figure 2.26.

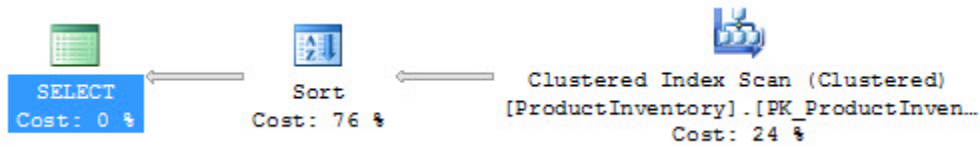


Figure 2.26

The **Clustered Index Scan** operator outputs into the **Sort** operator. Compared to many of the execution plan icons, the **Sort** operator is very straightforward. It shows when the query optimizer is sorting data within the execution plan. If an **ORDER BY** clause does not specify order, the default order is ascending, as you will see from the ToolTip for the **Sort** icon (see Figure 2.27).

Sort	
Sort the input.	
Physical Operation	Sort
Logical Operation	Sort
Actual Number of Rows	1069
Estimated I/O Cost	0.0112613
Estimated CPU Cost	0.0168799
Number of Executions	1
Estimated Number of Executions	1
Estimated Operator Cost	0.0281412 (76%)
Estimated Subtree Cost	0.0370435
Estimated Number of Rows	1069
Estimated Row Size	21 B
Actual Rebinds	1
Actual Rewinds	0
Node ID	0
Output List	
[AdventureWorks2008R2].[Production]. [ProductInventory].Shelf	
Order By	
[AdventureWorks2008R2].[Production]. [ProductInventory].Shelf Ascending	

Figure 2.27

Pull up the ToolTip window for the arrow leading to the **Sort** icon (see Figure 2.28) and you'll see that the **Clustered Index Scan** passes 1069 rows to the **Sort** operator, which sorts these 1069 rows and then passes them on, in sorted order.

Actual Number of Rows	1069
Estimated Number of Rows	1069
Estimated Row Size	54 B
Estimated Data Size	56 KB

Figure 2.28

The most interesting point to note is that the **Sort** operation is 76% of the cost of the query. There is no index on this column, so SQL Server performs the **Sort** operation within the query execution.

As a rule of thumb, if sorting takes more than 25% of a query's total execution time, then you need to review it carefully and optimize it, if possible. In our case, the reason why we are breaking this rule is straightforward: we are missing a **WHERE** clause. Most likely, this query is returning far more rows than necessary, all of which SQL Server then needs to sort. Even if a **WHERE** clause exists, you need to ensure that it limits the amount of rows to only the required number of rows to be sorted, not rows that will never be used.

A good question to ask if you spot an expensive **Sort** operation is, "Is the **Sort** really necessary?" As hinted above, SQL Server often performs the **Sort** operation within the query execution due to the lack of an appropriate index. With the appropriate clustered index, the data may come pre-sorted. It is not always possible to create the appropriate clustered index, but if it is, you will save sorting overhead.

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.

If we change the query as shown in Listing 2.11, we get the execution plan shown in Figure 2.29.

```
SELECT *  
FROM Production.ProductInventory  
ORDER BY ProductID
```

Listing 2.11

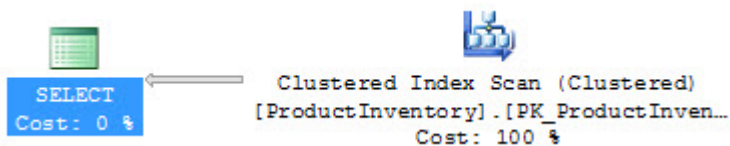


Figure 2.29

Although this query is almost identical to the previous query, and it includes an **ORDER BY** clause, we don't see a **Sort** operator in the execution plan. This is because the column we are sorting by has changed, and this new column has a clustered index on it, which means that the returned data does not have to be sorted again, as it is already sorted as a byproduct of being the clustered index. The query optimizer is smart enough to recognize that the data, as retrieved through the index, is already ordered, and does not have to be ordered again.

If you have no choice but to sort a lot of data, you should consider using trace events, **Performance:Showplan XML**, to see if any **Sort Warnings** are generated. To boost performance, SQL Server attempts to perform sorting in memory instead of on disk, since sorting in RAM is much faster than sorting on disk. However, if the **Sort** operation is large, SQL Server may have to write data to the **tempdb** database and sort on disk. Whenever this occurs, SQL Server generates a **Sort Warning** event, which we can capture using trace events. If your server is performing many **Sorts**, and generating many **Sort Warnings**, then you may need to add more RAM to your server, or to speed up **tempdb** access.

Hash Match (aggregate)



Earlier in this chapter, we looked at the **Hash Match** operator for joins. This same **Hash Match** operator can also occur when aggregations are present within a query. Let's consider a simple aggregate query against a single table using the **Count** operator.

```
SELECT  [City],
        COUNT([City]) AS CityCount
FROM    [Person].[Address]
GROUP BY [City]
```

Listing 2.12

Figure 2.30 shows the actual execution plan.

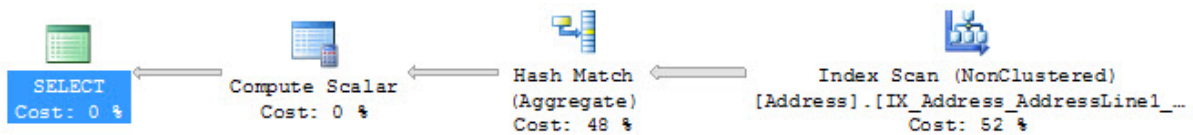


Figure 2.30

The query execution begins with an **Index Scan**, because all of the rows are returned for the query; there is no **WHERE** clause to filter the rows. Next, the optimizer aggregates these rows in order to perform the requested **COUNT** aggregate operation. In order for the query optimizer to count each row for each separate city, the optimizer chooses to perform a **Hash Match** operation. Notice that the word "Aggregate" appears, within parentheses, underneath **Hash Match** in the execution plan. This is to distinguish it from a **Hash Match** operation for a join. 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 order to count the number of rows that match the **GROUP BY** column, which in this case is **City**. With the results aggregated, they are passed back to us.

Quite often, aggregations within queries can be expensive operations. About the only way to "speed" the performance of an aggregation via code is to ensure that you have a restrictive `WHERE` clause to limit the number of rows that need to be aggregated, thus reducing the amount of aggregation that needs to be done. You can also pre-aggregate data by using an indexed view.

Filter



If we add a simple `HAVING` clause to our previous query, our execution plan gets more complex.

```
SELECT [City],
       COUNT([City]) AS CityCount
FROM   [Person].[Address]
GROUP BY [City]
HAVING COUNT([City]) > 1
```

Listing 2.13

The execution plan now looks as shown in Figure 2.31.

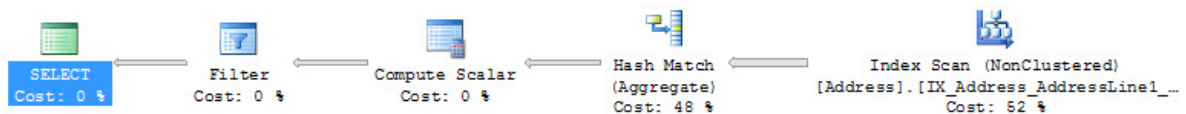


Figure 2.31

By adding the `HAVING` clause, the **Filter** operator appears in the execution plan. The **Filter** operator is applied to limit the output to those values of the column, `City`, that are greater than 1, in order to satisfy the `HAVING` clause. One useful bit of knowledge to take away from this plan is that the optimizer does not apply the `HAVING` clause until all the

aggregation of the data is complete. We can see this by noting that the actual number of rows in the **Hash Match** operator is 575 and in the **Filter** operator it's 348.

Actual Number of Rows	575
Estimated Number of Rows	575
Estimated Row Size	32 B
Estimated Data Size	18 KB

Rows After Aggregate

Actual Number of Rows	348
Estimated Number of Rows	575
Estimated Row Size	32 B
Estimated Data Size	18 KB

Rows After Filter

Figure 2.32

While adding a **HAVING** clause reduces the amount of data returned, it actually adds to the resources needed to produce the query results, because the **HAVING** clause does not come into play until after the aggregation. This hurts performance. As with the previous example, if you want to speed the performance of a query with aggregations, the only way to do so in code is to add a **WHERE** clause to the query to limit the number of rows that need to be selected and aggregated.

A brief aside on rebinds and rewinds

While examining the ToolTips for physical operators, throughout this chapter, you may have noticed these terms several times:

- actual rebinds or estimated rebinds
- actual rewinds or estimated rewinds.

Most of the time in this chapter, the value for both the rebinds and rewinds has been zero, but for the **Sort** operator example, a little earlier, we saw that there was **one** actual rebind and **zero** actual rewinds. Figure 2.33 shows the relevant section of the **Properties** page for that operator.

⊕ Actual Rebinds	1
⊕ Actual Rewinds	0

Figure 2.33

In order to understand what these values mean, we need some background. Whenever a physical operator, such as the **Sort** operator in an execution plan occurs, three things happen.

- First, the physical operator is initialized and any required data structures are set up. This is called the `Init()` method. In all cases, this happens once for an operator, although it is possible for it to happen many times.
- Second, the physical operator gets (or receives) the rows of data that it is to act on. This is called the `GetNext()` method. Depending on the type of operator, it may receive none, or many `GetNext()` calls.
- Third, once the operator has performed its function, it needs to clean itself up and shut itself down. This is called the `Close()` method. A physical operator only ever receives a single `Close()` call.

A rebind or rewind is a count of the number of times the `Init()` method is called by an operator. A rebind and a rewind both count the number of times the `Init()` method is called, but do so under different circumstances.

Only certain operators record values for rebind and rewind:

- **Non-Clustered Index Spool**
- **Remote Query**
- **Row Count Spool**
- **Sort**
- **Table Spool**
- **Table-Valued Function.**

If the following operators occur, the rebind and rewind counts will only be populated when the `StartupExpression` for the physical operation is set to `TRUE`, which can vary depending on how the query optimizer evaluates the query. This is set by Microsoft in code and is something over which we have no control.

- **Assert**
- **Filter.**

For all other physical operators, the counts for rebind and rewind are not populated and will be zero.

For the operators affected, multiple rebind or rewind events only occur in relation to a **Nested Loops** join operation, specifically on the inner (lower) set of data. A **rebind** occurs, increasing the rebind count, when one or more of the correlated parameters of the **Nested Loops** join change and the inner side must be reevaluated. A **rewind** occurs, increasing the rewind count, when none of the correlated parameters change and the prior inner result set may be reused.

So, what does it mean when you see a value for either rebind or rewind for the eight operators where rebind and rewind may be populated?

If you see an operator where rebind equals one and rewinds equals zero, this means that an `Init()` method was called one time on a physical operator that is **not** on the inner side of a loop join. If the physical operator is on the inner side of a loop join used by an operator, then the sum of the rebinds and rewinds will equal the number of rows processed on the outer side of a join used by the operator.

This query will result in a **Table Spool** on the inner side of a **Loop Join** operator.

```
SELECT sod.SalesOrderDetailID
FROM Sales.SalesOrderDetail AS sod
WHERE LineTotal < (SELECT AVG(dos.LineTotal)
                  FROM Sales.SalesOrderDetail AS dos
                  WHERE dos.ModifiedDate < sod.ModifiedDate
                  )
```

Listing 2.14

If you examine the property sheet of the operator, you will see the relationship between the actual rows returned and the rebinds and rewinds necessary to return them.

Since this is a natural part of the **Nested Loops** join operator, it's not an issue when you see these values but, depending on the number of executions of these operators, they can be indicators of performance issues.

Execution Plans for INSERT, UPDATE and DELETE Statements

The optimizer generates execution plans for all queries against the database, in order for the storage engine to figure out how best to undertake the submitted request. While the previous examples have been for **Select** queries, in this section we will look at the execution plans of **INSERT**, **UPDATE**, and **DELETE** queries.

INSERT statements

INSERT statements are always against a single table. This would lead you to believe that execution plans will be equally simplistic. However, in order to take into account rollbacks and data integrity checks, execution plans for insert queries can be quite complicated.

Listing 2.15 contains a very simple INSERT statement.

```
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 2.15

There are two ways I could run the query above to generate the execution plan. I could retrieve an estimated plan, which is what I did. An estimated plan is a plan that is compiled, but not run. The other method for getting an execution plan would be to wrap the query in a transaction and roll back that transaction after capturing the execution plan. I chose to use the estimated plan because it was easier and still suited our needs.

This plan may be a little difficult to read on the printed page. Please execute the code on your own machine in order to see the plan clearly, and follow along with the explanations.

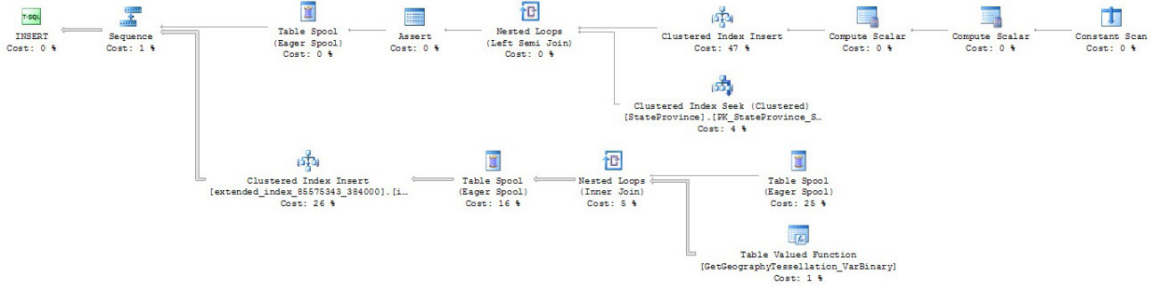


Figure 2.34

The physical structures of the tables the query accesses can affect the resulting execution plans. For example, this table has an **IDENTITY** column, **FOREIGN KEY** constraints and a spatial data column. All these objects will affect the execution plan.

The physical operation of the execution plan starts off, reading right to left, with an operator that is new to us: **Constant Scan**. This operator introduces a constant number of rows into a query. In our case, it's building a row in order for the next two operators to have a place to add their output. The first of these is a **Compute Scalar** operator to call a function called `getidentity`. This is the point within the query plan when SQL Server generates an identity value, for the data to follow. Note that this is the first operation within the plan, which helps explain why, when an **INSERT** fails, you get a gap in the identity values for a table.

The next **Scalar** operation creates a series of placeholders for the rest of the data and creates the new `uniqueidentifier` value, and the date and time from the `GETDATE` function. You can tell this is what happens by looking at the **Property** sheet for the operator and then looking at the **Defined Values** property. There is an ellipsis next to that property which, when clicked, will open a window showing what is being built by the **Compute Scalar** operation as shown in Figure 2.35.

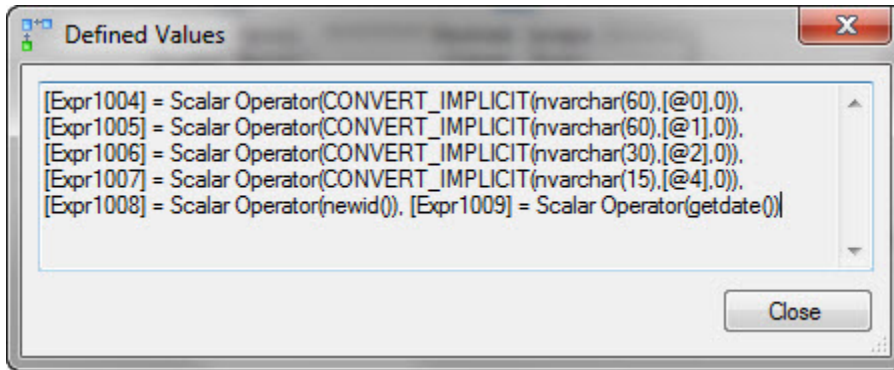


Figure 2.35

All of this is passed to the **Clustered Index Insert** operator, where the majority of the cost of this plan is realized. Note the output value from the **INSERT** statement, the **Person.Address.StateProvinceId**. This is passed to the next operator, the **Nested Loop** join, which also gets input from the **Clustered Index Seek** against the **Person.StateProvince** table. In other words, we had a read during the **INSERT** to check for referential integrity on the foreign key of **StateProvinceId**. The join then outputs a new expression which is tested by the next operator, **Assert**. An **Assert** operator verifies that a particular condition exists. This one checks that the value of **Expr1004** equals zero. Or, in other words, that the data to be inserted into the **Person.Address.StateProvinceId** field matched a piece of data in the **Person.StateProvince** table; this was the referential check.

The subtree of operations represents the data being added to the spatial data column. I cover some of the special data types in Chapter 7, so I won't go over those operators here.

UPDATE statements

UPDATE statements are also against one table at a time. Depending on the structure of the table, and the values to be updated, the impact on the execution plan could be as severe as that shown above for the INSERT query. Consider the UPDATE statement in Listing 2.16.

```
UPDATE [Person].[Address]
SET    [City] = 'Munro',
       [ModifiedDate] = GETDATE()
WHERE  [City] = 'Monroe' ;
```

Listing 2.16

Figure 2.36 shows the estimated execution plan.

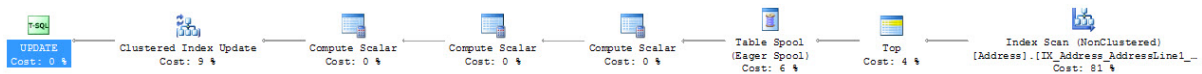


Figure 2.36

Let's begin reading this execution plan in the physical operation order, from right to left. The first operator is a **NonClustered Index Scan**, which retrieves all of the necessary rows from a non-clustered index, scanning through them, one row at a time. This is not particularly efficient and should be a flag to you that perhaps the table needs better indexes to speed performance. The purpose of this operator is to identify all the rows WHERE [City] = 'Monroe', and then send them to the next operator.

The next operator is **Top**. In an execution plan for an UPDATE statement, the **Top** operator enforces row count limits, if there are any. In this case, no limits were enforced because the TOP clause was not used in the UPDATE query as you can see in Listing 2.16.

Note

*If the **Top** operator is found in a **SELECT** statement, and not an **UPDATE** statement, it indicates that a specified number, or percent, of rows are being requested, based on the **TOP** command used in the **SELECT** statement.*

The next operator is an **Eager Spool** (a form of a **Table Spool**). This obscure sounding operator essentially takes each of the rows to be updated and stores them in a hidden temporary object stored in the `tempdb` database. The rewind is put in place in an update query like this as part of prevention for the Halloween problem (for a good definition of the Halloween Problem, read the document at [HTTP://EN.WIKIPEDIA.ORG/WIKI/HALLOWEEN_PROBLEM](http://en.wikipedia.org/wiki/Halloween_Problem)). The **Eager Spool** is in place in order to facilitate rollback operations.

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.

Now we get to the core of the **UPDATE** statement, the **Clustered Index Update** operator. In this case, the values being updated are part of a clustered index. So this operator identifies the rows to be updated, and updates them.

Last of all, we see the generic **T-SQL Language Element Catchall** operator, which tells us that an **UPDATE** operation has been completed.

From a performance perspective, one of the things to watch for is how the rows to be updated are retrieved. In this example, a **Non-Clustered Index Scan** was performed, which is not very efficient. Ideally, a **Clustered** or **Non-Clustered Index Seek** would be preferred from a performance standpoint, as either one of them would use less I/O to perform the requested **UPDATE**.

DELETE statements

What kind of execution plan is created with a DELETE statement? For example, let's run the following code and check out the execution plan.

```
BEGIN TRAN
DELETE FROM Person.EmailAddress
WHERE BusinessEntityID = 42
ROLLBACK TRAN
```

Listing 2.17

Figure 2.37 shows the estimated execution plan.

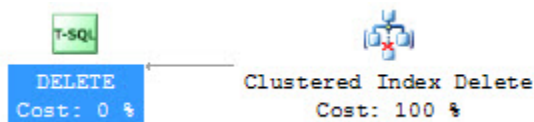


Figure 2.37

Not all execution plans are complicated and hard to understand. In this case, a direct access to the clustered index was used to identify the row that needed to be removed and it was deleted. You may still run into situations where you'll need to validate referential integrity with DELETE operations, just as you do with INSERTs.

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 the most common type of operators and we only looked at simple queries. So, if you decide to analyze a 200-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 the most complex of execution plans.

Chapter 3: Text and XML Execution Plans for Basic Queries

Chapter 2 described how to read graphical execution plans, building your understanding of what operators do and how the data flows from one operator to the next. Learning how to read graphical execution plans is not wasted time, because what you learned there also applies to reading text and XML execution plans. While these plans don't have icons and Property sheets, they consist of the exact same operators and properties; so, by learning how to read graphical execution plans, you also learn how to read text and XML execution plans.

In early versions of SQL Server, only text-based execution plans were available and many people found them hard to read, especially when dealing with complex plans. Microsoft eventually relented and introduced graphical execution plans in SQL Server 7, in addition to offering text execution plans. I find graphical execution plans much easier to read than text plans, and I guess I'm not the only database professional who feels this way, as text execution plans are on the SQL Server deprecation list and will eventually go away.

In SQL Server 2005, Microsoft modified the internal structure of execution plans. Instead of the proprietary binary format that they were formerly, now all graphical plans are actually XML underneath. We can access the XML in all these plans directly. Like text-based plans, XML plans can be difficult to read and analyze if you look straight at the raw XML code. So why did Microsoft decide to use XML for execution plans if they are difficult to read? There are several reasons for the change. Essentially, XML is a common file format that we can use programmatically, unlike text-based execution plans. This means you can use XQuery to access data within XML plans. XML plans also provide a much richer environment to store more execution plan details than ever before. In addition, XML plans are stored in a portable format that makes them easy to share with others. For example, I can send an XML plan to a fellow DBA, and she can use SSMS to graphically display and analyze it or run queries directly against it. Text-based plans, on the other hand, don't offer any of these benefits.

Text Execution Plans

So why should you even bother to learn about text execution plans if this feature is being deprecated? That's a question only you can answer. If you are working with SQL Server 2005 or later, for the most part, I suggest you focus your efforts on learning graphical execution plans, and understanding the benefits and uses of the XML plan file format. On the other hand, if you are still managing older versions of SQL Server, you may want to learn how to read text plans because they still crop up in books and articles about these older versions, and knowing how to read them might prove useful.

A text plan for a simple query

Let's start by examining the text plan for a query we saw in the previous chapter. First, as a reminder, we'll capture the graphical plan for the following query.

```
SELECT  e.BusinessEntityID ,
        e.JobTitle ,
        e.LoginID
FROM    HumanResources.Employee AS e
WHERE   e.LoginID = 'adventure-works\marc0';
```

Listing 3.1

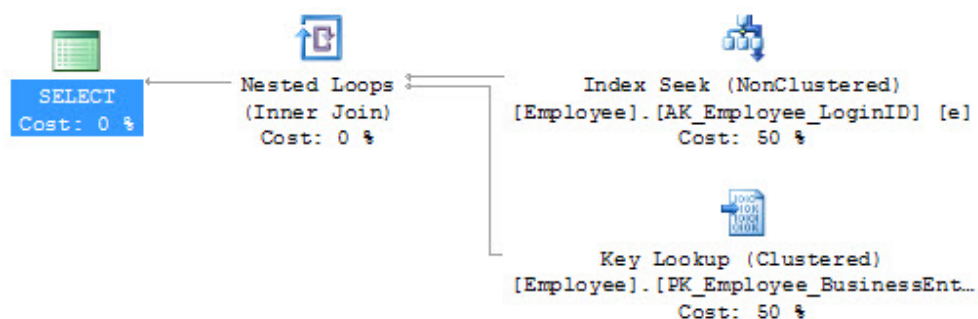


Figure 3.1

Now, we'll capture the equivalent text plan. There is no button to turn on in the GUI to capture text plans. Instead, we'll use a T-SQL statement to output the text plan. Turning on `SHOWPLAN_ALL` will allow you to collect *estimated* execution plans. This is a change to the behavior of your query window. No T-SQL code submitted after this statement is actually executed until you turn `SHOWPLAN_ALL` off again.

```
SET SHOWPLAN_ALL ON;
GO
SELECT  e.BusinessEntityID ,
        e.JobTitle ,
        e.LoginID
FROM    HumanResources.Employee AS e
WHERE   e.LoginID = 'adventure-works\marc0';
GO
SET SHOWPLAN_ALL OFF;
```

Listing 3.2

By default, the text plan displays in a spreadsheet-style grid format, as shown in Figure 3.2.¹

	StmtText	StmtId	NodeId	Parent	PhysicalOp	LogicalOp	Argument
1	SELECT e.BusinessEntityID, e.JobTitle, e...	1	1	0	NULL	NULL	1
2	↳Nested Loops(Inner Join, OUTER REFERENCES:(...	1	2	1	Nested Loops	Inner Join	OUTER REFERENCES:([e].[BusinessEntityID])
3	↳Index Seek(OBJECT:([AdventureWorks2008R2]...	1	3	2	Index Seek	Index Seek	OBJECT:([AdventureWorks2008R2].[HumanResources]...
4	↳Clustered Index Seek(OBJECT:([AdventureWor...	1	5	2	Clustered Index Seek	Clustered Index Seek	OBJECT:([AdventureWorks2008R2].[HumanResources]...

Figure 3.2

The layout of the results of the text plan consists of rows of information where each row is an operator. The plan displays the operators in the logical processing order from top to bottom, similar to reading a graphical plan from left to right. In this example, Row 1 is the parent node, the first operator in the execution plan, and the **StmtText** column for this row contains the text for the T-SQL statement. Scrolling right through the results,

¹ If you right-click in the query window of SSMS, you can select **Results To | Results to Text**, which offers a more conventional view of the text execution plan.

you'll find additional columns, not shown in Figure 3.2, such as **Type**, **Defined Values** and **EstimateRows**. The **Type** column (Figure 3.3) describes the node type, which is the type of operator.

Type
SELECT
PLAN_ROW
PLAN_ROW
PLAN_ROW

Figure 3.3

The parent node shows **Select** in this column, since that is the type of SQL statement executed. All other rows define the type as **PLAN_ROW**. For all **PLAN_ROW** nodes, the **StmtText** column defines the type of operator that the node represents.

A quick glance at the **StmtText** column for the remaining rows reveals that three operations took place: a **Nested Loops** inner join, an **Index Seek**, and a **Clustered Index Seek**.

While the logical order of the text plan is easy to follow, the physical order is much more difficult to discern. In order to understand the physical flow of operations, we are helped by the indentation of the data and the use of the pipe (|) to connect the statements, parent to child. We can also refer to the **NodeID** and **Parent** columns, which indicate the IDs of the current node and its parent node, respectively. You can see this in Figure 3.2 where the first row has a **NodeID** of 1 and a **Parent** of 0 while the second row has a **NodeID** of 2 and a **Parent** of 1. Within each indentation, or for every row that has the same **Parent** number, the operators execute from top to bottom. In this example, the **Index Seek** occurs before the **Clustered Index Seek**.

Moving to the two most indented rows, we start at Row 3 (**NodeID** 3) with the **Index Seek** operation. By extending the **StmtText** column (or by examining the **Argument** column), we can see that the **Index Seek** was against the **HumanResources.Employee** table (I've imposed some additional formatting just to make it a little easier to read).

```
--Index Seek
(OBJECT: ([AdventureWorks2008R2].[HumanResources].[Employee].[AK_Employee_LoginID] AS [e]),
 SEEK: ([e].[LoginID]=CONVERT_IMPLICIT(nvarchar(4000),[@1],0)) ORDERED FORWARD)
```

Listing 3.3

The **DefinedValues** column for a given **PLAN_ROW** contains a comma-delimited list of the values that the operator introduces to the data flow between operators. These values may be present in the current query (in the **SELECT** or **WHERE** clauses), as is the case for our **Index Seek** operator, where this column contains:

```
[e].[BusinessEntityID], [e].[LoginID]
```

Listing 3.4

Alternatively, they could be internal (intermediate) values that the query processor needs in order to process the query. Think of **DefineValues** as a temporary holding place for values used by the query optimizer as the operators execute.

Finally, note from the **EstimateRows** column that the **Index Seek** operation produces an estimated one row.

Next, Line 4 (**NodeID 5**), a **Clustered Index Seek** against **HumanResources.Employee**. This immediately illustrates what I would regard as a weakness with text plan format. It is not immediately obvious, as it was with the graphical plan, that this operation is, in fact, a **Key Lookup** operation. We need to scan through the whole contents of the **StmtText** column to find this out (Listing 3.5).

```
--Clustered Index Seek
(OBJECT: ([AdventureWorks2008R2].[HumanResources].[Employee].[PK_Employee_BusinessEntityID] AS [e]),
 SEEK: ([e].[BusinessEntityID]=[AdventureWorks2008R2].[HumanResources].[Employee].[BusinessEntityID] as [e].[BusinessEntityID])
 LOOKUP ORDERED FORWARD)
```

Listing 3.5

The **EstimateExecutions** column does not have a direct parallel in the graphical execution plan. If you capture the graphical plan for this query and examine the properties of the **Key Lookup** operator, rather than the number of executions, you will see values for the number of rebinds and rewinds. In this case, one rebind and zero rewinds. As you may remember from Chapter 1, a rebind or rewind occurs each time an operator is reinitialized.

We then move up and out one Node to Row 2 (**NodeID 2**), where we see the **Nested Loop** inner join that combines the results from the previous two operations. In this case, **DefinedValues** displays **Null**, meaning that the operation introduces no new values, and **OutputList** shows the **BusinessEntityId**, **JobTitle** and **LoginId** columns required by our query.

The remainder of the columns in the results grid, such as **EstimateRows**, **EstimateIO**, **TotalSubTreeCost**, and so on, mirror the information found in the ToolTips or the properties for graphical execution plans, so we won't cover them again here.

A text plan for a slightly more complex query

The text-based plan for the previous simple query was straightforward to read. However, with more complex queries, it quickly gets more difficult to read the text plan. Let's look at the estimated text plan for the query in Listing 3.6, containing a couple of joins and a **WHERE** clause.

```
SET SHOWPLAN_ALL ON;
GO

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

SET SHOWPLAN_ALL OFF;
GO

```

Listing 3.6

After you execute the above query, the results pane reveals the estimated text plan. Figure 3.4 shows the **StmtText** column of the results.

	StmtText
1	SELECT c.CustomerID, a.City, s.Name, sp.Name FROM Sales.Customer AS c JOIN Sales.Store AS s ...
2	I-Nested Loops(Inner Join, OUTER REFERENCES:([c].[TerritoryID]))
3	I-Nested Loops(Inner Join, OUTER REFERENCES:([c].[StoreID], [Expr1014]) WITH UNORDERED PREFETCH)
4	I I-Hash Match(Inner Join, HASH:([bea].[BusinessEntityID])=([c].[CustomerID]))
5	I I I-Nested Loops(Inner Join, OUTER REFERENCES:([a].[AddressID], [Expr1013]) WITH UNORDERED PREFETCH)
6	I I I I-Nested Loops(Inner Join, OUTER REFERENCES:([a].[AddressID], [Expr1012]) WITH UNORDERED PREFETCH)
7	I I I I I-Nested Loops(Inner Join, OUTER REFERENCES:([sp].[StateProvinceID]))
8	I I I I I-Index Seek(OBJECT:([AdventureWorks2008R2].[Person].[StateProvince].[AK_StateProvince_Name] AS [sp]), SE...
9	I I I I I-Index Seek(OBJECT:([AdventureWorks2008R2].[Person].[Address].[IX_Address_StateProvinceID] AS [a]), SEEK:(...
10	I I I I-Clustered Index Seek(OBJECT:([AdventureWorks2008R2].[Person].[Address].[PK_Address_AddressID] AS [a]), SEE...
11	I I I I-Index Seek(OBJECT:([AdventureWorks2008R2].[Person].[BusinessEntityAddress].[IX_BusinessEntityAddress_AddressID]...
12	I I I-Clustered Index Scan(OBJECT:([AdventureWorks2008R2].[Sales].[Customer].[PK_Customer_CustomerID] AS [c]))
13	I I-Clustered Index Seek(OBJECT:([AdventureWorks2008R2].[Sales].[Store].[PK_Store_BusinessEntityID] AS [s]), SEEK:([s].[Busi...
14	I-Clustered Index Seek(OBJECT:([AdventureWorks2008R2].[Sales].[SalesTerritory].[PK_SalesTerritory_TerritoryID] AS [st]), SEEK:(...

Figure 3.4

This is where the indentation of the data, and the use of the pipe (|) character to connect parent to child, really starts to be useful. Tracking to the innermost set of statements, we see an **Index Seek** operation against **IX_Address_StateProvinceId** on the **Address** table.

StmtText	StmtId	NodeID	Parent	PhysicalOp	LogicalOp	Argument	DefinedValues
1 SELECT c.[LastName], a.[City], cu.[AccountNum...	1	1	0	NULL	NULL	1	NULL
2 --Nested Loops(Inner Join, OUTER REFERENCES: ([c...	1	2	1	Nested Loops	Inner Join	OUTER REFERENCES: ([cu].[TerritoryID])	NULL
3 --Nested Loops(Inner Join, OUTER REFERENCES:...	1	3	2	Nested Loops	Inner Join	OUTER REFERENCES: ([i].[ContactID])	NULL
4 --Nested Loops(Inner Join, OUTER REFERENC...	1	4	3	Nested Loops	Inner Join	OUTER REFERENCES: ([cu].[CustomerID]) OPTIMIZED	NULL
5 --Compute Scalar(DEFINE: ([cu].[AccountNum...	1	6	4	Compute Scalar	Compute Scalar	DEFINE: ([cu].[AccountNumber]=[AdventureWorks].[S...	[cu].[AccountNum...
6 --Nested Loops(Inner Join, OUTER REFE...	1	7	6	Nested Loops	Inner Join	OUTER REFERENCES: ([ca].[CustomerID])	NULL
7 --Hash Match(Inner Join, HASH: ([a].[Ad...	1	8	7	Hash Match	Inner Join	HASH: ([a].[AddressID])=([ca].[AddressID])	NULL
8 --Nested Loops(Inner Join, OUTER ...	1	9	8	Nested Loops	Inner Join	OUTER REFERENCES: ([a].[AddressID])	NULL
9 --Index Seek(OBJECT: ([Adventure...	1	10	9	Index Seek	Index Seek	OBJECT: ([AdventureWorks].[Person].[Address].[PK_Ad...	[a].[AddressID]
10 --Clustered Index Seek(OBJECT: ([...	1	12	9	Clustered Index Seek	Clustered Index Seek	OBJECT: ([AdventureWorks].[Person].[Address].[PK_A...	[a].[City]
11 --Index Scan(OBJECT: ([AdventureW...	1	16	8	Index Scan	Index Scan	OBJECT: ([AdventureWorks].[Sales].[CustomerAddress...	[ca].[CustomerID]
12 --Compute Scalar(DEFINE: ([cu].[Accou...	1	18	7	Compute Scalar	Compute Scalar	DEFINE: ([cu].[AccountNumber]=isnull('AW'+[Adventur...	[cu].[AccountNum...
13 --Clustered Index Seek(OBJECT: ([Ad...	1	19	18	Clustered Index Seek	Clustered Index Seek	OBJECT: ([AdventureWorks].[Sales].[Customer].[PK_C...	[cu].[CustomerID]
14 --Clustered Index Seek(OBJECT: ([Adventur...	1	26	4	Clustered Index Seek	Clustered Index Seek	OBJECT: ([AdventureWorks].[Sales].[Individual].[PK_In...	[i].[ContactID]
15 --Clustered Index Seek(OBJECT: ([AdventureWor...	1	27	3	Clustered Index Seek	Clustered Index Seek	OBJECT: ([AdventureWorks].[Person].[Contact].[PK_C...	[c].[LastName]
16 --Clustered Index Seek(OBJECT: ([AdventureWorks...	1	28	2	Clustered Index Seek	Clustered Index Seek	OBJECT: ([AdventureWorks].[Sales].[SalesTerritory].[P...	[st].[Name]
8 --Index Seek(OBJECT: ([AdventureWorks2008R2].[Person].[StateProvince].[AK_StateProvince_Name] AS [sp]), SEEK: ([sp].[N...							
9 --Index Seek(OBJECT: ([AdventureWorks2008R2].[Person].[Address].[IX_Address_StateProvinceID] AS [a]), SEEK: ([a].[StatePr...							

Figure 3.5

This is how the plan displays the **WHERE** clause statement that limits the number of rows returned.

```
--Index Seek(OBJECT:
([AdventureWorks2008R2].[Person].[StateProvince].[AK_StateProvince_Name]
AS [sp]),
SEEK: ([sp].[Name]=N'New York') ORDERED FORWARD)
```

Listing 3.7

The output from this operator is the **StateProvinceId**, not a part of the **Select** list, but necessary for the operators that follow. This operator starts the query with a minimum number of rows to be used in all subsequent processing.

The **Index Seek** is followed by another **Index Seek** against the **Person.Address.IX_Address_StateProvinceID** table clustered index, using the **StateProvinceId** from the previous **Index Seek**.

Stepping out one level, the output from these two operations is joined via a **Nested Loop** join (Row 7).

	StmtText	StmtId	Nod...	Parent	PhysicalOp	LogicalOp	Argument	DefinedValues
1	SELECT c.[LastName] ,a.[City] ,cu.[AccountN...	1	1	0	NULL	NULL	1	NULL
2	I-Nested Loops(Inner Join, OUTER REFERENCES:[c...	1	2	1	Nested Loops	Inner Join	OUTER REFERENCES:[(cu].[TerritoryID])	NULL
3	I-Nested Loops(Inner Join, OUTER REFERENCES:[c...	1	3	2	Nested Loops	Inner Join	OUTER REFERENCES:[(i].[ContactID])	NULL
4	I-Nested Loops(Inner Join, OUTER REFERENCES:[c...	1	4	3	Nested Loops	Inner Join	OUTER REFERENCES:[(cu].[CustomerID]) OPTIMIZED	NULL
5	I-I-Compute Scalar(DEFINE:[(cu].[AccountNum...	1	6	4	Compute Scalar	Compute Scalar	DEFINE:[(cu].[AccountNumber]=[AdventureWorks].[S...	[cu].[AccountNur
6	I-I-I-I-Nested Loops(Inner Join, OUTER REFE...	1	7	6	Nested Loops	Inner Join	OUTER REFERENCES:[(ca].[CustomerID])	NULL
7	I-I-I-I-Hash Match(Inner Join, HASH:[(a).[Ad...	1	8	7	Hash Match	Inner Join	HASH:[(a].[AddressID])=[(ca).[AddressID])	NULL
8	I-I-I-I-I-Nested Loops(Inner Join, OUTER ...	1	9	8	Nested Loops	Inner Join	OUTER REFERENCES:[(a).[AddressID])	NULL
9	I-I-I-I-I-I-Index Seek(OBJECT:[Adventure...	1	10	9	Index Seek	Index Seek	OBJECT:[(AdventureWorks].[Person].[Address].[IX_Ad...	[a].[AddressID]
10	I-I-I-I-I-I-Clustered Index Seek(OBJECT:[(...	1	12	9	Clustered Index Seek	Clustered Index Seek	OBJECT:[(AdventureWorks].[Person].[Address].[PK_A...	[a].[City]
11	I-I-I-I-I-I-I-Index Scan(OBJECT:[AdventureW...	1	16	8	Index Scan	Index Scan	OBJECT:[(AdventureWorks].[Sales].[CustomerAddress...	[ca].[CustomerID]
12	I-I-I-I-I-I-I-Compute Scalar(DEFINE:[(cu).[Accou...	1	18	7	Compute Scalar	Compute Scalar	DEFINE:[(cu].[AccountNumber]=isnull(AW'+[Adventur...	[cu].[AccountNur
13	I-I-I-I-I-I-I-Clustered Index Seek(OBJECT:[(Ad...	1	19	18	Clustered Index Seek	Clustered Index Seek	OBJECT:[(AdventureWorks].[Sales].[Customer].[PK_C...	[cu].[CustomerID]
14	I-I-I-I-I-I-I-Clustered Index Seek(OBJECT:[(Adventu...	1	26	4	Clustered Index Seek	Clustered Index Seek	OBJECT:[(AdventureWorks].[Sales].[Individual].[PK_In...	[i].[ContactID]
15	I-I-Clustered Index Seek(OBJECT:[(AdventureWor...	1	27	3	Clustered Index Seek	Clustered Index Seek	OBJECT:[(AdventureWorks].[Person].[Contact].[PK_C...	[c].[LastName]
16	I-Clustered Index Seek(OBJECT:[(AdventureWorks)...	1	28	2	Clustered Index Seek	Clustered Index Seek	OBJECT:[(AdventureWorks].[Sales].[SalesTerritory].[P...	[st].[Name]
7	I-I-I-I-I-Nested Loops(Inner Join, OUTER REFERENCES:[(sp).[StateProvinceID])							
8	I-I-I-I-I-I-Index Seek(OBJECT:[AdventureWorks2008R2].[Person].[StateProvince].[AK_StateProvince_Name] AS [sp]), SEEK:[(sp].[N...							
9	I-I-I-I-I-I-I-Index Seek(OBJECT:[AdventureWorks2008R2].[Person].[Address].[IX_Address_StateProvinceID] AS [a]), SEEK:[(a).[StatePr...							

Figure 3.6

Following the pipes down from Row 9, the **Index Seek** operation, we reach Row 10, which holds one of the estimated, costliest operations in this query, an **Index Seek** against the entire **CustomerAddress** clustered index, another **Key Lookup** operation.

