

High Performance SQL Server



Complete Showplan Operators

Fabiano Amorim



ISBN: 978-1-906434-71-7

Complete Showplan Operators

By Fabiano Amorim

First published by Simple Talk Publishing June 2011

Copyright Fabiano Amorim 2011

ISBN 978-1-906434-71-7

The right of Fabiano Amorim 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.

Typeset by Gower Associates.

Table of Contents

About the author	6
Preface	6
Chapter 1: Assert	8
Assert and check constraints	8
Assert checking foreign keys.....	10
Assert checking a subquery.....	12
Chapter 2: Concatenation	14
Chapter 3: Compute Scalar	18
Chapter 4: BookMark/Key Lookup	27
Chapter 5: Spools – Eager Spool.....	33
Spool operators.....	33
Eager Spool.....	34
The Halloween Problem	35
Chapter 6: Spools – Lazy Spool	44
Chapter 7: Spools – Non-Clustered Index Spool	53
Understanding rebind and rewind	58
Rebinds and rewinds with Table Spool (Lazy Spool)	59
Rebinds and rewinds with Index Spool (Lazy Spool).....	63
Summary.....	66
Chapter 8: Spools – Row Count Spool	67
Chapter 9: Stream Aggregate	73
Scalar aggregations.....	75
Group Aggregations	78
A myth is born.....	80

Chapter 10: SORT.....	83
SORT into execution plans.....	83
SORT in memory/disk.....	85
How to avoid SORT operations.....	86
Chapter 11: Merges – Merge Join	88
Introduction.....	88
SORT Merge Join.....	92
Residual predicate	95
One to Many and Many to Many Merge Join	96
Chapter 12: Merges – Merge Interval	99
Creating sample data	99
Merge Interval.....	101
Finally	107
Chapter 13: Split, Sort, Collapse	108
Introduction.....	108
Unique Index.....	108
Creating sample data	109
Querying a Unique Index.....	110
Trivial plan.....	112
Full Optimization.....	118
More about querying a Unique Index.....	119
Non-Unique Index and updates.....	121
Unique Index and updates.....	126
And finally.....	131

About the author

Fascinated by the SQL Server Query Processor and the way it works to Optimize queries, procedures and functions, Fabiano is a Data Platform Architect at SolidQ Brazil, and graduated as a Technical Processor from Colégio Bezerra de Menezes, SP – Brazil. He has also worked for several years with SQL Server, focusing on SQL Server Development and BI Projects for many companies in Brazil and Argentina. Fabiano is an MCP for SQL Server 2000, MCTS and MCITP Data Base Developer for SQL Server 2005 and 2008. He is also actively involved in the SQL Server community through forums such as MSDN and TechNet Brazil, and he writes articles for Simple-Talk and SQL Server Magazine, Brazil, and presents online webcasts and in-person events for Microsoft Brazil. His blog is at [HTTP://FABIANOSQLSERVER.SPACES.LIVE.COM/](http://fabianosqlserver.spaces.live.com/), and you can follow him on Twitter as [@MCFlyAmorim](https://twitter.com/MCFlyAmorim).

Preface

Writing good TSQL code is not an easy task. Then you submit the code to the query optimizer and strange things happen. The one good view you have into what the optimizer decided to do is provided by the execution plans. Understanding execution plans is a lot of work. Trust me on that. What you need to really understand your queries is as much knowledge as you can get. That's where this excellent collection of articles on some of the more common execution plan operators comes in.

Fabiano Amorim has taken the time to really drill into the behavior of a small set of execution plan operators in an effort to explain the optimizer's behavior. He's explored why things happen, how you can change them, positively or negatively, and he's done it

all in an approachable style. You want information and knowledge in order to achieve understanding.

When I wrote my book on execution plans, I really did try to focus on the plan as a whole. So while I spent time talking about individual operators, what they did, and why they did it, I was frequently not as interested in discussing everything that an individual operator might do once I had established their role in a given plan. Having someone like Fabiano come along and go the opposite route, sort of ignoring the whole plan in an effort to spend time exploring the operator, acts to fill in gaps. Where I tried to teach how to read an execution plan, Fabiano is trying to teach what a given operator does. It's all worthwhile and it all accumulates to give you more knowledge.

Time to stop listening to me blather, turn the page, and start learning from Fabiano.

Grant Fritchey

Chapter 1: Assert

Showplan operators are used by the Query Optimizer (QO) to build the query plan in order to perform a specified operation. A query plan will consist of many physical operators. The Query Optimizer uses a simple language that represents each physical operation by an operator, and each operator is represented in the graphical execution plan by an icon.

I'm going to mention only of those that are more common: the first being the **Assert**.

The **Assert** is used to verify a certain condition, it validates a Constraint on every row to ensure that the condition was met. If, for example, our DDL includes a check constraint which specifies only two valid values for a column, the **Assert** will, for every row, validate the value passed to the column to ensure that input is consistent with the check constraint.

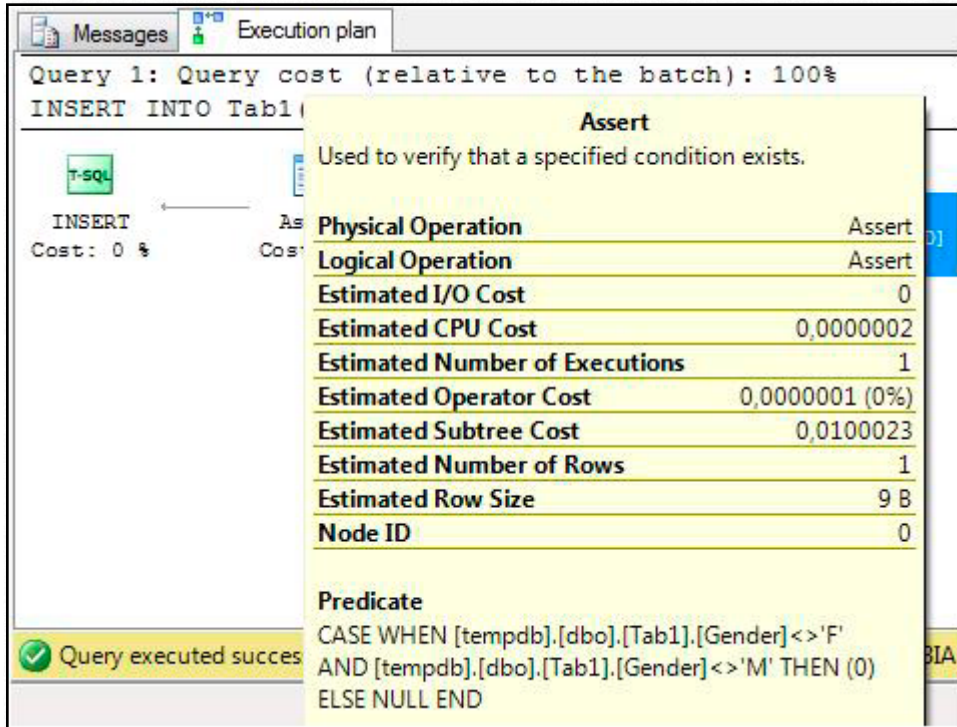
Assert and check constraints

Let's see where the SQL Server uses that information in practice. Take the following T-SQL:

```
IF OBJECT_ID('Tab1') IS NOT NULL
    DROP TABLE Tab1
GO
CREATE TABLE Tab1(ID Integer, Gender CHAR(1))
GO
ALTER TABLE TAB1 ADD CONSTRAINT ck_Gender_M_F CHECK(Gender IN('M','F'))
GO
INSERT INTO Tab1(ID, Gender) VALUES(1, 'X')
GO
```

Chapter 1: Assert

To the command above, the SQL Server has generated the following execution plan:



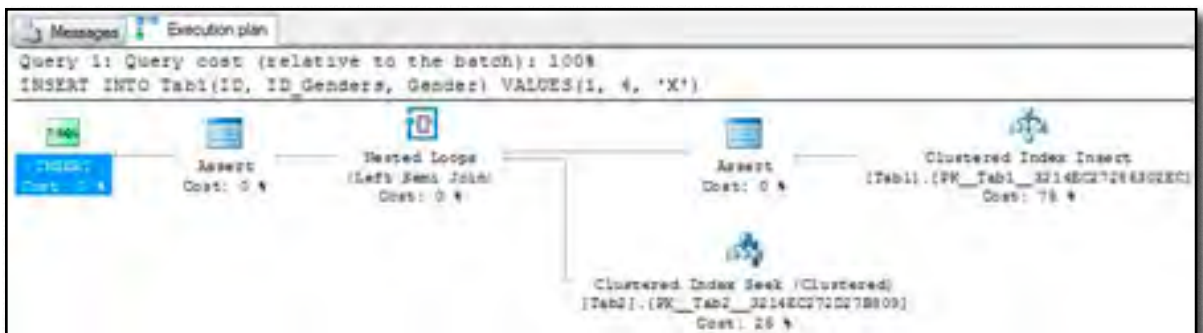
As we can see, the execution plan uses the **Assert** operator to check that the inserted value doesn't violate the Check Constraint. In this specific case, the **Assert** applies the rule, "if the value is different to 'F' and different to 'M' then return 0 otherwise return NULL."

The **Assert** operator is programmed to show an error if the returned value is not NULL; in other words, the returned value is not a "M" or "F".

Assert checking foreign keys

Now let's take a look at an example where the **Assert** is used to validate a foreign key constraint. Suppose we have this query:

```
ALTER TABLE Tab1 ADD ID_Genders INT
GO
IF OBJECT_ID('Tab2') IS NOT NULL
    DROP TABLE Tab2
GO
CREATE TABLE Tab2(ID Integer PRIMARY KEY, Gender CHAR(1))
GO
INSERT INTO Tab2(ID, Gender) VALUES(1, 'F')
INSERT INTO Tab2(ID, Gender) VALUES(2, 'M')
INSERT INTO Tab2(ID, Gender) VALUES(3, 'N')
GO
ALTER TABLE Tab1 ADD CONSTRAINT fk_Tab2 FOREIGN KEY (ID_Genders) REFERENCES
Tab2 (ID)
GO
INSERT INTO Tab1(ID, ID_Genders, Gender) VALUES(1, 4, 'X') 4
```



Let's look at the text execution plan to see what these **Assert** operators were doing. To see the text execution plan just execute `SET SHOWPLAN_TEXT ON` before run the insert command.

```
--Assert(WHERE:(CASE WHEN NOT [Pass1008] AND [Expr1007] IS NULL THEN (0) ELSE NULL
END))
  |--Nested Loops(Left Semi Join, PASSTHRU:([Tab1].[ID_Genders] IS NULL), OUTER
REFERENCES:([Tab1].[ID_Genders]), DEFINE:([Expr1007] = [PROBE VALUE]))
    |--Assert(WHERE:(CASE WHEN [Tab1].[Gender]<>'F' AND [Tab1].[Gender]<>'M'
THEN (0) ELSE NULL END))
      |--Clustered Index Insert(OBJECT:([Tab1].[PK]), SET:([Tab1].[ID] =
RaiseIfNullInsert(@1),[Tab1].[ID_Genders] = @2,[Tab1].[Gender] = [Expr1003]),
DEFINE:([Expr1003]=CONVERT_IMPLICIT(char(1),@3),0))
        |--Clustered Index Seek(OBJECT:([Tab2].[PK]), SEEK:([Tab2].[ID]=[Tab1].
[ID_Genders]) ORDERED FORWARD)
```

Here we can see the **Assert** operator twice, first (looking down to up in the text plan and the right to left in the graphical plan) validating the Check Constraint. The same concept showed above is used, if the exit value is "0" than keep running the query, but if NULL is returned shows an exception.