```
--Clustered Index Seek
(OBJECT: ([AdventureWorks2008R2].[Person].[Address].[PK_Address_AddressID]
AS [a]),
SEEK: ([a].[AddressID]=[AdventureWorks2008R2].[Person].[Address].[AddressID]
as [a].[AddressID]) LOOKUP ORDERED FORWARD)
```

Listing 3.8

The **Clustered Index Seek** produces an estimated one row in order to provide output for the next step out, a **Nested Loops** join that combines the output of the previous operations with the necessary data from the clustered index.

	StmtText
1	SELECT c.CustomerID, a.City, s.Name, st.Name FROM Sales.Customer AS c JOIN Sales.Store AS s ON c.StoreID = ...
2	-Nested Loops(Inner Join, OUTER REFERENCES:([c].[TerritoryID]))
3	-Nested Loops(Inner Join, OUTER REFERENCES:([c].[StoreID], [Expr1014]) WITH UNORDERED PREFETCH)
4	-Hash Match(Inner Join, HASH:([bea].[BusinessEntityID])=([c].[CustomerID]))
5	-Nested Loops(Inner Join, OUTER REFERENCES:([a].[AddressID], [Expr1013]) WITH UNORDERED PREFETCH)
6	-Nested Loops(Inner Join, OUTER REFERENCES:([a].[AddressID], [Expr1012]) WITH UNORDERED PREFETCH)
7	-Nested Loops(Inner Join, OUTER REFERENCES:([sp].[StateProvinceID]))
8	-Index Seek(OBJECT:([AdventureWorks2008R2].[Person].[StateProvince].[AK_StateProvince_Name] AS [sp]), SEEK:([sp].[Name]...
9	-Index Seek(OBJECT:([AdventureWorks2008R2].[Person].[Address].[IX_Address_StateProvinceID] AS [a]), SEEK:([a].[StateProvi...
10	-Clustered Index Seek(OBJECT:([AdventureWorks2008R2].[Person].[Address].[PK_Address_AddressID] AS [a]), SEEK:([a].[AddressI...
11	-Index Seek(OBJECT:([AdventureWorks2008R2].[Person].[BusinessEntityAddress].[IX_BusinessEntityAddress_AddressID] AS [bea]), SE...
12	-Clustered Index Scan(OBJECT:([AdventureWorks2008R2].[Sales].[Customer].[PK_Customer_CustomerID] AS [c]))
13	-Clustered Index Seek(OBJECT:([AdventureWorks2008R2].[Sales].[Store].[PK_Store_BusinessEntityID] AS [s]), SEEK:([s].[BusinessEntityID]=...
14	-Clustered Index Seek(OBJECT:([AdventureWorks2008R2].[Sales].[SalesTerritory].[PK_SalesTerritory_TerritoryID] AS [st]), SEEK:([st].[TerritoryID]...

Figure 3.7

Following the pipe characters down from the **Nested Loops** operation, we arrive at a **Clustered Index Scan** operation (Row 12). The scan operation's output in Row 12 is combined with the **Nested Loops** in a **Hash Match** from Row 4.

The **Hash Match** in Row 4 combines its output with another **Clustered Index Seek** in Row 13 (Figure 3.8).

	Stmt Text
1	SELECT c.CustomerID, a.City, s.Name, st.Name FROM Sales.Customer AS c JOIN Sales.Store AS s ON c.StoreID = ...
2	-Nested Loops(Inner Join, OUTER REFERENCES:([c].[TerritoryID]))
3	-Nested Loops(Inner Join, OUTER REFERENCES:([c].[StoreID], [Expr1014]) WITH UNORDERED PREFETCH)
4	-Hash Match(Inner Join, HASH:([bea].[BusinessEntityID])=([c].[CustomerID]))
5	-Nested Loops(Inner Join, OUTER REFERENCES:([a].[AddressID], [Expr1013]) WITH UNORDERED PREFETCH)
6	-Nested Loops(Inner Join, OUTER REFERENCES:([a].[AddressID], [Expr1012]) WITH UNORDERED PREFETCH)
7	-Nested Loops(Inner Join, OUTER REFERENCES:([sp].[StateProvinceID]))
8	-Index Seek(OBJECT:([AdventureWorks2008R2].[Person].[StateProvince].[AK_StateProvince_Name] AS [sp]), SEEK:([sp].[Name]...
9	-Index Seek(OBJECT:([AdventureWorks2008R2].[Person].[Address].[IX_Address_StateProvinceID] AS [a]), SEEK:([a].[StateProvi...
10	-Clustered Index Seek(OBJECT:([AdventureWorks2008R2].[Person].[Address].[PK_Address_AddressID] AS [a]), SEEK:([a].[AddressI...
11	-Index Seek(OBJECT:([AdventureWorks2008R2].[Person].[BusinessEntityAddress].[IX_BusinessEntityAddress_AddressID] AS [bea]), SE...
12	-Clustered Index Scan(OBJECT:([AdventureWorks2008R2].[Sales].[Customer].[PK_Customer_CustomerID] AS [c]))
13	-Clustered Index Seek(OBJECT:([AdventureWorks2008R2].[Sales].[Store].[PK_Store_BusinessEntityID] AS [s]), SEEK:([s].[BusinessEntityID]=...
14	-Clustered Index Seek(OBJECT:([AdventureWorks2008R2].[Sales].[SalesTerritory].[PK_SalesTerritory_TerritoryID] AS [st]), SEEK:([st].[TerritoryID]...

Figure 3.8

Finally, you can see the last steps which consist of another **Clustered Index Seek** in Row 14 that combines with the Loop Joins in Row 2 to put everything together for the **Select** statement.

As you see, reading text-based execution plans is not easy, and we have only taken a brief look at a couple of simple queries. Longer queries generate much more complicated plans, sometimes running to dozens of pages. I would suggest you focus on graphical execution plans, unless you have some special need where only text-based execution plans will meet your needs.

XML Execution Plans

I think it's safe to say that most database professionals prefer to view execution plans in graphical format. However, the big drawback in SQL Server 2000 and earlier, was that there was no "file format" for graphical execution plans, so there was no easy way to pass them around. This limitation was removed in SQL Server 2005, with the introduction of the XML plan behind every graphical plan.

To most people, an XML plan is simply a common file format in which to store a graphical execution plan, so that they can share it with other DBAs and developers.

I would imagine that very few people would prefer to read execution plans in the raw XML format. Having said that, there is one over-riding reason why you will absolutely want to use the raw XML data as a tool in your toolbox, and that is 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. This means we can display specific details of a plan and, by performing a search on specific terms, such as "Index Scan," we can track down specific, potentially problematic, aspects of a query. We'll see an example of this a little later.

XML plans can also be used in plan forcing, where we essentially dictate to the query optimizer that it should use only a specified plan to execute the query. We'll cover this topic in detail in Chapter 8, but it's worth noting here that, contrary to what you might hear, plan forcing does not bypass the optimization process.

In the following section, we take a brief look at the structure of XML as it exists behind graphical plans and in the plan cache.

An estimated XML plan

This section will show the information available to you in an XML plan. Just to get a sense of the XML structure, you'll follow through this example and "read" the XML in the same way as you have done previously for graphical and text plans. However, this example is not representative of how you're really going to use XML execution plans in your day-to-day work. You won't generally be browsing through the raw XML data in an attempt to find information.

We'll use the same execution plan that we evaluated as a text plan. Then, I'll show you how to retrieve estimated plans from the plan cache and retrieve information out of them directly, using XQuery, which is a much more useful exercise.

We can issue the `SHOWPLAN_XML` command in order to start capturing an estimated execution plan in the XML format (remember that any statements that follow the command will not be executed). We can then execute the required statement and then immediately deactivate `SHOWPLAN_XML` so that SQL Server will execute any subsequent statements we issue.

```
SET SHOWPLAN_XML ON;
GO
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
SET SHOWPLAN_XML OFF;
GO
```

Listing 3.9

Being able to wrap individual statements in this manner is a great way to capture an execution plan for an isolated statement within the larger set of statements that can make up many of the more complex queries. When you run the query above, you won't see results, but rather a link.

Click on the link and it will open as a graphical execution plan. In order to see the underlying XML, right-click on the graphical execution plan and select **Show Execution Plan XML**, and you will see something that looks like Figure 3.9.

```
<?xml version="1.0" encoding="utf-16"?>
<ShowPlanXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementCompId="1" StatementEstRows="2.0063" StatementId="1" StatementOptmLevel="FULL" Sta
        <StatementSetOptions ANSI_NULLS="true" ANSI_PADDING="true" ANSI_WARNINGS="true" ARITHABORT="true" CON
        <QueryPlan CachedPlanSize="72" CompileTime="30" CompileCPU="30" CompileMemory="1168">
          <RelOp AvgRowSize="116" EstimateCPU="6.38809E-05" EstimateIO="0" EstimateRebinds="0" EstimateRewind
            <OutputList>
              <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]" Table="[Customer]" Alias="[
              <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]" Table="[Store]" Alias="[s]"
              <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]" Table="[SalesTerritory]" Al
              <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Person]" Table="[Address]" Alias="[
            </OutputList>
            <NestedLoops Optimized="false">
              <OuterReferences>
                <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]" Table="[Customer]" Alias=
              </OuterReferences>
              <RelOp AvgRowSize="101" EstimateCPU="0.00101883" EstimateIO="0" EstimateRebinds="0" EstimateRew
                <OutputList>
                  <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]" Table="[Customer]" Alia
                  <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]" Table="[Customer]" Alia
                  <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]" Table="[Store]" Alias="
                  <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Person]" Table="[Address]" Alia
                </OutputList>
                <NestedLoops Optimized="false" WithUnorderedPrefetch="true">
                  <OuterReferences>
                    <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]" Table="[Customer]" Al
                    <ColumnReference Column="Expr1014" />
                  </OuterReferences>
                  <RelOp AvgRowSize="53" EstimateCPU="0.115178" EstimateIO="0" EstimateRebinds="0" EstimateRe
                    <OutputList>
                      <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]" Table="[Customer]"
                      <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]" Table="[Customer]"
                      <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]" Table="[Customer]"
                      <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Person]" Table="[Address]"
                    </OutputList>
                    <MemoryFractions Input="0" Output="0" />
                  <Hash>
                    <DefinedValues />
                    <HashKeysBuild>
                      <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Person]" Table="[Business
                    </HashKeysBuild>
                    <HashKeysProbe>
                      <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]" Table="[Customer]
```

Figure 3.9

The results are far too large to display here, but Figure 3.9 shows the first portion of the raw XML code used to create the graphical execution plan. XML data is more difficult than the graphical execution plans to take in all at once, but with the ability to expand and collapse elements using the "+" and "-" nodules down the left-hand side, the hierarchy of the data being processed becomes somewhat clearer.

The main reason to use XML is that it offers a defined and published format for people to use in programming, and to share the files. 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 [HTTP://SCHEMAS.MICROSOFT.COM/SQLSERVER/2004/07/SHOWPLAN/](http://schemas.microsoft.com/sqlserver/2004/07/showplan/).

After the familiar BatchSequence, Batch, Statements and StmtSimple elements (described in Chapter 1), the first point of real interest is in the physical attributes of the QueryPlan.

```
<QueryPlan CachedPlanSize ="52" CompileTime="29293" CompileCPU="6277"  
          CompileMemory="520">
```

Listing 3.10

This describes the size of the plan in the plan cache, along with the amount of time, CPU cycles and memory used by the plan. This information is available in graphical execution plans; you just have to look at the ToolTips window or the **Properties** page for the root operator (in this case it would be a **Select** operator). In SQL Server 2008 and above, all information available in the XML plan is available in the graphical execution plan. In 2005, certain values, such as missing index information, were only available in the XML data.

In some cases, but not this example, we can see an element labeled MissingIndexes. This contains information about tables and columns that do not have an index available to the execution plan created by the optimizer. 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. You also have ability to correlate the behavior of this query with other queries running on the system, so taking the missing index information in isolation can be a problematic approach.

```
<MissingIndexes>
  <MissingIndexGroup Impact="30.8535">
    <MissingIndex Database="[AdventureWorks]" Schema="[Sales]"
                  Table="[CustomerAddress]">
      <ColumnGroup Usage="EQUALITY">
        <Column Name="[AddressID]" ColumnId="2" />
      </ColumnGroup>
      <ColumnGroup Usage="INCLUDE">
        <Column Name="[CustomerID]" ColumnId="1" />
      </ColumnGroup>
    </MissingIndex>
  </MissingIndexGroup>
</MissingIndexes>
```

Listing 3.11

The execution plan then lists, via the RelOp nodes, the XML element name for operators, the various physical operations that it anticipates performing, according to the data supplied by the optimizer. The first node, with NodeId=0, refers to the first operator in logical order, which is NestedLoops (Listing 3.12).

```
<RelOp NodeId="0" PhysicalOp="Nested Loops" LogicalOp="Inner Join"
EstimateRows="2.0063" EstimateIO="0" EstimateCPU="6.38809e-005" AvgRowSize="116"
EstimatedTotalSubtreeCost="1.09119" Parallel="0" EstimateRebinds="0"
EstimateRewinds="0">
  <OutputList>
    <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]"
                    Table="[Customer]" Alias="[c]" Column="CustomerID" />
    <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]"
                    Table="[Store]" Alias="[s]" Column="Name" />
    <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]"
                    Table="[SalesTerritory]" Alias="[st]" Column="Name" />
    <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Person]"
                    Table="[Address]" Alias="[a]" Column="City" />
  </OutputList>
```

```
<NestedLoops Optimized="0">
  <OuterReferences>
    <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]"
      Table="[Customer]" Alias="[c]" Column="TerritoryID" />
  </OuterReferences>
```

Listing 3.12

The information within this element will be familiar to you from the ToolTip window and **Properties** sheets from the graphical execution plans. Notice that, unlike text plans, which just displayed "EstimateExecutions=0.001581," the XML plan includes the estimated number of rebinds and rewinds. This can often give you a more accurate idea of what occurred within the query, such as how many times the operator executed.

For example, for NodeId="20", the final **Clustered Index Seek**, associated with the **Nested Loops** join in NodeId="0", we see:

```
<RelOp NodeId="20" PhysicalOp="Clustered Index Seek"
  LogicalOp="Clustered Index Seek"
  EstimateRows="1" EstimateIO="0.003125" EstimateCPU="0.0001581"
  AvgRowSize="28" EstimatedTotalSubtreeCost="0.00554117"
  TableCardinality="10" Parallel="0" EstimateRebinds="12.8482"
  EstimateRewinds="1.43427">
```

Listing 3.13

Returning to **Node 0** at the top of the XML, the next element listed is the **OutputList** element with a list of **ColumnReference** elements, each containing a set of attributes to describe that column (Listing 3.14).

```
<OutputList>
  <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]"
    Table="[Customer]" Alias="[c]" Column="CustomerID" />
  <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]"
    Table="[Store]" Alias="[s]" Column="Name" />
  <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]"
    Table="[SalesTerritory]" Alias="[st]" Column="Name" />
```



```
<ColumnReference Database="[AdventureWorks2008R2]" Schema="[Person]"
                 Table="[Address]" Alias="[a]" Column="City" />
</OutputList>
```

Listing 3.14

This makes XML not only easier to read, but much more readily translated directly back to the original query. The output described above is from the references to the schema `Sales` and the tables `Customer` (aliased as "c"), `Store` (aliased as "s") and `SalesTerritory` (aliased as "st"), as well as the schema `Person` and the table `Address` (aliased as "a"), in order to output the required columns (`CustomerID`, `Name`, `Name`, and `City`). The names of the operator elements are the same as the operators you would see in the graphical execution plans, and the details within the attributes are usually those represented in the ToolTip windows or in the **Properties** window.

Finally, for Node 0 in the estimated plan, in Listing 3.15 we see some more information about the `Nested Loops` operation, such as the table involved, along with the table's alias.

```
<NestedLoops Optimized="0">
  <OuterReferences>
    <ColumnReference Database="[AdventureWorks2008R2]" Schema="[Sales]"
                     Table="[Customer]" Alias="[c]" Column="TerritoryID" />
  </OuterReferences>
```

Listing 3.15

As you can see, reading plans directly through the XML is not easy. Before moving on to querying the XML directly, let's quickly examine the differences between estimated and actual plans in an XML plan.

An actual XML plan

Now let's try executing the same query as we just did, but this time execute and collect the actual XML plan. We won't go through the plan in detail again, just highlight the main differences between estimated and actual XML plans.

```
SET STATISTICS XML ON;
GO
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
SET STATISTICS XML OFF;
GO
```

Listing 3.16

We open the raw XML code for the actual plan in the same way as for the estimated plan. Listing 3.17 shows that the `QueryPlan` element contains additional information, including the `DegreeOfParallelism` (more on parallelism in Chapter 8) and `MemoryGrant`, which is the amount of memory needed for the execution of the query.

```
<QueryPlan DegreeOfParallelism="0" MemoryGrant="1024" CachedPlanSize="80"
CompileTime="36" CompileCPU="35" CompileMemory="1224">
```

Listing 3.17

The other major difference between the actual XML execution plan and the estimated one is that the actual plan includes an element called `RunTimeInformation`, showing the thread, actual rows, and the number of executions prior to the same final **Nested Loop** information.

```
<RunTimeInformation>
  <RunTimeCountersPerThread Thread="0" ActualRows="1" ActualEndOfScans="1"
                                ActualExecutions="1" />
</RunTimeInformation>...
```

Listing 3.18

Querying the XML

One of the real advantages of having plans in an XML format is the ability to use XQuery to interrogate them. 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 in SQL Server.

XML querying is inherently costly, and queries against XML in the plan cache might seriously affect performance on the server, so apply due diligence when running these types of queries. However, certain information, such as the missing index information mentioned previously, is only available through the XML in query plans, so you will want to use them when appropriate.

A good practice is to apply some filtering criteria, via the DMVs that give us access to the plan cache, and so limit the amount of data accessed. I show a simple example of that in Listing 3.18. Better still, we could move the XML plans into a second table and then run the XQuery against that table, in order to avoid placing too much load directly against the plan cache.

The simple example in Listing 3.18 returns a list of operators within queries currently stored in the plan cache. It illustrates how we can construct queries against the plan cache, using the DMVs, but I would still hesitate before running this query on a production system if that system was already under stress. Exercise caution using these XML queries.

```
SELECT TOP 3
    RelOp.op.value('declare default element namespace "http://schemas.
microsoft.com/sqlserver/2004/07/showplan";
    @PhysicalOp', 'varchar(50)') AS PhysicalOp,
    dest.text,
    deqs.execution_count,
    RelOp.op.value('declare default element namespace "http://schemas.
microsoft.com/sqlserver/2004/07/showplan";
    @EstimatedTotalSubtreeCost', 'float') AS EstimatedCost
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
    CROSS APPLY deqp.query_plan.nodes('declare default element namespace
"http://schemas.microsoft.com/sqlserver/2004/07/showplan";
    //RelOp') RelOp (op)
ORDER BY deqs.execution_count DESC
```

Listing 3.18

This query returns the first three operators from the most frequently called query in the plan cache. The basic construct is easy enough to follow. In the FROM clause, three DMVs are referenced, `sys.dm_exec_query_stats`, `sys.dm_exec_sql_text`, and `sys.dm_exec_query_plan`, along with a reference to the `query_plan` column within `sys.dm_exec_query_plan` and to the `.nodes` of the XML. Apart from that, it's all XQuery. The results from my system look as shown in Figure 3.10.

	PhysicalOp	text	execution_count	EstimatedCost
1	Nested Loops	SELECT this_[Id] as Id1_0_0_, this_[CreatedDat...	9805	0.00689714
2	Index Scan	SELECT this_[Id] as Id1_0_0_, this_[CreatedDat...	9805	0.0032853
3	Clustered Index Seek	SELECT this_[Id] as Id1_0_0_, this_[CreatedDat...	9805	0.0035993

Figure 3.10

Listed along with the physical operator is the query text, the number of executions of that query, and the estimated cost. Any of the values stored within the XML are available for querying. This completely opens up what is possible with XML and execution plans (see Chapter 7 for more details).

Summary

Text plans are easier to read than XML, but the information in them is much more limited than that provided by XML or a graphical plan. As the queries get more complex, it also gets harder to understand the flow of data within a text plan. Since Microsoft plans to deprecate text plans, I would only spend time working on using them if you are in an older version of SQL Server and need to share execution plans with others.

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, 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.

Chapter 4: Understanding More Complex Query Plans

As we've seen, even relatively simple queries can generate complicated execution plans. Complex T-SQL statements generate ever-expanding execution plans that become more 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.

In Chapters 2 and 3, we dealt with single statement T-SQL queries. In this chapter, we'll extend that to consider stored procedures, temporary tables, table variables, `MERGE` statements, and more.

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 and so on.

Stored procedures

The best place to get started is with stored procedures. We'll create a new one for AdventureWorks2008R2.

```
CREATE PROCEDURE [Sales].[spTaxRateByState]
    @CountryRegionCode NVARCHAR(3)
AS
    SET NOCOUNT ON ;

    SELECT  [st].[SalesTaxRateID],
            [st].[Name],
            [st].[TaxRate],
            [st].[TaxType],
            [sp].[Name] AS StateName
```

Chapter 4: Understanding More Complex Query Plans

```
FROM [Sales].[SalesTaxRate] st
JOIN [Person].[StateProvince] sp
ON [st].[StateProvinceID] = [sp].[StateProvinceID]
WHERE [sp].[CountryRegionCode] = @CountryRegionCode
ORDER BY [StateName] ;
GO
```

Listing 4.1

Which we can then execute:

```
EXEC [Sales].[spTaxRateByState] @CountryRegionCode = 'US';
```

Listing 4.2

The resulting actual execution plan is quite simple, as shown in Figure 4.1.

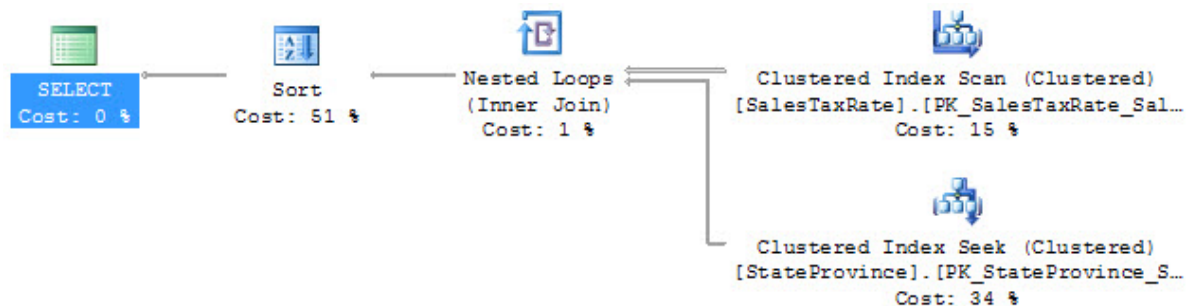


Figure 4.1

Starting from the right, we see a **Clustered Index Scan** operator, which gets the list of tax rates. The query combines this data with data pulled from the **States** table, based on the parameter, **@CountryRegionCode**, visible in the ToolTip or the **Properties** window, through a **Nested Loops** operation.

The combined data is passed to a **Sort** operation, which, at 51%, is the operator with the highest estimated cost in the plan. Looking at the properties in Figure 4.2, you can see that the optimizer feels that it has found the best possible plan.

Misc	
Cached plan size	24 B
CompileCPU	7
CompileMemory	352
CompileTime	7
Degree of Parallelism	1
Estimated Number of Rows	29
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0227401
Logical Operation	
Memory Grant	1024
Optimization Level	FULL
Parameter List	@CountryRegionCode
Column	@CountryRegionCode
Parameter Compiled Value	N'US'
Parameter Runtime Value	N'US'
Physical Operation	
QueryHash	0x8C888ED1D6B1989D
QueryPlanHash	0x5411436BC7EBF7A7
Reason For Early Termination Of State	Good Enough Plan Found
Set Options	ANSI_NULLS: True, ANSI_PADDING: True, ANSI_WARNINGS
Statement	SELECT [st].[SalesTaxRateID], [st].[Name], [st].[

Figure 4.2

You can see this in the **Reason for Early Termination Of Statement** property, where it says **Good Enough Plan Found**. Another interesting point is that you can see, in the **Parameter Compiled Value** property, the value that the optimizer used to compile the plan. Below it is the **Parameter Runtime Value**, showing the value when this query was called.

These are useful properties to help you better understand the decisions made by the optimizer and how the plan was compiled. The section on *Statistics and indexes*, later in this chapter, provides more information about compiled values.

While this plan isn't complex, the 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.

Using a sub-select

A common but sometimes problematic approach to querying data is to select information from other tables within the query, but not as part of a `JOIN` statement. Instead, a completely different query is used in the `WHERE` clause, as a filtering mechanism. Using AdventureWorks2008R2 as an example, the `Production.ProductListPriceHistory` table maintains a running list of changes to product price. We'll use a sub-select within the `ON` clause of the join to limit the data to only the latest versions of the `ListPrice`.

```
SELECT  p.Name,
        p.ProductNumber,
        ph.ListPrice
FROM    Production.Product p
        INNER JOIN Production.ProductListPriceHistory ph
ON      p.ProductID = ph.ProductID
        AND ph.StartDate = (SELECT TOP (1)
                               ph2.StartDate
                              FROM    Production.ProductListPriceHistory ph2
                              WHERE   ph2.ProductID = p.ProductID
                              ORDER BY ph2.StartDate DESC
                             ) ;
```

Listing 4.3

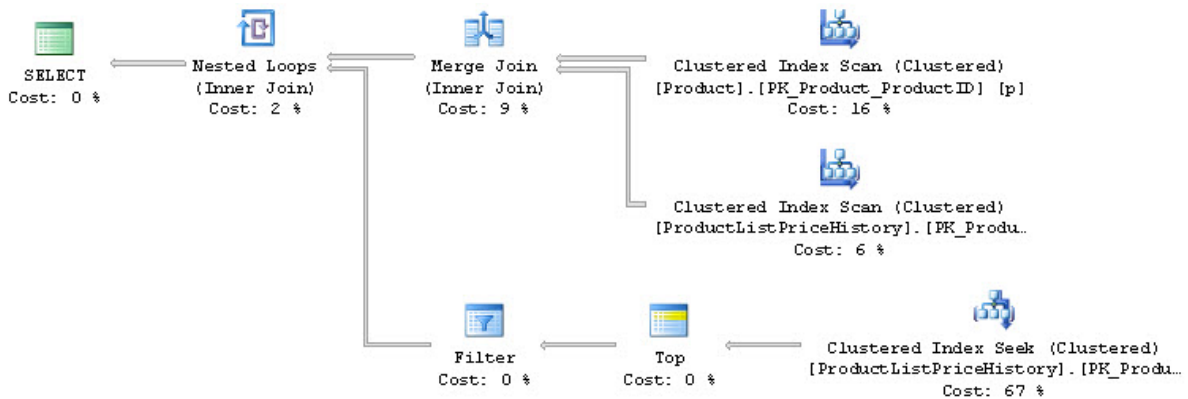


Figure 4.3

What appears to be a somewhat complicated query turns out to have a straightforward execution plan. Reading it in physical operator order, there are two **Clustered Index Scans** against `Production.Product` and `Production.ProductListPriceHistory`. These two data streams are combined using the **Merge Join** operator.

The **Merge Join** requires that both data inputs are ordered on the join key, in this case, `ProductId`. The data resulting from a clustered index is always retrieved in an ordered fashion, so no additional sort operation is required here.

A **Merge Join** takes a row each from the two ordered inputs and compares them. The inputs are sorted, so the operation will only scan each input one time (except in the case of a many-to-many join; more on this shortly). The operation will scan through the right side of the operation until it reaches a row that is different from the row on the left side. At that point, it will step the left side forward a row and begin scanning the right side until it reaches a changed data point. This operation continues until all the rows are processed. With the data already sorted, as in this case, it makes for a very fast operation.

Although we don't have this situation in our example, many-to-many **Merge Joins** create a worktable to complete the operation. The creation of a worktable adds a great deal of cost to the operation because it will mean writing data out to `tempdb`. However, the

Merge Join operation will generally still be less costly than the use of a **Hash Match** join, which is the other choice the query optimizer can make to solve the same type of join. We can see that a worktable was not necessary, because the ToolTips property labeled **Many to Many** (see figure below) is set to **False**.

Merge Join	
Match rows from two suitably sorted input tables exploiting their sort order.	
Physical Operation	Merge Join
Logical Operation	Inner Join
Actual Number of Rows	395
Estimated I/O Cost	0
Estimated CPU Cost	0.0075174
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0.0075204 (9%)
Estimated Subtree Cost	0.0254437
Estimated Number of Rows	294,969
Estimated Row Size	108 B
Actual Rebinds	0
Actual Rewinds	0
Many to Many	False
Node ID	1
Where (join columns)	
([AdventureWorks2008R2].[Production].[ProductListPriceHistory].ProductID) = ([AdventureWorks2008R2].[Production].[Product].ProductID)	
Output List	
[AdventureWorks2008R2].[Production].[Product].ProductID, [AdventureWorks2008R2].[Production].[Product].Name, [AdventureWorks2008R2].[Production].[Product].ProductNumber, [AdventureWorks2008R2].[Production].[ProductListPriceHistory].StartDate, [AdventureWorks2008R2].[Production].[ProductListPriceHistory].ListPrice	

Figure 4.4

Next, we move down to the **Clustered Index Seek** operator in the lower right side of the execution plan. Interestingly enough, this step accounts for 67% of the cost of the query because the seek operation returned all 395 rows from the query. The data was only limited to the TOP (1) after the rows were returned. A scan in this case, since all the rows were returned, might have worked better. The only way to know for sure would be to add a hint to the query to force a **Table Scan** and see if performance is better or worse.

The **Top** operator simply limits the number of returned rows to the value supplied within the query, in this case "1." The **Filter** operator is then applied to limit the returned values to only those where the dates match the main table. In other words, a join occurs between the [Production].[Product] table and the [Production].[ProductList-PriceHistory] table, where the column [StartDate] is equal in each. See the ToolTip in Figure 4.5.

Filter	
Restricting the set of rows based on a predicate.	
Physical Operation	Filter
Logical Operation	Filter
Actual Number of Rows	293
Estimated I/O Cost	0
Estimated CPU Cost	0.0000005
Estimated Number of Executions	294.969
Number of Executions	395
Estimated Operator Cost	0.0001416 (0%)
Estimated Subtree Cost	0.0545963
Estimated Number of Rows	1
Estimated Row Size	9 B
Actual Rebinds	0
Actual Rewinds	0
Node ID	5
Predicate	
[AdventureWorks2008R2].[Production].	
[ProductListPriceHistory].[StartDate] as [ph2].	
[StartDate]=[AdventureWorks2008R2].[Production].	
[ProductListPriceHistory].[StartDate] as [ph].[StartDate]	

Figure 4.5

The two data streams are then joined through a **Nested Loops** operator, to produce the final result.

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 second `SELECT` statement within a set of parentheses in the `FROM` clause of an outer `SELECT` query. Once you apply an alias, this `SELECT` statement is treated as a table by the T-SQL code, outside of the derived table definition. 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 have to maintain history.

SQL Server 2005 introduced a new type of derived table, created using one of the two forms of the **Apply** operator, **Cross Apply** or **Outer Apply**. The **Apply** operator allows us to use a table-valued function, or a derived table, to compare values between the function and each row of the table to which the function is being "applied."

If you are not familiar with the **Apply** operator, check out [HTTP://TECHNET.MICROSOFT.COM/EN-US/LIBRARY/MSI75156.ASPX](http://technet.microsoft.com/en-us/library/msi75156.aspx).

Listing 4.4 shows the rewritten query. Remember, both this and the query in Listing 4.3 return identical data; they are just written differently.

```
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 4.4

The introduction of this new functionality changes the execution plan substantially, as shown in Figure 4.6.

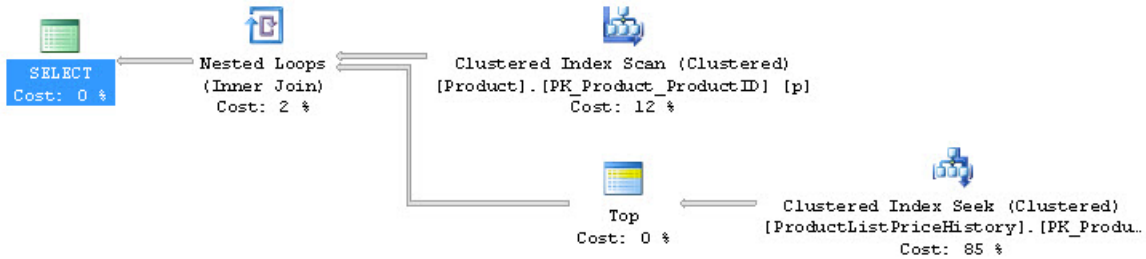


Figure 4.6

The TOP statement is now be applied row by row within the control of the APPLY function, so the second **Index Scan** against the ProductListPriceHistory table, and the **Merge Join** that joined the tables together, are no longer needed. Furthermore, only the **Index Seek** and Top operations are required to provide data back for the Nested Loops operation.

So, which method of writing this query do you think is the most efficient? One way to find out is to run each query with the SET STATISTICS IO option set to ON. With this option set, SQL Server displays I/O statistics as part of the Messages returned by the query.

When we run the first query, which uses the sub-select, the results are:

```
(293 row(s) affected)
Table 'ProductListPriceHistory'. Scan count 396, logical reads 795
Table 'Product'. Scan count 1, logical reads 15, physical reads 0
```

If we run the query using a derived table, the results are:

```
(293 row(s) affected)
Table 'ProductListPriceHistory'. Scan count 504, logical reads 1008
Table 'Product'. Scan count 1, logical reads 15, physical reads 0,
```

Although both queries returned identical result sets, the sub-select query uses fewer logical reads (795) versus the query written using the derived table (1008 logical reads), along with fewer scans (396 to 504). This is a result of the extra work done by the **Nested Loops** compared to the **Merge Join** operator used in Figure 4.3.

This gets more interesting if we add the following **WHERE** clause in Listing 4.5 to each of the previous queries.

```
WHERE [p].[ProductID] = '839'
```

Listing 4.5

When we re-run the original query with the added **WHERE** clause, we get the plan shown in Figure 4.7.

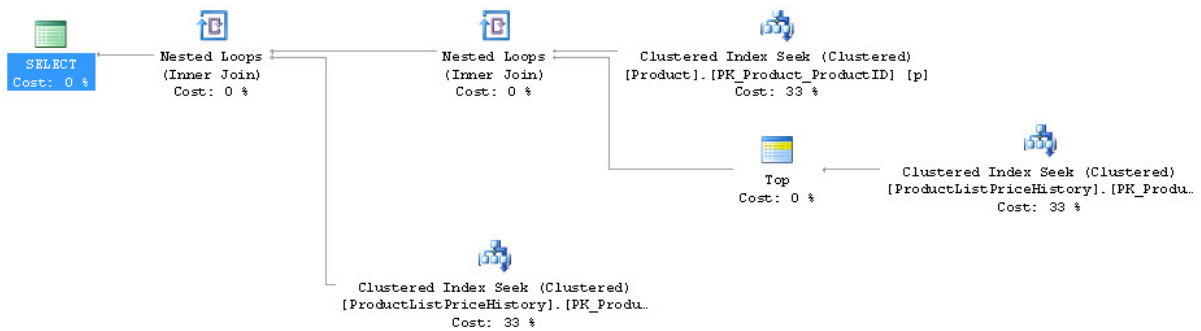


Figure 4.7

The **Filter** operator is gone but, more interestingly, the operators and costs have changed. Instead of **Index Scans** and the inefficient (in this case) **Index Seeks** mixed together, we have three, clean **Clustered Index Seeks** with an equal cost distribution. That can be an indication for a well-tuned query.

If we add the **WHERE** clause to the derived table query, we see the plan shown in Figure 4.8.

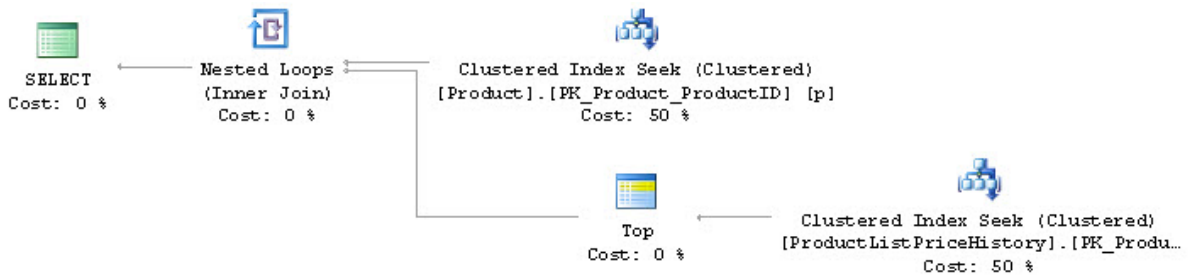


Figure 4.8

This plan is almost identical to the one seen in Figure 4.6, 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 rows needed, rather than having to scan through them all in order to find the correct rows to return.

Now, let's compare the I/O statistics for each of the queries, which return the same physical row. When we run the query with the sub-select, we get:

```
(1 row(s) affected)
Table 'ProductListPriceHistory'. Scan count 1, logical reads 4
Table 'Product'. Scan count 0, logical reads 2, physical reads 0
```

When we run the query with the derived table, we get:

```
(1 row(s) affected)
Table 'ProductListPriceHistory'. Scan count 1, logical reads 2
Table 'Product'. Scan count 0, logical reads 2, physical reads 0
```

Now, with the addition of a **WHERE** clause, the derived query is more efficient, with only 2 logical reads, versus the sub-select query with 4 logical reads.

The lesson to learn here is that in one set of circumstances a particular T-SQL method may be exactly what you need, and yet, in another circumstance, that same syntax negatively affects performance. The **Merge Join** made for a very efficient query when we were dealing with inherent scans of the data, but was not used, nor applicable, when the introduction of the **WHERE** clause reduced the data set. With the **WHERE** clause in place, the sub-select became, relatively, more costly 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.

Common table expressions

SQL Server 2005 introduced a T-SQL construct, whose behavior appears similar to derived tables, called a common table expression (CTE). 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 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. (For more details on CTEs, check out this article on Simple-Talk: WWW.SIMPLE-TALK.COM/SQL/SQL-SERVER-2005/SQL-SERVER-2005-COMMON-TABLE-EXPRESSIONS/.)

One of the classic use cases for a CTE is to create a **recursive query**. AdventureWorks2008R2 takes advantage of this functionality in a classic recursive exercise, listing employees and their managers. The procedure in question, `uspGetEmployeeManagers`, is as shown in Listing 4.6.

```
ALTER PROCEDURE [dbo].[uspGetManagerEmployees]
    @BusinessEntityID [int]
AS
BEGIN
    SET NOCOUNT ON;
    WITH [EMP_cte] ( [BusinessEntityID], [OrganizationNode],
                    [FirstName], [LastName], [RecursionLevel] )
        -- CTE name and columns
```

```

AS ( SELECT  e.[BusinessEntityID] ,
            e.[OrganizationNode] ,
            p.[FirstName] ,
            p.[LastName] ,
            0 -- Get the 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 )
END;

```

Listing 4.6

Let's execute this procedure, capturing the actual XML plan.

```
SET STATISTICS XML ON;  
GO  
EXEC [dbo].[uspGetEmployeeManagers] @EmployeeID = 9;  
GO  
SET STATISTICS XML OFF;  
GO
```

Listing 4.7

We get an execution plan of reasonable complexity, so let's break it down into sections in order to evaluate it. We will examine parts of the XML plan alongside the graphical plan.

Figure 4.9 displays the top right-hand section of the graphical plan.

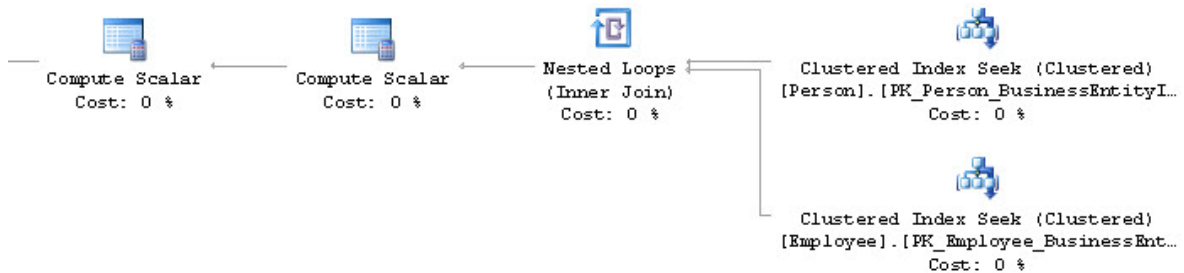


Figure 4.9

A **Nested Loops** join takes the data from two **Clustered Index Seeks** against **HumanResources.Employee** and **Person.Person**. The **Scalar** operator puts in the constant "0" from the original query for the derived column, **RecursionLevel**. The second scalar, carried to a later operator, is an identifier used as part of the **Concatenation** operation.

This data is fed into a **Concatenation** operator, which scans multiple inputs and produces one output. The optimizer uses it most commonly to implement the UNION ALL operation from T-SQL.

Figure 4.10 displays the bottom right-hand section of the plan.

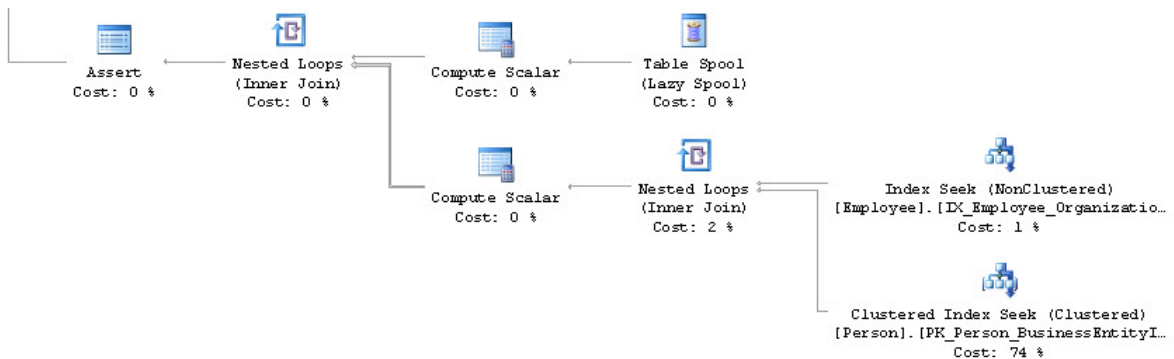


Figure 4.10

This is where things get interesting. SQL Server implements the recursion methods via the **Table Spool** operator. This operator provides the mechanism for looping through the records multiple times. As noted in Chapter 2, this operator takes each of the rows and stores them in a hidden temporary object stored in the tempdb database. Later in the execution plan, if the operator is rewound (say, due to the use of a **Nested Loops** operator in the execution plan) and no rebinding is required, the spooled data can be reused, instead of having to rescan the data again. As the operator loops through the records, they are joined to the data from the tables as defined in the second part of the UNION ALL definition within the CTE.

You can see the identifier for each operator in the **Properties** page. For our current execution plan, the **Table Spool** operator has a **NodeId=21**. If you look up **NodeId 21** in the XML plan, you can see the **RunTimeInformation** element. This element contains the actual query information, data that would not be available for an estimated plan.

Listing 4.8

Figure 4.II shows the final section of the graphical plan.



153

MERGE

With SQL Server 2008, Microsoft introduced the `MERGE` statement. This is a method for updating data in your database in a single statement, instead of one statement for `INSERTs`, one for `UPDATEs` and another for `DELETEs`. The nickname for this is an "upsert." The simplest application of the `MERGE` statement 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 4.9 `UPDATEs` or `INSERTs` rows to the `Purchasing.Vendor` table.

```
DECLARE @BusinessEntityId INT = 42,
        @AccountNumber NVARCHAR(15) = 'SSHI',
        @Name NVARCHAR(50) = 'Shotz Beer',
        @CreditRating TINYINT = 2,
        @PreferredVendorStatus BIT = 0,
        @ActiveFlag BIT = 1,
        @PurchasingWebServiceURL NVARCHAR(1024) = '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 Name = vn.Name,
          CreditRating = vn.CreditRating,
          PreferredVendorStatus = vn.PreferredVendorStatus,
          ActiveFlag = vn.ActiveFlag,
```

```
PurchasingWebServiceURL = vn.PurchasingWebServiceURL,  
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 4.9

I use a rollback on the transaction above to avoid changing the data in the AdventureWorks2008R2 database. Despite the fact that a rollback occurs, it's still possible to get an actual execution plan from the query because the query did complete its operations. This query uses the alternate key that exists on the table on the AccountNumber column. If the value matches, the query will run an UPDATE, and if it doesn't, it will perform an INSERT. The resulting execution plan is a bit large, so I'll break it down in physical operator order, working from the right.

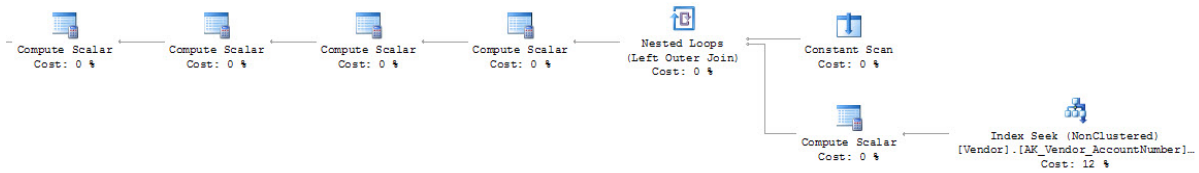


Figure 4.12

This first section, in essence, contains a series of steps to prepare for the main operations to come. The **Constant Scan** operator creates a row into which all the **Compute Scalar** operations will be able to load data. The **Index Seek** operation, against the **Vendor**. **AK_Vendor_AccountNumber** index, pulls back matching rows, if any. In this case, the ToolTip for the data flow between the **Index Seek** and the first **Compute Scalar**, reveals zero rows.

Actual Number of Rows	0
Estimated Number of Rows	1
Estimated Row Size	37 B
Estimated Data Size	37 B

Figure 4.13

The hard part of reading a plan like this is trying to figure out what each of the **Compute Scalar** operations are doing. To find out, we have to walk through two values in particular, the **Defined Values** and the **Output List**, both of which we can access through the **Properties** window, although the Output List is also available in the operator's ToolTip. Working from the right again, the first **Compute Scalar** creates a value called **TrgPrb1002** and sets it equal to "1." Via a **Nested Loops** operator, the plan combines this value with the row from the **Constant Scan**.

The next **Compute Scalar** operator performs a little calculation:

```
ForceOrder(CASE WHEN [TrgPrb1002] IS NOT NULL THEN (1) ELSE (4) END)
```

It creates a new value, `Action1004`, and since `TrgPrb1002` is not null, the value is set to "1." The next **Compute Scalar** operator loads all the variable values into the row, and performs one other calculation:

```
Scalar Operator(CASE WHEN [Action1004]=(4) THEN [@AccountNumber] ELSE  
[AdventureWorks2008R2].[Purchasing].[Vendor].[AccountNumber] as [v].[AccountNumber]  
END)
```

Here, we can begin to understand what's happening. The first **Compute Scalar** output, `TrgPrb1002`, determined if the row existed in the table. If it existed, then the second scalar would have set `Action1004` equal to 4, meaning that the row did exist, and this new **Compute Scalar** would have used the value from the table, but instead, it's using the variable `@AccountNumber`, since an INSERT is needed.

Moving to the left, the next **Scalar** operator validates what `Action1004` is and sets a new value, `Expr1042`, based on this formula:

```
Scalar Operator(CASE WHEN [Action1004] = (1) THEN (0) ELSE [Action1004] END)
```

We know that `Action1004` is set to 1, so this expression will be, as well.

The final **Scalar** operator adds two values for later use in the execution plan. Finally, we're ready to move on with the rest of the execution plan:

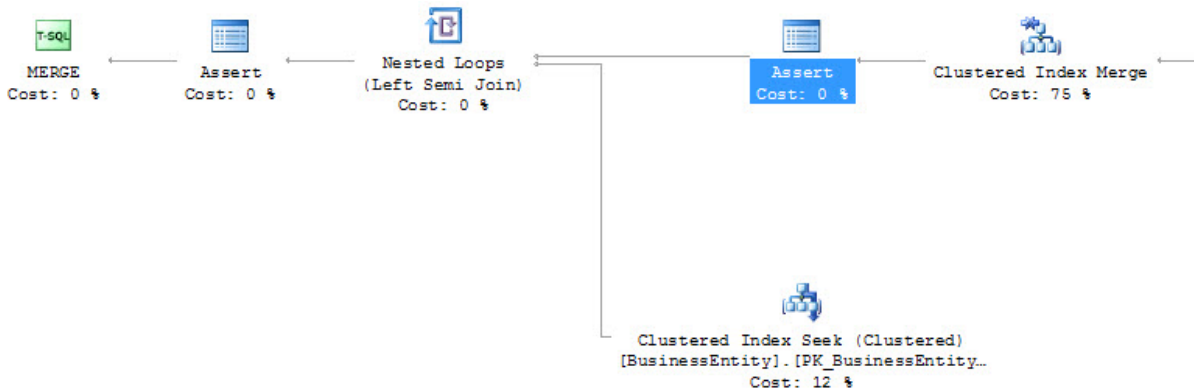


Figure 4.14

The **Clustered Index Merge** receives all of the information added to the data stream by the execution plan, uses it to determine if the action is an INSERT, an UPDATE, or a DELETE, and performs that action. Appropriately, in this case, because of all the determinations that the merge operation must perform, 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 validates that the `BusinessEntityId` referential integrity is intact, through the **Clustered Index Seek** against the `BusinessEntity` table. The information gathered by that join passes to another **Assert** operator, which validates the referential integrity, and the query is completed.

As you can see, a lot of action takes place within execution plans but, with careful review, it is possible to identify what is going on.

Prior to the MERGE statement, you may have done a query of this type dynamically. You either had different procedures for each of the processes, or different statements 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 4.9 and change one simple value like this...

```
...  
@AccountNumber NVARCHAR(15) = 'SPEEDCO0001',  
...
```

Listing 4.10

...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, and the **Merge** operator, creates a single reusable plan for all the data manipulation operations it supports.

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 actually 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.

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 4.11.

```
SELECT *  
FROM Sales.vIndividualCustomer  
WHERE BusinessEntityId = 8743;
```

Listing 4.11

Figure 4.15 shows the resulting graphical execution plan.

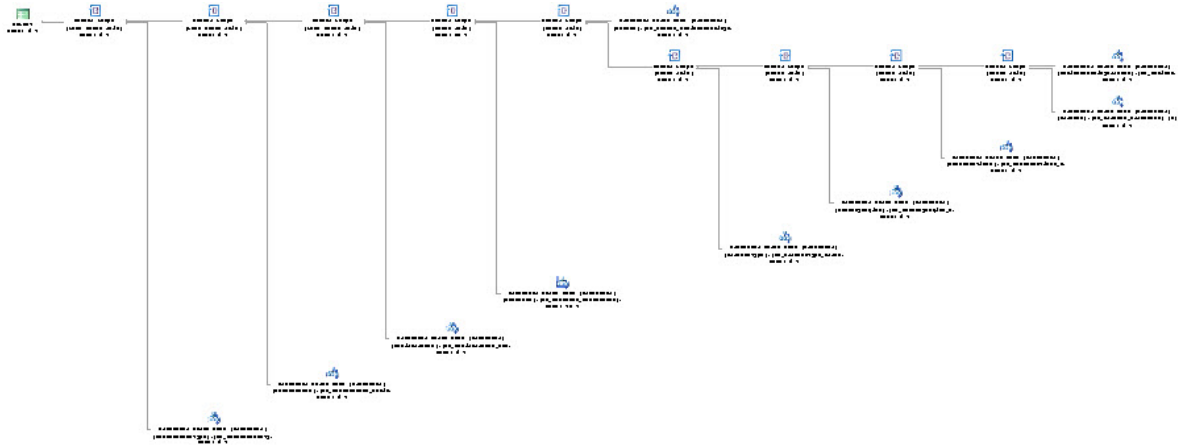


Figure 4.15

This is another plan that is very difficult to read on the printed page, so Figure 4.16 shows an exploded view of just the five operators on the right-hand side of the plan.

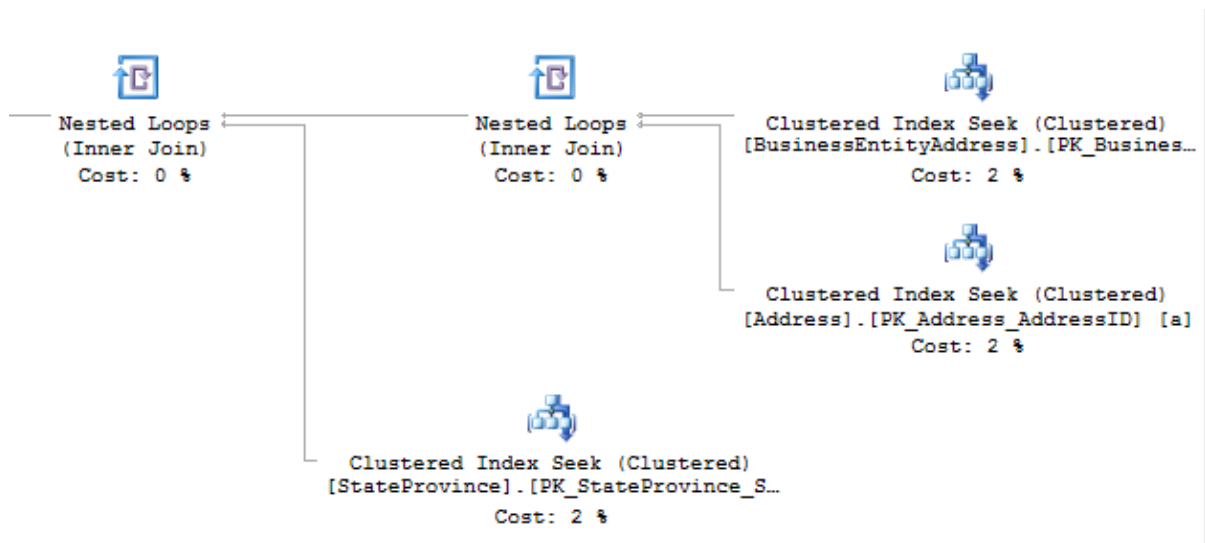


Figure 4.16

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 named construct that sits on top of the base tables from which they derive. The optimizer, during binding, resolves all those component parts in order to arrive at an execution plan to access the data. In effect, the query optimizer ignores the view object, and instead deals directly with 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 necessities 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.

Indexed views

An indexed view, also called a materialized 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 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.

Maintaining an index view is a different story. 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 have to decide if the overhead associated with maintaining 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.

For example, one of the indexed views supplied with AdventureWorks2008R2 is `vStateProvinceCountryRegion`. This combines the `StateProvince` table and the `CountryRegion` table into a single entity for querying, such as the query in Listing 4.12.

```
SELECT *  
FROM Person.vStateProvinceCountryRegion;
```

Listing 4.12

Figure 4.17 shows the execution plan for this query.

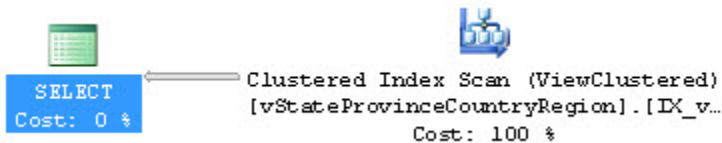


Figure 4.17

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 4.13 gives the exact same execution plan as the one shown in Figure 4.17, because the optimizer recognizes the index as the best way to access the data.

```
SELECT  sp.Name AS StateProvinceName,  
        cr.Name AS CountryRegionName  
FROM    Person.StateProvince sp  
        INNER JOIN Person.CountryRegion cr ON  
            sp.CountryRegionCode = cr.CountryRegionCode ;
```

Listing 4.13

However, as the execution plan grows in complexity, this behavior is neither automatic nor guaranteed. For example, consider the query in Listing 4.14.

```
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 4.14

If you expected to see a join between the indexed view and the Person.Address table, you would be disappointed.

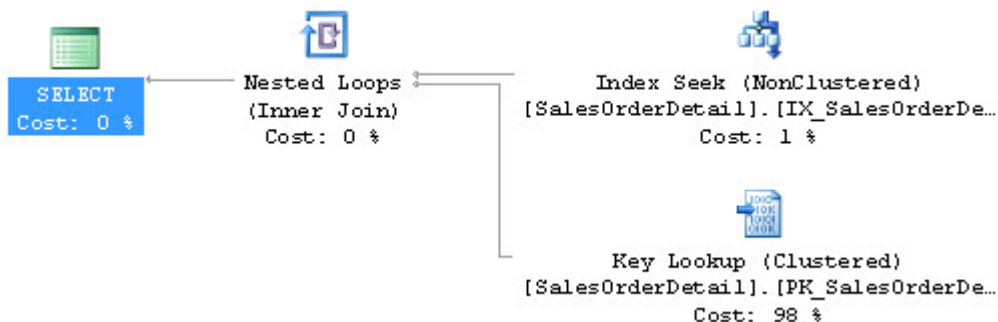


Figure 4.18

Instead of using the clustered index that defines the materialized view, as we saw in Figure 4.17, the optimizer 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.¹ SQL Server will expand views when the optimizer determines that direct table access will be less costly than using the indexed view.

Indexes

A big part of any tuning effort involves choosing the right indexes to include in a database. In most people's minds, the importance of using indexes is already well established. A frequently asked question however, is "How come some of my indexes are used and others are not?"

The availability of a useful index can directly affect the choices made by the query optimizer. The right index leads the optimizer to the selection of the right plan. However, a lack of indexes or, even worse, a poor choice of indexes created on the database, can directly lead to poor execution plans and poor query performance.

Included indexes: avoiding Bookmark Lookups

One of the more pernicious problems when tuning a query is the **Bookmark Lookup**. Type "avoid bookmark lookup" into Google and you'll get quite a few hits. As we discovered in Chapter 2, SQL Server 2005 and later no longer refers directly to **Bookmark Lookup** operators, although it does use the same term for the operation within its documentation.

¹ There is a way around this, as will be explained when we encounter the NOEXPAND hint, in the *Table Hints* section of Chapter 5.

To recap, a **Bookmark Lookup** occurs when a non-clustered index is used to retrieve the row or rows of interest, but the index is not covering (does not contain all the columns requested by the query). In this situation, the optimizer is forced to send the query to a clustered index, if one exists (a **Key Lookup**), otherwise, to a heap (a **RID Lookup**), in order to retrieve the data.

A **Bookmark Lookup** is not necessarily a bad thing, but the operation required to first read from the index, followed by an extra read to retrieve the data from the clustered index or heap, can lead to performance problems.

We can demonstrate this with the simple query in Listing 4.15.

```
SELECT sod.ProductID ,
       sod.OrderQty ,
       sod.UnitPrice
FROM   Sales.SalesOrderDetail sod
WHERE  sod.ProductID = 897;
```

Listing 4.15

This query returns the execution plan in Figure 4.19, which demonstrates the cost of lookups.

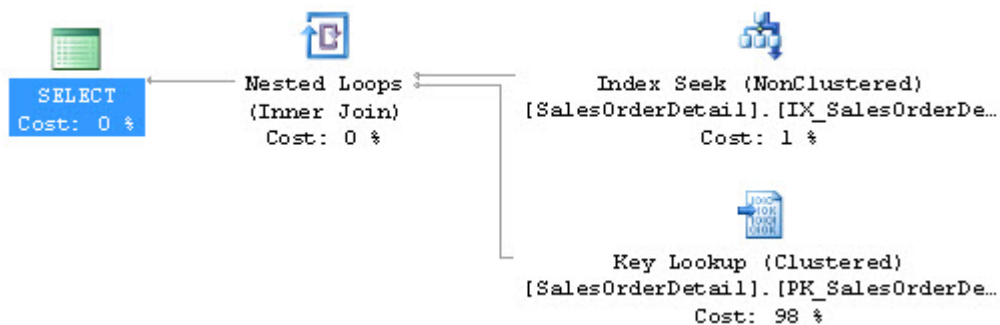


Figure 4.19

The **Index Seek** operator pulls back the four rows we need, quickly and efficiently. Unfortunately, the only data available on that index is the `ProductId` because the index in this case only stores the key value, `ProductId`, and the clustered key, so all other data is returned from another location, the clustered index.

Index Seek (NonClustered)	
Scan a particular range of rows from a nonclustered index.	
Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Number of Rows	2
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0002402
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0.0033652 (1%)
Estimated Subtree Cost	0.0033652
Estimated Number of Rows	75.6667
Estimated Row Size	19 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	2
Object	
[AdventureWorks2008R2].[Sales].[SalesOrderDetail]. [IX_SalesOrderDetail_ProductID] [sod]	
Output List	
[AdventureWorks2008R2].[Sales]. [SalesOrderDetail].SalesOrderID, [AdventureWorks2008R2].[Sales]. [SalesOrderDetail].SalesOrderDetailID, [AdventureWorks2008R2].[Sales]. [SalesOrderDetail].ProductID	
Seek Predicates	
Seek Keys[1]: Prefix: [AdventureWorks2008R2].[Sales]. [SalesOrderDetail].ProductID = Scalar Operator((897))	

Figure 4.20

As you can see from Figure 4.20, the **Index Seek** also outputs columns that define the clustered index, in this case `SalesOrderId` and `SalesOrderDetailId`. These values are used to keep the index synchronized with the clustered index and the data itself.

We then get the **Key Lookup**, whereby the optimizer retrieves the other columns required by the query, `OrderQty` and `UnitPrice`, from the clustered index. In SQL Server 2000, the only way around this would be to modify the existing index used by this plan, `IX_SalesOrderDetail_ProductID`, to use all three columns. However, in SQL Server 2005 and above, we have the additional option of using the `INCLUDE` attribute within a non-clustered index.

The `INCLUDE` attribute was added to non-clustered indexes in SQL Server 2005 specifically to solve this type of problem. `INCLUDE` allows you to add columns to the index at the leaf level for storage only. `INCLUDE` does not make these columns a part of the key of the index itself. This means that the columns added do not affect the sorting or lookup values of the index. Adding the columns needed by the query can turn the index into a covering index, eliminating the need for the **Lookup** operation. This does come at the cost of added disk space and additional overhead for the server to maintain the index. Due consideration must be paid prior to implementing this as a solution.

In Listing 4.16, we create a new index using the `INCLUDE` attribute. In order to get an execution plan in the middle of running all these statements together, we set `STATISTICS XML` to `ON`, and turn it `OFF` when we are done. The code that appears after we turn `STATISTICS XML` back `OFF` recreates the original index so that everything is in place for any further tests down the road.

```
IF EXISTS ( SELECT *
            FROM   sys.indexes
            WHERE  OBJECT_ID = OBJECT_ID(N'Sales.SalesOrderDetail')
                  AND name = N'IX_SalesOrderDetail_ProductID' )
    DROP INDEX IX_SalesOrderDetail_ProductID
    ON Sales.SalesOrderDetail
    WITH ( ONLINE = OFF );
CREATE NONCLUSTERED INDEX IX_SalesOrderDetail_ProductID
ON Sales.SalesOrderDetail
(ProductID ASC)
INCLUDE ( OrderQty, UnitPrice ) WITH ( PAD_INDEX = OFF,
STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY = OFF,
DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON )
ON [PRIMARY];
GO
```

```
SET STATISTICS XML ON;
GO

SELECT  sod.ProductID ,
        sod.OrderQty ,
        sod.UnitPrice
FROM    Sales.SalesOrderDetail sod
WHERE   sod.ProductID = 897;
GO
SET STATISTICS XML OFF;
GO

--Recreate original index
IF EXISTS ( SELECT *
            FROM   sys.indexes
            WHERE  OBJECT_ID = OBJECT_ID(N'Sales.SalesOrderDetail')
                AND name = N'IX_SalesOrderDetail_ProductID' )
    DROP INDEX IX_SalesOrderDetail_ProductID
        ON Sales.SalesOrderDetail
        WITH ( ONLINE = OFF );
CREATE NONCLUSTERED INDEX IX_SalesOrderDetail_ProductID
ON Sales.SalesOrderDetail
(ProductID ASC)
WITH ( PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
      SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY = OFF,
      DROP_EXISTING = OFF,
      ONLINE = OFF, ALLOW_ROW_LOCKS = ON,
      ALLOW_PAGE_LOCKS = ON ) ON [PRIMARY];
GO

EXEC sys.sp_addextendedproperty @name = N'MS_Description',
    @value = N'Nonclustered index.', @level0type = N'SHEMA',
    @level0name = N'Sales', @level1type = N'TABLE',
    @level1name = N'SalesOrderDetail', @level2type = N'INDEX',
    @level2name = N'IX_SalesOrderDetail_ProductID';
```

Listing 4.16

Run this code in Management Studio with the **Include Actual Execution Plan** option turned on, and you will see the plan shown in Figure 4.21.

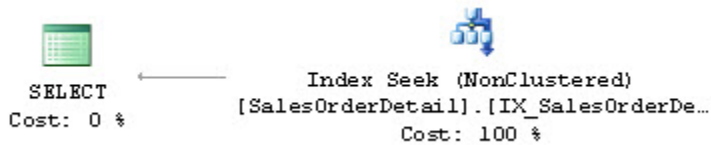


Figure 4.21

The index is now covering, so the execution plan is able to use a single operator to find and return all the data we need.

Index selectivity

Let's now move on to the questions of which indexes the optimizer will use, and why it sometimes avoids using available indexes.

First, let's briefly review the definition of the two kinds of indexes: clustered and non-clustered. A clustered index stores the data along with the key values of the index and it sorts the data, physically. A non-clustered index sorts the column, or columns, that define the key of the index. The non-clustered index also contains the key of the clustered index as a means of connecting the non-clustered index to the appropriate row in the clustered index.

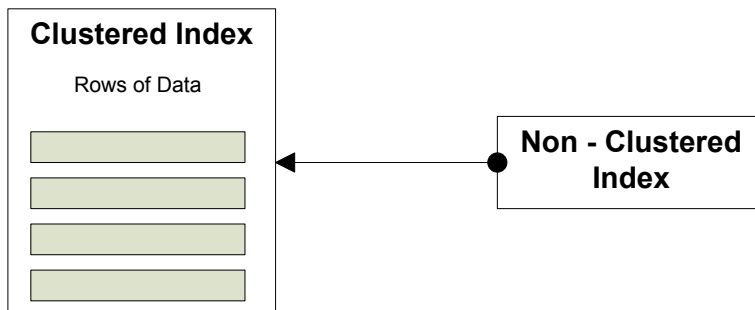


Figure 4.22

As described in Chapter 1, for each index the optimizer automatically generates statistics that describe the data distribution within the index, and therefore determine its potential utility for resolving a certain query. The key indicator of the usefulness of an index is its **selectivity**.

An index's selectivity describes the distribution of the distinct values within a given data set. To put it more simply, you count the number of rows and then you count the number of unique values for a given column across all the rows. After that, divide the unique values by the number of rows. This results in a ratio that is the selectivity of the index. The higher the selectivity, the more useful the index, and the more likely it will be used by the optimizer, because a high degree of selectivity means that the optimizer can count on the index being very accurate.

For example, on the `Sales.SalesOrderDetail` table there is an index, `IX_SalesOrderDetail_ProductID`, on the `ProductID` column. To see the statistics for that index, use the `DBCC SHOW_STATISTICS` command.

```
DBCC SHOW_STATISTICS('Sales.SalesOrderDetail',  
                      'IX_SalesOrderDetail_ProductID');
```

Listing 4.17

This returns three result sets with various amounts of data. For the purposes of selectivity, the second result set is the most interesting:

All density	Average Length	Columns
0.003759399	4	ProductID
8.242868E-06	8	ProductID, SalesOrderID
8.242868E-06	12	ProductID, SalesOrderID, SalesOrderDetailID

The density is inverse to the selectivity, meaning that the lower the density, the higher the selectivity. So an index like the one above, with a density of .003759399, a small number, therefore its inverse will be fairly high, indicating high selectivity, will very likely be used

by the optimizer. The other rows refer to the key columns from the clustered index, adding to the selectivity of this index. Non-clustered indexes have a pointer back to the clustered index, since that's where the data is stored. If no clustered index is present, then a pointer to the data itself, referred to as a **heap**, is generated. That's why the columns of the clustered index are included as part of the selectivity of the index.

Low selectivity can cause the optimizer to shun an index. Let's create a situation where you've taken the time to create an index on a frequently searched column, and yet you're not seeing a performance benefit. The business represented in AdventureWorks2008R2 has decided that they're going to be giving away prizes based on the quantity of items purchased. This means a query very similar to the one from Listing 4.18.

```
SELECT sod.OrderQty ,
       sod.SalesOrderID ,
       sod.SalesOrderDetailID ,
       sod.LineTotal
FROM   Sales.SalesOrderDetail sod
WHERE  sod.OrderQty = 10;
```

Listing 4.18

Figure 4.23 shows the execution plan for this query.



Figure 4.23

We see a **Clustered Index Scan** against the entire table, and then a simple **Filter** operation to derive the final results sets, where `OrderQty = 10`.

Let's now create an index that our query can use:

```
CREATE NONCLUSTERED INDEX IX_SalesOrderDetail_OrderQty
ON Sales.SalesOrderDetail ( OrderQty ASC )
WITH ( PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY = OFF,
DROP_EXISTING = OFF, ONLINE = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON )
ON [PRIMARY];
```

Listing 4.19

Unfortunately, if you capture the plan again, you'll see that it's identical to the one shown in Figure 4.23; in other words, the optimizer has completely ignored our new index. Since we know that selectivity, along with the number of pages, the type of index, and other factors, determines when, or if, the optimizer will use an index, let's examine the new index using `DBCC SHOW_STATISTICS`.

All density	Average Length	Columns
-----	-----	-----
0.02439024	2	OrderQty
2.18055E-05	6	OrderQty, SalesOrderID
8.242868E-06	10	OrderQty, SalesOrderID, SalesOrderDetailID

We can see that the density of the `OrderQty` column is 10 times higher than for the `ProductID` column, meaning that our `OrderQty` index is ten times less selective. To express this in more quantifiable terms, there are 121,317 rows in the `SalesOrderDetail` table on my system. There are only 41 distinct values for the `OrderQty` column. With so few distinct values out of all those rows, the chances of the optimizer finding an index on that column to be useful for most queries is exceedingly small. To make the index more selective and more useful, there would have to be more unique values, or the addition of more columns. This column just isn't, by itself, an adequate index to make a difference in the query plan.

If we really had to make this query run well, the answer would be to make the index selective enough to be useful to the optimizer. We could also try forcing the optimizer to

use the index by using a query hint, but in this case, it wouldn't help the performance of the query (I cover hints in detail in Chapter 5). Remember that adding an index, however selective, comes at a price during INSERTs, UPDATEs and DELETEs as the data within the index is reordered, added or removed.

If you're following along in AdventureWorks2008R2, you'll want to be sure to drop the index we created.

```
DROP INDEX Sales.SalesOrderDetail.IX_SalesOrderDetail_OrderQty;
```

Listing 4.20

Statistics and indexes

As noted previously, for each index, the optimizer generates statistics that describe what kind of data it stores and how that data is distributed. Without accurate and up-to-date statistics, the optimizer may ignore even well-constructed and highly selective indexes and so create suboptimal execution plans.

The following example is somewhat contrived, but it demonstrate how, as the data changes, the exact same query can result in two different execution plans. Listing 4.21 creates a new table, along with an index.

```
IF EXISTS ( SELECT *
            FROM sys.objects
            WHERE object_id = OBJECT_ID(N'[NewOrders]')
              AND type IN ( N'U' ) )
    DROP TABLE [NewOrders]
GO
SELECT *
INTO NewOrders
FROM Sales.SalesOrderDetail
GO
CREATE INDEX IX_NewOrders_ProductID ON NewOrders ( ProductID )
GO
```

Listing 4.21

In Listing 4.22, we capture an estimated plan for a simple query against `NewOrders`. Next, we start a transaction and execute an `UPDATE` statement where the intent is to change significantly the distribution of data in the `ProductID` column, making it much less selective. We then run the same simple query and capture the actual execution plan.

```
-- Estimated Plan
SET SHOWPLAN_XML ON
GO
SELECT  OrderQty ,
        CarrierTrackingNumber
FROM    NewOrders
WHERE   ProductID = 897
GO
SET SHOWPLAN_XML OFF
GO

BEGIN TRAN
UPDATE  NewOrders
SET     ProductID = 897
WHERE   ProductID BETWEEN 800 AND 900
GO

-- Actual Plan
SET STATISTICS XML ON
GO
SELECT  OrderQty ,
        CarrierTrackingNumber
FROM    NewOrders
WHERE   ProductID = 897

ROLLBACK TRAN
GO
SET STATISTICS XML OFF
GO
```

Listing 4.22

Use of `SET SHOWPLAN_XML` statements and batches allows us to capture only the execution plans for those specific batches (in this case, omitting the plan generated for the `UPDATE` statement). First, the estimated plan for the query before the data update.

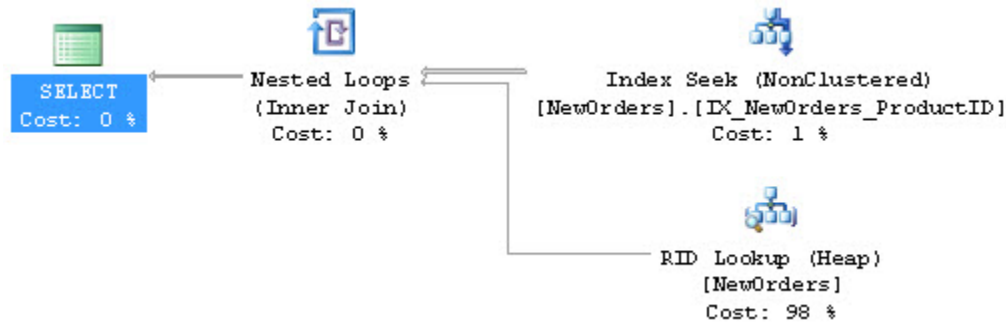


Figure 4.24

Then the actual execution plan, after the data update.

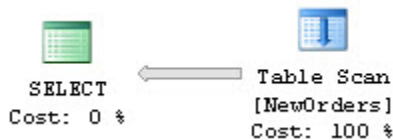


Figure 4.25

Go to the top and right of Figure 4.24 to find the **Index Seek** operator. Clearly, prior to the updates, the data and statistics within the index were selective enough that the query could use a **Seek** operation. Then, because the data requested is not included in the index itself, we see a **RID Lookup** operation against a heap table, using the row identifier to bring back the data from the correct row.

However, after the update, our index is much less selective and returns much more data, so the optimizer ignores it and instead retrieves the data by scanning the whole table, as we can see from the **Table Scan** operator in Figure 4.25.

What has happened is that the statistics that existed prior to the update of the data supported one execution plan. After the data update, the optimizer updated the statistics and, because of this, despite the same indexes and table structures being in place, a new execution plan was necessary. This shows the importance of statistics to the optimizer and to the plans that it generates.

Summary

This chapter demonstrated the sort of execution plans that we can expect to see when our code uses stored procedures, views, derived tables, and CTEs. 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.

It's difficult to understate the impact of indexes and their supporting statistics on the quality of the plans that the optimizer generates. Simply adding an index doesn't necessarily mean you've solved a performance problem. You need to ensure that the index is selective, and you have to make appropriate choices regarding the addition or inclusion of columns in your indexes, both clustered and non-clustered. You will also need to be sure that your statistics accurately reflect the data that is stored within the index.

Chapter 5: Controlling Execution Plans with Hints

It is possible, using various available hints, to impose your will on the optimizer and, to some degree, control its behavior. There are three categories of hints, which include:

- **Query hints** tell the optimizer to apply a hint throughout the execution of the entire query.
- **Join hints** tell the optimizer to use a particular join at a particular point in the query.
- **Table hints** control **Table Scans** and the use of a particular index for a table.

In this chapter, I'll describe how to use each type of hint, but I can't stress the following enough: hints are *dangerous*. Hints detract from the optimizer's ability to make choices. Appropriate use of the right hint on the right query can improve query performance. 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.

If you find yourself putting hints on a majority of your queries and stored procedures, then you're *doing something wrong*. As part of describing what the hints do, I'll lay out the problem that you're hoping to solve by applying that hint. Some of the examples will improve performance or change the behavior in a positive manner, and some will negatively affect performance.

Query Hints

There are many different query hints that perform many different tasks. Some are useful occasionally, while a few are for rare circumstances. Query hints take control of an entire query. The other types of hints, join and table hints, are for more granular control over a particular aspect of the plan, or for controlling plans for specific objects.

We specify query hints in the `OPTION` clause. Listing 5.1 shows the basic syntax.

```
SELECT ...  
OPTION (<hint>,<hint>...)
```

Listing 5.1

We can't apply query hints to `INSERT` statements, except as part of a `SELECT` operation, and we can't use query hints in sub-select statements.

Before we proceed, let me take this opportunity to warn you, once again, that injudicious use of these hints can cause you more problems than they solve!

HASH|ORDER GROUP

These two hints – `HASH GROUP` and `ORDER GROUP` – directly apply to a `GROUP BY` aggregate operation (as well as to `DISTINCT` or `COMPUTE` clauses). They instruct the optimizer to apply either hashing, or grouping to the aggregation, respectively.

In Listing 5.2, we have a simple `GROUP BY` query that returns the number of distinct values in the `Suffix` column of the `Person` table.

```
SELECT  p.Suffix ,  
        COUNT(p.Suffix) AS SuffixUsageCount  
FROM    Person.Person AS p  
GROUP BY p.Suffix;
```

Listing 5.2

As the DBA, you 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 5.2 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 5.1.

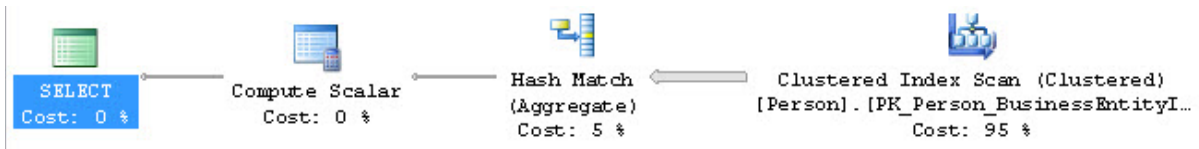


Figure 5.1

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, selecting distinct values from the data supplied by the **Clustered Index Scan**. Then the counts are derived, based on the matched values. The query ran in about 14ms and performed a single scan on the **Person** table, resulting in 3,821 reads. This plan has an estimated cost of 2.99377, which you can see in the **Select** operator's ToolTip in Figure 5.2.

SELECT	
Cached plan size	24 B
Degree of Parallelism	1
Memory Grant	1024
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	2.99377
Estimated Number of Rows	6
Statement	
SELECT p.Suffix, COUNT(p.Suffix) AS SuffixUsageCount FROM Person.Person AS p GROUP BY p.Suffix;	

Figure 5.2

You wonder if the best way to improve the performance of the query is to try to get the optimizer to use the data from the **Clustered Index Scan** in an ordered fashion. Although not the most expensive operation in the plan (that's the scan itself), the unordered **Hash Match** aggregation is a cost that could be saved. If you get the data out in order, you don't have to deal with the overhead of a hash table being built and populated in the tempdb.

To accomplish this, you add 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 5.3

Figure 5.3 shows the new plan.



Figure 5.3

We've told the optimizer to use ordering rather than hashing, via the `ORDER GROUP` hint so, instead of the hash table, it's been forced to use a **Sort** operator to feed rows into the **Stream Aggregate** operator, which works with ordered data to arrive at an aggregation.

As per my repeated warning, this query now runs in 20ms instead of the original 14ms, a 42% increase. You can even see it reflected in the estimated cost of the plan, which has now jumped to 4.18041, a 39% increase, which closely matches the actual increase in execution time. The source of the increased cost is ordering the data as it comes out of the **Clustered Index Scan**.

Depending on your situation, you may find an instance where, using our example above, the data is already ordered, yet the optimizer chooses to use the **Hash Match** operator instead of the **Stream Aggregate**. In that case, the optimizer would recognize that the data was ordered and accept the hint gracefully, increasing performance.

While query hints allow you to control the behavior of the optimizer, it doesn't mean your choices are necessarily better than the optimizer's choices. To optimize this query, you may want to consider adding a different index or modifying the clustered index. Also, remember that a hint applied today may work well but, over time, as data and statistics shift, the hint may no longer work as expected.

MERGE |HASH |CONCAT UNION

These 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.

The example query in Listing 5.4 is not running fast enough to satisfy the demands of the application.

```
SELECT [pm1].[Name] ,
       [pm1].[ModifiedDate]
FROM   [Production].[ProductModel] pm1
UNION
SELECT [pm2].[Name] ,
       [pm2].[ModifiedDate]
FROM   [Production].[ProductModel] pm2;
```

Listing 5.4

When a query has been identified as running slow, it's time to look at the execution plan, as seen in Figure 5.4.