The second **Assert** is validating the result of the **Tab1** and **Tab2** join. It is interesting to see the "[**Expr1007**] IS NULL". To understand that you need to know what this **Expr1007** is, look at the Probe Value (green text) in the text plan and you will see that it is the result of the join. If the value passed to the INSERT at the column **ID_Gender** exists in the table **Tab2**, then that probe will return the join value; otherwise it will return NULL. So the **Assert** is checking the value of the search at the **Tab2**; if the value that is passed to the INSERT is not found then **Assert** will show one exception.

If the value passed to the column **ID_Genders** is NULL than the SQL can't show a exception, in that case it returns "0" and keeps running the query.

If you run the **INSERT** above, the SQL will show an exception because of the "X" value, but if you change the "X" to "F" and run again, it will show an exception because of the value "4". If you change the value "4" to NULL, 1, 2 or 3 the insert will be executed without any error.

Assert checking a subquery

The **Assert** operator is also used to check one subquery. As we know, one scalar subquery can't validly return more than one value. Sometimes, however, a mistake happens, and a subquery attempts to return more than one value. Here the **Assert** comes into play by validating the condition that a scalar subquery returns just one value.

Take the following query:

```
INSERT INTO Tab1(ID_TipoSexo, Sexo) VALUES((SELECT ID_TipoSexo FROM Tab1), 'F')
INSERT INTO Tab1(ID_TipoSexo, Sexo) VALUES((SELECT ID_TipoSexo FROM Tab1), 'F')
|--Assert(WHERE:(CASE WHEN NOT [Pass1016] AND [Expr1015] IS NULL THEN (0) ELSE NULL
END))
|--Nested Loops(Left Semi Join, PASSTHRU:([tempdb].[dbo].[Tab1].[ID_
TipoSexo] IS NULL), OUTER REFERENCES:([tempdb].[dbo].[Tab1].[ID_TipoSexo]),
DEFINE:([Expr1015] = [PROBE VALUE]))
|--Assert(WHERE:([Expr1017]))
|--Compute Scalar(DEFINE:([Expr1017]=CASE WHEN [tempdb].[dbo].
[Tab1].[Sexo]<>'F' AND [tempdb].[dbo].[Tab1].[Sexo]<>'M' THEN (0) ELSE NULL END))
|--Clustered Index Insert(OBJECT:([tempdb].[dbo].[Tab1].
[PK__Tab1__3214EC277097A3C8]), SET:([tempdb].[dbo].[Tab1].[ID_TipoSexo] =
[Expr1008],[tempdb].[dbo].[Tab1].[Sexo] = [Expr1009],[tempdb].[dbo].[Tab1].[ID] =
[Expr1003]))
|--Top(TOP EXPRESSION:((1)))
|--Compute Scalar(DEFINE:([Expr1008]=[Expr1014],
[Expr1009]='F'))
|--Nested Loops(Left Outer Join)
|--Compute Scalar(DEFINE:([Expr1003]=geti
dentity((1856985942),(2),NULL)))
|--Constant Scan
|--Assert(WHERE:(CASE WHEN [Expr1013]>(1)
THEN (0) ELSE NULL END))
|--Stream Aggregate(DEFINE:([Expr101
3]=Count(*), [Expr1014]=ANY([tempdb].[dbo].[Tab1].[ID_TipoSexo])))
|--Clustered Index
Scan(OBJECT:([tempdb].[dbo].[Tab1].[PK__Tab1__3214EC277097A3C8]))
|--Clustered Index Seek(OBJECT:([tempdb].[dbo].[Tab2].[PK__
Tab2__3214EC27755C58E5]), SEEK:([tempdb].[dbo].[Tab2].[ID]=[tempdb].[dbo].[Tab1].
[ID_TipoSexo]) ORDERED FORWARD)
```


You can see from this text Showplan that SQL Server as generated a Stream Aggregate to count how many rows the SubQuery will return, This value is then passed to the **Assert** which then does its job by checking its validity.

It's very interesting to see that the Query Optimizer is smart enough be able to avoid using **assert** operators when they are not necessary. For instance:

```
INSERT INTO Tab1(ID_TipoSexo, Sexo) VALUES((SELECT ID_TipoSexo FROM Tab1 WHERE ID = 1), 'F')

INSERT INTO Tab1(ID_TipoSexo, Sexo) VALUES((SELECT TOP 1 ID_TipoSexo FROM Tab1), 'F')
```

For both these INSERTs, the Query Optimizer is smart enough to know that only one row will ever be returned, so there is no need to use the **Assert**.

Chapter 2: Concatenation

Showplan operators are used by SQL Server's Query Optimizer (QO) to perform a particular operation within a query plan. A query plan will usually contain several of these physical operators. Each physical operation is represented in the Query Plan by an operator, and each operator is shown in the graphical execution plan by an icon. In this chapter, we'll be featuring the Concatenation Showplan operator. Its behavior is quite simple; it receives one or more input streams and returns all the rows from each input stream in turn. We can see its effect whenever we use the Transact-SQL **UNION ALL** command.

Concatenation is a classic operator that can receive more than one input. It is both a logical and a physical operator.

Before we start to talk about concatenation, we need to understand some important points about Showplan operators and execution plans.

All operators used in execution plans, implement three methods called **Init()**, **GetNext()** and **Close()**. Some operators can receive more than one input, so, these inputs will be processed at the **Init()** method. The concatenation is one example of these operators.

At the **Init()** method, the concatenation will initialize itself and set up any required data structures. After that, it will run the **GetNext()** method to read the first or the subsequent row of the input data, it runs this method until it has read all rows from the input data.

Let's take the following query as a sample:

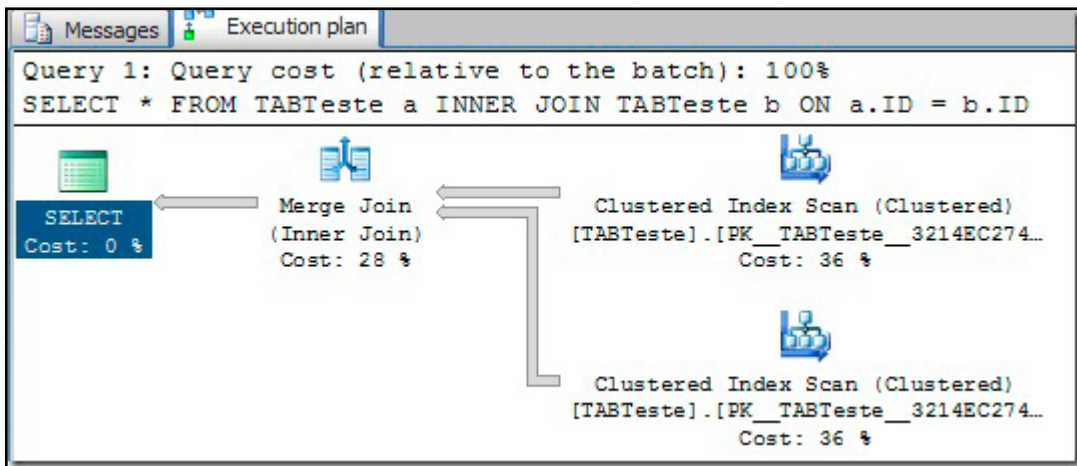
The following script will create a table **TabTeste** and populate with some garbage data.

Chapter 2: Concatenation

```
USE tempdb
GO
CREATE TABLE TABTeste(ID Int Identity(1,1) PRIMARY KEY,
                       Nome VarChar(250) DEFAULT NewID())
GO
SET NOCOUNT ON
GO
INSERT INTO TABTeste DEFAULT VALUES
GO 10000
```

The script above will populate 10000 rows at the **TABTeste** table. Now let's run one query sample to look at the execution plan.

```
SELECT *
FROM TABTeste a
INNER JOIN TABTeste b
ON a.ID = b.ID
```



Graphical execution plan.

Chapter 2: Concatenation

```
SELECT * FROM TABTeste a INNER JOIN TABTeste b ON a.ID = b.ID
|--Merge Join(Inner Join, MERGE:([b].[ID])=([a].[ID]), RESIDUAL:([TABTeste].[ID]
as [b].[ID]=[TABTeste].[ID] as [a].[ID]))

|--Clustered Index Scan(OBJECT:([TABTeste].[PK_] AS [b]), ORDERED FORWARD)

|--Clustered Index Scan(OBJECT:([TABTeste].[PK_] AS [a]), ORDERED FORWARD)
```

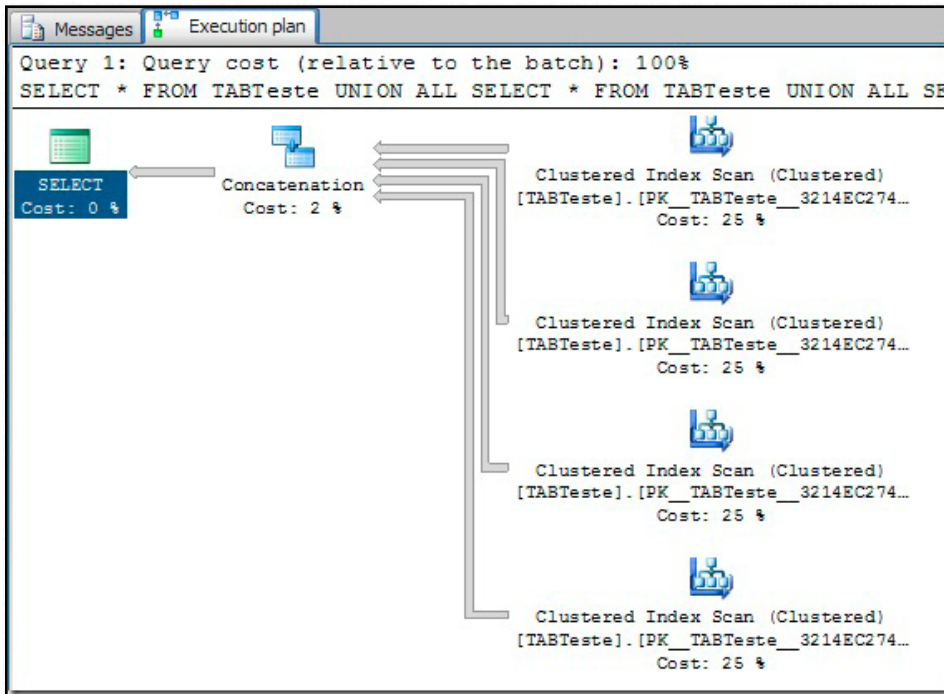
Text execution plan.

As we can see, this query is using one operator called **Merge** to join the tables, in the plan, the **Merge** operator is receiving two inputs (the table **TabTeste** twice).

Concatenation is a good example of an operator that receives more than one input. If, for example, we run the following query, we will see that it receives four inputs.

```
SELECT * FROM TABTeste
UNION ALL
SELECT * FROM TABTeste
UNION ALL
SELECT * FROM TABTeste
UNION ALL
SELECT * FROM TABTeste
```

Chapter 2: Concatenation



Graphical execution plan.

```
--Concatenation
|--Clustered Index Scan(OBJECT:([TABTeste].[PK_]
|--Clustered Index Scan(OBJECT:([TABTeste].[PK_]
|--Clustered Index Scan(OBJECT:([TABTeste].[PK_]
|--Clustered Index Scan(OBJECT:([TABTeste].[PK_]

```

Text execution plan.