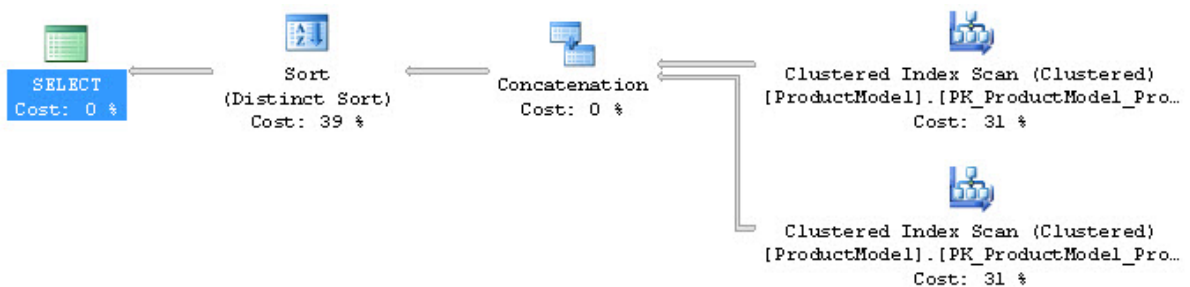


Figure 5.4

You can see that the **Concatenation** operation that the optimizer chooses to use is, in the context of the plan, very cheap. The **Sort** operation that follows it is relatively expensive. The overall estimated cost of the plan is 0.0377 and it took about 104ms to run with 2 scans and 28 reads.

In a test to see if changing the implementation of the **UNION** operation will affect overall performance, you apply the **MERGE UNION** hint.

```
SELECT pm1.Name ,  
       pm1.ModifiedDate  
FROM   Production.ProductModel pm1  
UNION  
SELECT pm2.Name ,  
       pm2.ModifiedDate  
FROM   Production.ProductModel pm2  
OPTION ( MERGE UNION );
```

Listing 5.5

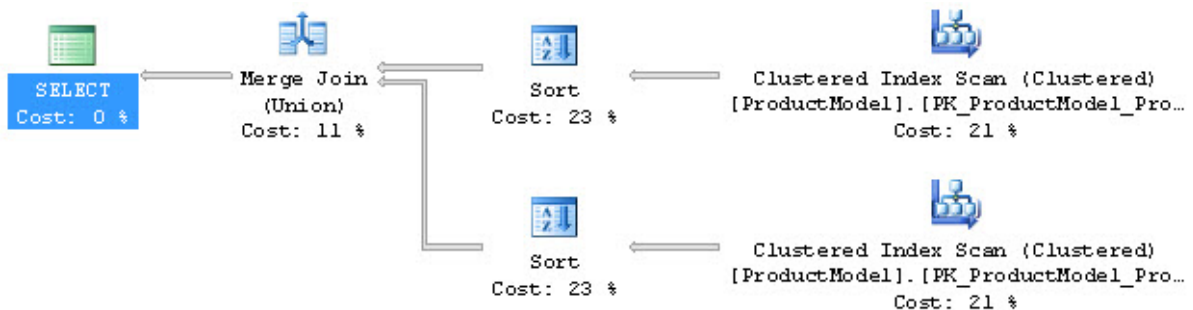


Figure 5.5

You have forced the **UNION** operation to use the **Merge Join** instead of the **Concatenation** operator. However, since the **Merge Join** only works with sorted data feeds, we've also forced the optimizer to add two **Sort** operators. The estimated cost for the query has gone from 0.0377 to 0.0548, and the execution time went up to 135ms from 100ms. Clearly, this didn't work.

What if you tried the HASH UNION hint?

```
SELECT pm1.Name ,  
       pm1.ModifiedDate  
FROM   Production.ProductModel pm1  
UNION  
SELECT pm2.Name ,  
       pm2.ModifiedDate  
FROM   Production.ProductModel pm2  
OPTION ( HASH UNION );
```

Listing 5.6

Figure 5.6 shows the new execution plan.

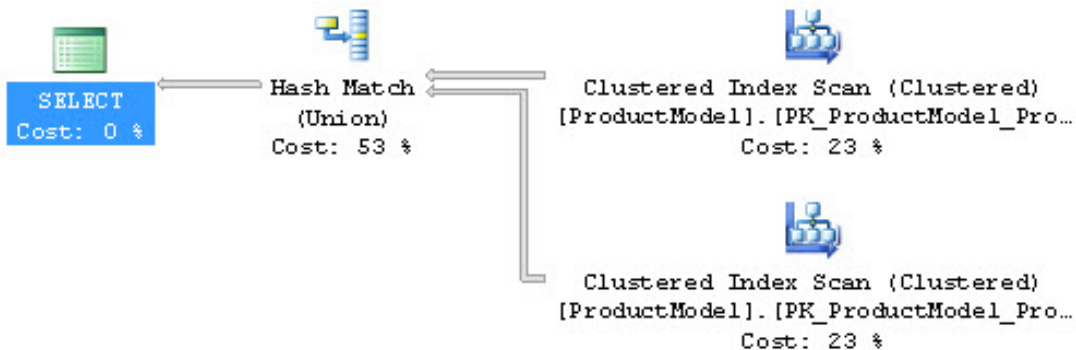


Figure 5.6

The execution plan is simpler, with the **Sort** operations eliminated. The estimated cost has gone up considerably, from 0.0377 to 0.0497, but the execution time has dropped from 100ms to 50ms. This is a positive development.

In this situation, the hint is working to modify the behavior of the query in a positive way. This example shows how it is possible to wring extra performance out of a query by applying hints.

LOOP|MERGE|HASH JOIN

These hint methods make all the join operations in a particular query use the method supplied by the hint. However, note that, if we also apply a join hint (covered later) to a specific join, then the more granular join hint takes precedence over the general query hint.

In this situation, we've found that our system is suffering from poor disk I/O, so we need to reduce the number of scans and reads that our queries generate. By collecting data from Profiler and Performance Monitor, we identify the query in Listing 5.7 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 5.7

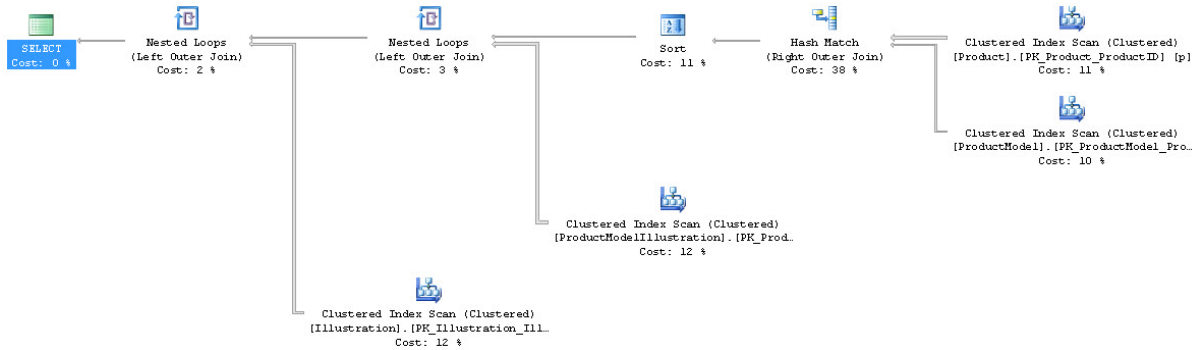


Figure 5.7

As you can see, the query uses a mix of **Nested Loops** and **Hash Match** operators to put the data together, and it runs in around 214ms. Now, let's view the I/O output of the query. This can be done by navigating from the main menu, **Query | Query Options**, selecting the **Advanced** tab and selecting the **Set Statistics I/O** check box.

```
Table 'Illustration'. Scan count 1, logical reads 273...
Table 'ProductModelIllustration'. Scan count 1, logical reads 183...
Table 'Worktable'. Scan count 0, logical reads 0...
Table 'ProductModel'. Scan count 1, logical reads 14...
Table 'Product'. Scan count 1, logical reads 15...
```

Listing 5.8

Most of the reads come from the **Illustration** and **ProductModelIllustration** tables. It occurs to you that allowing the query to perform all those **Hash Match** join operations may be slowing it down because of the reads and writes in tempdb, so you decide to change the behavior by adding the **LOOP JOIN** hint to the end of the query.

```
OPTION ( LOOP JOIN );
```

Listing 5.9

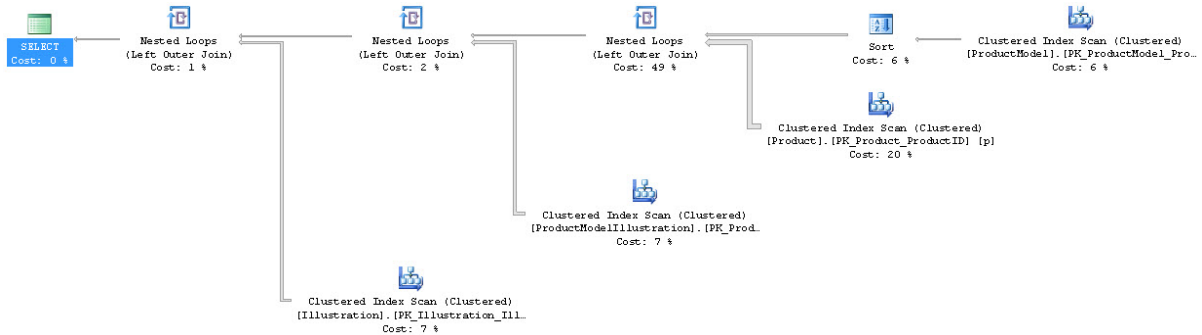


Figure 5.8

Now the **Hash Match** join is a **Nested Loops** join. This situation could be interesting. If you look at the operations that underpin the query execution plan, you'll see that the second query, with the hint, eliminates the creation of a worktable. In addition, the estimated cost of the plan is slightly higher, and the execution time went up just a bit. You can tell why if you look at the scans and reads.

```
Table 'Illustration'. Scan count 1, logical reads 273...
Table 'ProductModelIllustration'. Scan count 1, logical reads 183...
Table 'Product'. Scan count 1, logical reads 555...
Table 'ProductModel'. Scan count 1, logical reads 14...
```

Listing 5.10

Not only have we been unsuccessful in reducing the reads, despite the elimination of the worktable, but we've actually increased the number of reads on the Product table. What if we modify the query to use the MERGE JOIN hint, instead?

```
OPTION ( MERGE JOIN );
```

Listing 5.11

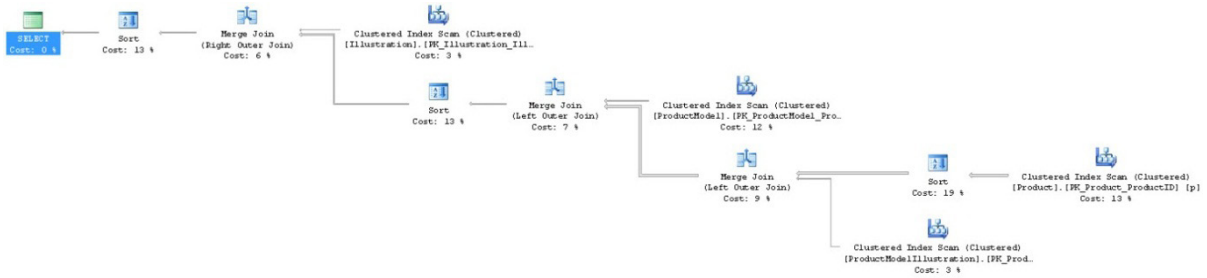


Figure 5.9

This query executes a little faster than the original but did we solve our read problem?

```
Table 'Worktable'. Scan count 2, logical reads 17...
Table 'ProductModelIllustration'. Scan count 1, logical reads 2...
Table 'Product'. Scan count 1, logical reads 15...
Table 'ProductModel'. Scan count 1, logical reads 14...
Table 'Illustration'. Scan count 1, logical reads 3...
```

Listing 5.12

We've re-introduced a worktable, but it does appear that we've eliminated the large number of reads. We may have a solution. However, before we conclude the experiment, we may as well as try out the `HASH JOIN` hint, to see what it might do.

```
OPTION ( HASH JOIN );
```

Listing 5.13

We're back to a simplified execution plan using a **Hash Match** join operation, as compared to the **Merge Join**. The execution time is about half what the **Merge Join** was, and a third of the original query and the I/O looks as shown in Listing 5.14.

Chapter 5: Controlling Execution Plans with Hints

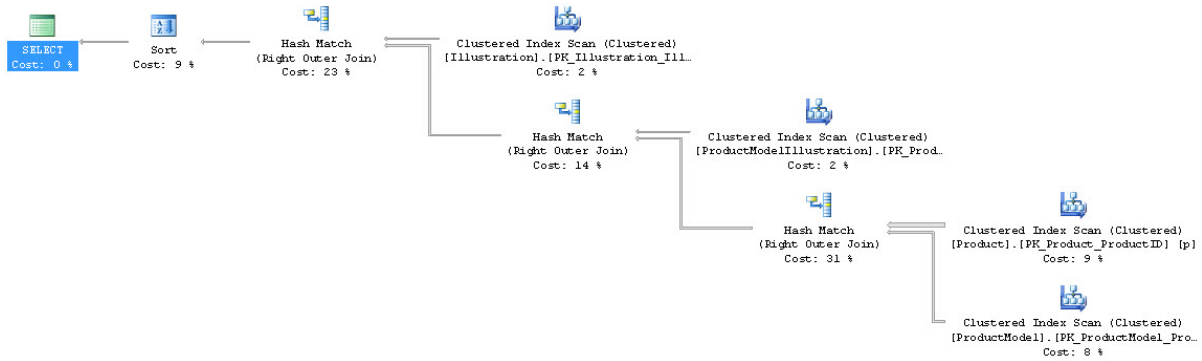


Figure 5.10

```
Table 'Worktable'. Scan count 0, logical reads 0...
Table 'ProductModel'. Scan count 1, logical reads 14...
Table 'Product'. Scan count 1, logical reads 15...
Table 'ProductModelIllustration'. Scan count 1...
Table 'Illustration'. Scan count 1, logical reads 3...
```

Listing 5.14

For the example above, using the **HASH JOIN** hint appears to be the best bet for reducing the I/O costs of the query. There will be the added overhead of the creation of the worktable in **tempdb** but the benefit of increased speed and reduced reads seems to offset the worktable cost.

Most of the cost savings in this case seem to come from moving the **Sort** operation to the end of the data movement as opposed to earlier when the query was using **Loop Joins**. Why didn't the optimizer pick this plan in the first place? It's not always possible to know. It could be that optimizer deemed it less expensive to place the **Sort** operator earlier in the plan. Remember, the optimizer tries to get a good enough plan, not a perfect plan. This is another situation where, with careful testing, you can achieve a performance improvement using query hints.

FAST n

Let's assume that we are not concerned about the performance of the database. This time, we're concerned 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 have to wait 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 getting the execution plan to return the first "n" rows as fast as possible, where "n" is a positive integer value. Consider the following query and execution plan.

```
SELECT *
FROM Sales.SalesOrderDetail sod
JOIN Sales.SalesOrderHeader soh
ON sod.SalesOrderID = soh.SalesOrderID;
```

Listing 5.15

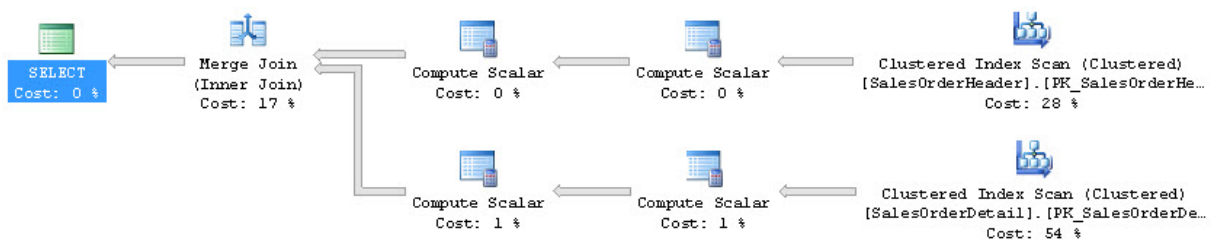


Figure 5.11

This query performs adequately considering the fact that it's selecting all the data from the tables without any sort of filtering operation, but there is a delay before the end-users see any results, so we can try to fix this by adding the FAST N hint to return the first 10 rows as quickly as possible.

```
OPTION ( FAST 10 );
```

Listing 5.16

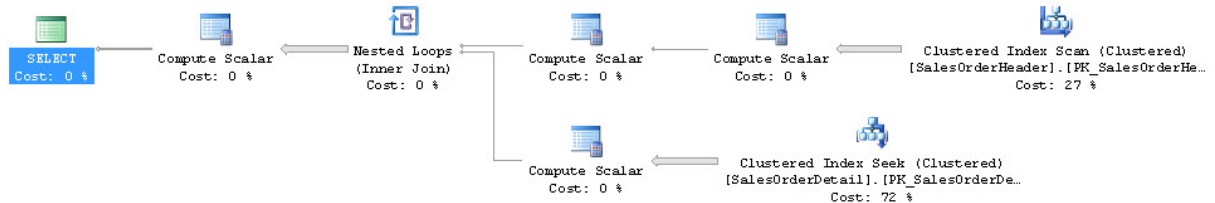


Figure 5.12

Instead of the **Merge Join** operator for the join, the optimizer attempts to use a **Nested Loops** operator. The loop join returns the 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 just on getting the first ten rows back as soon as possible.

The total estimated cost for the original query was 1.961. The hint reduced that cost to 0.012 (for the first 10 rows). The number of logical reads increases dramatically, from 1,238 for the un-hinted query to 101,827 for the hinted query. The actual speed of the execution of the query increases only marginally, from around 4.6 seconds to 4.5 seconds. This slight improvement is only going to get the first few rows back to the screen quickly. The overall query won't run as fast once there is a load on the system and this query has to share disk access with other queries, because of all those additional reads.

FORCE ORDER

Once again, our monitoring tools have identified a query that is performing poorly. It's a long query with a number of tables being joined, as shown in Listing 5.17, which could be somewhat concerning, 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 fairly certain that your join order is better than that supplied by the optimizer, or if you're experiencing timeouts.

```
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
```

Chapter 5: Controlling Execution Plans with Hints

```
LEFT JOIN Production.ProductDescription AS pdr
      ON pmpdc.ProductDescriptionID = pdr.ProductDescriptionID
LEFT JOIN Production.Culture AS c
      ON c.CultureID = pmpdc.CultureID;
```

Listing 5.17

Based on your knowledge of the data, you're confident that you've put the joins in the correct order. Figure 5.13 shows the current execution plan.

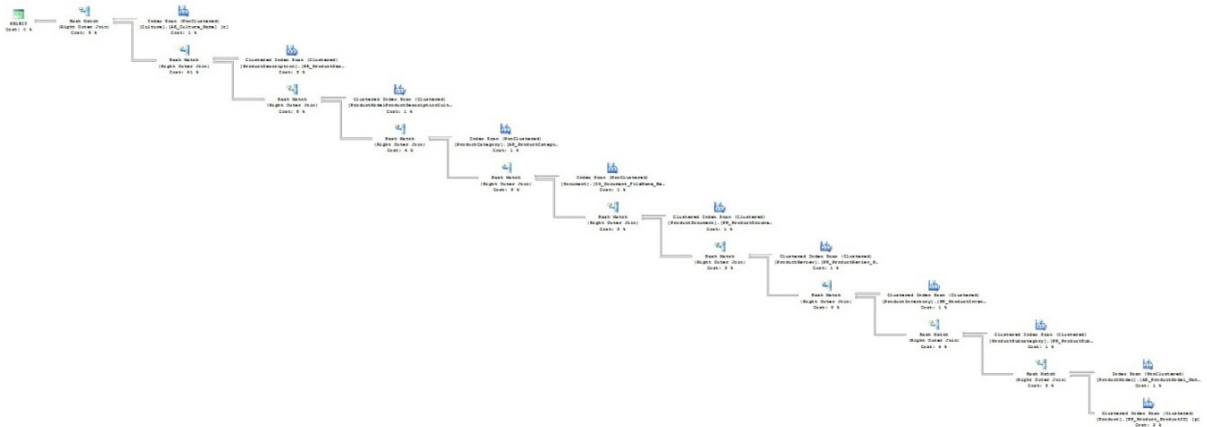


Figure 5.13

Again, this plan is far too large to review on the page, but it gives you a good idea of its overall structure. Figure 5.14 shows an exploded view of just a few of the tables and the order in which they are being joined.

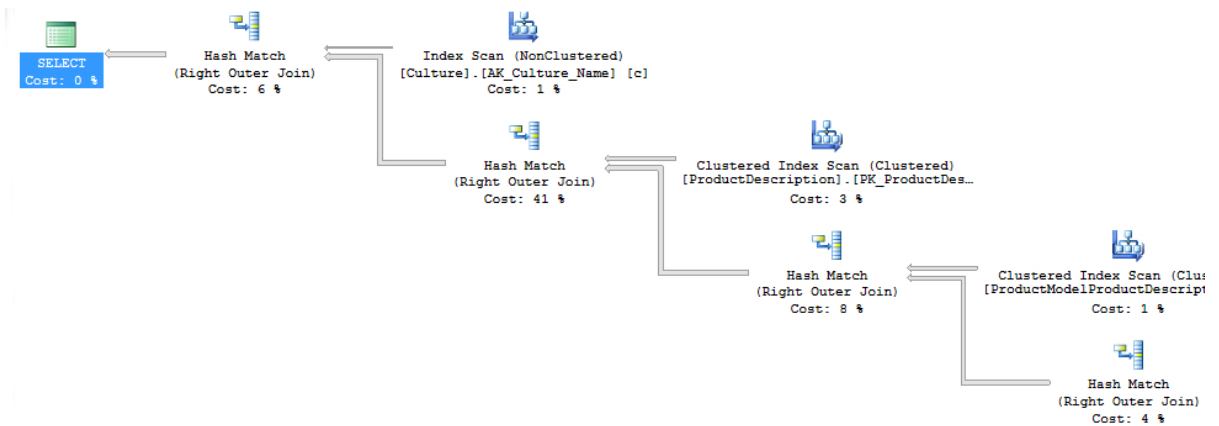


Figure 5.14

The physical join order, based on the order of the joins in the query in Listing 5.17, would be Product, followed by ProductModel and then ProductSubcategory and so on. However, the optimizer actually starts right at the other end, with Culture, then ProductDescription, then ProductModelProductDescriptionCulture and so on.

Take the same query and apply the `FORCE ORDER` query hint.

```
OPTION (FORCE ORDER);
```

Listing 5.18

It results in the plan shown in Figure 5.15.

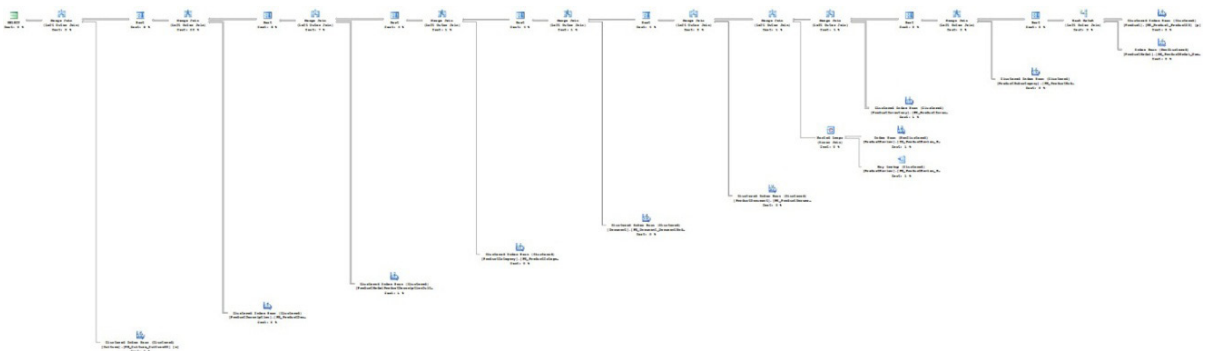


Figure 5.15

You can tell, just by comparing the shapes of the plan in Figure 5.13 to the one in Figure 5.15 that a substantial change has occurred. The optimizer is now accessing the tables in the order specified by the query. Again, we'll zoom in on the first operators so that you can see how the physical order has changed.

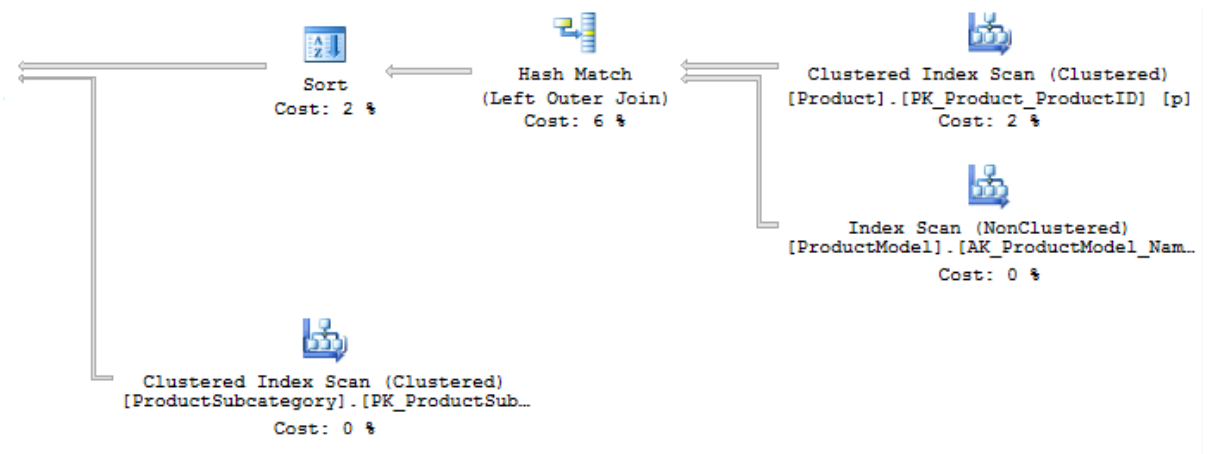


Figure 5.16

Now the order is from the Product table, followed by the ProductModel and then ProductSubcategory and so on from there.

However, the interesting thing here is that the execution time went from 848ms in the first query to 588ms in the second, or about a 31% improvement. It is possible to get direct control over the optimizer in order to achieve positive results.

Why would I be able to get a better execution plan than the optimizer and thus improve performance? If you look at the **Select** operator to get the basic information about the plan, you'll find that the first plan showed that the reason the optimizer stopped attempting to find a good plan was because it timed out before it found the least-cost plan. By forcing the order, I was able to get a better plan than the optimizer could during the time it allotted itself.

MAXDOP

In this example, we have one of those very nasty problems, a query that sometimes runs just fine, but sometimes runs incredibly slowly. We have investigated the issue, using SQL Server Profiler to capture the execution plan of this procedure, over time, with various parameters. We finally arrive at two execution plans. Figure 5.17 shows the execution plan that runs quickly.

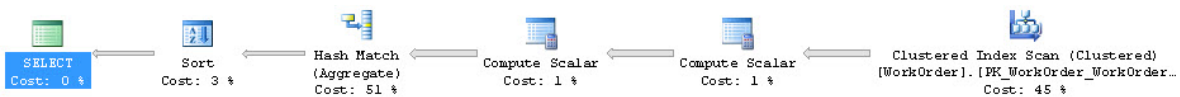


Figure 5.17

Figure 5.18 shows the slow execution plan (note that I split this image in order to make it more readable).

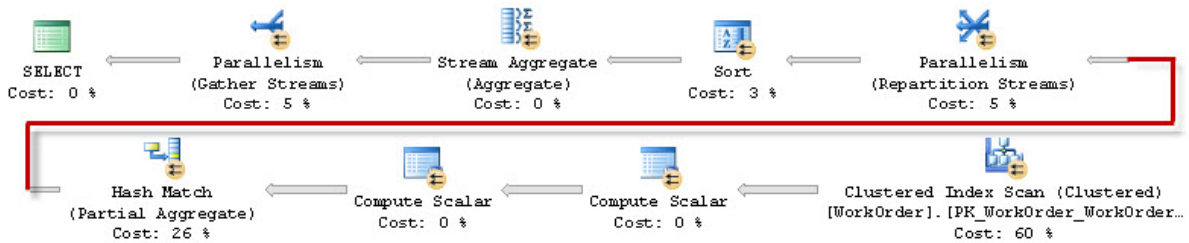


Figure 5.18

What we have here is an example of where parallelism (covered in more detail in Chapter 8), which should be helping the performance of your system, is instead hurting performance. We're probably seeing a situation where the estimated cost of executing the plan serially exceeds the 'cost threshold for parallelism' `sp_configure` option and therefore the optimizer decided to introduce parallelism, whereby the work required to execute the query is split across multiple CPUs.

Parallelism is normally turned on and off at the server level, but let's assume that there are a number of other queries running on the server that are benefiting from parallelism, so you can'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. However, having done the work of setting up the server, you still have the occasional outlier, and it's for instances like these that the `MAXDOP` hint becomes useful.

The `MAXDOP` query hint controls the use of parallelism within an individual query, rather than working using the server-wide setting of max degree of parallelism.

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, in order to force this query to be run in parallel.

```
sp_configure 'cost threshold for parallelism', 1;
GO

RECONFIGURE WITH OVERRIDE;
GO

SELECT  wo.DueDate ,
        MIN(wo.OrderQty) MinOrderQty ,
        MIN(wo.StockedQty) MinStockedQty ,
        MIN(wo.ScrapedQty) MinScrappedQty ,
        MAX(wo.OrderQty) MaxOrderQty ,
        MAX(wo.StockedQty) MaxStockedQty ,
        MAX(wo.ScrapedQty) MaxScrappedQty
FROM      Production.WorkOrder wo
GROUP BY wo.DueDate
ORDER BY wo.DueDate;
GO

sp_configure 'cost threshold for parallelism', 50;
GO

RECONFIGURE WITH OVERRIDE;
GO
```

Listing 5.19

This will result in an execution plan that takes full advantage of parallel processing, as shown in Figure 5.18. You can examine the properties of the **Clustered Index Scan** operator, by selecting that icon on the plan in Management Studio. The property, **Actual Number of Rows** can be expanded by clicking on the plus (+) icon. Depending on your system, you'll see multiple threads, the number of threads spawned by the parallel operation. On my machine, it was eight separate threads.

However, we know that when our example query uses parallel processing, it runs slowly. We have no desire to change the overall behavior of parallelism within the server itself, so we directly affect the query that is causing problems by modifying the query to include the MAXDOP hint.

```
OPTION ( MAXDOP 1 );
```

Listing 5.20

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 5.19 and then rerun the code.

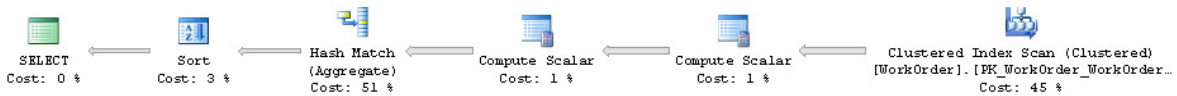


Figure 5.19

Comparing Figure 5.19 to Figure 5.18, we can see that limiting parallelism didn't fundamentally change the operations on the execution plan, since it's still using a **Clustered Index Scan** to get the initial data set. The plan still puts the data set through two **Compute Scalar** operators to deal with the `StockedQty` column, a calculated column. We still see the same **Hash Match** join operator between the table and itself as part of aggregating the data, and then, finally, a **Sort** operator puts the data into the correct order before the **Select** operator adds the column aliases back in. The only real changes are the removal of the operators necessary for parallel execution. The reason, in this instance, is that the performance that was worse on the production machine was due to the extra steps required to take the data from a single stream to a set of parallel streams, and then bring it all back together again. While the optimizer may determine this should work better, it's not always correct.

OPTIMIZE FOR

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 result, most of the time, in an execution plan that performs very well. Sometimes,

though, a certain value, or subset of values, supplied to the parameters results in an execution plan that performs extremely poorly. This is the **parameter sniffing** problem.

Parameter sniffing is a process that occurs with all stored procedures; it is normally at the very least benign, and often very beneficial to the performance of the system. As values are passed to a stored procedure or parameterized query, the optimizer uses those values to evaluate how well an index will satisfy the query requirements, based on the available statistics. In most cases, this produces more accurate execution plans.

However, situations can arise whereby the data distribution of a particular table or index is such that most parameters will result in a good plan, but some parameters can result in a bad one. Problems occur when the first execution of the parameterized query or stored procedure happens to use a very non-representative parameter value. The resulting plan will be stored in cache and reused, and often turns out to be highly inefficient for the parameter values that are more representative. Alternatively, a "good" plan may be aged out of the cache, or recompiled due to changes in the statistics or the code, and replaced with a "bad" plan. As such, it becomes, to a degree, a gamble as to where and when the problematic execution plan is the one that is created and cached.

In SQL Server 2000, only two options are available:

1. Recompile the plan every time, using the `RECOMPILE` hint.
2. Get a good plan and keep it, using the `KEEPFIXED PLAN` hint.

Both of these solutions (covered later in this chapter) could create as many problems as they solve since, depending on its complexity and size, the `RECOMPILE` of the query could be longer than the execution itself. The `KEEPFIXED PLAN` hint could be applied to the problematic values as well as the useful ones.

In SQL Server 2005 and above, when you're hitting a bad parameter sniffing situation, you can use the `OPTIMIZE FOR` hint. This hint allows you to instruct the optimizer to optimize the query for the value that you supply, rather than the value passed to the

parameter of the query. OPTIMIZE FOR supplies you with one more tool in the arsenal for dealing with bad parameter sniffing.

We can demonstrate the utility of this hint with a very simple set of queries.

```
SELECT *
FROM Person.Address
WHERE City = 'Mentor'

SELECT *
FROM Person.Address
WHERE City = 'London'
```

Listing 5.21

We'll run these at the same time, and we get two different execution plans.

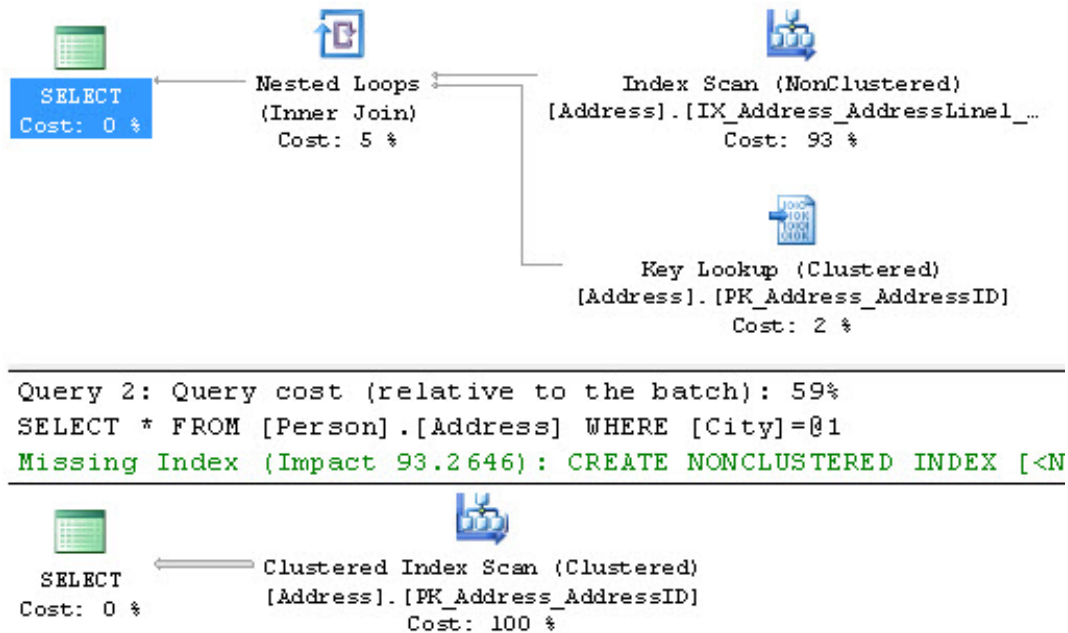


Figure 5.20

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'` is able to take advantage of the selectivity of the non-clustered index, for this value, to perform a **Seek** operation. 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, which you can see in the second execution plan in Figure 5.20.

Let's see what happens if we parameterize our T-SQL, as shown in Listing 5.22.

```
DECLARE @City NVARCHAR(30)

SET @City = 'Mentor'
SELECT *
FROM Person.Address
WHERE City = @City

SET @City = 'London'
SELECT *
FROM Person.Address
WHERE City = @City;
```

Listing 5.22

Now, we get a standard execution plan for both queries.

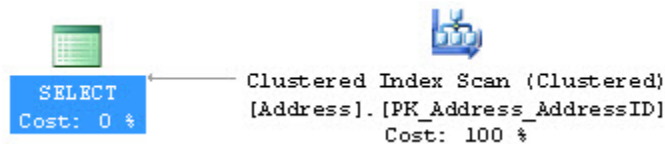


Figure 5.21

It's using the clustered index for both queries now because the optimizer is not sure which of the values available in the table is most likely going to be passed in as @City. That means it samples the values in the statistics and chooses a generic plan based on an average value, from the sampled data.

Let's make one more modification. In the second query, we instruct the optimizer to optimize for 'Mentor'.

```
DECLARE @City NVARCHAR(30)

SET @City = 'London'
SELECT *
FROM Person.Address
WHERE City = @City

SET @City = 'London'
SELECT *
FROM Person.Address
WHERE City = @City
OPTION ( OPTIMIZE FOR ( @City = 'Mentor' ) );
```

Listing 5.23

The value 'London' has a very low level of selectivity within the index (i.e. there are a lot of values equal to 'London'), and this is displayed by the **Clustered Index Scan** in the first query. Despite the fact that the second query looks up the same value, it's now the faster of the two queries for values other than those of 'London'. The OPTIMIZE FOR query hint was able to trick the optimizer into creating a plan that assumed that the data was highly selective, even though it was not. The execution plan created was one for the more selective value 'Mentor' and helps that execution, but hurts that of 'London'.

Use of this 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. You could supply the hint for a value that doesn't represent the most frequently referenced data set, which would mean you've hurt performance instead of helping.

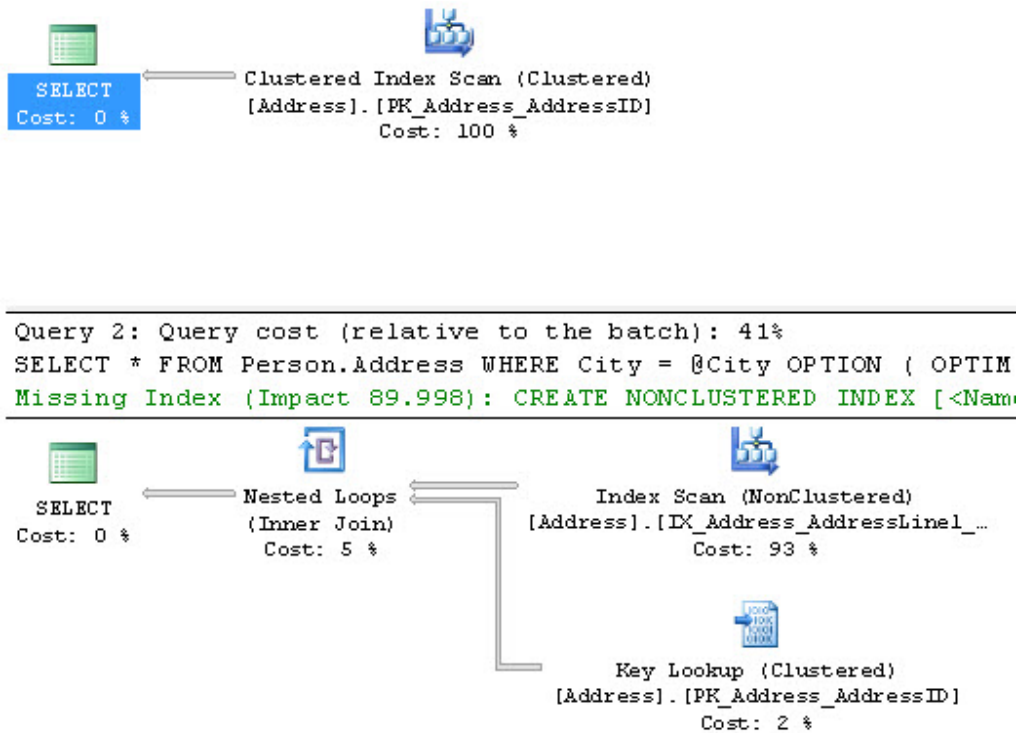


Figure 5.22

In the example above, there was only a single parameter, so there was only a single hint needed. If you have multiple parameters, you can set as many hints as you use parameters within the query.

SQL Server 2008 introduced a new addition to the OPTIMIZE FOR query hint, OPTIMIZE FOR UNKNOWN, where UNKNOWN takes the place of a specified value and makes the optimizer use sampled values from the statistics. In this situation, instead of trying to decide which specific value generates the best plan, you decided that optimizing for the most common value, as determined by the available statistics, is preferable. This situation frequently arises when there is either no discernible "best" plan or when the values passed to the query are extremely inconsistent, so you can only use a generic plan. In such cases, OPTIMIZE FOR UNKNOWN is a good option to control a bad parameter sniffing situation.

PARAMETERIZATION SIMPLE|FORCED

The `PARAMETERIZATION` query hint, forced and simple, can be applied only within a plan guide, so full explanation of this hint is deferred to the section on Plan Guides, in Chapter 8.

RECOMPILE

You have yet another problem query, with a stored procedure, which performs slowly in an intermittent fashion. Investigation and experimentation lead you to the realization that this is an ad hoc query (one that uses T-SQL statements or other code to generate SQL statements) that the optimizer could execute in completely different ways, because each time the query is passed to SQL Server, it has slightly different parameter values, and possibly even a different structure.

While plans are being cached for the query, many of these plans cannot be reused and could simply bloat the cache to the point where you observe memory pressure. These plans could even be problematic due to issues like parameter sniffing; the execution plan that works well for one set of parameter values may work horribly for another set.

The `RECOMPILE` hint was introduced in SQL 2005. It instructs the optimizer to mark the plan created so that it is not stored in cache at all. This hint might be useful in situations such as that just described, where there is a lot of ad hoc T-SQL in the query, or the data is relatively volatile, meaning that the plan created isn't likely to be useful to any of the following calls.

Regardless of the cause, we've determined that the cost of recompiling the procedure each time it executes is worth the query execution time saved by that recompile. In other words, because problematic execution plans are being cached that cause the query performance to degrade to the point where it's worth taking the "hit" of recompiling the query each time it's called.

Alternatively, this determination might be based on the occurrence of memory pressure in the procedure cache, or because you are seeing bad parameter sniffing issues.

You can also add the instruction to recompile the plan to the code that creates the stored procedure, but the `RECOMPILE` query hint offers greater control, in that we can recompile individual statements within a stored procedure or query, rather than just the whole thing. This means that, for example, we can recompile just the single statement within the query or procedure that uses ad hoc T-SQL, as opposed to the entire procedure. When a statement recompiles within a procedure, all local variables are initialized and the parameters used for the plan are those supplied to the procedure. If you use local variables in your queries, the optimizer makes a guess as to what value may work best for the plan, and stores this guess in the cache. Consider the pair of queries in Listing 5.24.

```
DECLARE @PersonId INT = 277;
SELECT soh.SalesOrderNumber ,
       soh.OrderDate ,
       soh.SubTotal ,
       soh.TotalDue
FROM Sales.SalesOrderHeader soh
WHERE soh.SalesPersonID = @PersonId;

SET @PersonId = 288;
SELECT soh.SalesOrderNumber ,
       soh.OrderDate ,
       soh.SubTotal ,
       soh.TotalDue
FROM Sales.SalesOrderHeader soh
WHERE soh.SalesPersonID = @PersonId;
```

Listing 5.24

These queries result in an identical pair of execution plans. The `PersonId` column is not very selective for the value of 277, so we get a **Clustered Index Scan** to retrieve the initial data. The two **Compute Scalar** operations are for the calculated columns, `Subtotal` and `TotalDue`, and then the data is finally compared to the value of 277 in the **Filter** operator, which reduces the number of rows returned. Both plans work this way, as shown in Figure 5.23.

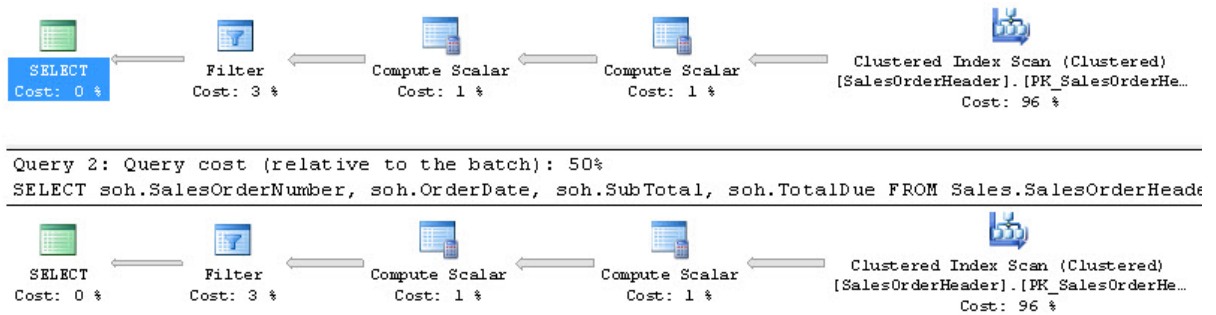


Figure 5.23

With a full knowledge of your system that you have from examining the indexes and statistics on the tables, you know that the plan for the second query should be completely different because the value passed (288) is much more selective and a useful index exists on that column. So, you modify the queries using the `RECOMPILE` hint. In this instance (Listing 5.25) I'm adding it to both queries so that you can see that the performance gain in the second query is due to the `RECOMPILE` and the subsequent improved execution plan, while the same `RECOMPILE` on the first query leads to the original plan.

```
DECLARE @PersonId INT = 277;
SELECT  soh.SalesOrderNumber ,
        soh.OrderDate ,
        soh.SubTotal ,
        soh.TotalDue
FROM    Sales.SalesOrderHeader soh
WHERE   soh.SalesPersonID = @PersonId
OPTION  ( RECOMPILE );

SET @PersonId = 288;
SELECT  soh.SalesOrderNumber ,
        soh.OrderDate ,
        soh.SubTotal ,
        soh.TotalDue
FROM    Sales.SalesOrderHeader soh
WHERE   soh.SalesPersonID = @PersonId
OPTION  ( RECOMPILE );
```

Listing 5.25

This results in the mismatched set of query plans in Figure 5.24.

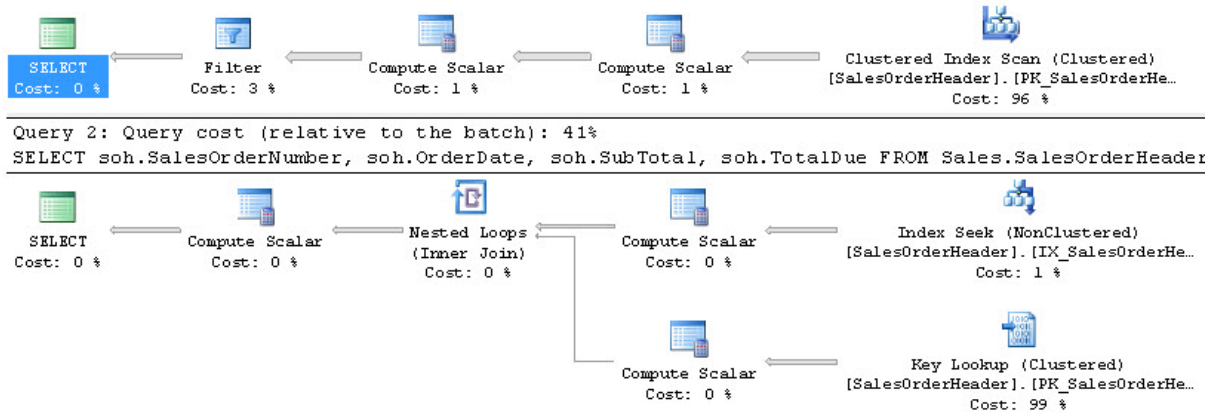


Figure 5.24

Note that the second query is now using our `IX_SalesOrderHeader_SalesPersonID` index, which resulted in a slightly longer execution time, but only 409 reads compared to the 686 of the original query.

ROBUST PLAN

This query hint is used when you need to work with very wide rows. For example:

- a row that contains one or more variable length columns set to very large size or even the `MAX` size introduced in 2005
- a row that contains one or more large objects (LOB) such as `BINARY`, `XML` or `TEXT` data types.

Sometimes, when processing these rows, it's possible for some operators to encounter errors, usually when creating worktables as part of the plan. The `ROBUST PLAN` hint ensures that a plan that could cause errors won't be chosen by the optimizer.

While this will eliminate errors, it will almost certainly result in longer query times since the optimizer won't be able to choose the optimal plan over the "robust" plan.

You should only ever use this hint if you have a set of wide rows that cause this rare error condition.

KEEP PLAN

As the data in a table changes, gets inserted, or deleted, the statistics describing the data also change. Once the volume of statistics changed passes a certain threshold, any queries referencing those statistics are marked for recompile in the plan cache, on the assumption that the optimizer might, based on the new statistics, be able to generate a more efficient plan.

Setting the `KEEP PLAN` hint doesn't prevent recompiles, but it does cause the optimizer to use less stringent rules when determining the need for a recompile. This means that, with more volatile data, you can keep recompiles to a minimum. The hint causes the optimizer to treat temporary tables within the plan in the same way as permanent tables, reducing the number of recompiles caused by the temporary table. This reduces the time and cost of recompiling a plan, which can be quite large, depending on the query.

However, problems may arise because the old plans may not be as efficient as newer plans could be. This happens because of the changing data; a plan that was previously adequate for a given data set is no longer appropriate because of the changes to that data set.

KEEPFIXED PLAN

The `KEEPFIXED PLAN` query hint is similar to `KEEP PLAN` but, instead of simply limiting the number of recompiles, `KEEPFIXED PLAN` eliminates any recompile due to changes in statistics.

Use this hint with extreme caution. The whole point of letting SQL Server maintain statistics is to aid the performance of your queries. If you prevent the optimizer from using the new statistics, it can lead to severe performance issues.

As with `KEEP PLAN`, the plan will remain in the cache unless the schema of the tables referenced in the query changes, or `sp_recompile` is run against the query, forcing a recompile.

EXPAND VIEWS

The `EXPAND VIEWS` query hint eliminates the use of the index 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. This behavior can be overridden on a view-by-view basis by adding the `WITH (NOEXPAND)` clause to any indexed views within the query.

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. However, if the data in the indexed view is not up to date, this hint can address that issue, usually at the cost of performance. Test this hint to ensure its use doesn't negatively affect performance.

Using one of the indexed views supplied with `AdventureWorks2008R2`, we can run the simple query in Listing 5.26.

```
SELECT *
FROM Person.vStateProvinceCountryRegion;
```

Listing 5.26

Figure 5.25 shows the resulting execution plan.

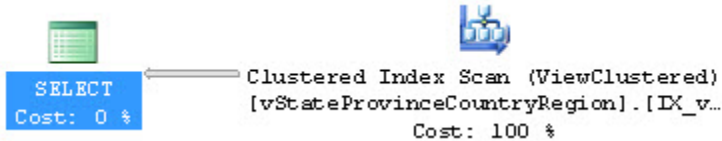


Figure 5.25

An indexed view is effectively a clustered index, so this execution plan makes perfect sense since the data needed to satisfy the query is available in the materialized view. Things change, as we see in Figure 5.26, if we add the query hint, `OPTION (EXPAND VIEWS)`.

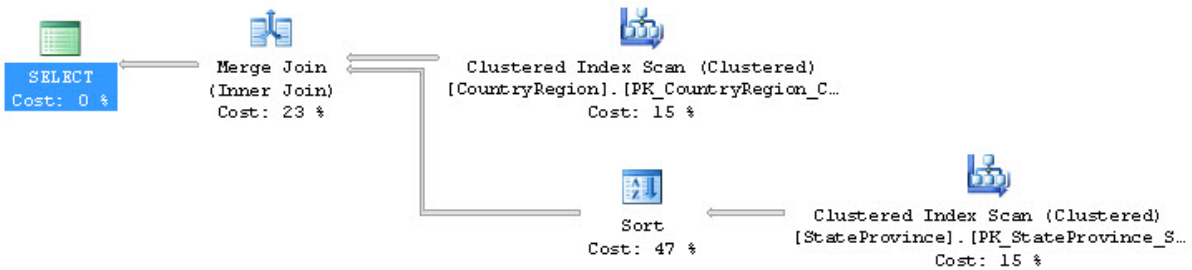


Figure 5.26

Now we're no longer scanning the indexed view. Within the optimizer, the view has been expanded into its definition so 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 214ms, but the second ran in about 342ms, so we're talking a substantial decrease in performance to use the hint in this situation.

MAXRECURSION

The addition of the Common Table Expression (see Chapter 4) to SQL Server offered a very simple method for calling recursive queries. The **MAXRECURSION** hint places an upper limit on the number of recursions within a query.

Valid values are between 0 and 32,767. Setting the value to zero allows for infinite recursion. The default number of recursions is 100. When the number is reached, an error is returned and the recursive loop is exited. This will cause any open transactions to roll back. Using this option doesn't change the execution plan. However, if you do get an error, then an actual execution plan will not be returned.

USE PLAN

This hint simply substitutes any plan the optimizer may have created with the XML plan supplied with the hint. We cover this in detail in Chapter 8.

Join Hints

A **join hint** provides a means to force SQL Server to use one of the three join methods that we've encountered previously, in a given part of a query. To recap, these join methods are:

- **Nested Loops** join: compares each row from one table ("outer table") to each row in another table ("inner table") and returns rows that satisfy the join predicate. Cost is proportional to the product of the rows in the two tables. Very efficient for smaller data sets.

- **Merge Join:** compares two **sorted** inputs, one row at a time. Cost is proportional to the sum of the total number of rows. Requires an equi-join condition. Efficient for larger data sets.
- **Hash Match join:** reads rows from one input, hashes the rows, based on the equi-join condition, into an in-memory hash table. Does the same for the second input and then returns matching rows. Most useful for very large data sets (especially data warehouses).

By including one of the join hints in your T-SQL, you will potentially override the optimizer's choice of the most efficient join method. In general, this is not a good idea, and if you're not careful you could seriously impede performance.¹

Application of the join hint applies to any query (SELECT, INSERT, or DELETE) where joins can be applied. Join hints are specified between two tables.

LOOP

Consider a simple query that lists Product Models, Products and Illustrations from AdventureWorks2008R2.

```
SELECT pm.Name ,
       pm.CatalogDescription ,
       p.Name AS ProductName ,
       i.Diagram
FROM   Production.ProductModel pm
       LEFT JOIN Production.Product p
           ON pm.ProductModelID = p.ProductModelID
       LEFT JOIN Production.ProductModelIllustration pmi
           ON pm.ProductModelID = pmi.ProductModelID
```

¹ There is a fourth join method, the **Remote** join, that is used when dealing with data from a remote server. It 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.

```

LEFT JOIN Production.Illustration i
      ON pmi.IllustrationID = i.IllustrationID
WHERE pm.Name LIKE '%Mountain%'
ORDER BY pm.Name;

```

Listing 5.27

We'll get the execution plan shown in Figure 5.27.

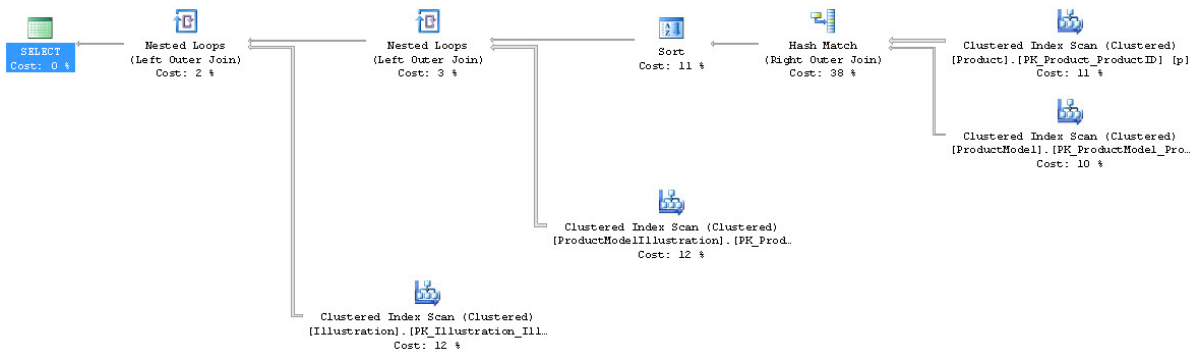


Figure 5.27

This is a straightforward plan. WHERE pm.name LIKE '%Mountain%', the query predicate, is non-SARGable, meaning that it can't be used by the optimizer in an **Index Seek**, and so the **Clustered Index Scan** operators on the Product and ProductModel table make sense. They are joined using a **Hash Match** operator, accounting for 46% of the cost of the query. Once the data is joined, the **Sort** operator implements the ORDER BY command. The plan continues with the **Clustered Index Scan** against the ProductModelIllustration table that joins to the data stream with a **Nested Loops** operator. This pattern repeats, with another **Clustered Index Scan** against the Illustration table and a join to the data stream with a **Nested Loops** operator. The total estimated cost for these operations comes to 0.1121 and the query runs in about 199ms.

What happens if we decide that we're smarter than the optimizer and that it really should be using a **Nested Loops** join instead of that **Hash Match** join? We can force the issue by adding the **LOOP** hint to the join condition between **Product** and **ProductModel**.

```
SELECT  pm.Name ,
        pm.CatalogDescription ,
        p.Name AS ProductName ,
        i.Diagram
FROM    Production.ProductModel pm
        LEFT LOOP JOIN Production.Product p
            ON pm.ProductModelID = p.ProductModelID
        LEFT JOIN Production.ProductModelIllustration pmi
            ON pm.ProductModelID = pmi.ProductModelID
        LEFT JOIN Production.Illustration i
            ON pmi.IllustrationID = i.IllustrationID
WHERE   pm.Name LIKE '%Mountain%'
ORDER BY pm.Name;
```

Listing 5.28

If we execute this new query, we'll see the plan shown in Figure 5.28.

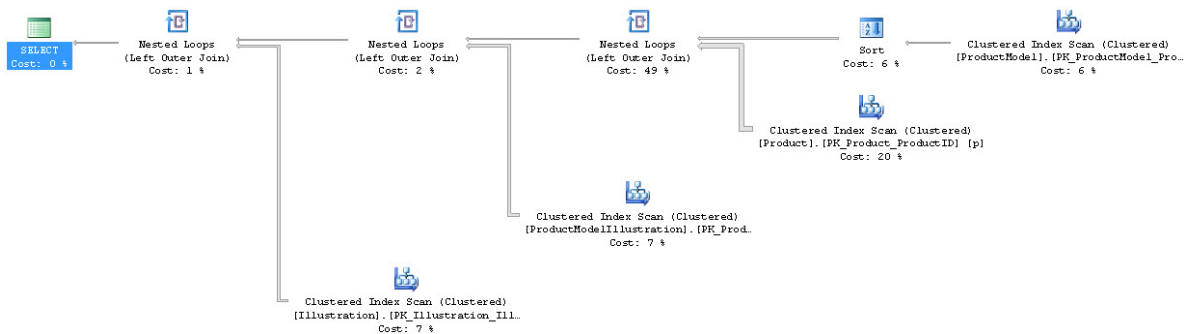


Figure 5.28

Sure enough, where previously we saw a **Hash Match** operator, we now see the **Nested Loops** operator. Also, the **Sort** moved before the join in order to feed ordered data into the **Nested Loops** operation, which means that the original data is sorted instead of the

joined data. This adds to the overall cost. Also, note that the **Nested Loops** join accounts for 56% of the cost, whereas the original **Hash Match** accounted for only 46%. All this resulted in a total, higher cost of 0.1834. The execution time was only slightly higher at 209ms, but overall, you can see how choosing the wrong join can negatively affect performance.

MERGE

If you replace the previous **LOOP** hint with the **MERGE** hint, you'll see the plan shown in Figure 5.29.

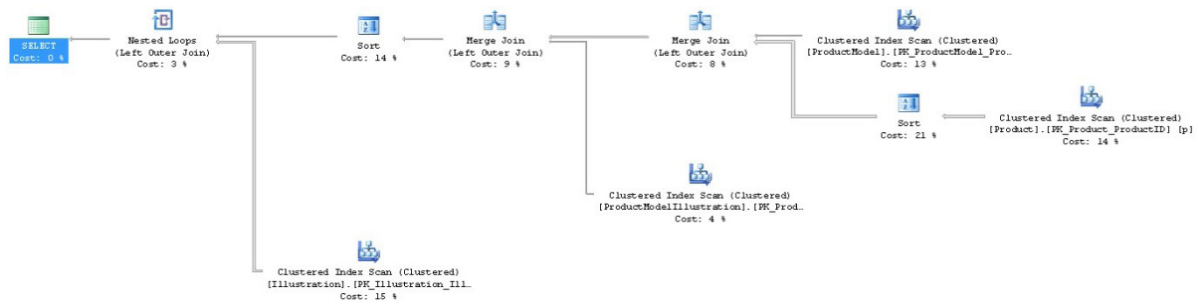


Figure 5.29

The **Nested Loops** becomes a **Merge Join** operator and the overall estimated cost of the plan drops to 0.0888, apparently offering us a performance benefit. In fact, the execution time dropped to 176ms.

The **Merge Join** plus the **Sort** operator, which is required to make sure it uses ordered data, turns out to cost less than the **Hash Match** or the **Nested Loop**.

In order to understand the origin of the performance increase, we can change the query options so that it shows us the I/O costs of each query. The output of all three queries is listed in part, here.

Original (Hash)

```
Table 'Illustration'. Scan count 1, logical reads 273
Table 'ProductModelIllustration'. Scan count 1, logical reads 183
Table 'Worktable'. Scan count 0, logical reads 0
Table 'ProductModel'. Scan count 1, logical reads 14
Table 'Product'. Scan count 1, logical reads 15
```

Loop

```
Table 'Illustration'. Scan count 1, logical reads 273
Table 'ProductModelIllustration'. Scan count 1, logical reads 183
Table 'Product'. Scan count 1, logical reads 555
Table 'ProductModel'. Scan count 1, logical reads 14
```

Merge

```
Table 'Illustration'. Scan count 1, logical reads 273
Table 'Worktable'. Scan count 0, logical reads 0
Table 'ProductModelIllustration'. Scan count 1, logical reads 2
Table 'Product'. Scan count 1, logical reads 15
Table 'ProductModel'. Scan count 1, logical reads 14
```

This shows us that the **Hash Match** and **Nested Loops** joins required almost exactly the same number of reads as the **Merge Join** to arrive at the required data set. The differences arise when we see that, in order to support the **Nested Loops** join, 555 reads were required instead of 15 for both the **Merge Join** and **Hash Match**. The key difference is the number of reads of the `ProductModelIllustration` table in the **Merge Join**; it's only 2, as opposed to the 183 in the other two queries.

This illustrates the point that the optimizer does not always choose an optimal plan. It's always best to look at the properties of the **Select** operator to see if full optimization occurred and if there was a timeout. Neither was the case here; it was a full optimization and the optimizer thought it found the best possible plan in the time allotted. Based on the statistics in the index and the amount of time it had to calculate its results, it must have decided that the **Hash Match** would perform faster.

Although we've positively influenced query performance for the time being, the downside is that, as the data changes over time within the tables, it's possible that the **Merge Join** will cease to function better. However, because we've hard coded the join, no new plan will be generated by the optimizer as the data changes, as would normally be the case.

Table Hints

Table hints enable you to control how the optimizer "uses" a particular 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 to 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!

Some of the table hints are primarily concerned with locking strategies. Since some of these don't affect execution plans, we won't be covering them. The three table hints covered below have a direct impact on the execution plans. For a full list of table hints, please refer to the Books Online supplied with SQL Server 2005 and later.

Table hint syntax

The correct syntax in SQL Server 2005 and above is to use the **WITH** keyword and list the hints within a set of parentheses, as shown in Listing 5.29.

```
FROM TableName WITH (hint, hint,...)
```

Listing 5.29

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 multiple indexed views are referenced within a query, the use of the **NOEXPAND** table hint will override the **EXPAND VIEWS** query hint that we saw 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. This allows for a more granular control over which of the indexed views is forced to resolve to its base tables, and which simply pull their data from their base clustered index.

SQL 2005 and 2008 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 can force the optimizer to use the index from the indexed view. In Chapter 4, we used a query that referenced one of the Indexed Views, `vStateProvinceCountryRegion`, in `AdventureWorks` (Listing 4.14). The optimizer expanded the view and we saw an execution plan that featured a three-table join. Via use of the **NOEXPAND** table hint, in Listing 5.30, 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 5.30

Now, instead of a 3-table join, we get the execution plan in Figure 5.30.

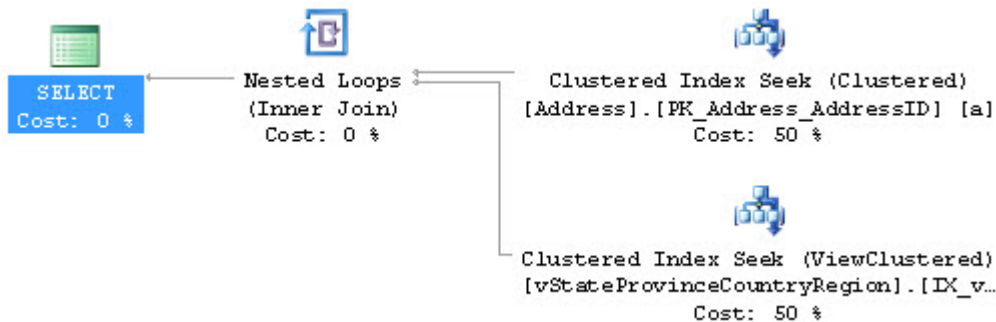


Figure 5.30

Now, not only are we using the clustered index defined on the view, but we're also seeing a performance increase, with the execution time going from 105ms to 81ms. 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. Firstly, numbering the index, starting at 0, which represents a clustered index, if any, and proceeding one at a time through the rest of the indexes on the table (Listing 5.31).

```
FROM TableName WITH (INDEX(0))
```

Listing 5.31

Or, secondly, simply referring to the index by name, which I recommend, because the order in which indexes are applied to a table can change (although the clustered index will always be 0). For an example, see Listing 5.32.

```
FROM TableName WITH (INDEX ([IndexName]))
```

Listing 5.32

You can only have a single `INDEX()` hint for a given table, but you can define multiple indexes within that one hint. 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 de  
       JOIN HumanResources.EmployeeDepartmentHistory edh  
         ON de.DepartmentID = edh.DepartmentID  
       JOIN HumanResources.Employee e  
         ON edh.BusinessEntityID = e.BusinessEntityID  
       JOIN Person.Person p  
         ON e.BusinessEntityID = p.BusinessEntityID  
WHERE  de.Name LIKE 'P%'
```

Listing 5.33

We get a reasonably straightforward execution plan, as shown in Figure 5.31.

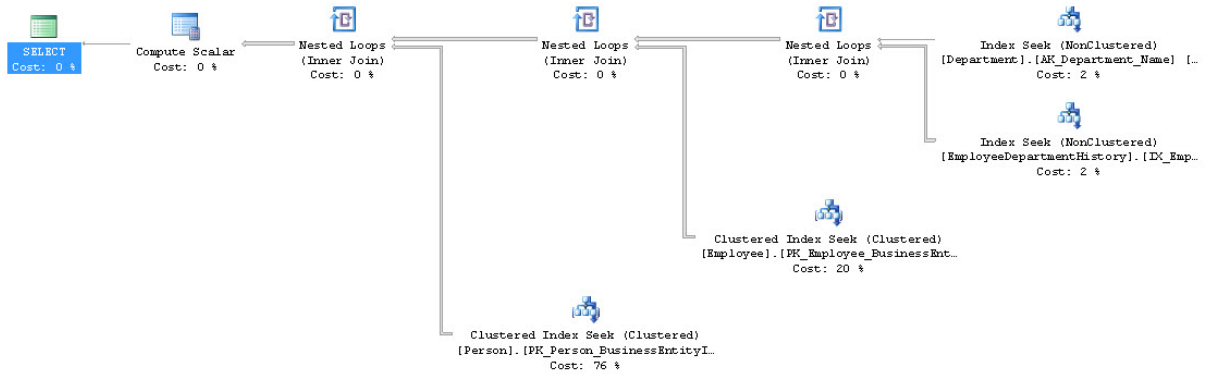


Figure 5.31

We see a series of **Index Seek** and **Clustered Index Seek** operations, joined together by **Nested Loop** operations. 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 5.34.

```
SELECT de.Name,
       e.JobTitle,
       p.LastName + ', ' + p.FirstName
FROM   HumanResources.Department de WITH (INDEX (PK_Department_DepartmentID))
JOIN   HumanResources.EmployeeDepartmentHistory edh
      ON de.DepartmentID = edh.DepartmentID
JOIN   HumanResources.Employee e
      ON edh.BusinessEntityID = e.BusinessEntityID
JOIN   Person.Person p
      ON e.BusinessEntityID = p.BusinessEntityID
WHERE  de.Name LIKE 'P%';
```

Listing 5.34

Figure 5.32 shows the resulting execution plan.

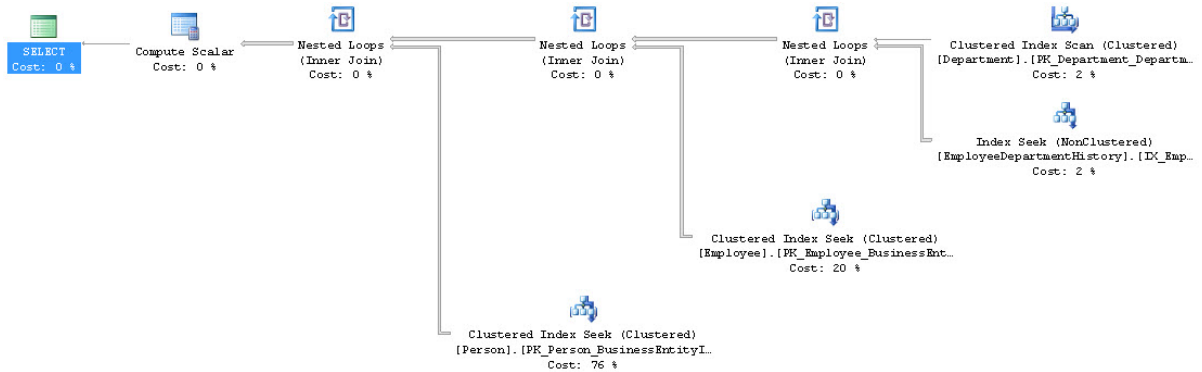


Figure 5.32

After the hint is added, we can see a **Clustered Index Scan** replacing the **Index Seek**. This change results in a more expensive query, with the execution time coming in at 252ms as opposed to 154ms. While the **Index Seek** is certainly faster than the scan, the difference at this time is small because the scan is only hitting a few more rows than the seek, in such a small table.

However, because the data was not as selective in the clustered index, the query resulted in a scan of this index rather than a seek, and so the performance of the query suffered.

FASTFIRSTROW

Just like the **FAST n** query hint outlined earlier, **FASTFIRSTROW** forces the optimizer to choose a plan that will return the first row as fast as possible for the table in question. Functionally, **FASTFIRSTROW** is equivalent to the **FAST n** query hint, but it is more granular. Be aware, though, that Microsoft is deprecating this hint in the next version of SQL Server, so I suggest you avoid using it.

The intent of the query in Listing 5.35 is to get a summation of the available inventory by product model name and product name.

```

SELECT  pm.Name AS ProductModelName,
        p.Name AS ProductName,
        SUM(pin.Quantity)
FROM    Production.ProductModel pm
        JOIN Production.Product p
        ON pm.ProductModelID = p.ProductModelID
        JOIN Production.ProductInventory pin
        ON p.ProductID = pin.ProductID
GROUP BY pm.Name,
        p.Name ;

```

Listing 5.35

It results in the execution plan in Figure 5.33.

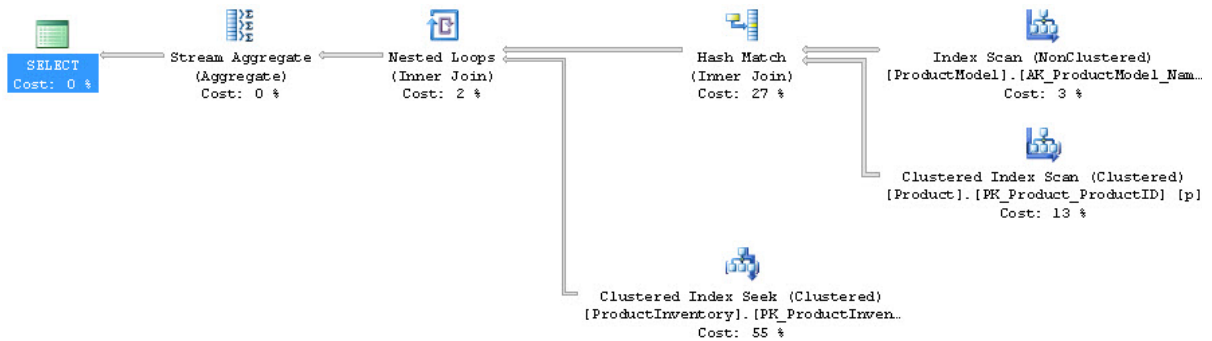


Figure 5.33

As you can see, an **Index Scan** operation against the ProductModel database returns the first stream of data. This is joined against a **Clustered Index Scan** operation from the Product table, through a **Hash Match** join operator. The data from the ProductInventory table can be retrieved through a **Clustered Index Seek** and this is then joined to the other data through a **Nested Loops** join. Finally, the optimizer builds the summation information through a **Stream Aggregate** operator.

If we decided that we thought that getting the Product information a bit quicker might make a difference in the behavior of the query, we could add the **FASTFIRSTROW** table hint to that table.

```

SELECT  pm.Name AS ProductModelName,
        p.Name AS ProductName,
        SUM(pin.Quantity)
FROM    Production.ProductModel pm
        JOIN Production.Product p WITH (FASTFIRSTROW)
        ON pm.ProductModelID = p.ProductModelID
        JOIN Production.ProductInventory pin
        ON p.ProductID = pin.ProductID
GROUP BY pm.Name,
        p.Name ;

```

Listing 5.36

This results in the execution plan in Figure 5.34.

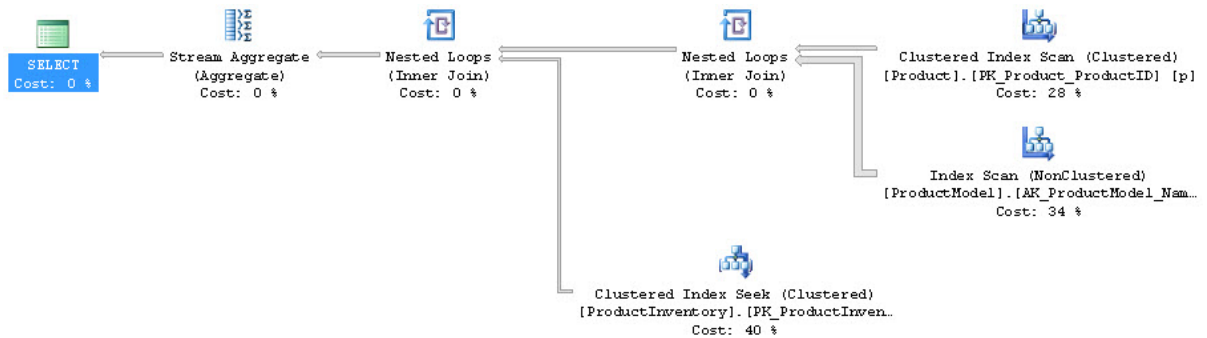


Figure 5.34

This makes the optimizer choose a different path through the data. Instead of hitting the `ProductModel` table first, it's now collecting the `Product` information first. A **Nested Loops** operator loops through the smaller set of rows from the `Product` table and compares them to the larger data set from the `ProductModel` table.

The rest of the plan is the same. The net result is that, rather than building the worktable to support the **Hash Match** join, most of the work occurs in accessing the data through the **Index Scans** and **Seeks**, with cheap **Nested Loops** joins replacing the **Hash Joins**. The cost execution time increases from 195ms in the first query to 215ms in the second.

The performance penalty comes from the fact that there was a single scan and two reads on the `ProductModel` table in the original query. Use of the **Nested Loops** join changed the behavior to a single scan with 1,009 reads. Reading more data will have an impact on the performance of queries.

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. 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. Use of these hints should be a last resort, not a standard method of operation.

Chapter 6: Cursor Operations

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 may still be occasions when a cursor is the more appropriate or more expedient way to resolve a problem. Certainly, most query processing to support application behavior, reporting and other uses, will be best solved by concentrating on set-based solutions. However, certain maintenance routines will be more easily implemented using cursors (although even these may need to be set-based in order to reduce the maintenance footprint in a production system).

A specific set of operators, within execution plans, describe the effects of the operations of a cursor. The operators, similar to those for data manipulation, are split between logical (or estimated) and physical (or actual) operators. In the case of the data manipulation operators, these represent the possible path and the actual path through a query. For cursors, there are bigger differences between the logical and physical operators. Logical operators give more information about the actions that will occur while a cursor is created, opened, fetched, closed and de-allocated. The physical operators show the functions that are part of the actual execution of the query, with less regard to the operations of the cursor.

As with all the previous execution plans, we can view plans containing cursors graphically, as text, or as XML. This chapter will use only graphical plans and will describe all of the operators that represent the action of cursors.

Simple cursors

In Listing 6.1, we declare a cursor with no options, with all defaults, and then traverse it using the `FETCH NEXT` method, returning a list of all the `CurrencyCodes` used in the `AdventureWorks2008R2` database. I'm going to use this same basic query throughout the section on cursors, because it returns a small number of rows, and because we

can easily see how changes to cursor properties affect the execution plans. The data is returned as multiple result sets, as shown in Figure 6.1.

```
DECLARE CurrencyList CURSOR
FOR
    SELECT  CurrencyCode
    FROM    Sales.Currency
    WHERE   Name LIKE '%Dollar%'

OPEN CurrencyList

FETCH NEXT FROM CurrencyList

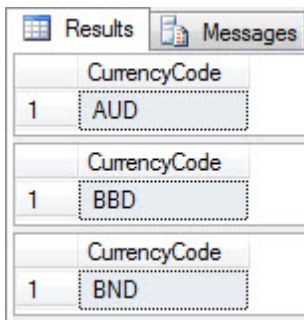
WHILE @@FETCH_STATUS = 0
    BEGIN

        -- Normally there would be operations here using data from cursor

        FETCH NEXT FROM CurrencyList
    END

CLOSE CurrencyList
DEALLOCATE CurrencyList
GO
```

Listing 6.1



The screenshot shows the SQL Server Results window with two tabs: 'Results' and 'Messages'. The 'Results' tab is active and displays three separate result sets. Each result set has a single column header 'CurrencyCode' and one row of data. The first result set contains 'AUD', the second contains 'BBD', and the third contains 'BND'. Each row is preceded by a small icon indicating the row number (1).

	CurrencyCode
1	AUD

	CurrencyCode
1	BBD

	CurrencyCode
1	BND

Figure 6.1

Logical operators

Using a graphical execution plan, we can see that the query consists of six distinct statements and therefore six distinct plans, as shown in Figure 6.2.

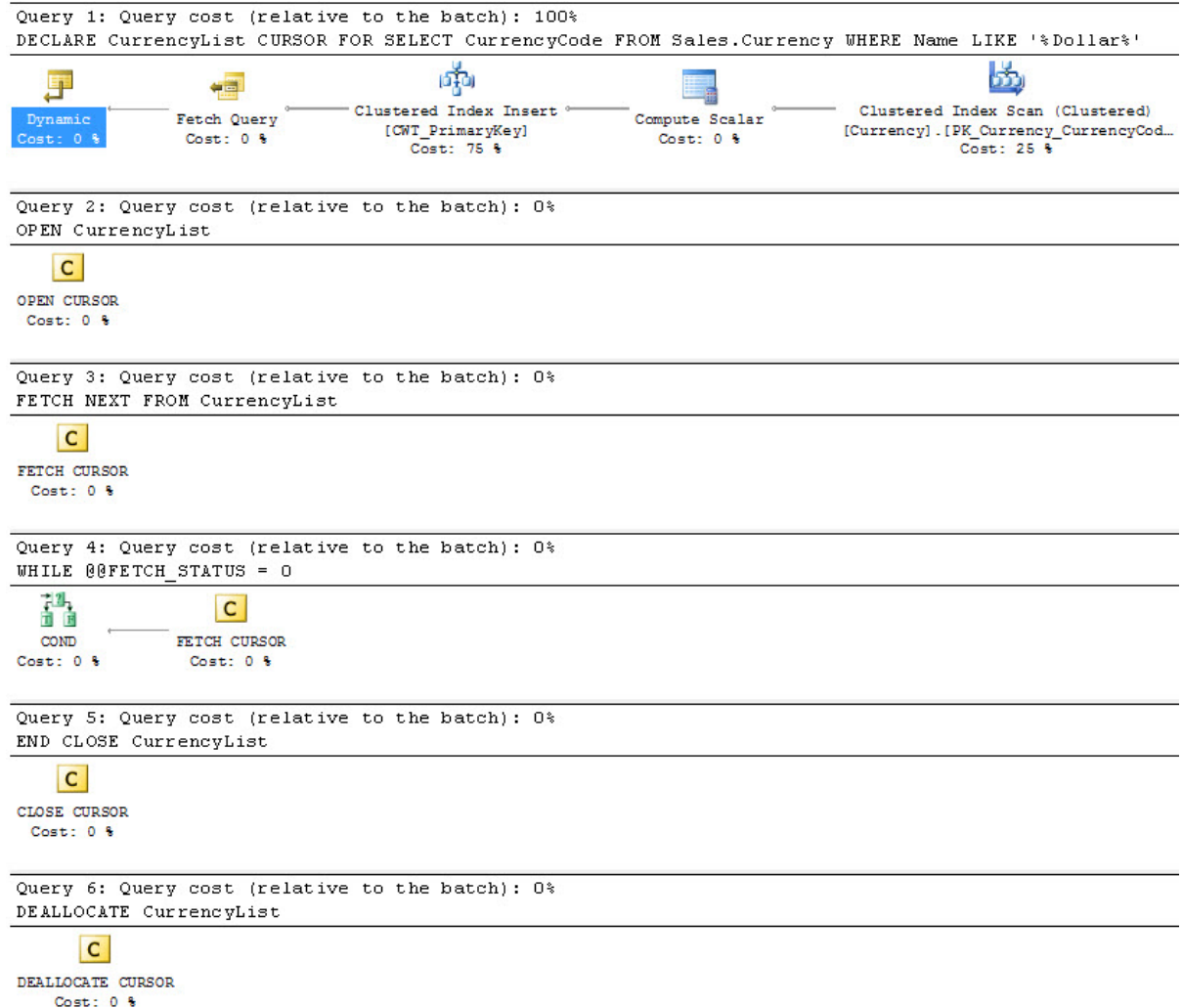


Figure 6.2

Let's look at each part of the plan individually. The top section shows the T-SQL definition of the cursor.