The concatenation operator receives the result of all **clustered index scan** and copies all the rows to one output calling the methods **Init()** and **GetNext()**. These methods are called to each Input.

The Query Processor will execute this plan in the order that the operators appear in the plan, the first is the top one and the last is the end one.

Chapter 3: Compute Scalar

The previous two chapters covered two of the most important Showplan operators, Concatenation and Assert. It is useful to know about such Showplan operators if you are programming in SQL Server, because they are used by SQL Server's Query Optimizer (QO) to perform a particular operation within a query plan. Each physical operation in the Query Plan is performed by an operator. When you look at a graphical execution plan, you will see each operator represented by an icon. This chapter covers the **Compute Scalar** Showplan operator. This operator is very common, and we can see it in many execution plans.

As is obvious from its name, **Compute Scalar** performs a scalar computation and returns a computed value. This calculation can be as simple as a conversion of value, or a concatenation of values.

Most of the time, it is ignored by SQL users because it represents a minimal cost when compared to the cost of the entire execution plan, but, it can become well-worth looking at when we are dealing with cursors and some huge loops, and especially if you are having a CPU problem.

To start with, let's take a simple use of **Compute Scalar**. One simple conversion of data from **Int** to **Char** can be done without much problem but, if we execute this conversion one million times, it becomes a different matter. If we change the query so as to not execute this conversion step, we will have an optimization in CPU use, and a consequential improvement in the speed of execution.

Let's take the following query as a sample. The following script will create a table, **TabTeste**, and populate with some garbage data.

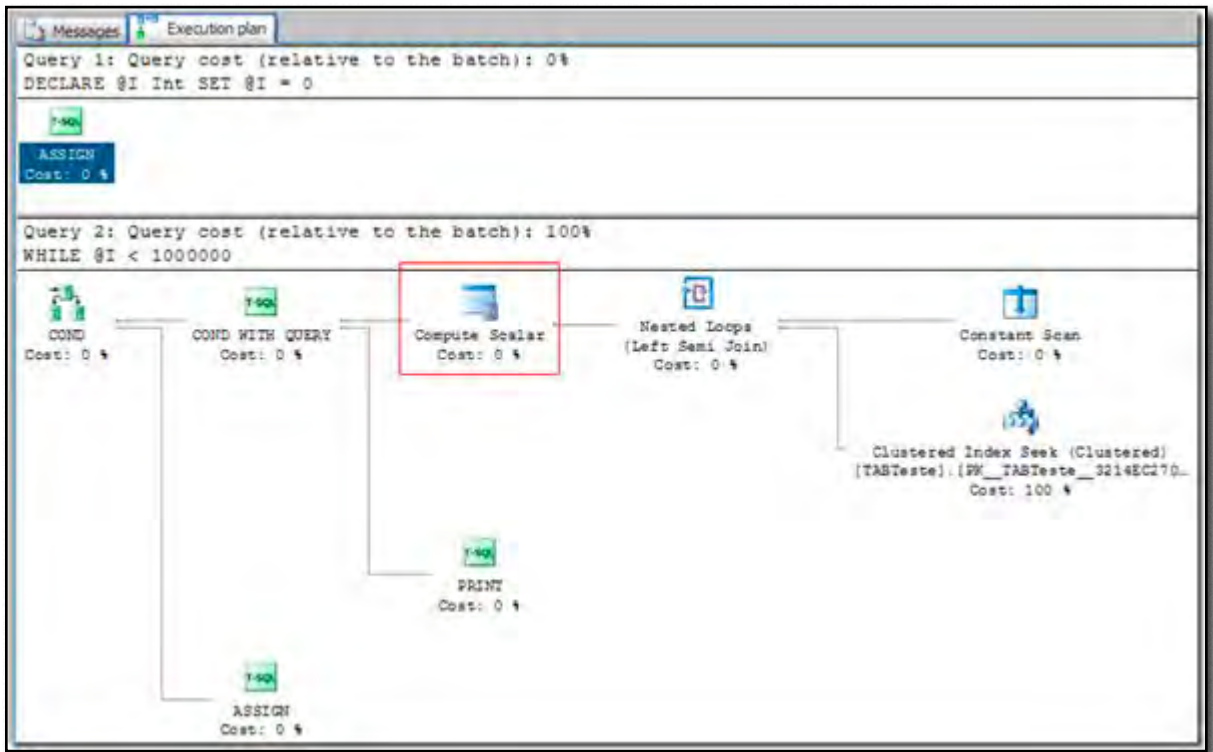
Chapter 3: Compute Scalar

```
USE tempdb
GO
CREATE TABLE TABTeste(ID    Int Identity(1,1) PRIMARY KEY,
                       Nome  VarChar(250)      DEFAULT NewID())
GO
SET NOCOUNT ON
GO
INSERT INTO TABTeste DEFAULT VALUES
GO 10000
```

Now, the code below will pass to the loop one million times.

```
DECLARE @I Int
SET @I = 0
WHILE @I < 1000000
BEGIN
    IF EXISTS(SELECT ID FROM TABTeste WHERE ID = @I)
    BEGIN
        PRINT 'Entrou no IF'
    END
    SET @I = @I + 1;
END
GO
```

Chapter 3: Compute Scalar



Graphical execution plan.

As we can see, the operator **Compute Scalar** is used; let's take a look at the text execution plan to see more details about that operation.

```
--Compute Scalar(DEFINE:([Expr1003]=CASE WHEN [Expr1004] THEN (1) ELSE (0) END))
  |--Nested Loops(Left Semi Join, DEFINE:([Expr1004] = [PROBE VALUE]))
    |--Constant Scan
    |--Clustered Index Seek(OBJECT:([tempdb].[dbo].[TABTeste].[PK__
TABTeste__3214EC27096F09E1]), SEEK:([tempdb].[dbo].[TABTeste].[ID]=[@I]) ORDERED
FORWARD)
```

Text execution plan.

Chapter 3: Compute Scalar

This plan is using the Compute Scalar to check if the Nested Loop returns any rows, on the other words; it is doing the IF EXISTS Job.

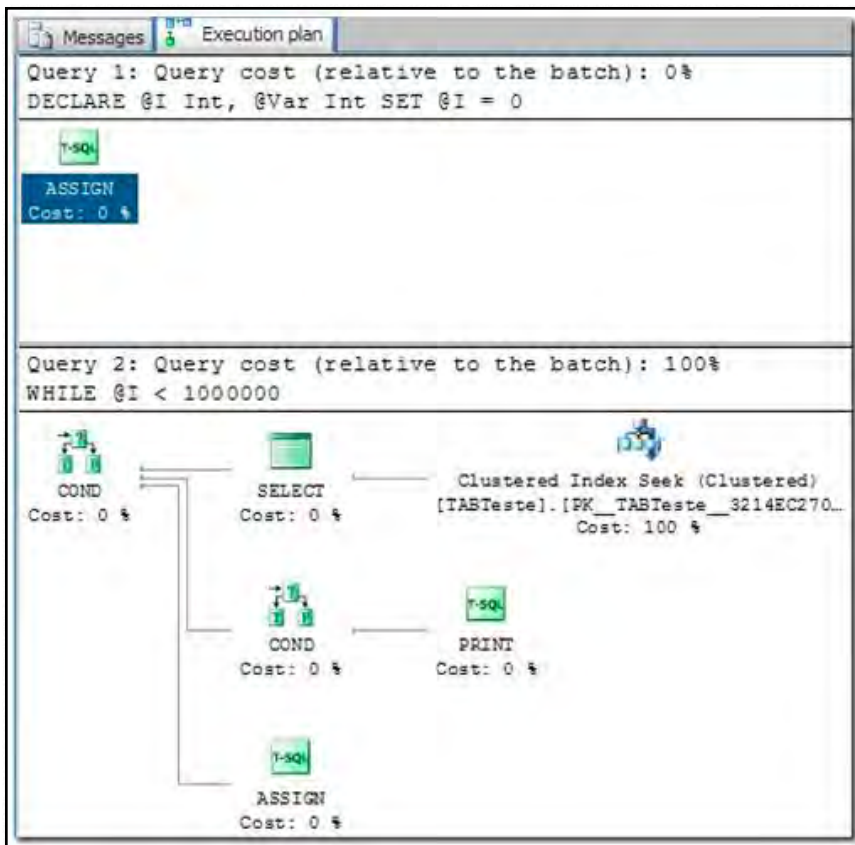
If we look at the profiler results we can see the column, CPU, which shows us how much CPU the query above uses.

EventClass	TextData	CPU	Reads	Writes	Duration
SQL:BatchStarting	DECLARE @I...				
SQL:BatchCompl...	DECLARE @I...	12828	2000000	0	13379

Now let's change the code to remove the **Compute Scalar** operator.

```
DECLARE @I Int, @Var Int
SET @I = 0
WHILE @I < 1000000
BEGIN
    SELECT @Var = ID FROM TABTeste WHERE ID = @I
    IF @@ROWCOUNT > 0
    BEGIN
        PRINT 'Entrou no IF'
    END
    SET @I = @I + 1;
END
GO
```

Chapter 3: Compute Scalar



Graphical execution plan.

```
--Clustered Index Seek(OBJECT:([tempdb].[dbo].[TABTeste].[PK__  
TABTeste__3214EC27096F09E1]), SEEK:([tempdb].[dbo].[TABTeste].[ID]=[@I]) ORDERED  
FORWARD)
```

Text execution plan.

Now that SQL Server does not use the **Compute Scalar**, let's take a look at the CPU costs.

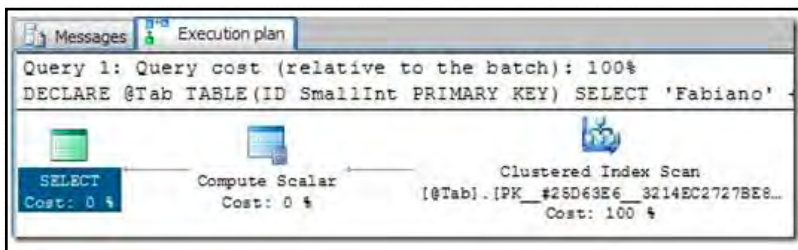
Chapter 3: Compute Scalar

EventClass	TextData	CPU	Reads	Writes	Duration
SQL:Bat...	DECLAR...				
SQL:Bat...	DECLAR...	10750	2000000	0	11163

As you can see, SQL Server uses less CPU and finishes the execution of the query faster than it does with the first query. I'm not trying to show you the better way to check whether a particular value exists, I'm just showing the **Compute Scalar** behavior. However, if you have never seen this kind of validation using @@RowCount, it may be that it could help you a little bit in your coding. Some time ago I changed one procedure that uses a lot of **IF Exists** in much the same way, with very satisfactory result for the performance of the procedure.

Let's take a look at more practical examples of **Compute Scalar**.

```
DECLARE @Tab TABLE(ID SmallInt PRIMARY KEY)
SELECT 'Fabiano' + ' - ' + 'Amorim' FROM @Tab
```



Graphical execution plan.

```
--Compute Scalar(DEFINE:([Expr1003]='Fabiano - Amorim'))
--Clustered Index Scan(OBJECT:(@Tab))
```

Text execution plan.

Chapter 3: Compute Scalar

The plan was generated using the **Compute Scalar** just to make the concatenation between "Fabiano", "-" and "Amorim". Quite simple.

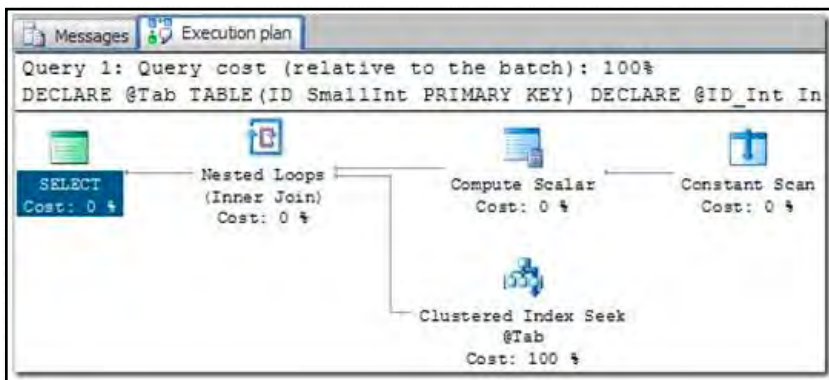
Now we'll see one very interesting behavior of **Compute Scalar** that it changes in SQL Server 2005/2008.

Consider the following query:

```
DECLARE @Tab TABLE(ID SmallInt PRIMARY KEY)
DECLARE @ID_Int Integer
SELECT *
FROM @Tab
WHERE ID = @ID_Int
```

Notice that the Column ID is a **SmallInt** type, and the variable **@ID_Int** is a Integer, that means SQL Server as to convert the value of **@ID_Int** to be able to compare the value with ID Column.

At SQL Server 2000 we have the following plans:



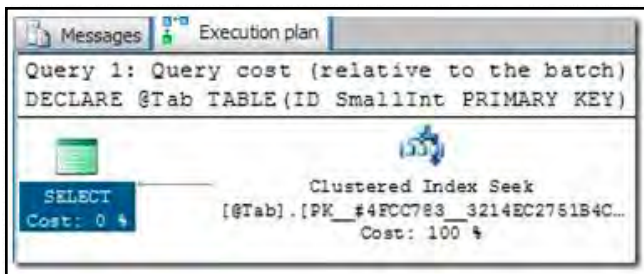
SQL 2000 Graphical execution plan.

Chapter 3: Compute Scalar

```
--Nested Loops (Inner Join, OUTER REFERENCES: ([Expr1002], [Expr1003],  
[Expr1004]))  
  
|--Compute Scalar (DEFINE: ([Expr1002]=Convert([@ID_Int])-1,  
[Expr1003]=Convert([@ID_Int])+1, [Expr1004]=If (Convert([@ID_Int])-1=NULL) then 0  
else 6|If (Convert([@ID_Int])+1=NULL) then 0 else 10))  
|--Constant Scan  
|--Clustered Index Seek (OBJECT: (@Tab), SEEK: (@Tab.[ID] > [Expr1002] AND @  
Tab.[ID] < [Expr1003]), WHERE: (Convert(@Tab.[ID])=[@ID_Int]) ORDERED FORWARD)
```

SQL 2000 Text execution plan.

Wow, it's quite hard work, don't you think? Now let's take a look at what happens if we run this code at SQL 2005/2008.



SQL 2005/2008 Graphical execution plan.

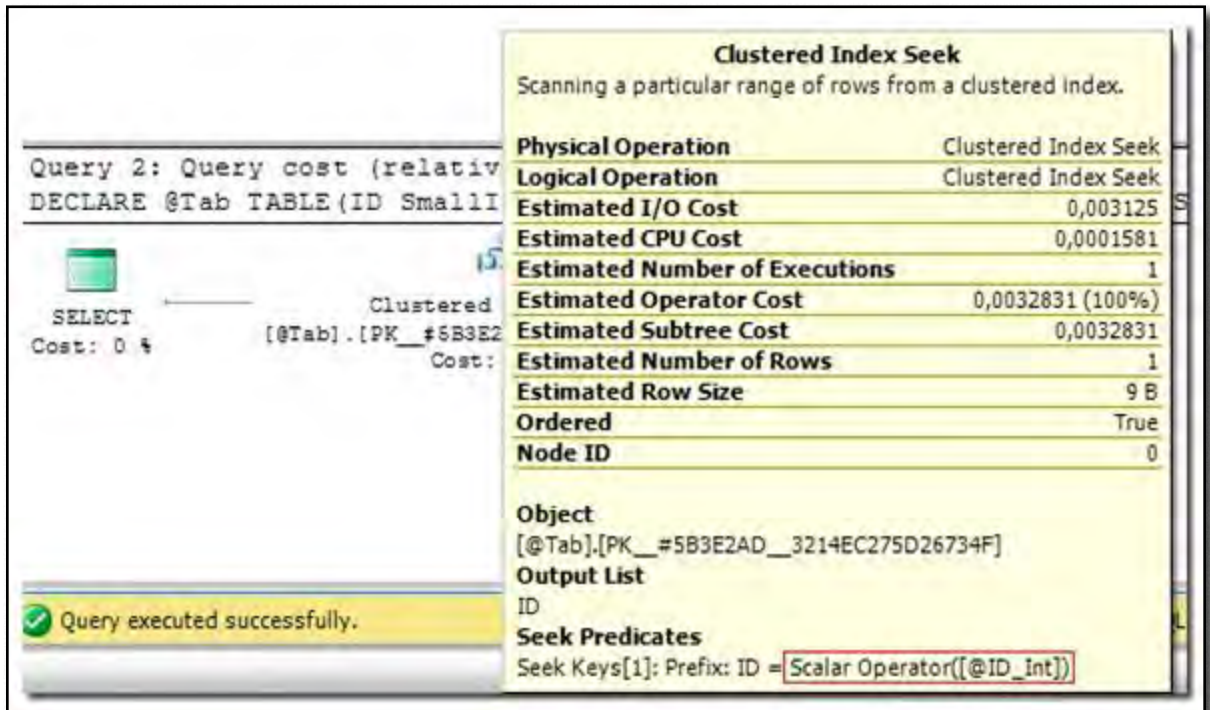
```
--Clustered Index Seek (OBJECT: (@Tab), SEEK: ([ID]=[@ID_Int]) ORDERED FORWARD)
```

SQL 2005/2008 Text execution plan.

Yep, now we have a much simpler plan (which was nothing about the band). But wait a minute, what is this? Now SQL Server does not convert the value!

Let's look at the execution plan to understand what is going on with the **Clustered Index Seek** show plan operator.

Chapter 3: Compute Scalar



Graphical execution plan.

As we can see, the SQL Server Dev Team has changed the Engine to use a function called **Scalar Operator** to convert the value to the appropriate datatype; that's interesting.

Chapter 4: BookMark/Key Lookup

It's time to talk about a film star amongst operators – **Key Lookup** is such a famous operator that I couldn't write about it without giving it the red carpet treatment.

Get yourself comfortable, grab some popcorn, and relax. I realize that you know this smooth operator, but I hope you discover something new about Key Lookup behavior and learn some good tips here. I hope to give you a fuller understanding of the Bookmark/Key Lookup by the time you finish the article.

As if seeking anonymity, this operator has changed its identity three times in the last three versions of SQL Server. In SQL 2000 it was called **BookMark Lookup**. SQL 2005 comes, and it was renamed and showed as a simple **Clustered Index Seek** operation. Since SQL Server 2005 SP2 it has been called a **Key Lookup**, which makes more sense to me. In a text execution plan, it is represented as a **Clustered Index Seek**.

First, I'll explain a bit about the way that SQL Server uses an index, and why unordered bookmark operations are so very expensive. I'll also create a procedure to return some information about when lookup operations are good, and when a scan turns out to be better than a lookup.



Icon in SQL Server 2000.



Icon in SQL Server 2005 SP2 and 2008.

Before you get distracted into thinking about RIDs/Heaps, let me say one thing – I'll be talking about RID Lookup another time. The main focus here is about the **Key Lookup** operator.

In summary, the **Key Lookup** is used to fetch values via a clustered index, when the required data isn't in a non-clustered index. Let's take a look further.

I once heard Kimberly Tripp give a very good and practical analogy: imagine a book, and suppose we have two indexes, the first is a clustered index, the Table of Contents that appears in beginning of the book; and the non-clustered index is the index in the back of the book. When you need to search for some information in the index at the back, you have a pointer, the page number, to the page that mentions the subject. This "page number" is what we call a **BookMark**, in SQL terms, that is the **Cluster Key**.

This **BookMark** action goes to the "back" index, finds the page number that contains the information, and goes to the page. This is what a "bookmark lookup" does.

Suppose I want to know more about "bookmark lookups"; well, since I have the *Inside Microsoft SQL Server 2005 – Query Tuning and Optimization* book, I can go to the back index and see if there is information somewhere in the book. At the "B" word I have a text "BookMark Lookup" and the number of the page that talks about the subject. The index at the end of the book is very useful.

But wait a minute, there is a snag. A Key lookup is a very expensive operation because it performs a random I/O into the clustered index. For every row of the non-clustered index, SQL Server has to go to the Clustered Index to read their data. We can take advantage of knowing this to improve the query performance. To be honest, when I see an execution plan that is using a **Key Lookup** it makes me happy, because I know I've a good chance of improving the query performance just creating a covering index. A Covering index is a non-clustered "composite" index which contains all columns required by the query.

Let's demonstrate this with some code.

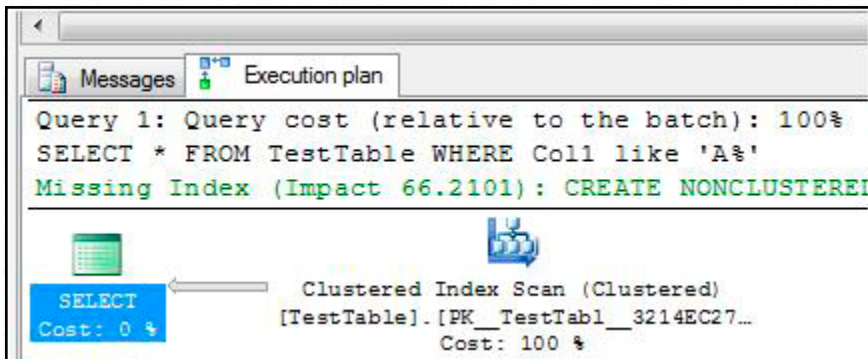
The following script will create a table called **TestTable** and will insert 100,000 rows with garbage data. An index called **ix_Test** will be created to be used in our lookup samples.