```
DECLARE CurrencyList CURSOR
FOR
    SELECT  CurrencyCode
    FROM    Sales.Currency
    WHERE   Name LIKE '%Dollar%'
```

Listing 6.2

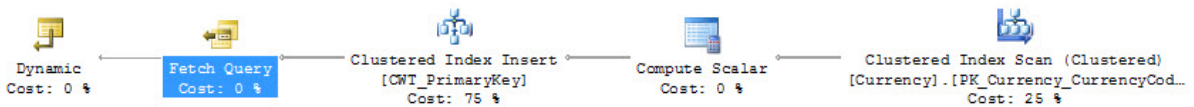


Figure 6.3

This definition in the header includes the **SELECT** statement that will provide the data that the cursor uses. This plan contains our first two cursor-specific operators but, as usual, we'll read this execution plan, starting from the right. First, there is a **Clustered Index Scan** against the **Sales.Currency** table.

Clustered Index Scan (Clustered)	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0002725
Estimated Number of Executions	1
Estimated Operator Cost	0.0033975 (25%)
Estimated Subtree Cost	0.0033975
Estimated Number of Rows	14
Estimated Row Size	45 B
Ordered	True
Node ID	2
Predicate	
[AdventureWorks2008R2].[Sales].[Currency].[Name] like N'%Dollar%'	
Object	
[AdventureWorks2008R2].[Sales].[Currency]. [PK_Currency_CurrencyCode]	
Output List	
Chk1002, [AdventureWorks2008R2].[Sales]. [Currency].CurrencyCode	

Figure 6.4

The **Clustered Index Scan** retrieves an estimated 14 rows. Following, is a **Compute Scalar** operator, which creates a unique identifier to identify the data returned by the query, independent of any unique keys on the table or tables from which the data was selected (see Figure 6.5).

Compute Scalar	
Compute new values from existing values in a row.	
Physical Operation	Compute Scalar
Logical Operation	Compute Scalar
Estimated I/O Cost	0
Estimated CPU Cost	0.0000014
Estimated Number of Executions	1
Estimated Operator Cost	0.0000203 (0%)
Estimated Subtree Cost	0.0034178
Estimated Number of Rows	14
Estimated Row Size	21 B
Node ID	1
Output List	
Chk1002, [AdventureWorks2008R2].[Sales]. [Currency].CurrencyCode, Expr1005	

Figure 6.5

With a new key value, these rows are inserted into a temporary clustered index, created in tempdb. This clustered index, commonly referred to as a worktable, is the "cursor" mechanism by which the server is able to walk through a set of data (Figure 6.6).

Clustered Index Insert	
Insert rows in a clustered index.	
Physical Operation	Clustered Index Insert
Logical Operation	Insert
Estimated I/O Cost	0.01
Estimated CPU Cost	0.000014
Estimated Number of Executions	1
Estimated Operator Cost	0.010014 (75%)
Estimated Subtree Cost	0.0134318
Estimated Number of Rows	14
Estimated Row Size	17 B
Node ID	0
Object	
[tempdb].[CWT_PrimaryKey]	
Output List	
[AdventureWorks2008R2].[Sales].[Currency].CurrencyCode, Expr1005	

Figure 6.6

After that, we get our first cursor operator, **Fetch Query**.

Fetch Query



The **Fetch Query** operator retrieves the rows from the cursor, the clustered index created above, when the **FETCH** command is issued. The ToolTip in Figure 6.7 displays the familiar information (which doesn't provide much that's immediately useful).

Fetch Query	
The query used to retrieve rows when a fetch is issued against a cursor.	
Cached plan size	24 B
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0134318

Figure 6.7

Finally, instead of yet another **Select** operator, we finish with a **Dynamic** operator.

Dynamic



The **Dynamic** operator contains the definition of the cursor. In this case, the default cursor type is a dynamic cursor, which means that it sees data changes made by others to the underlying data, including **inserts**, as they occur. This means that the data within

the cursor can change over the life of the cursor. For example, if data in the table is modified before it has been passed through the cursor, the modified data will be picked up by the cursor. This time, the ToolTip shows some slightly different, more detailed and useful information.

Dynamic	
Cursor that can see all changes made by others.	
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0134318
Statement	
DECLARE CurrencyList CURSOR	
FOR	
SELECT CurrencyCode	
FROM Sales.Currency	
WHERE Name LIKE '%Dollar%'	

Figure 6.8

We see a view of the direct T-SQL that defines the cursor.

Cursor Catchall



The next five sections of our execution plan, from Figure 6.2, all feature a generic icon known as the **Cursor Catchall**. In general, a catchall icon covers for operations that Microsoft determined didn't need their own special graphic.

In Query 2 and Query 3, we see catchall icons for the **Open Cursor** operation and the **Fetch Cursor** operation.

```
OPEN CurrencyList

FETCH NEXT FROM CurrencyList
```

Listing 6.3

```
Query 2: Query cost (relative to the batch): 0%
OPEN CurrencyList
```



```
OPEN CURSOR
Cost: 0 %
```

```
Query 3: Query cost (relative to the batch): 0%
FETCH NEXT FROM CurrencyList
```



```
FETCH CURSOR
Cost: 0 %
```

Figure 6.9

Query 4 shows the next time within the T-SQL that the `FETCH CURSOR` command was used, and it shows a language element icon, for the `WHILE` loop, as a **Cond** or **Conditional** operator. This **Conditional** operator is performing a check against the information returned from the **Fetch** operation.

```
WHILE @@FETCH_STATUS = 0
BEGIN
    --Normally there would be operations here using data from cursor
    FETCH NEXT FROM CurrencyList
END
```

Listing 6.4



Figure 6.10

Finally, Query 5 closes the cursor and Query 6 de-allocates it, removing the cursor from the tempdb.

```
CLOSE CurrencyList  
DEALLOCATE CurrencyList
```

Listing 6.5

```
Query 5: Query cost (relative to the batch): 0%  
END CLOSE CurrencyList
```



```
CLOSE CURSOR  
Cost: 0 %
```

```
Query 6: Query cost (relative to the batch): 0%  
DEALLOCATE CurrencyList
```



```
DEALLOCATE CURSOR  
Cost: 0 %
```

Figure 6.11

Physical operators

When we execute the same script, using the actual execution plan, we find that it doesn't mirror the estimated plan. Instead, we see the plan shown in Figure 6.12.

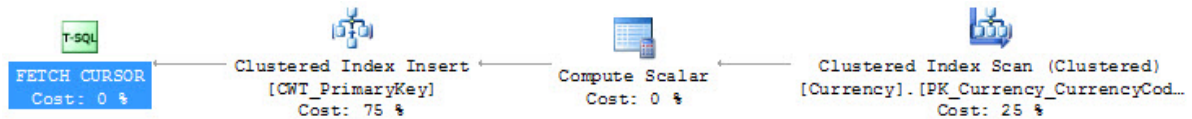


Figure 6.12

This simple plan is repeated 15 times, once for each row of data added to the cursor. The slight discrepancy between the actual number of rows, 15, and the estimated 14 rows you'll see in the ToolTip is caused by a minor disparity between the actual data and the statistics.

One interesting thing to note is that no cursor icons are present in the plan. Instead, the one cursor command immediately visible, `FETCH CURSOR`, is represented by the generic T-SQL operator icon. This is because all the physical operations that occur with a cursor are represented by the actual operations being performed, and the `FETCH` is roughly equivalent to the `SELECT` statement.

Hopefully, this execution plan demonstrates why a dynamic cursor may be costly to the system. It's performing a **Clustered Index Insert**, as well as the reads necessary to return the data to the cursor. It performs these actions as each of the 15 separate `FETCH` statements are called. The same query, outside a cursor, would return a very simple, one-step execution plan, as shown in Figure 6.13.

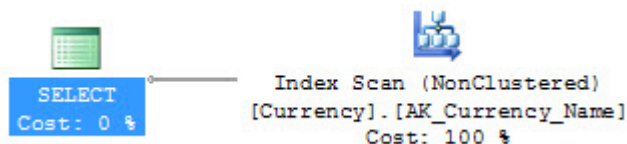


Figure 6.13

More cursor operations

Changing the settings and operations of the cursor result in differences in the plans generated. We've already seen the dynamic cursor; so let's now look at the three other types of cursor.

Static cursor

Unlike the `DYNAMIC` cursor outlined above, a `STATIC` cursor is a temporary copy of the data, created when the cursor is called. This means that it doesn't see any underlying changes to the data over the life of the cursor. To see this in action, change the cursor declaration as shown in Listing 6.6.

```
DECLARE CurrencyList CURSOR STATIC FOR
```

Listing 6.6

Logical operators

Let's first look at the estimated execution plan. You will see six distinct plans, just as with a `DYNAMIC` cursor. Figure 6.14 shows the plan for the first query, which represents the cursor definition. The remaining queries in the estimated plan look just like the `DYNAMIC` cursor in Figure 6.2.

Chapter 6: Cursor Operations

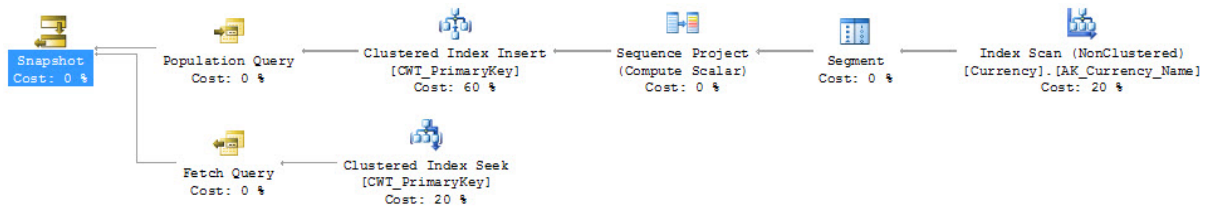


Figure 6.14

Reading the query in the direction of the physical operations, from the top right-hand side, we see an **Index Scan** to retrieve the data from the **Sales.Currency** table. This data passes to the **Segment** operator, which divides the input into segments, based on a particular column, or columns. In this case, as you can see in the ToolTip in Figure 6.15, it's based on a derived column called **Segment1006**. The derived column splits the data up in order to pass it to the next operation, which will assign the unique key.

Segment	
Segment.	
Physical Operation	Segment
Logical Operation	Segment
Estimated I/O Cost	0
Estimated CPU Cost	0.0000003
Estimated Number of Executions	1
Estimated Operator Cost	0.0000041 (0%)
Estimated Subtree Cost	0.0034016
Estimated Number of Rows	14
Estimated Row Size	17 B
Segment Column	Segment1006
Node ID	2
Output List	
[AdventureWorks2008R2].[Sales].	
[Currency].CurrencyCode, Segment1006	

Figure 6.15

Cursors require worktables and, to make them efficient, SQL Server creates them as a clustered index with a unique key. With a **STATIC** cursor, the key is generated after the segments are defined. The segments are passed on to the **Compute Scalar** operator, which adds a string valued "1" for the next operation, **Sequence Project**. This logical operator represents a physical task that results in a **Compute Scalar** operation. It's adding a new column as part of computations across the set of data. In this case, it's creating row numbers through an internal function called `i4_row_number`. These row numbers are used as the identifiers within the clustered index.

Sequence Project	
Adds columns to perform computations over an ordered set.	
Physical Operation	Sequence Project
Logical Operation	Compute Scalar
Estimated I/O Cost	0
Estimated CPU Cost	0.0000011
Estimated Number of Executions	1
Estimated Operator Cost	0.0000162 (0%)
Estimated Subtree Cost	0.0034178
Estimated Number of Rows	14
Estimated Row Size	17 B
Node ID	1
Output List	
[AdventureWorks2008R2].[Sales].	
[Currency].CurrencyCode, Expr1005	

Figure 6.16

The data, along with the new identifiers, is then passed to the **Clustered Index Insert** operator and then on to the **Population Query Cursor** operator.

Population Query



The **Population Query Cursor** operator, as stated in the description of the operator on the **Properties** sheet says, *"populates the work table for a cursor when the cursor is opened"* or, in other words, from a logical standpoint, this is when the data that has been marshaled by all the other operations is loaded into the worktable.

The **Fetch Query** operation retrieves the rows from the cursor via an **Index Seek** on the index in tempdb, the worktable created to manage the cursor. Notice that, in this case, the **Fetch Query** operation is defined in a separate sequence, independent from the **Population Query**. This is because this cursor is static, unlike the dynamic cursor, which reads its data each time it's accessed.

Snapshot



Finally, we see the **Snapshot** cursor operator, which represents a cursor that does not see changes made to the data by separate data modifications.

Clearly, with a single **INSERT** operation, and then a simple **Clustered Index Seek** to retrieve the data, this cursor will operate much faster than the dynamic cursor. The **Index Seek** and the **Fetch** operations show how the data will be retrieved from the cursor.

Physical operators

If we execute the query and display the actual execution plan, we get two distinct plans. The first plan is the query that loads the data into the cursor worktable, as represented by the clustered index. The second plan is repeated, and we see a series of plans identical to

the one shown for Query 2, below, which demonstrates how the cursor is looped through by the **WHILE** statement.

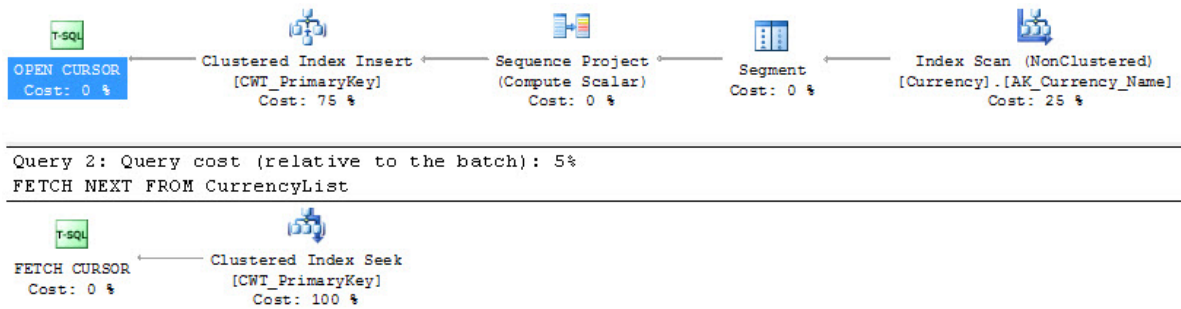


Figure 6.17

These execution plans accurately reflect what the estimated plan intended. Note that the cursor was loaded when the **OPEN CURSOR** statement was called. We can even look at the **Clustered Index Seek** operator to see it using the row identifier created during the population of the cursor.

Clustered Index Seek	
Scanning a particular range of rows from a clustered index.	
Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Number of Rows	1
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0.0032831 (100%)
Estimated Subtree Cost	0.0032831
Estimated Number of Rows	1
Estimated Row Size	17 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	0
Object	
[tempdb].[CWT_PrimaryKey]	
Output List	
[CWT].COLUMN0, [CWT].ROWID	
Seek Predicates	
Seek Keys[1]: Prefix: [CWT].ROWID = Scalar Operator (FETCH_RANGE((0)))	

Figure 6.18

Keyset cursor



The KEYSET cursor retrieves a defined set of keys as the data defined within the cursor. This means that it doesn't retrieve the actual data but, instead, a set of identifiers for finding that data later. The KEYSET cursor allows for the fact that data may be updated during the life of the cursor. This behavior leads to yet another execution plan, different from the previous two examples.

Let's change the cursor definition again.

```
DECLARE CurrencyList CURSOR KEYSET FOR
```

Listing 6.7

Logical operators

Figure 6.19 shows the estimated execution plan.

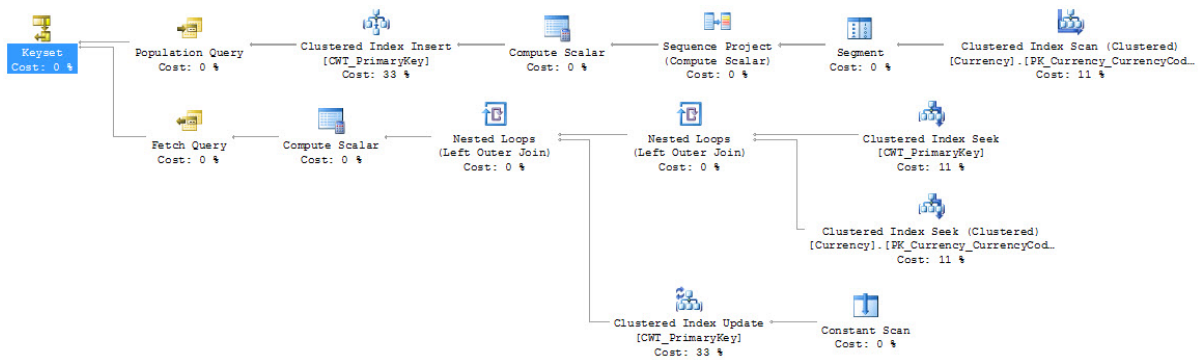


Figure 6.19

Now that we've worked with cursors a bit, it's easy to recognize the two paths defined in the estimated plan: one for populating the cursor and one for fetching the data from the cursor.

The top line of the plan, containing the **Population Query** operation, is almost exactly the same as that defined for a **Static** cursor. The second **Scalar** operation is added as a status check for the row. It ends with the **Keyset** operator, indicating that the cursor can see updates, but not inserts.

The major difference is evident in how the **Fetch Query** works, in order to support the updating of data after the cursor was built. Figure 6.20 shows that portion of the plan in more detail.

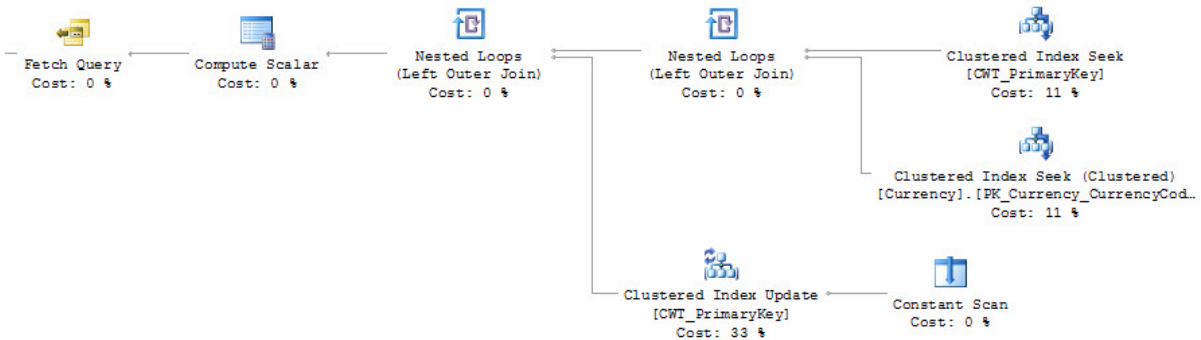


Figure 6.20

Going to the right and top of the **Fetch Query** definition, we find that it first retrieves the key from the index created in the **Population Query**. Then, to retrieve the data, it joins it through a **Nested Loop** operation to the **Sales.Currency** table. This is how the **Keyset cursor** manages to get updated data into the set returned while the cursor is active.

The **Constant Scan** operator scans an internal table of constants; basically, it's just a place for adding data later in the plan. The data from the **Constant Scan** feeds into the **Clustered Index Update** operator in order to be able to change the data stored inside the cursor, if necessary. This data is joined to the first set of data through a **Nested Loop** operation and finishes with a **Compute Scalar**, which represents the row number.

Physical operators

When the cursor is executed, we get the plan shown in Figure 6.21.

Chapter 6: Cursor Operations

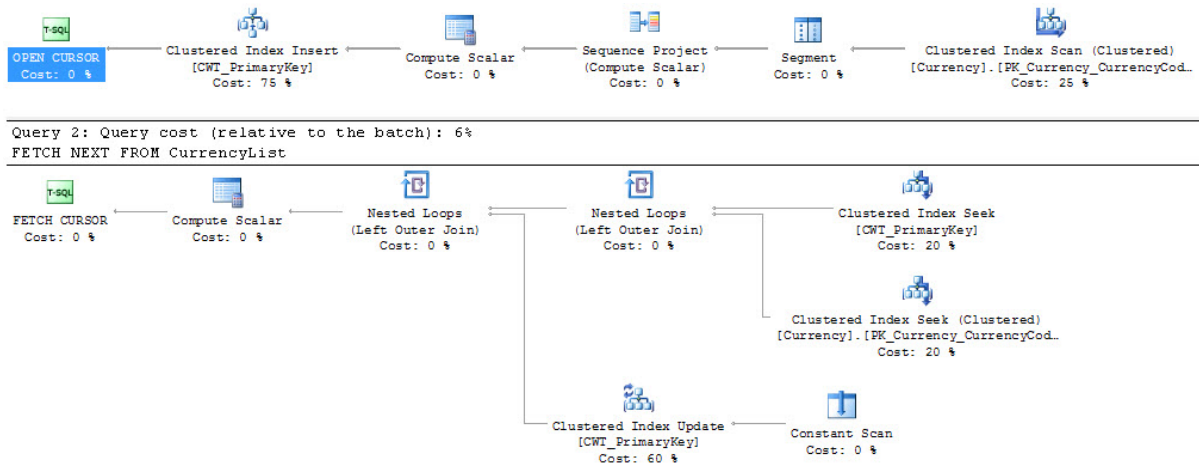


Figure 6.21

Query 1 contains the **Open Cursor** operator, and populates the key set exactly as the estimated plan envisioned.

In Query 2, the **FETCH NEXT** statements against the cursor activate the **Fetch Cursor** operation 15 times, as the cursor walks through the data. While this can be less costly than a dynamic cursor, it's clearly more costly than a **Static** cursor. The performance issues come from the fact that the cursor queries the data twice, once to load the key set and a second time to retrieve the row data. Depending on the number of rows retrieved into the worktable, this can be a costly operation.

READ_ONLY cursor

Each of the preceding cursors, except for **Static**, allowed the data within the cursor to be updated. If we define the cursor as **READ_ONLY** and look at the execution plan, we sacrifice the ability to capture changes in the data, but we create what is known as a **Fast Forward** cursor.

```
DECLARE CurrencyList CURSOR READ_ONLY FOR
```

Listing 6.8

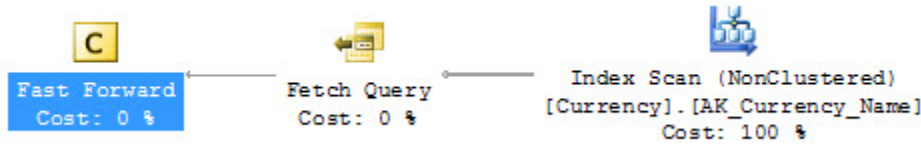


Figure 6.22

Clearly, this represents the simplest cursor definition plan that we've examined so far. Unlike for other types of cursor, there is no branch of operations within the estimated plan. It simply reads what it needs directly from the data. In our case, an **Index Scan** operation against the `CurrencyName` index shows how this is accomplished. The amount of I/O, compared to any of the other execution plans, is reduced since there is not a requirement to populate any worktables. Instead, there is a single step: get the data. The actual execution plan is identical except that it doesn't have to display the **Fast Forward** logical operator.

Cursors and performance

Cursors are notorious for their ability to cause performance problems within SQL Server. The following example shows, for each of the cursors, how you can tweak their performance. The example also demonstrates how to get rid of the cursor and use a set-based operation that performs even better.

Let's assume that, in order to satisfy a business requirement, we need a report that lists the number of sales by store, assigning an order to them, and then, depending on the amount sold, displays how good a sale it is considered. Here's a query, using a dynamic cursor, which might do the trick.

Chapter 6: Cursor Operations

```
DECLARE @WorkTable TABLE
(
    DateOrderNumber INT IDENTITY(1, 1) ,
    Name VARCHAR(50) ,
    OrderDate DATETIME ,
    TotalDue MONEY ,
    SaleType VARCHAR(50)
)

DECLARE @DateOrderNumber INT ,
        @TotalDue MONEY

INSERT INTO @WorkTable
( Name ,
  OrderDate ,
  TotalDue
)
SELECT s.Name ,
       soh.OrderDate ,
       soh.TotalDue
FROM   Sales.SalesOrderHeader AS soh
       JOIN Sales.Store AS s
         ON soh.SalesPersonID = s.SalesPersonID
WHERE  soh.CustomerID = 29731
ORDER BY soh.OrderDate

DECLARE ChangeData CURSOR
FOR
    SELECT DateOrderNumber ,
           TotalDue
    FROM   @WorkTable

OPEN ChangeData

FETCH NEXT FROM ChangeData INTO @DateOrderNumber, @TotalDue

WHILE @@FETCH_STATUS = 0
BEGIN
    -- Normally there would be operations here using data from cursor
    IF @TotalDue < 1000
        UPDATE @WorkTable
        SET     SaleType = 'Poor'
        WHERE  DateOrderNumber = @DateOrderNumber
    ELSE
```

Chapter 6: Cursor Operations

```
IF @TotalDue > 1000
    AND @TotalDue < 10000
    UPDATE @WorkTable
    SET     SaleType = 'OK'
    WHERE   DateOrderNumber = @DateOrderNumber
ELSE
    IF @TotalDue > 10000
        AND @TotalDue < 30000
        UPDATE @WorkTable
        SET     SaleType = 'Good'
        WHERE   DateOrderNumber = @DateOrderNumber
    ELSE
        UPDATE @WorkTable
        SET     SaleType = 'Great'
        WHERE   DateOrderNumber = @DateOrderNumber
    FETCH NEXT FROM ChangeData INTO @DateOrderNumber, @TotalDue
END

CLOSE ChangeData
DEALLOCATE ChangeData

SELECT *
FROM   @WorkTable
```

Listing 6.9

Whether or not you've written a query like this, I imagine that you've certainly seen one. The data returned from the query looks something like this:

Number	Name	OrderDate	TotalDue	SaleType
1	Trusted Catalog Store	2001-07-01	18830.1112	Good
2	Trusted Catalog Store	2001-10-01	13559.0006	Good
3	Trusted Catalog Store	2002-01-01	51251.2959	Great
4	Trusted Catalog Store	2002-04-01	78356.9835	Great
5	Trusted Catalog Store	2002-07-01	9712.8886	OK
6	Trusted Catalog Store	2002-10-01	2184.4578	OK
7	Trusted Catalog Store	2003-01-01	1684.8351	OK
8	Trusted Catalog Store	2003-04-01	1973.4799	OK
9	Trusted Catalog Store	2003-07-01	8897.326	OK
10	Trusted Catalog Store	2003-10-01	10745.818	Good
11	Trusted Catalog Store	2004-01-01	2026.9753	OK
12	Trusted Catalog Store	2004-04-01	702.9363	Poor

The estimated execution plan (not shown here) displays the plan for populating the temporary table, and updating the temporary table, as well as the plan for the execution of the cursor. The cost to execute this script, as a dynamic cursor, includes, not only the query against the database tables, `Sales.OrderHeader` and `Sales.Store`, but also the `INSERT` into the temporary table, all the `UPDATES` of the temporary table, and the final `SELECT` from the temporary table. The result is about 27 different scans and about 113 reads.

Let's take a look at a subsection of the actual execution plan, which shows the `FETCH` from the cursor and one of the updates.

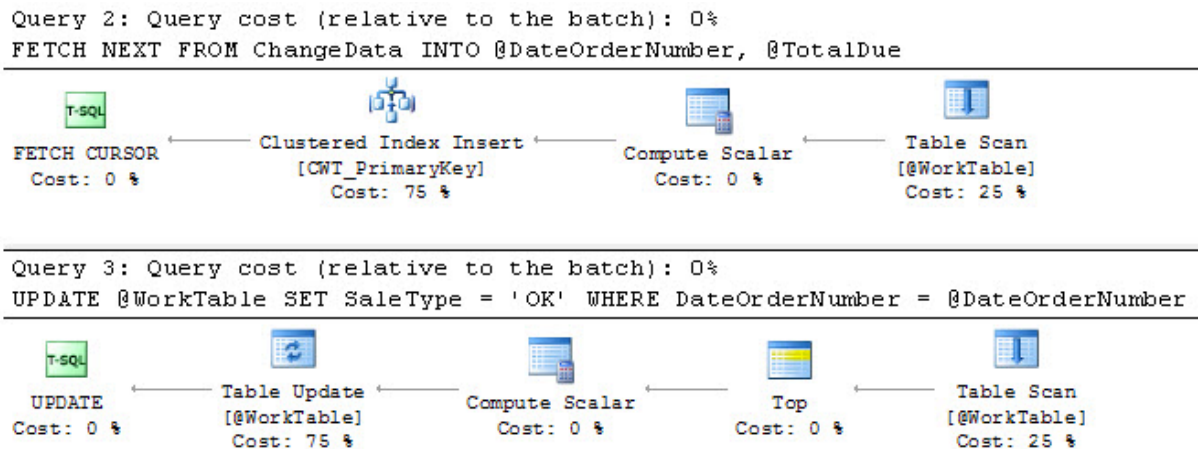


Figure 6.23

Simply counting scans and reads through the `STATISTICS I/O` output is not possible because the cursor repeats hundreds of times. Each iteration through this query, to return a row, invokes 2 scans and 105 reads on the tables involved. This repeats for each of the hundreds of rows that make up the cursor. The total execution time was about 185ms.

To see which cursor might perform better, let's change this **Dynamic** cursor to a **Static** cursor.

```
DECLARE ChangeData CURSOR STATIC
```

Listing 6.10

With this modification in place, execution time dropped to about 100ms, with less work being required to attempt to maintain data.

Next, let's see how the **Keyset Cursor** fares.

```
DECLARE ChangeData CURSOR KEYSET
```

Listing 6.11

This time execution time decreased as less work was required for maintaining the cursor. Execution time went down to 78ms. Let's change the cursor again so that it uses the read only option.

```
DECLARE ChangeData CURSOR READ_ONLY
```

Listing 6.12

Execution time held about the same. In some tests, it was even slightly longer, at 83ms.

The tests so far show that the **Keyset Cursor** is the fastest. Let's see if we can't make it a bit faster by changing the cursor declaration again.

```
DECLARE ChangeData CURSOR FAST_FORWARD
```

Listing 6.13

This **FAST_FORWARD** setting creates a **FORWARD_ONLY** cursor, which also results in a **READ_ONLY** cursor.

In many cases, setting the cursor to `FORWARD_ONLY` and `READ_ONLY`, through the `FAST_FORWARD` setting, results in the fastest performance. However, in this case, performance was unaffected for all the tests using this approach.

Let's see if we have any more luck by making the key set cursor to `FORWARD_ONLY`.

```
DECLARE ChangeData CURSOR FORWARD_ONLY KEYSET
```

Listing 6.14

The resulting execution plan is the same, and the performance hasn't really changed. So, short of tuning other parts of the procedure, the simple `KEYSET` is probably the quickest way to access this data.

However, what if we eliminate the cursor entirely? We can rewrite the script so that it looks as shown in Listing 6.15.

```
SELECT ROW_NUMBER() OVER ( ORDER BY soh.OrderDate ) ,
       s.Name ,
       soh.OrderDate ,
       soh.TotalDue ,
       CASE WHEN soh.TotalDue < 1000 THEN 'Poor'
            WHEN soh.TotalDue BETWEEN 1000 AND 10000 THEN 'OK'
            WHEN soh.TotalDue BETWEEN 10000 AND 30000 THEN 'Good'
            ELSE 'Great'
       END AS SaleType
FROM   Sales.SalesOrderHeader AS soh
       JOIN Sales.Store AS s ON soh.SalesPersonID = s.SalesPersonID
WHERE  soh.CustomerID = 29731
ORDER BY soh.OrderDate
```

Listing 6.15

This query returns exactly the same data, but the performance is radically different. It performs a single scan on the `SalesOrderHeader` table, and about 40 reads between the two tables. The execution time is recorded as 0 ms, which isn't true, but gives you an

indication of how much faster it is than the cursor. Instead of a stack of small execution plans, we have a single-step execution plan.

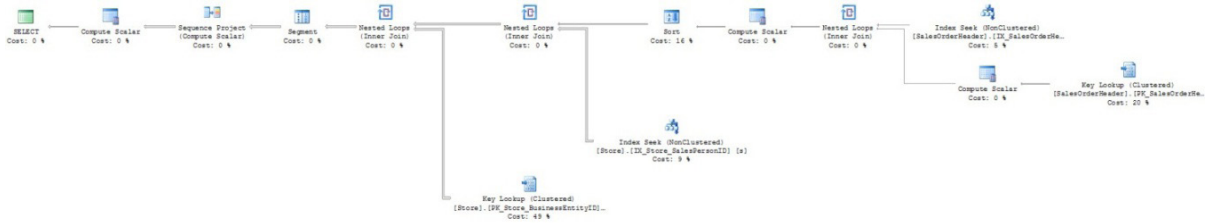


Figure 6.24

The plan is actually a bit too large to see clearly here but the key take-away is that the main cost for this query, accounting for 54% of the cost of the plan, is the **Key Lookup** operator in the lower right. That's a tuning opportunity, as we saw in a previous chapter. Eliminating the **Lookup** will make this query run faster.

This example was simple, the amount of data was relatively small, and most of the cursors operated well enough to be within the performance margins of most large-scale systems. However, even with a small data set it was possible to see differences among the types of cursors. Most significantly, it was possible to achieve a major performance increase with the elimination of the cursor.

It shouldn't be too difficult to see how, when working with 12,000 rows instead of 12, the cursor operations would be very costly, and just how much time and resource usage changing from cursors to set-based operations will save your production systems resources.

Summary

More often than not, we should avoid cursors in order to take advantage of the set-based nature of T-SQL and SQL Server. Set-based operations just work better. However, when faced with the necessity of a cursor, understanding what you're likely to see in the execution plans, estimated and actual, will assist you in using cursors appropriately.

Don't forget that the estimated plan shows both how the cursor will be created, in the top part of the plan, and how the data in the cursor will be accessed, in the bottom part of the plan. The primary differences between a plan generated from a cursor and one from a set-based operation are in the estimated execution plans. Other than that, as you have seen, reading these plans is no different than reading the plans from a set-based operation: start at the right and top and work your way to the left. There are just a lot more plans generated, because of the way in which cursors work.

Chapter 7: Special Datatypes and Execution Plans

The introduction of the Common Language Runtime (CLR), and with it the ability to run .NET code within SQL Server, may not have taken the database world by storm, but Microsoft has certainly started to put it to work. It has exploited the capabilities introduced with CLR to add a lot more functionality to SQL Server, through the addition of special data types.

In addition to the special functionality around XML, available since SQL Server 2005, SQL Server 2008 saw the introduction of the spatial data type and the hierarchy data type, to name just a couple, and more will be coming with new releases of SQL Server.

Some of these data types have no effect on execution plans, while others have a major effect. This chapter examines three of the more commonly used special data types now available within SQL Server. First, we'll go over XML in some detail. Then we'll cover the hierarchy data type. Finally, we'll look at spatial data and have a very short introduction to spatial indexes.

XML

XML plays an ever greater role in a large numbers of applications, and the use of XML within stored procedures affects the execution plans generated.

You can break down XML operations within SQL Server into four broad categories:

- **Storing XML** – The XML datatype is used to store XML, as well as to provide a mechanism for XQuery queries and XML indexes.

- **Output** relational data to XML – The `FOR XML` clause can be used to output XML from a query.
- **Inserting** XML into tables – `OPENXML` accepts XML as a parameter and opens it within a query for storage or manipulation, as structured data.
- **Querying** XML documents using `XQuery`.

Storing XML within the database is largely outside the scope of this chapter, but we will cover getting XML out of the database, and we can do it in a number of different ways. We will cover the various types of XML output, using the `FOR XML` commands. Each form of the `FOR XML` command requires different T-SQL, and will result in different execution plans, as well as differences in performance.

Getting XML into a format you can use within your T-SQL code requires special calls. You can read XML within SQL Server using either `OPENXML` or `XQuery`. `OPENXML` provides a rowset view of an XML document. We will explore its use via some execution plans for a basic `OPENXML` query, and will outline some potential performance implications. However, the true strength of querying XML within SQL Server is through `XQuery`. We'll examine a few simple `XQuery` examples, via execution plans, but this is a huge topic and we will barely scratch its surface here. In fact, to cover it in any depth at all would require an entire book of its own.

While all these methods of accessing and manipulating XML are very useful, they do come at a cost. XML can cause performance issues in one of two ways. Firstly, the XML parser, which is required to manipulate XML, uses memory and CPU cycles that you would normally have available only for T-SQL. In addition, the manner in which you use the XML, input or output, will affect the plans generated by SQL Server and can lead to performance issues.

Secondly, manipulating XML data uses good, old-fashioned T-SQL statements, and poorly written XML queries can impact performance just as any other query can, and we must tune them in the same manner as any other query.

FOR XML

If you want to output the result of a query in XML format, you use the `FOR XML` clause. You can use the `FOR XML` clause in any of the following four modes:

- **AUTO** – Returns results as nested XML elements in a simple hierarchy (think: table = XML element).
- **RAW** – Transforms each row in the results into an XML element, with a generic `<row />` identifier as the element tag.
- **EXPLICIT** – Allows you to define explicitly, in the query itself, the shape of the resulting XML tree.
- **PATH** – A simpler alternative to **EXPLICIT** for controlling elements, attributes, and the overall shape of the XML tree.

Each of these methods requires different T-SQL in order to arrive at the same output, and each is associated with different performance and maintenance issues. We will explore all four options and I'll point out where each has strengths and weaknesses.

Just to introduce the capabilities that are possible with `FOR XML`, you can see that the first example, in Listing 7.1, produces a list of employees and their addresses. There is no requirement for any type of direct manipulation of the XML output, and the query is simple and straightforward, so we'll use `XML AUTO` mode.

```
SELECT  p.FirstName ,
        p.LastName ,
        e.Gender ,
        a.AddressLine1 ,
        a.AddressLine2 ,
        a.City ,
        a.StateProvinceID ,
        a.PostalCode
FROM    Person.Person p
        INNER JOIN HumanResources.Employee e
```

```

        ON p.BusinessEntityID = e.BusinessEntityID
    INNER JOIN Person.BusinessEntityAddress AS bea
        ON e.BusinessEntityID = bea.BusinessEntityID
    INNER JOIN Person.Address a
        ON bea.AddressID = a.AddressID
FOR    XML AUTO;

```

Listing 7.1

This generates the actual execution plan shown in Figure 7.1.

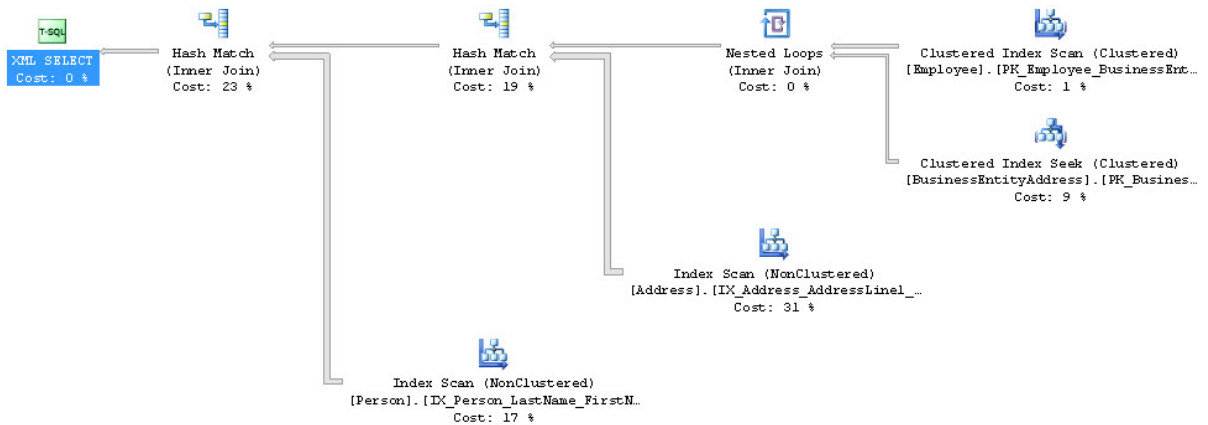


Figure 7.1

The difference between this execution plan and that for any "normal" query may be hard to spot. It's at the very beginning of the logical processing order. Instead of a **T-SQL Select** operation, we see an **XML Select** operation. That is the only change. Otherwise, it's simply a query.

To see all the various methods at work, instead of the slightly complicated example above, consider the second, somewhat simpler, query in Listing 7.2. We'll compare the output of this query using the various FOR XML modes, starting over again with AUTO mode.