Chapter 4: BookMark/Key Lookup

```
IF OBJECT_ID('TestTable') IS NOT NULL
BEGIN
    DROP TABLE TestTable
END
GO
CREATE TABLE TestTable(ID    Int Identity(1,1) PRIMARY KEY,
                        Col1  VarChar(250) NOT NULL DEFAULT NewID(),
                        Col2  VarChar(250) NOT NULL DEFAULT NewID(),
                        Col3  VarChar(250) NOT NULL DEFAULT NewID(),
                        Col4  DateTime      NOT NULL DEFAULT GetDate())
GO
SET NOCOUNT ON
GO
INSERT INTO TestTable DEFAULT VALUES
GO 100000
CREATE NONCLUSTERED INDEX ix_Test ON TestTable(Col1, Col2, Col3)
GO
```

Now suppose the following query:

```
SET STATISTICS IO ON
SELECT *
    FROM TestTable
WHERE Col1 like 'A%'
SET STATISTICS IO OFF
```



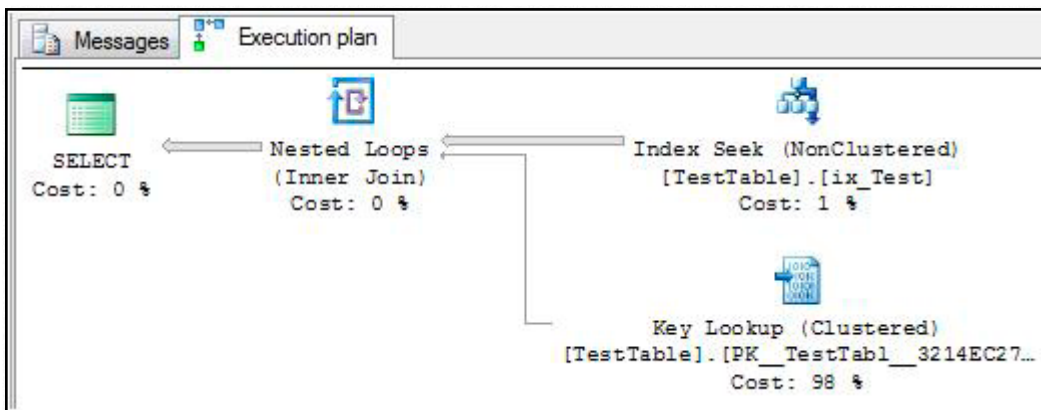
(6223 row(s) affected)

Table 'TestTable'. Scan count 1, logical reads 1895, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

As we can see in the execution plan above, the Query **Optimizer** chose not to use the **ix_Test** index, but instead chose to scan the clustered index to return the rows. 6,223 rows were returned and SQL made 1895 logical reads to return that data. That means the table had 1,895 pages, because SQL made a Scan on the table.

Now, let's suppose that I don't trust the Query Optimizer to create a good plan (don't hit me, but, do you?), and I decide add a hint to say how SQL Server should access the data. What will happen?

```
SET STATISTICS IO ON
SELECT *
  FROM TestTable WITH(INDEX = ix_Test)
WHERE Col1 like 'A%'
SET STATISTICS IO OFF
```



(6223 row(s) affected)

Table 'TestTable'. Scan count 1, logical reads 19159, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

Here the Key lookup operator is used to read the value of the column Col4 by referencing the clustered index, since that column is not into the non-clustered index **ix_Test**.

Cool, now our smartest plan uses the index, and return the same 6,223 rows, but, wait a minute, it read 19,159 pages! In other words, it has read 1,7264 more than the first plan, or ten times the table size.

As you can see, a Key Lookup is a good choice only when a few rows are returned; but the retrieval cost grows with the quantity of returned rows. As [JASON MASSIE](#) once said, there is a point where the threshold is crossed and a scan becomes more efficient than a lookup. A scan uses a sequential IO while a lookup does random IO.

Here, we can raise a question. How many rows before a key lookup stops being a good strategy? When is this threshold crossed, and a scan performs better?

To help you with that, I've created a procedure called `st_TestLookup` ([YOU CAN DOWNLOAD THE CODE HERE](#)). I'll not explain what I did, because this is not the intention, but you need to enable `xp_cmdShell` to run this code so, obviously, I don't recommend that you to run this in a production environment.

The code below runs the procedure to `TestTable`, the input parameters are three:

- `@Table_Name` where you input the table that you want to analyze
- `@Lookup_Index` where you input the non-clustered index that will be analyzed
- a valid Path to SQL Server to create a profiler trace to check the amount of logical IOs for each query.

Let's look at the results.

Chapter 4: BookMark/Key Lookup

```
EXEC dbo.st_TestLookup @Table_Name    = 'TestTable',
                        @Lookup_Index  = 'ix_Test',
                        @Trace_Path    = 'C:\TesteTrace.trc'

GO

Logical Reads to Scan 100000 rows of table TestTable: 1902
GoodPlan - Logical Reads to Lookup 100 rows of table : 339
GoodPlan - Logical Reads to Lookup 200 rows of table : 659
GoodPlan - Logical Reads to Lookup 300 rows of table : 981
GoodPlan - Logical Reads to Lookup 400 rows of table : 1301
GoodPlan - Logical Reads to Lookup 500 rows of table : 1620
BadPlan - Logical Reads to Lookup 600 rows of table : 1940

***** Scan
***
*****
*****
*****
*****
*****
*****
```

The first result line shows us how many rows are in the table, and how many logical reads are used to scan the table. This is our start point. Based on that value, I then do a loop, reading values of the table. When the number of lookup IOs cross the Scan, I write a line starting with "BadPlan", and I show the number of IOs to read "x" rows using the Key Lookup operator.

The lines with "*" are just a tentative way to show these results in a graphical mode.

Based on the procedure results, we know that, when we need more than 600 rows, a Scan is better than a Lookup.

Chapter 5: Spools – Eager Spool

Spool operators



Eager Spool



Lazy Spool



Row Count Spool



Table Spool



Non-Clustered Index
Spool

There are five types of Spool operators, each with its own behavior, and idiosyncrasies, but they all share the way that they save their intermediate query results on the **TempDb** database, and use this temporary area to search a value.

There are many tricks that Query Optimizer uses to avoid any logical problems and to perform queries better. The spool operators are a good example of this.

A spool reads the data and saves it on **TempDb**. This process is used whenever the **Optimizer** knows that the [DENSITY](#) of the column is high and the intermediate result is very complex to calculate. If this is the case, SQL makes the computation once, and stores the result in the temporary space so it can search it later.

The spool operators are always used together with another operator. As it stores values, it needs to know what these values are, and so it must receive them from another operator.

For instance, one spool can be used together with a **clustered index scan** in order to save the rows read by the scan. These rows can then be read back to an **update**, or a **select** operator. A quite simple graphic representation of this could be the following picture:



In the example represented by these icons above, the order of execution is: **Clustered Index Scan** sends rows to **Spool** and the **Select** reads all of these rows directly from the Spool.

Eager Spool

We'll start with the Eager Spool operator. The role of the Eager Spool is to catch **all** the rows received from another operator and store these rows in **TempDb**. The word "eager" means that the operator will read ALL rows from the previously operator at one time. In other words, it will take the entire input, storing each row received.

In our simple sample (SCAN ► EAGER SPOOL ► SELECT) the Eager will work like this: when the scanned rows are passed to Spool then it gets all the rows, not one row at the time you'll notice but the entire scan, and keeps them in a hidden temporary table.

Knowing that, we could say that The **Eager Spool** is a blocking operator. I believe that I haven't yet written about the difference between blocking operators and non-blocking operators. Let's open a big parenthesis here.

There are two categories of Showplan operators: the "non-blocking" operators and the "blocking" operators or "stop-and-go."

Non-blocking operators are those that read one row from their input and return the output for each read row. In the method known as `GetRow()`, the operator executes its function and returns the row to the next operator as soon as the first row has been read.

The "nested loop" operator is a good example of this "non-blocking" behavior. When the first row is read, SQL needs to perform a Join with the outer table; if the outer table joins with the inner table then a row is returned. This process repeats until the end of the table. To each received row, SQL tries to join the rows and return the required values.

A blocking operator needs to read all the rows from its input to perform some action and then return the data. A classic sample of a blocking operator is the **SORT** operator; it needs to read all rows, sort the data and then return the ordered rows. The execution of the query will wait until all rows to be read and ordered, before continuing with the command.

The Halloween Problem

There is a very interesting classic computing problem, called "The Halloween Problem." Microsoft Engineers take advantage of the blocking **Eager Spool** operator to avoid this problem.

In order to illustrate this, I will start with the following code to create a table called **Funcionarios** ("Employees" in Portuguese).

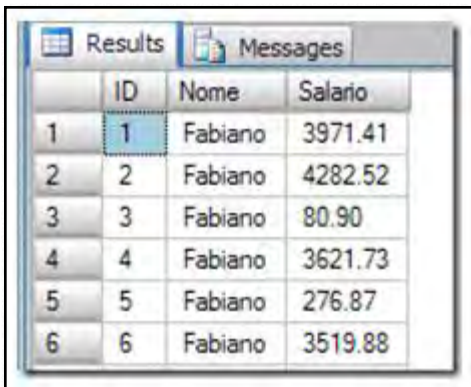
This script will create the table with three columns, **ID**, **Nome** (Name) and **Salario** (Salary), and populate them with some garbage data.

Chapter 5: Spools – Eager Spool

```
USE TempDB
GO
SET NOCOUNT ON
IF OBJECT_ID('Funcionarios') IS NOT NULL
    DROP TABLE Funcionarios
GO
CREATE TABLE Funcionarios(ID Int IDENTITY(1,1) PRIMARY KEY,
                           Nome VarChar(30),
                           Salario Numeric(18,2));
GO

DECLARE @I SmallInt
SET @I = 0
WHILE @I < 1000
BEGIN
    INSERT INTO Funcionarios(Nome, Salario)
    SELECT 'Fabiano', ABS(CONVERT(Numeric(18,2), (Checksum(NEWID()) / 500000.0)))
    SET @I = @I + 1
END
CREATE NONCLUSTERED INDEX ix_Salario ON Funcionarios(Salario)
GO
```

This is what the data looks like.



The screenshot shows a SQL Server Results window with a table containing 6 rows. The columns are ID, Nome, and Salario. The data is as follows:

	ID	Nome	Salario
1	1	Fabiano	3971.41
2	2	Fabiano	4282.52
3	3	Fabiano	80.90
4	4	Fabiano	3621.73
5	5	Fabiano	276.87
6	6	Fabiano	3519.88

OK, now that we have run the script, we can return to where we left off when describing the operator **Eager Spool**.

To show the functionality of the Eager Spool we'll go back in time a bit, to the time when I was just a project being planned, but some other geeks working intensively with databases.

It was Halloween; the cold winter's night was black as pitch, (I really don't know if was winter, but I thought it would sound more thrilling) and the wind howled in the trees. It was 1976 and the children was demanding "trick or treat" in the houses. The full moon shone and illuminated the whole city when suddenly, some clouds crossed the moon making the night even more dark and gloomy. It was possible to smell and taste the tension in the air like a stretched rubber band so close to bursting. People walking in the street felt that someone was observing them, and when they looked closely behind them, they could see two red eyes waiting and looking out for unprotected prey.

Was that their imagination? Or just the wrong night to work with databases?

Meanwhile, in a place not far away, an update was started on a database by a skeleton staff, to update the salary by 10% of all employees who earned less than \$ 25,000 dollars. Their feelings of impending doom increased as the query failed to complete in the expected time. When, at length, it did, they found to their horror that every employee had their pay increased to \$ 25,000. It was the stuff of DBA nightmares.

So begins the story of a problem known as "Halloween Problem." IBM engineers were the first to find the problem, but several databases have suffered [SIMILAR PROBLEMS OVER THE YEARS](#), including our lovely SQL Server as we can see [HERE](#). The Halloween problem happens when the write cursor interferes with the read cursor. The selection of the rows to update is affected by the actual update process. This can happen when there is a index on the particular column being updated. When the update is made, the same row can be updated several times.

All updates are executed in two steps; the first is the read step, and the second is the update. A read cursor identifies the rows to be updated and a write cursor performs the actual updates.

Chapter 5: Spools – Eager Spool

```
UPDATE Funcionarios SET Salario = 0  
WHERE ID = 10
```

For the execution of this query, the first step is to locate the records that need updating, and then the second step needs to update these records. If there is an index on the column being modified (**Salario**) then this index also needs to be updated.

As we know, all non-clustered indexes need to be updated when a value changes that is used in the index.

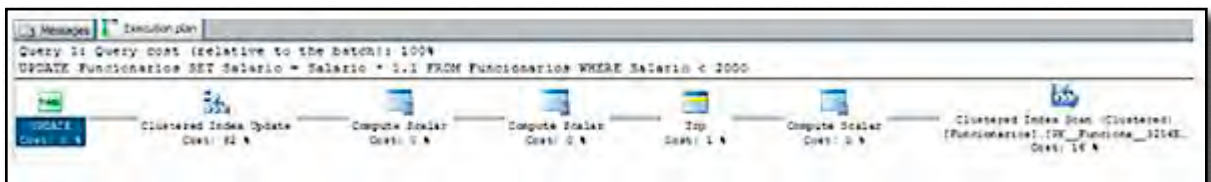
The problem can happen if SQL Server chooses to read the data using this index during the read step. This value can then change when the second step is being performed.

Let's see some examples using the same type of query that was being used when the problem was first found by the IBM engineers.

Suppose that I want to give 10% of increase salary to all employees that earn less than R\$2,000 reais (the Brazilian currency). I could run the following query:

```
UPDATE Funcionarios SET Salario = Salario * 1.1  
FROM Funcionarios  
WHERE Salario < 2000
```

We have the following execution plans:



Graphical execution plan.

Chapter 5: Spools – Eager Spool

```
--Clustered Index Update(OBJECT:([Funcionarios].[PK]), OBJECT:([Funcionarios].[ix_
Salario]), SET:([Funcionarios].[Salario] = [Expr1003]))
  |--Compute Scalar(DEFINE:([Expr1016]=[Expr1016]))
    |--Compute Scalar(DEFINE:([Expr1016]=CASE WHEN [Expr1007] THEN (0) ELSE
(1) END))
      |--Top(ROWCOUNT est 0)
        |--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(numeric
c(18,2),[Funcionarios].[Salario]*(1.1),0), [Expr1007]=CASE WHEN [Funcionarios].
[Salario] = CONVERT_IMPLICIT(numeric(18,2),[Funcionarios].[Salario]*(1.1),0) THEN
(1) ELSE (0) END))
          |--Clustered Index Scan(OBJECT:([Funcionarios].[PK]),
WHERE:([Funcionarios].[Salario]<(2000.00)) ORDERED)
```

Text execution plan.

For now, let us just look at the steps that I mentioned. In the **Clustered Index Scan**, we see that SQL selects all the rows that will be updated, and the **Clustered Index Update** then updates these rows in the cluster index (PK) and the non-clustered index **ix_salario**.

Well, if we follow this logic, what do we predict will happen? Let's take this a step at a time to see how this will work.

We know that the clustered index is ordered by ID column. SQL Server is using this index to look at what rows needs to be updated, so we have the following:

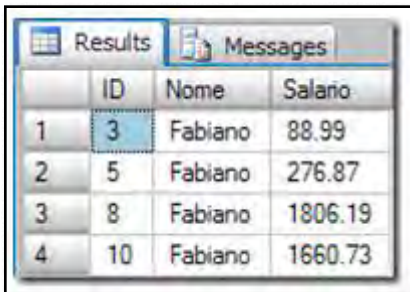


	ID	Nome	Salario
1	3	Fabiano	80.90
2	5	Fabiano	276.87
3	8	Fabiano	1806.19
4	10	Fabiano	1660.73

The first selected row is the ID = 3. After we select the row, we need to update the "**salario**" column. So, we could perform the update of the salary with the actual value * 1.1.

Chapter 5: Spools – Eager Spool

After the update we have the following:



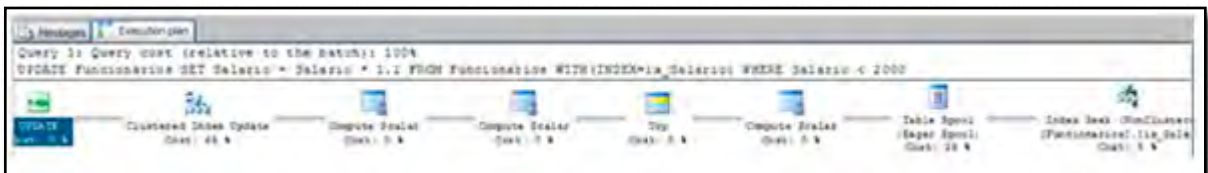
	ID	Nome	Salario
1	3	Fabiano	88.99
2	5	Fabiano	276.87
3	8	Fabiano	1806.19
4	10	Fabiano	1660.73

Note that the salary earned a 10% increase, from 80.90 to 88.99. We then continue by selecting the next row, ID = 5, and this process will then continue until the end of the selected rows.

So far, we can see that the use of this clustered index generates no error. If SQL Server continues with this logic until the end of the process, then all rows will be updated correctly. What if SQL chose, instead, to use the non-clustered index **ix_salario** to read the rows that will be updated (first step of the update), what will happen then?

Let's use the same illustration that we used above. But this time we'll force the use of the non-clustered index **ix_salario**.

```
UPDATE Funcionarios SET Salario = Salario * 1.1
FROM Funcionarios WITH(INDEX=ix_Salario)
WHERE Salario < 2000
```



Graphical execution plan.

Chapter 5: Spools – Eager Spool

```
--Clustered Index Update(OBJECT:([Funcionarios].[PK]), OBJECT:([Funcionarios].[ix_
Salario]), SET:([Funcionarios].[Salario] = [Expr1003]))
  |--Compute Scalar(DEFINE:([Expr1016]=[Expr1016]))
    |--Compute Scalar(DEFINE:([Expr1016]=CASE WHEN [Expr1007] THEN (0) ELSE
(1) END))
      |--Top(ROWCOUNT est 0)
        |--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(numeric
c(18,2),[Funcionarios].[Salario]*(1.1),0), [Expr1007]=CASE WHEN [Funcionarios].
[Salario] = CONVERT_IMPLICIT(numeric(18,2),[Funcionarios].[Salario]*(1.1),0) THEN
(1) ELSE (0) END))
          |--Table Spool
            |--Index Seek(OBJECT:([Funcionarios].[ix_Salario]),
SEEK:([Funcionarios].[Salario] < (2000.00)) ORDERED FORWARD)
```

Text execution plan.

You'll see from the execution plan that, this time, after reading the data using the index **ix_Salario**, SQL Server uses a Blocking operator called Table Spool (Eager Spool). As I mentioned earlier in this article, when the Eager is called the first time, it will read all data and then move to the next operator. In our example, the Eager Spool writes the data returned from the index **ix_Salario** into a temporary table. Later, the updates do not read **ix_salario** any more; instead all reads are performed using the Eager Spool.

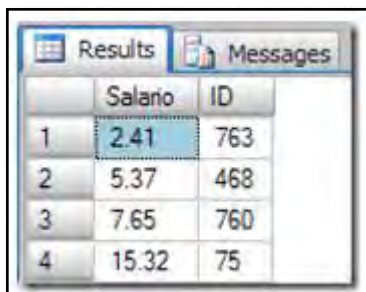
You may would be wondering, "Hey, Fabiano, where is the Halloween problem?" We will get there now.

Let's suppose that SQL Server hadn't used the Eager Spool operator. Let's assume that we have the same execution plan above but without the Eager Spool operator.

If SQL Server hadn't used the Eager Spool to read the data, it would have read rows directly from the index **ix_Salario**. It will read the first row, update the value with 10%, that get the next row and so on.

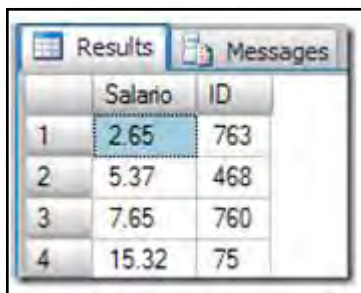
We know the index **ix_Salario** is ordered by column **Salario**, so let's draw the things again to see what will happen.

The data returned to the Index would be:



	Salario	ID
1	2.41	763
2	5.37	468
3	7.65	760
4	15.32	75

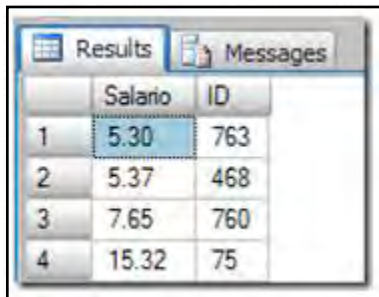
The first row is the ID 763, when the SQL Server updates the row the data in the index will be the following:



	Salario	ID
1	2.65	763
2	5.37	468
3	7.65	760
4	15.32	75

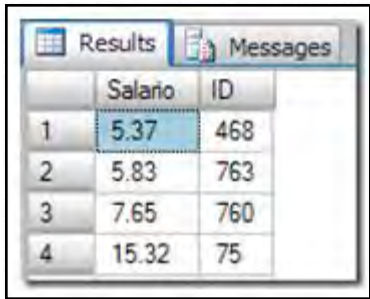
Now the next row is the ID 468.

Using this data we will not have a problem. But tell me one thing, if the data was distributed like that?



	Salario	ID
1	5.30	763
2	5.37	468
3	7.65	760
4	15.32	75

Well, that would be a very dangerous problem, Let's go again, get the first row, ID = 763. Update this value with 10%, the data in the index will be the following:



	Salario	ID
1	5.37	468
2	5.83	763
3	7.65	760
4	15.32	75

Now, get the next row, which was what? The ID 763 again? Yes, take it easy; let's understand why the row 763 ended up in the second row. Wasn't it in the first row?

Yes, it was, but when SQL Server updated the **Salario** column value by 10% it updated the non-clustered index **ix_Salario** (see into the execution plan, the clustered index updates **ix_Salario** and the PK) too, which meant that data was repositioned in the non-clustered index. The index needs to keep the data physically sorted by **salario**. Once a value changes, it resets this value in the balanced tree.

The outcome would be that the employee with ID = 763 has the salary increased by 20% (well that could be good, since he calls Fabiano). The problem found at IBM, the engineers said the end of the query all employees were earning \$ 25,000.00. And then they had to start to understand what had happened.

The team at Microsoft that develops the Query Processor uses blocking operators to ensure that the read data will be the same regardless whether there is a later update. In our query example, SQL uses the SQL Eager Spool. When it needs to read the rows for the second time, it will not use the index **ix_Salario** but will read from the Spool. The spool has a copy of rows of index **ix_Salario** in **TempDb** database.

Well, we saw that the Eager Spool can be used to avoid the problem known as "Halloween problem," but it can also be used in your queries by the Query Optimizer whenever it reckons that it pays to create a copy of the data. Keep eye out for more about the Spool operators. In the next chapter, we will describe the **Lazy Spool** operator.

Chapter 6: Spools – Lazy Spool



Lazy Spool

The **Lazy Spool** is actually very similar to the Eager Spool; the difference is just that Lazy Spool reads data *only* when individual rows are required. It creates a temporary table and builds this table in a "lazy" manner; that is, it reads and stores the rows in a temporary table only when the parent operator actually *asks* for a row, unlike Eager Spool, which reads all rows at once. To refer back to some material I covered in the Eager Spool explanation, the Lazy Spool is a non-blocking operator, whereas Eager Spool is a blocking operator.