```

SELECT  s.Name AS StoreName ,
        bec.PersonID ,
        bec.ContactTypeID
FROM    Sales.Store s
        JOIN Person.BusinessEntityContact AS bec
          ON s.BusinessEntityID = bec.BusinessEntityID
ORDER BY s.Name
FOR      XML AUTO;

```

Listing 7.2

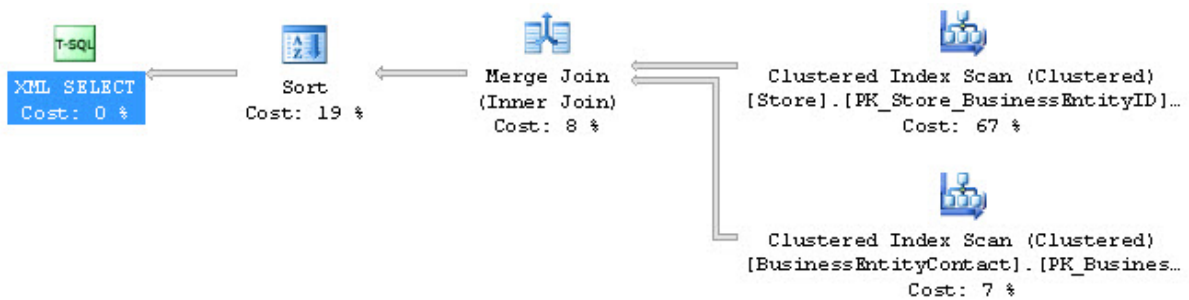


Figure 7.2

The execution plan is not terribly complicated. Following the logical processing, the **XML Select** operator requests data from the **Sort** operation, which is satisfying the **ORDER BY** clause in Listing 7.2. This data stream feeds into the **Sort** from the **Merge Join** operator, which puts together the output from the two different **Clustered Index Scan** operators. These operators, if you examine their **Properties** page, have an ordered output, which explains why the optimizer can use the **Merge** operator. The estimated cost of the plan is 0.12. The query runs in about 89ms and has 111 reads. Listing 7.3 shows the XML output.

```

<s StoreName="A Bicycle Association">
  <bec PersonID="2050" ContactTypeID="11" />
</s>
<s StoreName="A Bike Store">
  <bec PersonID="933" ContactTypeID="11" />
</s>

```

Listing 7.3

The same results are seen, in this case, if we use XML RAW mode.

XML EXPLICIT mode allows you to exert some control over the format of the XML generated by the query – for example, if the application or business requirements may need a very specific XML definition, rather than the generic one supplied by XML AUTO.

Without getting into a tutorial on XML EXPLICIT, you write the query in a way that dictates the structure of the XML output. Listing 7.4 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 7.4

Figure 7.3 shows the actual execution plan for this query, which is somewhat more complex.

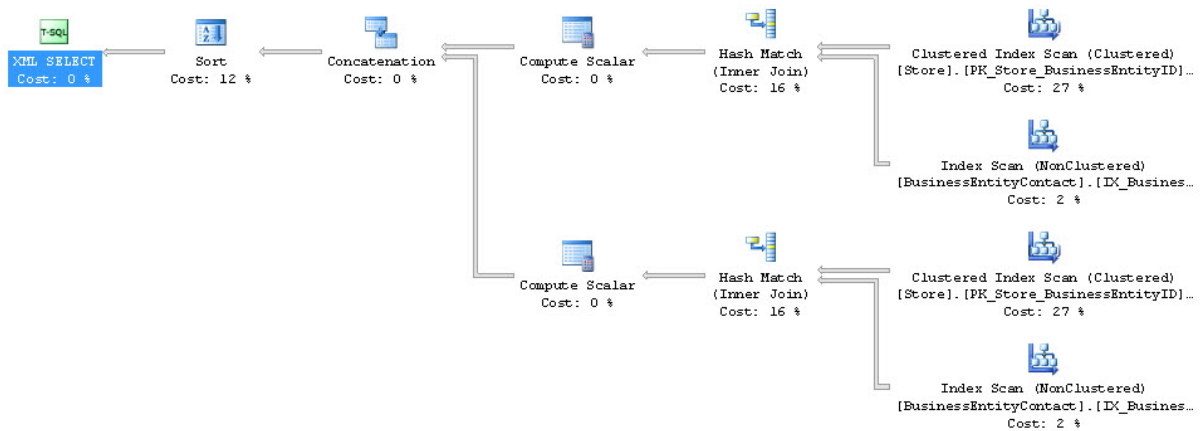


Figure 7.3

In order to build the hierarchy of XML, it's necessary to use the **UNION** clause in T-SQL. Reading this execution plan, again, in logical processing order, from left to right, you see the same **XML Select** and **Sort** operators as you did in the plan in Figure 7.3. However, this time, the next operation is the **Concatenation** operator, which combines the two different streams of data, as defined by the **UNION** clause. Each of the two branches of the plan are the same after that, which is not surprising, considering the query accesses the two tables in the same way, in both branches of the T-SQL statement. The **Compute Scalar** operator in each branch fills in the Tag and Parent columns. These columns are added to the output of a **Hash Match** operator that the optimizer, in this case, thought was more efficient than the **Merge Join** shown in Figure 7.2. The **Clustered Index Scan** operations for each branch are the same, and satisfy the definition of the **UNION** clause.

The estimated cost of the plan is much higher at 0.29. The execution time shoots up to 119ms and there are 214 reads. Listing 7.5 shows the XML output.

```
<store StoreName="A Bicycle Association">
  <BECContact PersonID="2050" ContactTypeID="11" />
</store>
<store StoreName="A Bike Store">
  <BECContact PersonID="933" ContactTypeID="11" />
</store>
```

Listing 7.5

Remove the `FOR XML EXPLICIT` clause, rerun the query and look at the new execution plan, and you'll see that, apart from seeing the **Select** instead of **XML Select** operator, the plans are the same, up to and including the cost of each of the operations. The difference isn't in the execution plan, but rather in the results. 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.

Even with this relatively simple example, you can see how, because of the multiple queries joined together via a `UNION`, while you get more control over the XML output, it comes at the cost of increased maintenance, due to the need for the `UNION` clause and the explicit naming standards. This leads to decreased performance due to the increased number of queries required to put the data together.

An extension of the `XML AUTO` mode allows you to specify the `TYPE` directive in order to better control the output of the results of the query as the XML datatype. The query in Listing 7.6 is essentially the same as the previous one, but expressed using this simpler syntax, available in SQL Server 2005 and later.

```
SELECT  s.Name AS StoreName ,
        ( SELECT      bec.BusinessEntityID ,
              bec.ContactTypeID
          FROM        Person.BusinessEntityContact bec
          WHERE       bec.BusinessEntityID = s.BusinessEntityID
        FOR
          XML AUTO ,
              TYPE ,
              ELEMENTS
        )
FROM    Sales.Store s
ORDER BY s.Name
FOR     XML AUTO ,
        TYPE;
```

Listing 7.6

The `ELEMENTS` directive specifies that the columns within the sub-select appear as sub-elements within the outer `SELECT` statement, as part of the structure of the XML.

```
<s StoreName="A Bicycle Association">
  <bec>
    <BusinessEntityID>2051</BusinessEntityID>
    <ContactTypeID>11</ContactTypeID>
  </bec>
</s>
```

Listing 7.7

Figure 7.4 shows the resulting execution plan, which does look a little different.

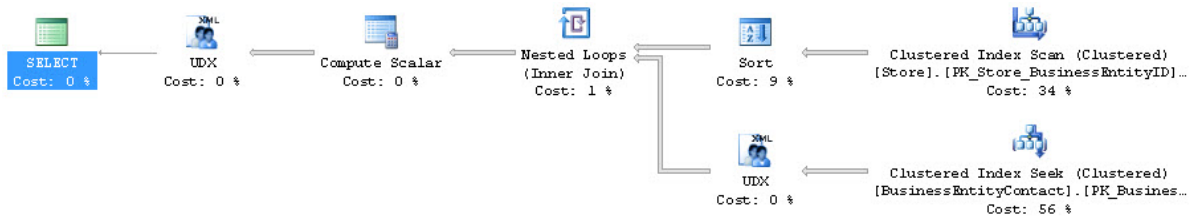


Figure 7.4

The estimated cost of the plan is 0.23, the execution time is now 99ms and there are a shocking 701 scans on `BusinessEntityContact`. The first thing to note is that the optimizer has introduced two **UDX** operators. The **UDX** operator is an extended operator used by `XPATH` and `XQUERY` operations. `XPATH` and `XQUERY` are two different ways to querying XML data directly. In our case, by examining the **Properties** window, we can see that the **UDX** operator on the lower right of the plan is creating the XML data.

[-] Misc	
[-] Actual Number of Rows	701
[-] Actual Rebinds	0
[-] Actual Rewinds	0
[-] Defined Values	Expr1004
Description	UDX.
Estimated CPU Cost	0.000001
Estimated I/O Cost	0
Estimated Number of Executions	701
Estimated Number of Rows	1
Estimated Operator Cost	0.000701 (0%)
Estimated Rebinds	700
Estimated Rewinds	0
Estimated Row Size	4035 B
Estimated Subtree Cost	0.130379
Logical Operation	UDX
Name	FOR XML
Node ID	5
Number of Executions	701
[-] Output List	Expr1004
Parallel	False
Physical Operation	UDX
[-] Used UDX Columns	[AdventureWorks2008R2].[Person].[BusinessEntit

Figure 7.5

The output is Expr1004, which consists of the two columns from the **BusinessEntity-Contact** table: **BusinessEntityID** and **ContactTypeID**. The optimizer uses a **Nested Loops** operator to join this data with the sorted data from the **Clustered Index Scan** on the **Stores** table. It then applies a **Compute Scalar** operator, probably some of the XML definitions or a checksum (calculation value). It then feeds this to the next **UDX** operator for the final output as fully-fledged XML.

Finally, the XML PATH mode simply outputs the XML data type, and makes it much easier to output mixed elements and attributes. Using this mode, the query we've already walked through twice now looks as shown in Listing 7.8.

```
SELECT  s.Name AS "@StoreName" ,
        bec.PersonID AS "BECContact/@PersonId" ,
        bec.ContactTypeID AS "BECContact/@ContactTypeID"
FROM    Sales.Store s
        JOIN Person.BusinessEntityContact AS bec
          ON s.BusinessEntityID = bec.BusinessEntityID
ORDER BY s.Name
FOR      XML PATH;
```

Listing 7.8

This results in the same execution plan as shown in Figure 7.2, as well as the same estimated cost (0.12) and approximately the same execution time at 78ms. Listing 7.9 shows the XML output.

```
<row StoreName="A Bicycle Association">
  <BusinessEntityContact BusinessEntityID="2051" ContactTypeID="11" />
</row>
<row StoreName="A Bike Store">
  <BusinessEntityContact BusinessEntityID="934" ContactTypeID="11" />
</row>
```

Listing 7.9

Of the various methods of arriving at the same XML output, XML PATH clearly results in the simplest execution plan as well as the most straightforward T-SQL. These factors make XML PATH probably the easiest code to maintain, while still exercising control over the format of the XML output. The optimizer transforms the output to XML only at the end of the process, using the familiar **XML Select** operator. This requires less overhead and processing power throughout the query.

From a performance standpoint, to get XML out of a query in the fastest way possible, you should use fewer XQuery or XPath operations. With that in mind, the least costly operations above, based on reads and scans, are the final `XML PATH` and the original `XML AUTO`, which both behaved more or less identically.

```
Table 'BusinessEntityContact'. Scan count 1, logical reads 8 ...  
Table 'Store'. Scan count 1, logical reads 103 ...
```

However, more often than not, the XML created in the `AUTO` won't meet with the application design, so you'll probably end up using `XML PATH` most often.

`XML EXPLICIT` performance was poor, with more scans and reads than the previous two options:

```
Table 'Worktable'. Scan count 0, logical reads 0 ...  
Table 'BusinessEntityContact'. Scan count 2, logical reads 8 ...  
Table 'Store'. Scan count 2, logical reads 206 ...
```

`XML AUTO` with `TYPE` was truly horrendous due to the inclusion of the `UDX` operations, causing a large number of reads and scans:

```
Table 'BusinessEntityContact'. Scan count 701, logical reads 1410 ...  
Table 'Store'. Scan count 1, logical reads 103, physical reads 0 ...
```

OPENXML

We can use `OPENXML` to read XML within SQL Server or XQuery. `OPENXML` takes in-memory XML data and converts it into a format that, for viewing purposes, can be treated as if it were a normal table. This allows us to use it within regular T-SQL operations. Most often, we use it to take data from the XML format and change it into structured storage within a normalized database. In order to test 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="1/1/2007" AverageRate=".553"
                  EndOfDayRate= ".558" />
    <CurrencyRate FromCurrencyCode="USD" ToCurrencyCode="UTE"
                  CurrencyRateDate="6/1/2007" AverageRate=".928"
                  EndOfDayRate= "1.057" />
  </Currency>
</ROOT>
```

Listing 7.10

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 7.11 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="1/1/2007" AverageRate=".553"
    EndOfDayRate= ".558" />
  <CurrencyRate FromCurrencyCode="USD" ToCurrencyCode="UTE"
    CurrencyRateDate="6/1/2007" AverageRate=".928"
    EndOfDayRate= "1.057" />
</Currency>
</ROOT>'

EXEC 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 sp_xml_removedocument @iDoc
ROLLBACK TRAN
```

Listing 7.11

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 7.6

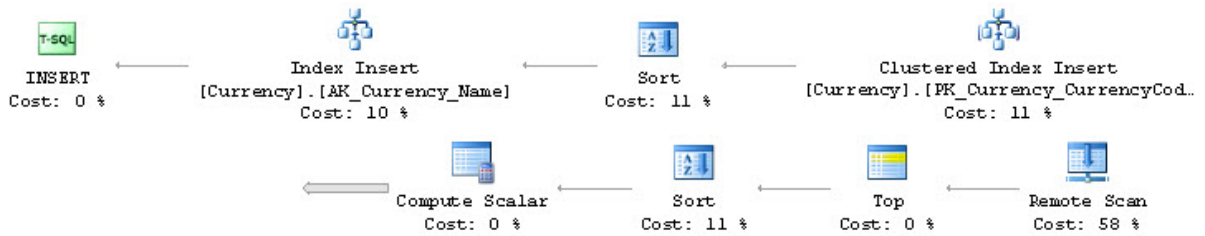



Figure 7.6

The physical flow starts on the top right of the plan, which is the bottom right portion of Figure 7.6 (to see the whole plan, I've graphically split it in half). A quick scan of the plan reveals no new XML icons. All the OPENXML statement processing is handled within the **Remote Scan** icon. This operator represents the opening of a remote object, meaning a DLL or some external process such as 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 a table 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.

Examining the estimated plan reveals none of the extensive XML statements that are present in this query: even the XML stored procedures `sp_xml_preparedocument` and `sp_xml_remove document` are referenced by simple logical T-SQL icons, as you can see in Figure 7.7.

```
Query 3: Query cost (relative to the batch): 0%
EXEC sp_xml_preparedocument @iDoc OUTPUT, @Xml
```



```
EXECUTE PROC
Cost: 0 %
```

Figure 7.7

The only place where we can really see the evidence of the XML is in the **Output List** for the **Remote Scan**. In Figure 7.8, 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, [OpenXML].Curren
[1]	[OpenXML].CurrencyCode
Column	CurrencyCode
Table	[OpenXML]
[2]	[OpenXML].CurrencyName
Column	CurrencyName
Table	[OpenXML]

Figure 7.8

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 second execution plan describes the INSERT against the CurrencyRate table.

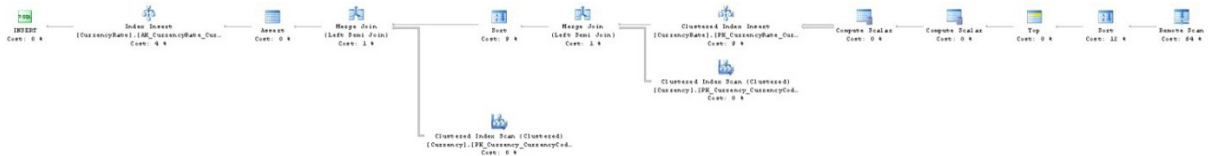


Figure 7.9

This query is the more complicated of the two because of the extra steps required for the maintenance of referential integrity between the Currency and CurrencyRate tables. 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 operation picked by the optimizer in this instance.

As you can see, it's easy to bring XML data into the database for use within your queries, or for inclusion within your database. As discussed previously, OPENXML is a useful tool for importing the semi-structured data within the XML documents into the well-maintained relational database structure. It can also allow you to pass in data for other uses. For example, you can pass in a list of variables to be used as a join in a `SELECT` statement. The main point to take away is that once the OPENXML has been formatted, you get to use it as if it were just another table within your queries.

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 parser as soon as possible. This will reduce the amount of time that the memory is allocated within your system.

XQuery

Along with the introduction of the XML data type in SQL Server 2005, came the introduction of XQuery as a method for querying XML data. Effectively, the inclusion of 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, each of which is reflected in execution plans in different ways.

- `.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 Bool 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 dataset.

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 this new language.

Using the .exist method

You are likely to use the `.exist` method quite frequently when working with XML data. In Listing 7.12, we query the résumés of all employees to find out which of the people hired were once sales managers.

```
SELECT  p.LastName ,
        p.FirstName ,
        e.HireDate ,
        e.JobTitle
FROM    Person.Person p
        INNER JOIN HumanResources.Employee e
            ON p.BusinessEntityID = e.BusinessEntityID
        INNER JOIN HumanResources.JobCandidate jc
            ON e.BusinessEntityID = jc.BusinessEntityID
            AND jc.Resume.exist(' declare namespace
Resume="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/
Resume";
/res:Resume/res:Employment/res:Emp.JobTitle[contains
(., "Sales Manager")]') = 1;
```

Listing 7.12

Figure 7.10 shows the actual execution plan for this query.



Then, another **Nested Loops** operator is used to combine data from a new operator, a **Table Valued Function**. This **Table Valued Function**, subtitled "XML Reader with XPath filter" operator 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 example, is actually a part of the query engine and is represented by a distinct icon.

273

Table Valued Function	
Table valued function.	
Physical Operation	Table Valued Function
Logical Operation	Table Valued Function
Actual Number of Rows	4
Estimated I/O Cost	0
Estimated CPU Cost	1.004
Estimated Number of Executions	1
Number of Executions	2
Estimated Operator Cost	0.0227048 (54%)
Estimated Subtree Cost	0.0227048
Estimated Number of Rows	4.52284
Estimated Row Size	4057 B
Actual Rebinds	2
Actual Rewinds	0
Node ID	8
Object	
[XML Reader with XPath filter]	
Output List	
[XML Reader with XPath filter].value, [XML Reader with XPath filter].lvalue	

Figure 7.11

These rows are passed to a **Filter** operator that determines if the XPath query we defined equals 1. 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.

Using the .query method

The `.query` method returns XML. 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. These constructs greatly extend the versatility of XQuery, making it comparable to T-SQL.

- **For** – Used to iterate XML nodes. The **For** expression binds some number of iterator variables, in this case, one, to input sequences, our `ss:StoreSurvey`. It works a lot like a **For/Each** loop.
- **Let** – This allows you to name and use repeating expressions within an XML-like variable.
- **Where** – You can limit the results using the **Where** expression. It works just like a **WHERE** clause in T-SQL.
- **Order** – Sorts the results, just like **ORDER BY** in T-SQL (not covered here).
- **Return** – Simply defines the results coming back, kind of like the **Select** clause in T-SQL except it includes all kinds of XML commands for formatting.

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. 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 7.13 shows our example query and execution plan.

```
SELECT 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 s
WHERE s.SalesPersonID = 279;
```

Listing 7.13

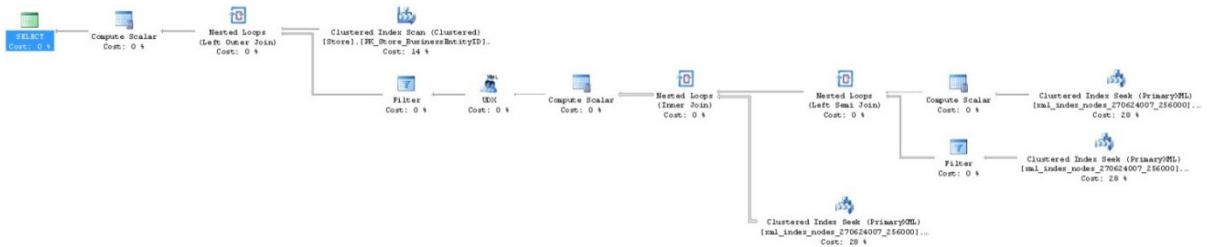


Figure 7.12

The query actually consisted of two simple queries:

- a regular T-SQL query against the Store table to return the rows where the SalesPersonId = 279
- a query 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.

As always, start at the top and right of Figure 7.12. The first operator is 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. Going over to the right to find the second stream of data for the join, we find a familiar operation: a **Clustered Index Seek**.

This time though, it's going against an XML clustered index. Figure 7.13 shows a blow-up of that part of the plan.

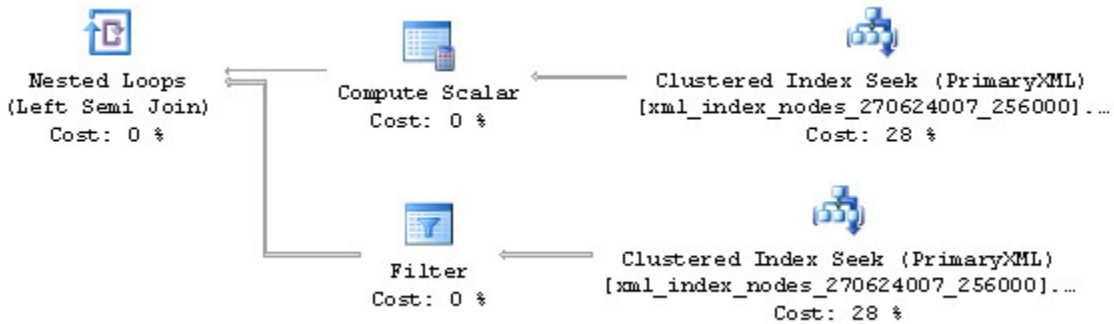


Figure 7.13

The data in the XML data type is stored separately from the rest of the table, and the **Clustered Index Seek** operations you see are the retrieval of that data.

You can see, in Figure 7.14, that the **Index Seek** is occurring on PXML_Store_Demographics, returning the 80 rows from the index that match the CustomerId field from the store. Below this, another **Clustered Index Seek** gathers data matching the CustomerId, but adds SquareFeet as part of the output. This data is filtered and then the outputs are combined through a left join.

From there, it feeds on out, joining against all the rest of the XML data, before going through a **UDX** operator that outputs the formatted XML data. This is combined with the original rows returned from the Store table. Of note is the fact that the XQuery information is being treated almost as if it were T-SQL. The data above is retrieved from an XML index, which stores all the data with multiple rows for each node, sacrificing disk space for speed of recovery. SQL Server creates an internal table to store this XML index. XML is stored as a binary large object (BLOB). The internal table is created from the definition of the index by shredding the XML (i.e., breaking it up into its component parts) ahead of time, as data is modified. These indexes can then speed access, but come at the cost of disk storage and processing time when data is stored.

Clustered Index Seek (PrimaryXML)	
Scanning a particular range of rows from a clustered index.	
Physical Operation	Clustered Index Seek
Logical Operation	Clustered Index Seek
Actual Number of Rows	80
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001713
Number of Executions	80
Estimated Number of Executions	80
Estimated Operator Cost	0.156065 (28%)
Estimated Subtree Cost	0.156065
Estimated Number of Rows	1
Estimated Row Size	22 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	9
Predicate	
[AdventureWorks2008R2].[sys].	
[xml_index_nodes_270624007_256000].[hid] as	
[StoreSurvey:1].[hid]='Å€'	
Object	
[AdventureWorks2008R2].[sys].	
[xml_index_nodes_270624007_256000].	
[PXML_Store_Demographics] [StoreSurvey:1]	
Output List	
[AdventureWorks2008R2].[sys].	
[xml_index_nodes_270624007_256000].id,	
[AdventureWorks2008R2].[sys].	
[xml_index_nodes_270624007_256000].hid,	
[AdventureWorks2008R2].[sys].	
[xml_index_nodes_270624007_256000].pk1	
Seek Predicates	
Seek Keys[1]: Prefix: [AdventureWorks2008R2].[sys].	
[xml_index_nodes_270624007_256000].pk1 = Scalar	
Operator([AdventureWorks2008R2].[Sales].[Store].	
[BusinessEntityID] as [s].[BusinessEntityID])	

Figure 7.14

These examples don't begin to cover the depth of what's available within XQuery. It really is a whole new language and syntax that you'll have to learn in order to take complete advantage of what it has to offer.

For an even more thorough introduction, read this white paper offered from Microsoft at <HTTP://MSDN2.MICROSOFT.COM/EN-US/LIBRARY/MS345122.ASPX>.

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.

Hierarchical Data

SQL Server can store hierarchical data using `HIERARCHYID`, a native CLR introduced in SQL Server 2008. 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 7.14 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;
DECLARE @BEId INT;

SET @BEId = 2;

SELECT @ManagerID = e.OrganizationNode
FROM HumanResources.Employee AS e
WHERE e.BusinessEntityID = @BEId;
```

```

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 7.14

This query returns fourteen rows and runs in about 214ms with 48 reads on the `Person.Person` table and three on the `HumanResources.Employee` table. Figure 7.15 shows the execution plan.

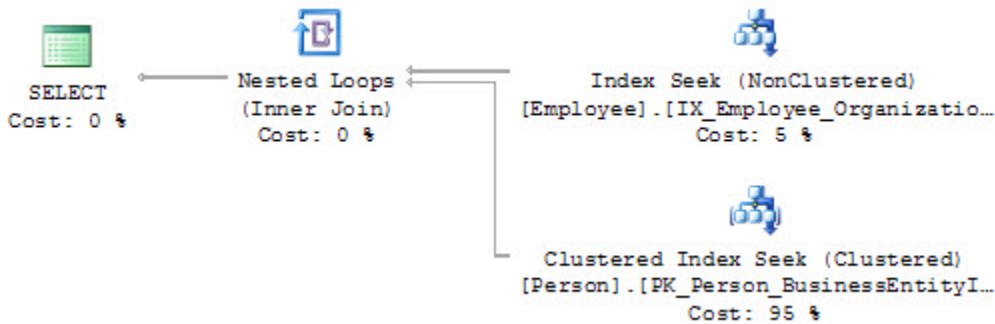


Figure 7.15

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** commands on the `Person.Person` table, which results in all the additional reads on that table. The interesting aspect of this plan is the Seek Predicate of the **Index Seek** operation, as shown in the ToolTip in Figure 7.16.

Index Seek (NonClustered)	
Scan a particular range of rows from a nonclustered index.	
Physical Operation	Index Seek
Logical Operation	Index Seek
Actual Number of Rows	14
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001857
Estimated Number of Executions	1
Number of Executions	1
Estimated Operator Cost	0.0033107 (5%)
Estimated Subtree Cost	0.0033107
Estimated Number of Rows	26.1
Estimated Row Size	11 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
Node ID	2
Object	
[AdventureWorks2008R2].[HumanResources]. [Employee].[IX_Employee_OrganizationNode] [e]	
Output List	
[AdventureWorks2008R2].[HumanResources]. [Employee].BusinessEntityID	
Seek Predicates	
Seek Keys[1]: Start: [AdventureWorks2008R2]. [HumanResources].[Employee].OrganizationNode >=	
Scalar Operator([@ManagerId]), End:	
[AdventureWorks2008R2].[HumanResources]. [Employee].OrganizationNode <= Scalar Operator	
([@ManagerId].DescendantLimit())	

Figure 7.16

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. These indexes are similar to other indexes, in that they contain key values, in this case, the `OrganizationNode`, and a lookup to the clustered index, but not other columns. If I had run the query and selected a different column, such as `JobTitle` from the `HumanResources.Employee` table, the query would have changed to a **Clustered Index Scan** since the index on `OrganizationNode` would no longer be a covering index.

We could explore a number of 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. 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 *Introduction to SQL Server 2008 Spatial Data* (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 non-clustered index in SQL Server, these indexes can, and do, work with functions, but not all functions. Listing 7.15 shows a query that could result in the use of an index on a SQL Server 2008 R2 database.

```
DECLARE @MyLocation GEOGRAPHY = GEOGRAPHY::STPointFromText
      ( 'POINT(-122.33383 47.61066 )',
        4326 )

SELECT  p.LastName + ', ' + p.FirstName ,
        a.AddressLine1 ,
        a.City ,
        a.PostalCode ,
        sp.Name AS StateName ,
        a.SpatialLocation
FROM    Person.Address AS a
        JOIN Person.BusinessEntityAddress AS bea
          ON a.AddressID = bea.AddressID
        JOIN Person.Person AS p
          ON bea.BusinessEntityID = p.BusinessEntityID
        JOIN Person.StateProvince AS sp
          ON a.StateProvinceID = sp.StateProvinceID
WHERE   @MyLocation.STDistance(a.spatiallocation) < 1000
```

Listing 7.15

This query creates a point, which coincides with the Seattle Sheraton, near where, most years, PASS hosts its annual summit. It then uses the `STDistance` calculation on that point to find all addresses in the database that are within a kilometer (1,000 meters) of that location. Unfortunately, there's no index on the `Address` table for our spatial query to use so, to see a spatial index in action, we can create the one shown in Listing 7.16.

```
CREATE SPATIAL INDEX [ix_Spatial] 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]
```

Listing 7.16

After running the query, we get an execution plan that is rather large and involved when you consider the number of tables in the query. Figure 7.18 focuses in on the operators directly applicable to the spatial index.

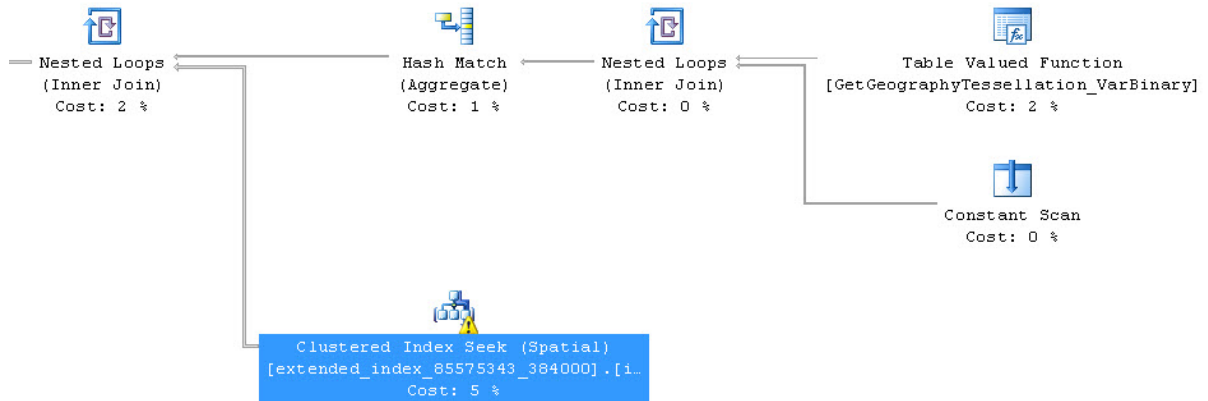


Figure 7.17

The operation on the index itself is at the bottom of Figure 7.18. You can see that the estimated cost for the operation is 5% of the overall cost of the plan, but you also have to take into account that the data is going to require other operations. The **Table Valued Function** is the creation of the spatial point, @MyLocation. This data is compared to the spatial data stored in the clustered index through the banal **Nested Loops** operator and a join.

You can see the complexity of the operations by drilling into the properties of the operators. The output of the **Nested Loops** operator is visible in Figure 7.19. You can see that multiple expressions are passed on and values that are not a part of the query itself, but are necessary for all the operations required to consume the data from the spatial index.

Outer References	Expr1075, Expr1076, Expr1084
[1]	Expr1075
Column	Expr1075
[2]	Expr1076
Column	Expr1076
[3]	Expr1084
Column	Expr1084
Output List	[AdventureWorks2008R2].[sys].[extended_index_85
[1]	[AdventureWorks2008R2].[sys].[extended_index_85
Column	Cell_Attributes
Database	[AdventureWorks2008R2]
Schema	[sys]
Table	[extended_index_85575343_384000]
[2]	[AdventureWorks2008R2].[sys].[extended_index_85
Column	SRID
Database	[AdventureWorks2008R2]
Schema	[sys]
Table	[extended_index_85575343_384000]
[3]	[AdventureWorks2008R2].[sys].[extended_index_85
Column	pk0
Database	[AdventureWorks2008R2]
Schema	[sys]
Table	[extended_index_85575343_384000]
[4]	Expr1073
Column	Expr1073
[5]	Expr1074
Column	Expr1074
Parallel	False
Physical Operation	Nested Loops
WithUnorderedPrefetch	True

Figure 7.18

The number of operators involved does make this plan more complicated. It's worth noting that the query without an index ran in about 2.2 seconds and in only 87ms with a spatial index.

While these functions are complex and require a lot more knowledge to implement, you can see that the execution plans still use the same tools to understand these operations. That's the point I'm trying to get across here, not specifically how spatial indexing works in all regards.

Summary

The introduction of these complex data types, XML, Hierarchical and Spatial, 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.

XML provides mechanisms for storage and retrieval of XML data and, as we covered earlier in the book, with XQuery we can retrieve information from execution plans directly from the procedure cache. The other data types don't have as profound an impact on what you should expect to see in execution plans, but their use does introduce some additional issues and concerns that you need to learn to keep an eye out for within execution plans. However, as you can see, generating and interpreting the plan still uses the same mechanisms and methods we've been working with throughout the book.

Chapter 8: Advanced Topics

In the previous chapters, we have discussed how the optimizer generates execution plans and how to interpret them, and we have examined plans for some moderately complex queries, including most of the common SQL Server objects, such as stored procedures, views, indexes, cursors, and so on. In our discussion of hints, we even walked through some ways in which we could exercise a measure of control over how the optimizer generates an execution plan.

In this chapter, we will take a tour of some of the more advanced topics related to the interpretation and manipulation of execution plans, covering the points below.

- **Large-scale execution plans** – How to interpret them.
- **Parallelism** – Why you might want to use parallelism, how to control it in your query execution and how to interpret parallel plans.
- **Forced parameterization** – Used to replace hard-coded literals with parameters and maximize the possibility of plan reuse; used mainly in systems subject to a large amount of ad hoc, or client-generated T-SQL.
- **Using plan guides** – To exercise control over a query through hints without changing the actual code; an invaluable tool when dealing with third-party applications.
- **Using plan forcing** – To capture and reuse an execution plan, the final word in controlling many of the decisions made by the optimizer.

Reading Large-scale Execution Plans

The most important thing to remember when dealing with execution plans that cover large numbers of tables and large numbers of individual plans is that the rules don't change. The optimizer uses the same criteria to determine the optimal type of join, type of index to use, and so on, whether you're dealing with two tables or two hundred.

However, the nature of the optimizer is such that, when faced with a truly large and complex plan, it's unlikely to spend too much time trying to find the perfect execution plan. This means, as execution plans become more complex, the need to understand what decisions the optimizer made, why, and how to change them, becomes much more important.

Let's look at what I'd consider a reasonably large-scale execution plan (although I've seen much larger). The stored procedure in Listing 8.1 returns the appropriate dataset, based on whether or not a particular individual used any special offers. In addition, if a particular special offer is requested, then the stored procedure executes a different query and returns a second, different result set.

```
DROP PROCEDURE Sales.uspGetDiscountRates;
GO
CREATE PROCEDURE Sales.uspGetDiscountRates
(
    @BusinessEntityId INT ,
    @SpecialOfferId INT
)
AS
BEGIN TRY
    -- determine if sale using special offer exists
    IF EXISTS ( SELECT *
                FROM      Person.Person AS p
                        INNER JOIN Sales.Customer AS c
                        ON p.BusinessEntityID = c.PersonID
                        INNER JOIN Sales.SalesOrderHeader AS soh
                        ON soh.CustomerID = c.CustomerID
                        INNER JOIN Sales.SalesOrderDetail AS sod
```

Chapter 8: Advanced Topics

```
        ON soh.SalesOrderID = sod.SalesOrderID
    INNER JOIN Sales.SpecialOffer AS spo
        ON sod.SpecialOfferID = spo.SpecialOfferID
    WHERE p.BusinessEntityID = @BusinessEntityId
        AND spo.[SpecialOfferID] = @SpecialOfferId )

BEGIN
    SELECT p.LastName + ', ' + p.FirstName ,
           ea.EmailAddress ,
           p.Demographics ,
           spo.Description ,
           spo.DiscountPct ,
           sod.LineTotal ,
           pr.Name ,
           pr.ListPrice ,
           sod.UnitPriceDiscount
    FROM Person.Person AS p
    INNER JOIN Person.EmailAddress AS ea
        ON p.BusinessEntityID = ea.BusinessEntityID
    INNER JOIN Sales.Customer AS c
        ON p.BusinessEntityID = c.PersonID
    INNER JOIN Sales.SalesOrderHeader AS soh
        ON c.CustomerID = soh.CustomerID
    INNER JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
    INNER JOIN Sales.SpecialOffer AS spo
        ON sod.SpecialOfferID = spo.SpecialOfferID
    INNER JOIN Production.Product pr
        ON sod.ProductID = pr.ProductID
    WHERE p.BusinessEntityID = @BusinessEntityId
        AND sod.[SpecialOfferID] = @SpecialOfferId;

END
-- use different query to return other data set
ELSE
    BEGIN
        SELECT p.LastName + ', ' + p.FirstName ,
               ea.EmailAddress ,
               p.Demographics ,
               soh.SalesOrderNumber ,
               sod.LineTotal ,
               pr.Name ,
               pr.ListPrice ,
               sod.UnitPrice ,
               st.Name AS StoreName ,
               ec.LastName + ', ' + ec.FirstName
```

```

                                AS SalesPersonName
FROM    Person.Person AS p
        INNER JOIN Person.EmailAddress AS ea
            ON p.BusinessEntityID = ea.BusinessEntityID
        INNER JOIN Sales.Customer AS c
            ON p.BusinessEntityID = c.PersonID
        INNER JOIN Sales.SalesOrderHeader AS soh
            ON c.CustomerID = soh.CustomerID
        INNER JOIN Sales.SalesOrderDetail AS sod
            ON soh.SalesOrderID = sod.SalesOrderID
        INNER JOIN Production.Product AS pr
            ON sod.ProductID = pr.ProductID
        LEFT JOIN Sales.SalesPerson AS sp
            ON soh.SalesPersonID = sp.BusinessEntityID
        LEFT JOIN Sales.Store AS st
            ON sp.BusinessEntityID = st.SalesPersonID
        LEFT JOIN HumanResources.Employee AS e
            ON st.BusinessEntityID = e.BusinessEntityID
        LEFT JOIN Person.Person AS ec
            ON e.BusinessEntityID = ec.BusinessEntityID
WHERE    p.BusinessEntityID = @BusinessEntityID;
END

--second result SET
IF @SpecialOfferId = 16
BEGIN
    SELECT    p.Name ,
              p.ProductLine
    FROM      Sales.SpecialOfferProduct sop
              INNER JOIN Production.Product p
                  ON sop.ProductID = p.ProductID
    WHERE     sop.SpecialOfferID = 16;
END

END TRY
BEGIN CATCH
    SELECT    ERROR_NUMBER() AS ErrorNumber ,
              ERROR_MESSAGE() AS ErrorMessage;
    RETURN ERROR_NUMBER();
END CATCH
RETURN 0;
```

Listing 8.1

This type of procedure does not represent an optimal way of accessing the required data. The first time we run the query, the optimizer creates a plan based on the initial parameters supplied. Because of the IF statement, the second and subsequent runs of the procedure can result in different statements within the query being run, using a very suboptimal plan.

Unfortunately, most DBAs are going to run into things like this at some point in their career. We'll execute the procedure with the parameters shown in Listing 8.2.