To highlight the Lazy Spool, we'll create a table called **Pedido** (which means "Order" in Portuguese). The following script will create a table and populate it with some garbage data.

```
IF OBJECT_ID('Pedido') IS NOT NULL
    DROP TABLE Pedido
GO
CREATE TABLE Pedido (ID INT IDENTITY(1,1) PRIMARY KEY,
    Cliente INT NOT NULL,
    Vendedor VARCHAR(30) NOT NULL,
    Quantidade SmallInt NOT NULL,
    Valor Numeric(18,2) NOT NULL,
    Data DATETIME NOT NULL)
DECLARE @I SmallInt
SET @I = 0
WHILE @I < 50
BEGIN
    INSERT INTO Pedido(Cliente, Vendedor, Quantidade, Valor, Data)
    SELECT ABS(CheckSUM(NEWID()) / 100000000),
        'Fabiano',
        ABS(CheckSUM(NEWID()) / 10000000),
        ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
        GETDATE() - (CheckSUM(NEWID()) / 1000000)
    END
```


Chapter 6: Spools – Lazy Spool

```
INSERT INTO Pedido(Cliente, Vendedor, Quantidade, Valor, Data)
SELECT ABS(CheckSUM(NEWID()) / 100000000),
       'Amorim',
       ABS(CheckSUM(NEWID()) / 100000000),
       ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
       GETDATE() - (CheckSUM(NEWID()) / 1000000)
INSERT INTO Pedido(Cliente, Vendedor, Quantidade, Valor, Data)
SELECT ABS(CheckSUM(NEWID()) / 100000000),
       'Coragem',
       ABS(CheckSUM(NEWID()) / 100000000),
       ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
       GETDATE() - (CheckSUM(NEWID()) / 1000000)
SET @I = @I + 1
END
SET @I = 1
WHILE @I < 3
BEGIN
    INSERT INTO Pedido(Cliente, Vendedor, Quantidade, Valor, Data)
    SELECT Cliente, Vendedor, Quantidade, Valor, Data
    FROM Pedido
    SET @I = @I + 1
END
GO
```

This is what the data looks like:

	ID	Cliente	Vendedor	Quantidade	Valor	Data
1	1	11	Fabiano	137	1639.92	2009-11-07 15:13:26.597
2	2	6	Amorim	166	1644.17	2004-11-04 15:13:26.597
3	3	2	Coragem	46	612.53	2007-12-06 15:13:26.597
4	4	3	Fabiano	69	315.08	2014-08-21 15:13:26.597
5	5	1	Amorim	200	1712.81	2003-05-20 15:13:26.597
6	6	15	Coragem	153	1160.99	2014-07-04 15:13:26.597
7	7	19	Fabiano	207	2100.44	2014-03-14 15:13:26.597
8	8	8	Amorim	25	993.69	2014-01-05 15:13:26.597

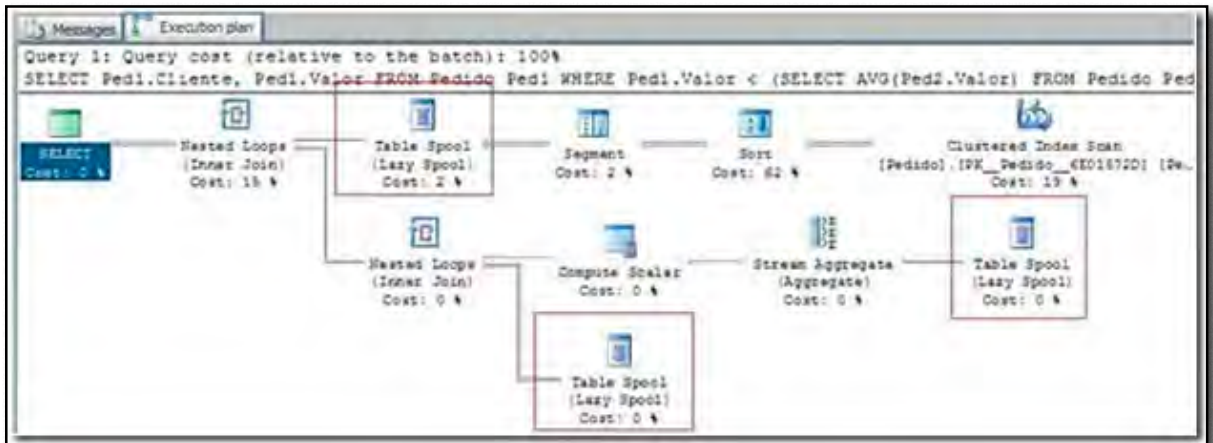
Chapter 6: Spools – Lazy Spool

To understand the Lazy Spool, I wrote a query that returns all Orders where the Order value is lower than the average value of all the relevant customer's orders. That sounds a little convoluted, so let's just look at the query.

```
SELECT Ped1.Cliente, Ped1.Valor
FROM Pedido Ped1
WHERE Ped1.Valor < (
  SELECT AVG(Ped2.Valor)
  FROM Pedido Ped2
  WHERE Ped2.Cliente = Ped1.Cliente)
```

Before we see the execution plan, let's make sure we understand the query a little better. First, for each customer in the **FROM** table (**Ped1.Cliente**), the SubQuery returns the average value of all orders (**AVG(Ped2.Valor)**). After that, the average is compared with the principal query and used to filter just each customer's orders with values lower than their average.

So, now we have the following plans:



Graphical execution plan.

Chapter 6: Spools – Lazy Spool

```
|--Nested Loops(Inner Join)
|   |--Table Spool
|       |--Segment
|           |--Sort(ORDER BY:([Ped1].[Cliente] ASC))
|               |--Clustered Index Scan(OBJECT:([Pedido].[PK__
Pedido__6E01572D] AS [Ped1]))
|                   |--Nested Loops(Inner Join, WHERE:([Pedido].[Valor] as [Ped1].
[Valor]<[Expr1004]))
|                       |--Compute Scalar(DEFINE:([Expr1004]=CASE WHEN [Expr1012]=(0) THEN NULL
ELSE
|                           [Expr1013]/CONVERT_IMPLICIT(numeric(19,0),[Expr1012],0) END))
|                               |--Stream Aggregate(DEFINE:([Expr1012]=Count(*),
|                                   [Expr1013]=SUM([Pedido].[Valor] as [Ped1].[Valor]))
|                                       |--Table Spool
|--Table Spool
```

Text execution plan.

As we can see, the Spool operator is displayed three times in the execution plan, but that doesn't mean that three temporary tables were created. All the Spools are actually using the same temporary table, which can be verified if you look at the operator's hints displayed in the graphical execution plan below.

Table Spool	Table Spool	Table Spool
Stores the data from the input into a temporary table in order to optimize rewinds.	Stores the data from the input into a temporary table in order to optimize rewinds.	Stores the data from the input into a temporary table in order to optimize rewinds.
Physical Operation	Table Spool	Physical Operation
Logical Operation	Lazy Spool	Logical Operation
Actual Number of Rows	22	Actual Number of Rows
Estimated I/O Cost	0	Estimated I/O Cost
Estimated CPU Cost	0	Estimated CPU Cost
Number of Executions	1	Number of Executions
Estimated Number of Executions	1	Estimated Number of Executions
Estimated Operator Cost	0,0006241 (2%)	Estimated Operator Cost
Estimated Subtree Cost	0,0274119	Estimated Subtree Cost
Estimated Number of Rows	21	Estimated Number of Rows
Estimated Row Size	20 B	Estimated Row Size
Actual Rebinds	1	Actual Rebinds
Actual Rewinds	0	Actual Rewinds
Node ID	2	Node ID
Primary Node ID	2	Primary Node ID
Output List	[tempdb].[dbo].[Pedido].Cliente; [tempdb].[dbo].[Pedido].Valor	Output List
		[tempdb].[dbo].[Pedido].Cliente; [tempdb].[dbo].[Pedido].Valor

As you can see, the first Spool hint has the Node ID equal to 2, and the other two operators are referenced to the Primary Node 2 as well. Now let's look at a step-by-step description of the execution plan so that we can understand exactly what it is doing. Note this is not the exact order of execution of the operators, but I think you'll understand things better in this way.

The **first step** in the execution plan is to read all of the data that will be used in the query, and then group that data by customer.



The Clustered Index Scan operator reads all rows from the **Cliente** and **Valor** (**Client** and **Value**) columns. So, the Input of this operator is the following:

	ID	Cliente	Vendedor	Quantidade	Valor	Data
1	1	11	Fabiano	137	1639.92	2009-11-07 15:13:26.597
2	2	6	Amorim	166	1644.17	2004-11-04 15:13:26.597
3	3	2	Coragem	46	612.53	2007-12-06 15:13:26.597
4	4	3	Fabiano	69	315.08	2014-08-21 15:13:26.597

... and the Output is *just* the **Client** and **Value** columns:

	Cliente	Valor
1	11	1639.92
2	6	1644.17
3	2	612.53



When the **SORT** operator receives the rows from the clustered index scan, its output is all the data, ordered by the **Client** column:

	Cliente	Valor
1	0	362.78
2	0	1799.71
3	0	1741.02



The **Segment** operator divides the data into many groups; in this case it receives all the rows, ordered by customers, and divides them into groups that share the same customer. So, the first segment produced by this operator will be all the rows where "**Client = 0**". Given that the data is already sorted by customers, the operator just needs to read down the rows until it finds a different value in order to create a segment.

When the value it is reading changes, it finishes its job and the next operator immediately receives the segment of all the data for "Customer 0". This process will repeat until all segments are completely read. The final output of the Segment operator is a series of segments dividing all the data according to customer, such that each segment contains all the rows for a particular customer. In this walkthrough, we'll look at all the rows where "**Client=0**".

	Cliente	Valor
1	0	362.78
2	0	1799.71
3	0	1741.02



Here we get on to the **Table Spool** operator, working as a "Lazy" Spool. It will create a temporary table in the **TempDb** database, and store all data returned from the Segment operator; in this case, all the data for customer o. The output of the Spool operator is just all data stored in the **TempDb** table.



The **Nested Loops** operator joins the first and second parts of the execution plan, or rather, the principle query with the subquery. As we now, the nested loops scan a table and join it with another table one row at time, and so for each row in the Table Spool (Item 4) the nested loop will join the result of Item 11. To give you a quick preview, this result will be the rows where the **Value** column in the Spool table is lower than the value calculated in the aggregation (Item 8 – the average value of the customer's orders). When this step is finished, the Spool operator (Item 4) is called again, and it in turn calls the Segment operator, which reads another segment of rows (i.e. processes another customer). This cycle repeats until all rows are read.

Now, let's go to the **second part** of this plan, which will run the SubQuery that returns the average order value for one customer.



To start with, the execution plan reads the data from the **Lazy Spool** and passes the results to the aggregate to calculate the average. Remember that the rows in the Spool operator are currently *only* the rows for "Customer o."



The **Stream Aggregate** operator will calculate the average of the **value** column, returning one row as an Output value.



The **Compute Scalar** operator, covered in an earlier chapter, will convert the result of the aggregation into a Numeric Datatype, and pass the Output row to the Nested Loops operator in Step 9.



The last **Table Spool** is used to once again read the "**Client=0**" rows from the Spool table, which will be joined with the result of the compute scalar.