```
EXEC [Sales].[uspGetDiscountRates]
    @BusinessEntityId = 1423, -- int
    @SpecialOfferId = 16 -- int
```

Listing 8.2

This query returns multiple datasets, depending on the values passed to it. Figure 8.1 shows the estimated execution plan.

Obviously, this plan is unreadable without drilling down. However, even from this high-level view, you can still see the logical steps of the query. The first grouping of operators, labeled "1," describes the first conditional query that checks for the existence of the special offer. The second group of operators, labeled "2," shows the first query in the IF statement. The third group of operators, labeled "3," contains the second query in the IF statement. Finally, the fourth group of operators describes the last query, which runs when the script receives the `SpecialOfferID = 16`.

While this execution plan may look intimidating, it is not doing anything that we haven't seen elsewhere. It's just doing a lot more of it. The key to investigating plans of this type is to be undaunted by their size and remember the basic methods for walking the plan. Start at the top and on the right, and work your way through.

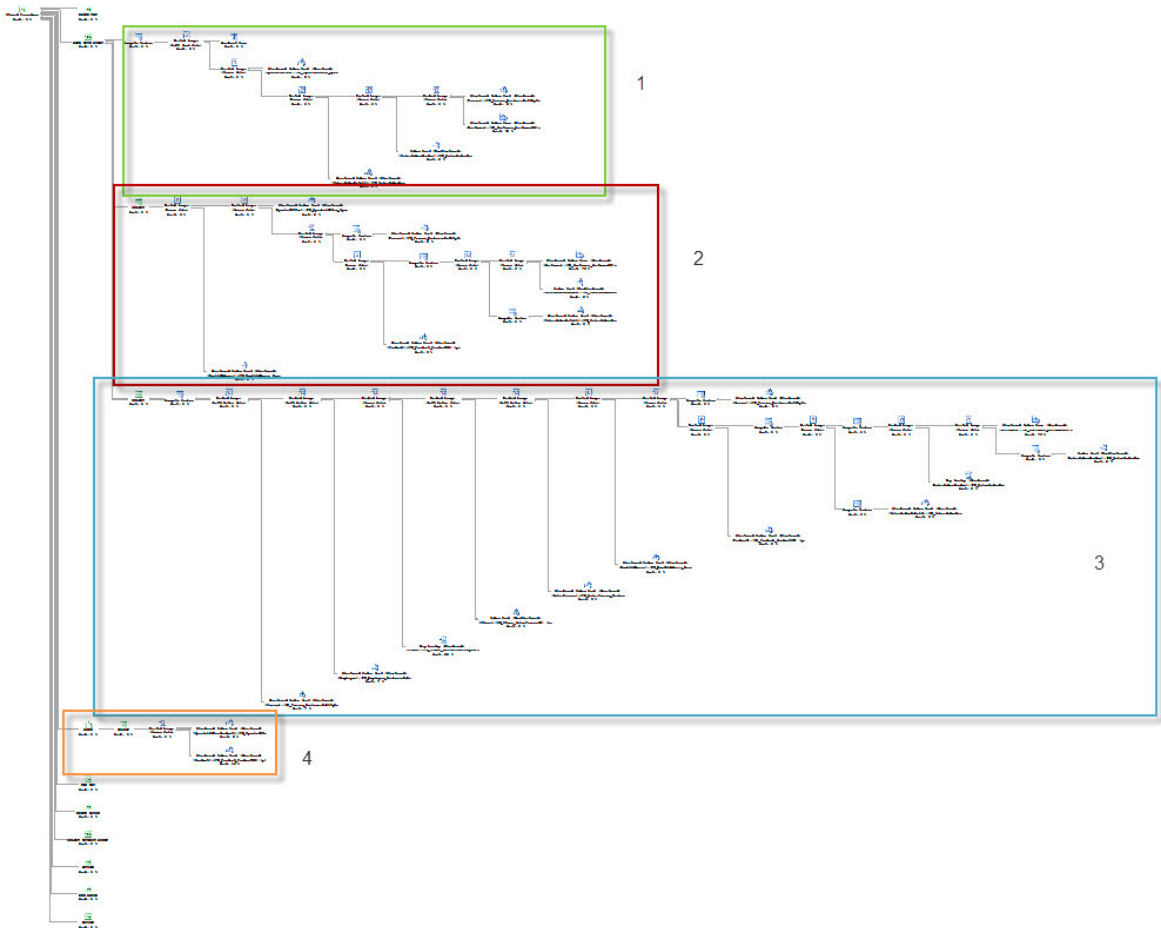


Figure 8.1

You have at least one tool that can help you when working with a large graphical plan. In the lower right of the results pane in the query window, when you're looking at an execution plan, you'll see a little plus sign, as shown in Figure 8.2.

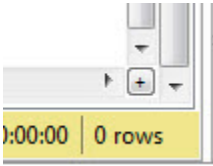


Figure 8.2

Click on the plus sign to open a little window, showing a representation of the entire execution plan. Keep your mouse button depressed, and drag the cursor across the window. You'll see that this moves a small "viewing rectangle" around the plan, as shown in Figure 8.3.

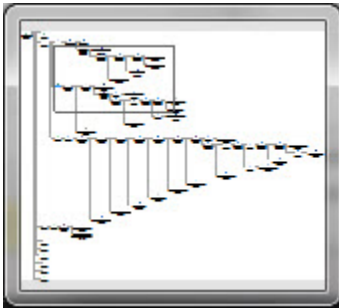


Figure 8.3

As you drag the viewable area, you'll notice that the main display in the results pane tracks your mouse movements. In this way, you can navigate around a large execution plan and keep track of where you are within the larger context of the plan, as well as view the individual operators that you're investigating.

You can scroll around each of the individual queries outlined above and identify which operators are likely to be problematic or in need of tuning. You can look at the estimated costs on the operators and the width of the pipes that are moving data between the various operators. You'll be able to look for operators which might be extraneous to the plan and which you could remove, by rewriting the query. You can even possibly identify

operators that could benefit from modification or the addition of indexes. For example, the first query has a single operator, a **Clustered Index Scan** on **Sales.Customer**, which takes 73% of the estimated cost of the query. That might be a good place to start tuning this query.

Another way to deal with extremely large execution plans, and one that is likely to be easier to use than simply scrolling around on the screen, is to take advantage of the fact that every execution plan is stored as XML. You can use XQuery to query the plan directly, as demonstrated in Listing 8.3.

```
WITH XMLNAMESPACES (DEFAULT N'http://schemas.microsoft.com/sqlserver/2004/07/
showplan'),
QueryPlans AS
(
SELECT RelOp.pln.value(N'@PhysicalOp', N'varchar(50)') AS OperatorName,
RelOp.pln.value(N'@NodeId', N'varchar(50)') AS NodeId,
RelOp.pln.value(N'@EstimateCPU', N'varchar(50)') AS CPOCost,
RelOp.pln.value(N'@EstimateIO', N'varchar(50)') AS IOCost,
dest.text
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
CROSS APPLY deqp.query_plan.nodes(N'//RelOp') RelOp (pln)
)

SELECT qp.OperatorName,
       qp.NodeId,
       qp.CPOCost,
       qp.IOCost,
       qp.CPOCost + qp.IOCost AS EstimatedCost
FROM   QueryPlans AS qp
WHERE  qp.text LIKE 'CREATE PROCEDURE Sales.uspGetDiscountRates%'
ORDER BY EstimatedCost DESC
```

Listing 8.3

This query accesses the plan cache directly by querying against the Dynamic Management Object (DMO), `sys.dm_exec_query_plan`. It joins against the XML that defines the plan so that you can pull out properties such as the `EstimatedCPU` or `EstimatedIO`.

This becomes incredibly useful for looking at large plans because it allows you to access the information programmatically rather than scrolling around on a screen. You can put together your own XQuery operation with this data in order to programmatically explore your large execution plans any way you need to.

In summary, the operations for a large-scale execution plan are no different from any other you have seen in this book; there are just more of them. Don't be intimidated by them. Just start at the top right, in the normal fashion, and work your way through in stages, using the scrolling window to navigate around, if necessary.

Parallelism in Execution Plans

SQL Server can take advantage of a server's multiple processors. It's able to take some operations and to spread the processing across the processors available to it, with the goal of dividing the work into smaller chunks so that the overall operation performs quicker. There are a couple of instance-wide settings that determine if, or when, the optimizer might generate "parallel execution plans."

- **Max degree of parallelism**, which determines the maximum number of processors that SQL Server can use when executing a parallel query; by default this is set to "0," which means that all available processors can potentially be used to execute a query.
- **Cost threshold for parallelism**, which specifies the threshold, or minimum cost, at which SQL Server creates and runs parallel plans; this cost is an estimated number of seconds in which the query will run, and the default value is "5." In other words, if the query optimizer determines that a query will take less than 5 seconds to execute, then it won't consider using parallelism.

Max degree of parallelism

SQL Server determines the optimal number of processors to execute a given parallel query (assuming that multiple processors are available). By default, SQL Server will use all available processors. 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 64.¹ Before you modify these values, you need to be sure you have a complete understanding of how parallelism benefits or damages your system and your queries. Without this knowledge, I recommend leaving parallelism on for most systems.

You can configure this option via the **sp_configure** system stored procedure, as shown in Listing 8.4.

```
sp_configure 'show advanced options', 1;
GO
RECONFIGURE WITH OVERRIDE;
GO
sp_configure 'max degree of parallelism', 3;
GO
RECONFIGURE WITH OVERRIDE;
GO
```

Listing 8.4

The first statement turns on the advanced options, necessary to access the degree of parallelism. The system is then reconfigured and we change the value of the `max degree of parallelism` setting to 3 from whatever it might have been previously, and then again reconfigure the system.

¹ In addition to these system settings, you can also affect the number of processors used by a query, by supplying the `MAXDOP` query hint, as described in Chapter 5.

Cost threshold for parallelism

The optimizer assigns costs to operations within the execution plan. These costs represent an estimation of the number of seconds each operation will take. If that estimated cost is greater than the **cost threshold for parallelism**, then that operation may be executed as a parallel operation.

The actual decision process used by the optimizer is outlined below.

- **Does the server have multiple processors?** Parallel processing requires the server to have more than one processor.
- **Are sufficient threads available?** Threads are an operating system construct that allow multiple concurrent operations, and SQL Server must check with the OS to determine if threads are available for use prior to launching a parallel process.
- **What type of query or index operation is being performed?** Queries that cost more, such as those that sort large amounts of data or do joins between large tables, and so on, lead to a higher estimated cost for the operation. It's this cost that is compared against the cost threshold.
- **Are there a sufficient number of rows to process?** The number of rows being processed directly affects the cost of each operation, which can lead to the process meeting or not meeting the cost threshold.
- **Are the statistics current?** Depending on the operation, if the statistics are not current, the optimizer may choose either to avoid parallelism, or to use a lower degree of parallelism.

When the optimizer determines that a query will benefit from parallel execution, it adds marshaling, meaning gathering, and control operators, called Exchange operators. These operators act to split the work done into multiple streams of data, pass it through the various parallel operators, and bring it all back together again.

When the optimizer creates an execution plan that uses parallelism, this plan is stored in cache *twice*: once for a plan that doesn't use parallelism and once for a plan that does. When a plan is reused, it is examined for the number of threads it used the last time. The query engine, at execution time, then determines whether that same number will be used, based on the current system load and the number of threads available.

Are parallel plans good or bad?

The one thing to remember about parallelism is that it comes at a cost. It takes processing time and power to divide an operation into various threads and marshall them back together. For long-running, processor-intensive, large-volume queries, parallelism makes a lot of sense. 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 more slowly. In other words, a query can actually run slower with a parallel execution plan than without one.

There is no hard and fast rule for determining when parallelism may be useful, or when it will be more costly. The best approach is to observe the execution times and wait states of queries that use parallelism, and where necessary, either change the system settings to increase the cost threshold, 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. If the times go down after you set the cost threshold to 3, then you're seeing a real benefit from parallel executions.

Examining a parallel execution plan

If you're performing these tests on a machine with a single processor, then you won't be able to see any parallel plans. Kalen Delaney supplied a method for simulating multiple processors in SQL Server Magazine, **InstantDoc #95497** (available only to subscribers). In the SQL Server Configuration Manager, right-click the appropriate SQL Server service and edit the startup properties. Add a property "-Pn" which represents the number of processors that you want to simulate. You must then restart the service. This simulates parallel execution plans, but it does not actually give you parallel execution on a single processor machine.

For more detail, read the article. However, I'll repeat the warning from the article: ***Never do this on a production system.***

We'll start with an aggregation query, of the sort that you might find in a data mart. If the dataset 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 >= '2003/02/01'
        AND so.ModifiedDate < DATEADD(mm, 3, '2003/02/01')
GROUP BY so.ProductID
ORDER BY so.ProductID
```

Listing 8.5

Figure 8.4 shows the estimated execution plan, which seems straightforward.

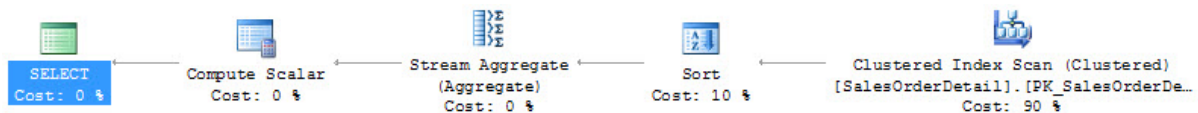


Figure 8.4

There is nothing in this plan that we haven't seen before, so we'll move on to see what would happen to this plan if it were executed with the use of multiple processors. In order to force the optimizer to use a parallel plan, we change the parallelism threshold to 1 from whatever value it is now (5, by default). Then, we can run this query and obtain a parallel execution plan.

```
EXEC sp_configure 'cost threshold for parallelism', 1;
GO
RECONFIGURE WITH OVERRIDE;
GO
SELECT  so.ProductID ,
        COUNT(*) AS Order_Count
FROM    Sales.SalesOrderDetail so
WHERE   so.ModifiedDate >= '2003/02/01'
        AND so.ModifiedDate < DATEADD(mm, 3, '2003/02/01')
GROUP BY so.ProductID
ORDER BY so.ProductID
GO
EXEC sp_configure 'cost threshold for parallelism', 5;
GO
```

Listing 8.6

Figure 8.5 shows the graphical execution.

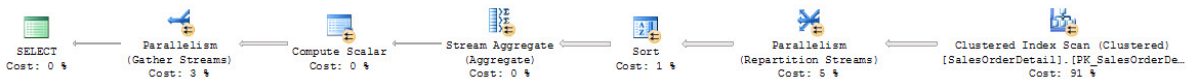


Figure 8.5

The first thing that will probably jump out at you, in addition to the new operators that support parallelism, is the small yellow icon with two arrows, attached to the otherwise familiar operators. This icon designates that the optimizer is using these operators within a parallel processing stream. If you look at the **Property** sheet (see Figure 8.6) for the **Select** operator, you can see whether the given query is using parallelism.

[-] Misc	
Cached plan size	24 B
CompileCPU	5
CompileMemory	400
CompileTime	5
Degree of Parallelism	8
Estimated Number of Rows	1
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	1.01567
Logical Operation	
Memory Grant	4160
[+] MissingIndexes	
Optimization Level	FULL
Physical Operation	
QueryHash	0x56C70718E91106D9
QueryPlanHash	0x7850768E3278B9B6
[+] Set Options	ANSI_NULLS: True, ANSI_PAD
Statement	SELECT so.ProductID ,CO

Figure 8.6

I've drawn a box around the **degree of parallelism** property so that it stands out. The value of 8, assigned to that property, indicates that the execution of this query will be split between each of the eight available processors. Looking at the graphical execution plan, we'll start from the right to look at the physical order of operations. First, we find a **Clustered Index Scan** operator.

Figure 8.7 shows the **Properties** sheet for that operator.

[-] Misc	
[-] Actual Number of Rows	2340
Thread 0	0
Thread 1	0
Thread 2	0
Thread 3	2262
Thread 4	0
Thread 5	0
Thread 6	0
Thread 7	78
Thread 8	0
[-] Actual Rebinds	0
[-] Actual Rewinds	0
[-] Defined Values	[AdventureWorks2008R2].[Sales].[SalesOrderDetail].ProductI
Description	Scanning a clustered index, entirely or only a range.
Estimated CPU Cost	0.0334014
Estimated I/O Cost	0.915718
Estimated Number of Executions	1
Estimated Number of Rows	2270.22
Estimated Operator Cost	0.949119 (91%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	19 B
Estimated Subtree Cost	0.949119
Forced Index	False
Logical Operation	Clustered Index Scan
Node ID	5
NoExpandHint	False
Number of Executions	8
[-] Object	[AdventureWorks2008R2].[Sales].[SalesOrderDetail].[PK_Sale
Ordered	False
[-] Output List	[AdventureWorks2008R2].[Sales].[SalesOrderDetail].ProductI
Parallel	True
Physical Operation	Clustered Index Scan
Predicate	[AdventureWorks2008R2].[Sales].[SalesOrderDetail].[Modifie
TableCardinality	121317

Figure 8.7

You can see in the **Properties** screen that the **Parallel** property is set to **True**, just like in the **Select** operator above. More interesting is that this operator, a **Clustered Index Scan**, was called 8 times, which you can see in the **Number of Executions** property. At the very top of the sheet, you can how the threads were split to retrieve the various rows needed for this operation, so you can see how the parallelism actually worked within this operator.

The data passes on to a **Parallelism** operator, which is the **Repartition Streams** operator. This operator is responsible for balancing the streams, trying to make sure that a roughly equal amount of work is performed by each stream. As you can see in Figure 8.7, the rows that were retrieved by the streams running in the **Clustered Index Scan** were retrieved at different rates by the various streams. This gets rebalanced by the **Repartition Streams** operator. It's also possible for this operator to reduce the number of rows, but that's not the case here. You can see this rebalancing in the operator ToolTip on display in Figure 8.8.

Look at the top of Figure 8.8, at the **Actual Number of Rows** property. Note how the threads have been better balanced out with a roughly even distribution of rows among them. The idea here is to make each thread do an equal amount of work so that no thread is waiting too long on the others. You begin to see where the added overhead of parallelism comes to play. You have to be dealing with quite large volumes of data for this to be beneficial. The amount of work that it takes to split the streams, balance the streams, and then bring them all back together can be quite costly. If only a few rows are involved, then that cost can far outweigh the benefits of putting multiple CPUs to work on the query.

The next several operators after the **Repartition Streams** operator in Figure 8.5 are quite common and we covered them in previous chapters. Each one, in this instance, is operating as a **Parallel** operator as designated by the yellow icon. Each of these operators shows how it's processing the data through a set of streams in its output listing. The next interesting operator is the final one, right before the **Select** operator, the **Parallelism Gather Streams** operator. This operator is self-explanatory. It pulls the streams back together in order to present the data gathered by the process as a single dataset to the query or operator calling it. The output from this operator is now a single thread of information.

[-] Misc	
[-] Actual Number of Rows	2340
Thread 0	0
Thread 1	354
Thread 2	259
Thread 3	308
Thread 4	224
Thread 5	294
Thread 6	247
Thread 7	368
Thread 8	286
[+] Actual Rebinds	0
[+] Actual Rewinds	0
Description	Repartition streams.
Estimated CPU Cost	0.02986
Estimated I/O Cost	0
Estimated Number of Executions	1
Estimated Number of Rows	2270.22
Estimated Operator Cost	0.056551 (5%)
Estimated Rebinds	0
Estimated Rewinds	0
Estimated Row Size	11 B
Estimated Subtree Cost	1.00567
Logical Operation	Repartition Streams
Node ID	4
Number of Executions	8
[+] Output List	[AdventureWorks2008R2].[Sales].[SalesOrderDetail]
Parallel	True
[+] Partition Columns	[AdventureWorks2008R2].[Sales].[SalesOrderDetail]
Partitioning Type	Hash
Physical Operation	Parallelism

Figure 8.8

Once you're done experimenting with parallelism, be sure to reset the parallelism threshold to where it was on your system (the default is 5).

```
sp_configure 'cost threshold for parallelism', 5;  
GO  
RECONFIGURE WITH OVERRIDE;  
GO
```

Listing 8.7

How Forced Parameterization Affects Execution Plans

The goal of using forced parameterization is to reduce recompiles as much as possible. Even when taking this more direct control over how the optimizer behaves, you have no control over the parameter name, nor can you count on the same name being used every time the execution plan is generated. The order in which parameters are created is also arbitrary. Crucially, you can't control the data types picked for parameterization, either. This means that, if the optimizer picks a particular data type that requires a CAST for comparisons to a given column, then it may as a result avoid using applicable indexes. In other words, using forced parameterization can result in suboptimal execution plans.

One example we saw in Chapter 2 was a simple DELETE statement, as shown in Listing 8.8.

```
DELETE FROM Person.EmailAddress  
WHERE BusinessEntityID = 42;
```

Listing 8.8

The search predicate in the **Clustered Index Delete** operation from this plan used a parameter instead of the hard-coded value 42, as you can see in the parameters listing of the **Properties** sheet from the **Delete** operator below.

[-] Misc	
Cached plan size	16 B
CompileCPU	1
CompileMemory	128
CompileTime	1
Degree of Parallelism	1
Estimated Number of Rows	1
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0232852
Logical Operation	
Optimization Level	TRIVIAL
[+] Parameter List	@1
Column	@1
Parameter Compiled Value	(42)
Parameter Runtime Value	(42)
Physical Operation	
QueryHash	0x6DCA8D9497F92154
QueryPlanHash	0xE400EE559CBB354D
[+] Set Options	ANSI_NULLS: True, ANSI_PADDING: On
Statement	DELETE [Person].[EmailAddress]

Figure 8.9

The optimizer performs this action in an effort to create plans that are more likely to be reused. The optimizer is only able to perform simple parameterization on relatively simple queries. The parameters created are as close to the correct data type as the optimizer can get, but since it's just estimation, it could be wrong.

The optimizer arbitrarily provides the names for these parameters as part of the process. It may or may not generate the same parameter names, in the same order, from one generation of the execution plan of the query in question to the next. As queries get more complex, the optimizer may be unable to determine whether or not a hard-coded value should be parameterized.

This is where forced parameterization comes into play. Instead of the occasional parameter replacing of a literal value, based on a simple set of rules, SQL Server attempts to replace all literal values with a parameter, with the following important exceptions:

- 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
- XQuery literals are not replaced with parameters.

Books Online details a very long list of other explicit exceptions.

With all these exceptions, why would you want to use forced parameterization? An application developed using stored procedures, with good parameters of appropriate data types, is very unlikely to benefit from forced parameterization. However, a system developed with most of the T-SQL being ad hoc or client-generated may contain nothing but hard-coded values. This type of system could benefit greatly from forced parameterization. As with any other attempts to force control out of the hands of the optimizer and the query engine, testing is necessary.

Normally, forced parameterization is set at the database level. You also have the option of choosing to set it on for a single query using the query hint, **PARAMETERIZATION FORCED**, but this hint is only available as a plan guide, which we cover in the next section.

In this example, we have several literals used as part of the query, which is a search to find email addresses that start with the literal, "david".

```
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
```

Chapter 8: Advanced Topics

```
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';
```

Listing 8.9

Run the query, and then let's examine the actual execution plan, shown in Figure 8.10. You can see the query in the **Select** operator is identical to what was written. In other words, no parameterization occurred.

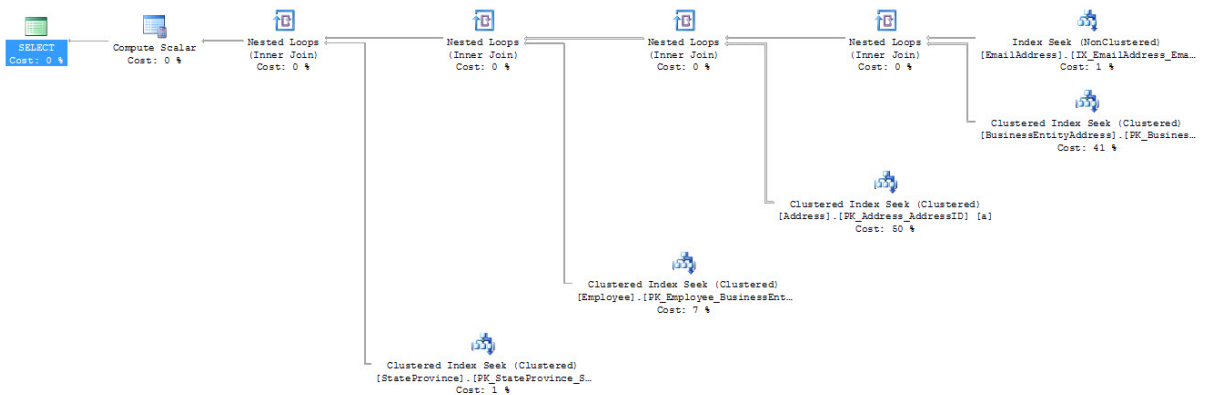


Figure 8.10

Let's now enable forced parameterization and clean out the buffer cache, so that we're sure to see a new execution plan.

```
ALTER DATABASE AdventureWorks2008R2
SET PARAMETERIZATION FORCED
GO
DBCC freeproccache
GO
```

Listing 8.10

If you rerun Listing 8.9, you'll see that the execution plan is the same as that shown in Figure 8.10. However, the query stored in the **Select** operator is not the same. It now looks as shown in Listing 8.11.

```
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 = @0
```

Listing 8.11

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. This could seriously affect performance, either positively or negatively. However, this does increase the likelihood that, if this query is called again with a different two- or three-character state code, the plan will be reused.

Before proceeding, be sure to reset the parameterization of the AdventureWorks2008R2 database.

```
ALTER DATABASE AdventureWorks2008R2
SET PARAMETERIZATION SIMPLE
GO
```

Listing 8.12

Using Plan Guides to Modify Execution Plans

Through most of the work we've been doing so far, if we wanted to change the behavior of a query, we could edit the T-SQL code, add or modify an index, add some hints to the query, or all of the above.

What do you do, however, when you're dealing with a third-party application where you cannot edit the T-SQL code, or where the structure and indexes are not under your control? This is where **plan guides** come in handy. Plan guides are simply a way of applying valid query hints to a query without actually editing the T-SQL code in any way. Plan guides can be created for stored procedures and other database objects, or for SQL statements that are not part of a database object.

The same caveat that applies to query hints obviously has to apply here: exercise due caution when implementing plan guides, because changing how the optimizer deals with a query can seriously impact its performance in a negative way if they are used incorrectly.

You create a plan guide by executing the procedure, `sp_create_plan_guide`. There are three available types of plan guide.

- **Object** plan guides – Applied to a stored procedure, function or DML trigger.
- **SQL** plan guides – Applied to strings in T-SQL statements and batches, which are outside the context of a database object.
- **Template** plan guides – Used specifically to control *how* a query is parameterized.

Object plan guides

Let's assume for a moment that we've noticed that the `AdventureWorks` procedure, `dbo.uspGetManagerEmployees`, is generating poor plans part of the time. Testing has led you to the conclusion that, ideally, you need to add a `RECOMPILE` hint to the stored procedure in order to get the best possible execution plan most of the time. However, this isn't a procedure you can edit. So, you decide to create a plan guide that will apply the recompile hint without editing the stored procedure.

Let's look at the plan guide and then I'll describe it in detail (I've modified the code for readability within the book, but if you do that with your actual code, it will prevent the guide from being used).

```
EXEC sp_create_plan_guide @name = N'MyFirstPlanGuide',
    @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 8.13

First, we use the `@name` parameter to give our plan guide a name, in this case `MyFirstPlanGuide`. Note that plan guide names operate within the context of the database, not the server.

The `@stmt` parameter has to be an exact match to the query that the query optimizer will be called on to match. White space and carriage returns don't matter but, in order to create the above, I had to include the CTE. Without it, I was getting errors. When the optimizer finds code that matches, it will look up and apply the correct plan guide.

The `@type` parameter is going to be a database object, so this is an object plan guide.

In the `@module_or_batch` parameter, we specify the name of the target object if we're creating an object plan guide, as in this case. We supply null otherwise.

We use `@params` only if we're using a template plan guide and forced parameterization. Since we're not, it's null in this case. If we were creating a template, this would be a comma-separated list of parameter names and data types.

Finally, the `@hints` parameter specifies any hints that need to be applied. 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, like in this case, we need the existing `OPTION` to be carried forward, we need to add it to the plan guide.

From this point forward, without making a single change to the actual definition of the stored procedure, each execution of this procedure will be followed by a recompile. You can identify that a guide has been used by looking at the **Select** operator of the resulting execution plan. If we were to execute the query as shown in Listing 8.14, it would result in the information shown in Figure 8.11 being on display in the execution plan's **Select** operator.

```
EXEC dbo.uspGetManagerEmployees @BusinessEntityID = 42 -- int
```

Listing 8.14

Misc	
Cached plan size	88 B
CompileCPU	117
CompileMemory	1032
CompileTime	127
Degree of Parallelism	1
Estimated Number of Rows	2063.01
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	1.29955
Logical Operation	
Memory Grant	11560
Optimization Level	FULL
Parameter List	@BusinessEntityID
Physical Operation	
PlanGuideDB	AdventureWorks2008R2
PlanGuideName	MyFirstPlanGuide
QueryHash	0x98ECA9AEF90F51BD
QueryPlanHash	0xCCDAF41E4260C267
Set Options	ANSI_NULLS: True, ANSI_PADDING:
Statement	WITH [EMP_cte]([BusinessEntityID],

Figure 8.11

So, you can see if a plan guide was applied to a stored procedure.

SQL plan guides

Let's look at another example. Let's assume that we have an application that submits primarily ad hoc T-SQL to a database. Once again, we're concerned about performance, and we've found that if only we can apply an `OPTIMIZE FOR` hint to the query, we'll get the execution plan that we want to see.

The simple query in Listing 8.15, where we look up addresses based on a city, should be familiar from Chapter 5.

```
SELECT *
FROM   Person.Address
WHERE  City = 'LONDON';
```

Listing 8.15

From Listing 5.23, we already know that we can improve the performance of the above query by applying the `(OPTIMIZE FOR (@City = 'Mentor'))` query hint, so let's enforce that behavior via a SQL plan guide.

In order to be sure of the formatting of the query with parameters as the optimizer will see it, you'll need to run the query through `sp_get_query_template`. This system procedure generates parameterized query output that we can use to verify that what we've done is the same as how the query will look when it has been parameterized by the system.

```
EXEC sp_create_plan_guide @name = N'MySecondPlanGuide',
    @stmt = N'SELECT * FROM Person.Address WHERE City
            = @0', @type = N'SQL', @module_or_batch = NULL,
    @params = N'@0 VARCHAR(8000)',
    @hints = N'OPTION(OPTIMIZE FOR (@0 = ''Mentor''))'
```

Listing 8.16

This returns two strings:

```
select * from Person . Address where City = @0
```

and

```
@0 varchar(8000)
```

You can see where we used these in the query above.

Now, when we run the query, with the plan guide created and enforced by the query engine, we get the execution plan we want, as shown in Figure 8.12.

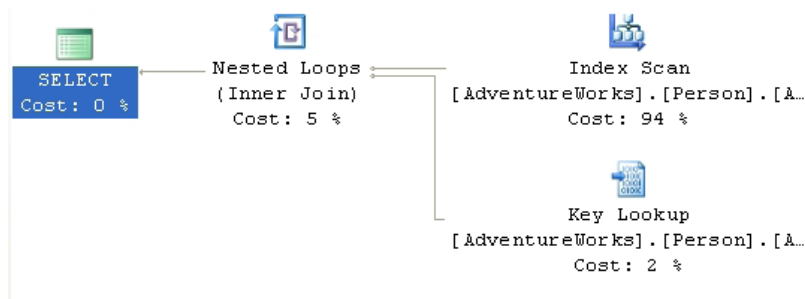


Figure 8.12

Template plan guides

As a final example, consider the query we used previously to demonstrate forced parameterization. If we determine that a procedure we cannot edit must have its `PARAMETERIZATION` set to `FORCED`, we can simply create a **template plan guide** rather than changing the settings on the entire database. A template plan guide will override parameterization settings in queries.

To get started, you have to have the query for which you're going to create a template. You will need to use `sp_get_query_template` again, in order to see the structure of the query, when parameterized. This will show you where SQL Server places the parameters.

You can then create the template guide, as shown in Listing 8.17.

```
EXEC sp_create_plan_guide @name = N'MyThirdPlanGuide',
    @stmt = 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' ;', @type = N'TEMPLATE',
    @module_or_batch = NULL, @params = N'@0 VARCHAR(8000)',
    @hints = N'OPTION(PARAMETERIZATION FORCED)'
```

Listing 8.17

Plan guide administration

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 8.18

Aside from the procedure to create plan guides, a second one, `sp_control_plan_guide`, allows you to drop, disable, 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.

```
EXEC sp_control_plan_guide @operation = N'DROP',  
                           @name = N'MyFourthPlanGuide'
```

Listing 8.19

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 store it "on the side," for example, inside a plan guide, and then to use that plan on the 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 then whatever plan the optimizer produces is not used. Instead, it uses the plan you're "forcing."

You cannot force a plan on:

- INSERT, UPDATE or DELETE queries
- queries that use cursors other than `static` and `fast_forward`
- distributed queries and full text queries.

Forcing a plan, just like all the other possible query hints, can result in poor performance. Proper testing and due diligence must be observed prior to applying `USE PLAN`.

Plan forcing can come in very handy if you have a poorly performing query and T-SQL code that you can't modify. This is frequently the case when working with third-party software. Plan forcing gives you the opportunity to get a preferred plan to work on the query. This can be a huge benefit for performance but, as code, structures, or the data changes, the "forced" plan may become suboptimal, hurting performance, or even inapplicable, at which point the optimizer can ignore the plan. As with hints, plan forcing should be a last resort, not a standard tactic.

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 8k in size, then SQL Server can no longer cache the query, because it exceeds the 8k 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. Further, you avoid having to deploy and redeploy the query to your production system, if you want to add or remove a plan.

Following is an example of a simple query, encapsulated within a stored procedure, for reporting some information from the `SalesOrderHeader` table.

```
ALTER PROCEDURE Sales.uspGetCreditInfo ( @SalesPersonID INT )
AS
    SELECT  soh.AccountNumber ,
            soh.CreditCardApprovalCode ,
            soh.CreditCardID ,
            soh.OnlineOrderFlag
    FROM    Sales.SalesOrderHeader AS soh
    WHERE   soh.SalesPersonID = @SalesPersonId;
```

Listing 8.20

When the procedure is run using the value for `@SalesPersonID = 277`, a **Clustered Index Scan** results, and the plan is quite costly.

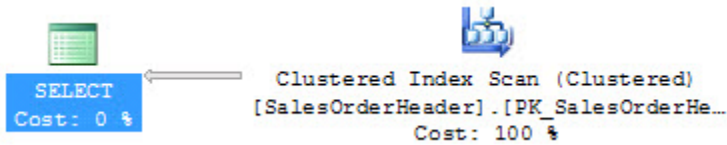


Figure 8.13

If we change the value to 288, we see an **Index Seek** with a **Bookmark Lookup**.

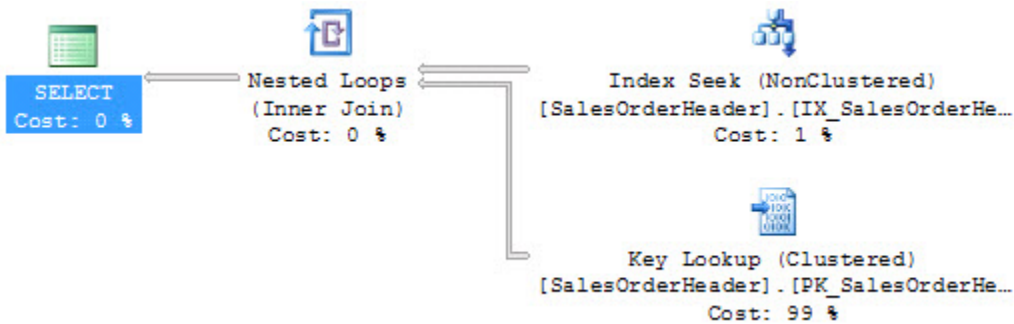


Figure 8.14

This is much faster than the **Clustered Index Scan**. If the execution plan for the procedure takes the first value for its plan, then the later values still use the **Clustered Index Scan**. While we could simply add a plan guide that uses the `OPTIMIZE FOR` hint, we're going to try `USE PLAN` instead.

First, we need to create an XML plan that behaves the way we want. We do this by taking the `SELECT` criteria out of the stored procedure and modifying it to behave the correct way. This results in the correct plan. In order to capture this plan, we'll wrap it with `STATISTICS XML`, which will generate an actual execution plan in XML.

```
SET STATISTICS XML ON
GO
SELECT  soh.AccountNumber ,
        soh.CreditCardApprovalCode ,
        soh.CreditCardID ,
        soh.OnlineOrderFlag
FROM    Sales.SalesOrderHeader AS soh
WHERE   soh.SalesPersonID = 288;
GO
SET STATISTICS XML OFF
GO
```

Listing 8.21

This simple query generates a 107-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.

```
EXEC sp_create_plan_guide
    @name = N'UsePlanPlanGuide',
    @stmt = N'SELECT  soh.AccountNumber
              ,soh.CreditCardApprovalCode
              ,soh.CreditCardID
              ,soh.OnlineOrderFlag]
FROM    Sales.SalesOrderHeader soh
WHERE   soh.SalesPersonID = @SalesPersonID --288 --277',
    @type = N'OBJECT',
    @module_or_batch = N'Sales.uspGetCreditInfo',
    @params = NULL,
    @hints = N'OPTION(USE PLAN N''<ShowPlanXML...
```

Listing 8.22

Now, when we execute the query using the values that generate a bad plan...

```
EXEC [Sales].uspGetCreditInfo @SalesPersonID = 277
```

Listing 8.23

...we still get the execution plan we want, as shown in Figure 8.15.

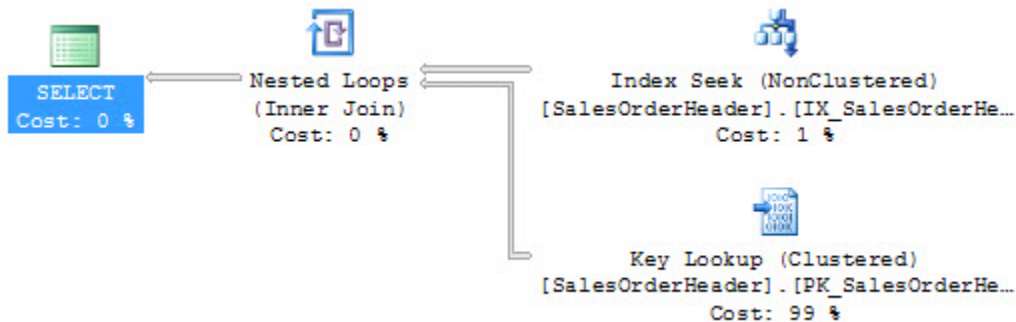


Figure 8.15

As a final reminder: using a plan guide, especially one that involves `USE PLAN`, 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.

Summary

All the methods outlined in this final chapter of the book come with some degree of risk. You can manage large execution plans using XQuery, but running large-scale, processor- and memory-intensive queries against a production server carries as many risks as it does benefits. Using plan guides and plan forcing, you can take direct control of 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 the methods outlined in this chapter. Take your time and test everything you do to your systems. Use the information you've gleaned from the other chapters in the book, in order to be sure the choices you're making are the right ones.

About Red Gate

You know those annoying jobs that spoil your day whenever they come up?

Writing out scripts to update your production database, or trawling through code to see why it's running so slow.

Red Gate makes tools to fix those problems for you. Many of our tools are now industry standards. In fact, at the last count, we had over 650,000 users.

But we try to go beyond that. We want to support you and the rest of the SQL Server and .NET communities in any way we can.

First, we publish a library of free books on .NET and SQL Server. You're reading one of them now. You can get dozens more from www.red-gate.com/books

Second, we commission and edit rigorously accurate articles from experts on the front line of application and database development. We publish them in our online journal **Simple Talk**, which is read by millions of technology professionals each year.



On **SQL Server Central**, we host the largest SQL Server community in the world. As well as lively forums, it puts out a daily dose of distilled SQL Server know-how through its newsletter, which now has nearly a million subscribers (and counting).

Third, we organize and sponsor events (about 50,000 of you came to them last year), including **SQL in the City**, a free event for SQL Server users in the US and Europe.

So, if you want more free books and articles, or to get sponsorship, or to try some tools that make your life easier, then head over to www.red-gate.com