The **Nested Loops** operator performs an iterative inner join; in this case, for each row returned by the computed scalar, it scans the Spool table and returns all rows that satisfy the condition of the join. Specifically, it returns the rows where the **Value** column in the Spool table is lower than the value calculated in the aggregation.

Tip

*If you create an index on the **Pedido** table covering the **Client** column and include the **Value** column, you will optimize the query because the **SORT** operator will not be necessary, and it costs 62% of the whole query.*

We saw that the Query Optimizer can use the Lazy Spool operator to optimize some queries by avoiding having to read the same values multiple times. Because SQL Server uses the Spool Lazy, the SQL works with just one chunk of the data in all operations, as opposed to having to constantly fetch new data with each iteration. Clearly, that translates into a great performance gain.

Chapter 7: Spools – Non-Clustered Index Spool



Non-Clustered Index Spool

"Indexes, Indexes, Indexes..." to misquote a certain prominent Microsoft employee. Indexing is a key issue when we are talking about databases and general performance problems, and so it's time to feature the **Non-Clustered Index Spool**. It's important that you read the earlier chapter on Lazy Spool before you get too deeply into this chapter.

The **Index Spool** is used to improve the read performance of a table which is not indexed and, as with other types of Spool operators, it can be used in a "Lazy" or an "Eager" manner. So, when SQL Server needs to read a table that is not indexed, it can choose to create a "temporary index" using the Spool, which can result in a huge performance improvement in your queries. To get started with understanding Index Spool, we'll use the usual table, called **Pedido** ("Order" in Portuguese). The following script will create a table and populate it with some garbage data:

```
USE tempdb
GO
IF OBJECT_ID('Pedido') IS NOT NULL
    DROP TABLE Pedido
GO
CREATE TABLE Pedido (ID#160;INT IDENTITY(1,1),
    Cliente#160;INT NOT NULL,
    Vendedor#160;VARCHAR(30) NOT NULL,
    Quantidade SmallInt NOT NULL,
    Valor#160;Numeric(18,2) NOT NULL,
    Data#160;DATETIME NOT NULL)
GO
CREATE CLUSTERED INDEX ix ON Pedido(ID)
GO
```

Chapter 7: Spools – Non-Clustered Index Spool

```
DECLARE @I SmallInt
SET @I = 0
WHILE @I < 50
BEGIN
    INSERT INTO Pedido(Cliente, Vendedor, Quantidade, Valor, Data)
    SELECT ABS(CheckSUM(NEWID()) / 100000000),
           'Fabiano',
           ABS(CheckSUM(NEWID()) / 100000000),
           ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
           GETDATE() - (CheckSUM(NEWID()) / 1000000)
    INSERT INTO Pedido(Cliente, Vendedor, Quantidade, Valor, Data)
    SELECT ABS(CheckSUM(NEWID()) / 100000000),
           'Amorim',
           ABS(CheckSUM(NEWID()) / 100000000),
           ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
           GETDATE() - (CheckSUM(NEWID()) / 1000000)
    INSERT INTO Pedido(Cliente, Vendedor, Quantidade, Valor, Data)
    SELECT ABS(CheckSUM(NEWID()) / 100000000),
           'Coragem',
           ABS(CheckSUM(NEWID()) / 100000000),
           ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
           GETDATE() - (CheckSUM(NEWID()) / 1000000)
    SET @I = @I + 1&#160;
END
SET @I = 0
WHILE @I < 2
BEGIN
    INSERT INTO Pedido(Cliente, Vendedor, Quantidade, Valor, Data)
    SELECT Cliente, Vendedor, Quantidade, Valor, Data
    FROM Pedido
    SET @I = @I + 1&#160;
END
GO&#160;
SELECT *
FROM Pedido Ped1
WHERE Ped1.Valor > (
    SELECT AVG(Ped2.Valor)
    FROM Pedido AS Ped2
    WHERE Ped2.Data < Ped1.Data)
```

Below is what the data looks like in SSMS:

	ID	Cliente	Vendedor	Quantidade	Valor	Data
1	1	18	Fabiano	107	401.88	2005-09-07 20:46:12.290
2	2	8	Amorim	11	1133.91	2008-03-03 20:46:12.310
3	3	1	Coragem	35	1224.57	2007-12-04 20:46:12.313
4	4	3	Fabiano	158	1319.03	2007-07-01 20:46:12.317
5	5	17	Amorim	64	591.47	2013-04-27 20:46:12.317
6	6	8	Coragem	27	5.38	2006-10-10 20:46:12.327
7	7	4	Fabiano	23	443.29	2005-06-07 20:46:12.423
8	8	9	Amorim	152	543.02	2007-12-17 20:46:12.423
9	9	8	Coragem	5	965.80	2008-09-15 20:46:12.423
10	10	17	Fabiano	209	762.74	2011-07-19 20:46:12.423
11	11	18	Amorim	56	1588.25	2011-10-07 20:46:12.427

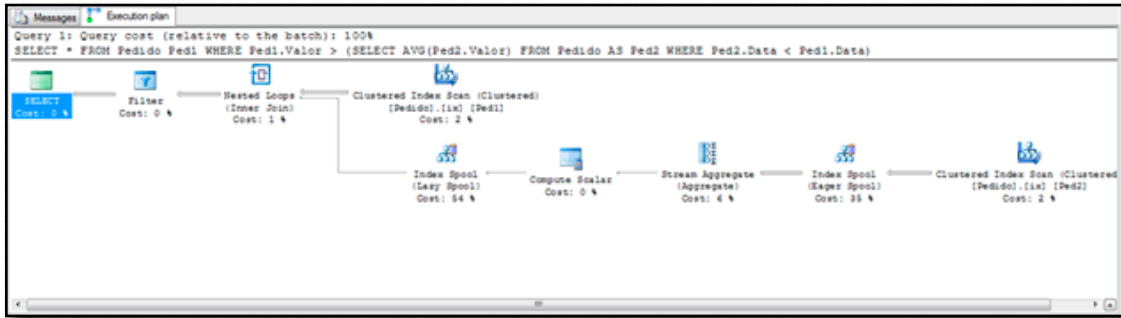
Query executed successfully.

To help us understand the Index Spool, I've written a query that returns all orders with a sale value higher than the average, as compared to all sales before the date of the order in question:

```
SELECT *
FROM Pedido Ped1
WHERE Ped1.Valor > (
    SELECT AVG(Ped2.Valor)
    FROM Pedido AS Ped2
    WHERE Ped2.Data < Ped1.Data)
```

Before we see the execution plan, let's make sure we understand the query a little better. The SubQuery returns the average value of all sales (**AVG(Ped2.Valor)**) dated before the order we're comparing them to. After that, the average is compared with the principal query, which determines whether the sale value in question is actually bigger than

the average. You'll notice that this query has a very similar form to the example in the previous chapter. So, now we have got the following execution plan:



```
--Filter(WHERE:([Pedido].[Valor] as [Ped1].[Valor]>[Expr1004]))
  |--Nested Loops(Inner Join, OUTER REFERENCES:([Ped1].[Data]))
    |--Clustered Index Scan(OBJECT:([Pedido].[PK_Pedido] AS [Ped1]))
      |--Index Spool(SEEK:([Ped1].[Data]=[Pedido].[Data] as [Ped1].[Data]))
        |--Compute Scalar(DEFINE:([Expr1004]=CASE WHEN [Expr1011]=(0) THEN
NULL ELSE [Expr1012]/CONVERT_IMPLICIT(numeric(19,0),[Expr1011],0) END))
          |--Stream Aggregate(DEFINE:([Expr1011]=Count(*),
[Expr1012]=SUM([Pedido].[Valor] as [Ped2].[Valor]))
            |--Index Spool(SEEK:([Ped2].[Data] < [Pedido].[Data] as
[Ped1].[Data]))
              |--Clustered Index Scan(OBJECT:([Pedido].[PK_Pedido]
AS [Ped2]))
```

This is a very interesting plan, and, as we can see, the Optimizer chose to use two Index Spool operators; one working as an Eager and the other as a Lazy spool. We can also see that, after the Nested Loops, SQL Server uses the Filter operator to select just the rows that satisfy the WHERE condition (**WHERE Ped1.Valor > ...**).

This execution plan is actually simpler than it looks; first, the Clustered Index Scan reads the rows in the **Pedido** table, returning the **Data** and **Valor** (Date and Value) columns to the Eager Index Spool. With these rows, the Optimizer uses the Index Spool to create a

temporary non-clustered index on **Data** and **Valor**, and as it is an Eager spool, it will read all the rows from the clustered scan to create the index.

Quick Tip

If you create an index on the Pedido table covering Data and Valor, you will optimize the query because the operator Index Spool (Eager) will not be necessary.

After that, the optimizer will calculate the average value of the sales, using the following rule: for each row of **Ped1**, the optimizer computes the average of any orders where the **Ped2.Data** is lower than **Ped1.Data** (i.e. the average of any orders which have a date earlier than the order in the given row of **Ped1**). To do this, SQL Server uses the Stream Aggregate and the Compute Scalar operators, in a manner similar to that discussed in the previous chapter.

I'll explain the Index Spool (Lazy) in just a moment, but for now I'll just say that it optimizes the Nested Loops join, which is joining the average calculated by the sub-query to the **Ped1.Data** column, and the result, as I mentioned, is then filtered to complete the query.

Now, let's look at what makes the Index Spool (Lazy) operator special. When SQL Server needs to read a value that it knows is repeated many times, then it can use a Spool to avoid having to do the same work each time it needs to find that value. For instance, suppose that the date column has a high [DENSITY](#) (i.e. it contains a lot of duplicated values); this will mean that SQL Server will have to do the same calculation more than once, since the Nested Loops operator will process the join row by row. However, if the value passed as a condition to the join is equal to a value that has already been calculated, you clearly shouldn't need to recalculate the same result each time. So how can we reuse the value that has already been found?

This is exactly what the Non-Clustered Index Spool operator (Lazy), is designed to do – optimize the process of the Join. It is optimized to predict precisely the case that I've

described above, so that a value that has already been calculated will not be *recalculated*, but instead read from the index, which is cached (**TempDb**). So, from the point of view of the Spool, it is very important to know if the required value still needs to be calculated (rebind), or has already *been* calculated (rewind). Now that this simple example has illustrated that point, it's time to dive a little deeper.

Understanding rebind and rewind

First of all you need understand that all types of Spool operators use temporary storage to cache the values used in the execution plan, although this temporary storage is truncated for each new read of the operator. This means that, if I use a Lazy Spool to calculate an aggregation and keep this calculated value in cache, I can use this cached data in many parts of the plan, and potentially work with just single chunk of data in all the plan's steps. However, to do this, we need reset the cache for each newly calculated value , otherwise by the end of the plan we'll be working with the whole table! Thus, for a spool, it is very important to distinguish between executions need the *same* value (rewinds) and executions needing an *different/new* value (rebinds).

A rewind is defined as an execution using the same value as the immediately preceding execution, whereas a rebind is defined as an execution using a different value. I know this is a little confusing to understand for the first time, so I'll try and explain it step by step, with some code and practical examples.

Rebinds and rewinds with Table Spool (Lazy Spool)

To understand a little more about rebind and rewind, let's suppose our **Pedido** table has some rows in the **Data** (Date) column in the following order: "19831203", "19831203", "20102206" and "19831203". A representation of rewind and rebind in a table spool operator would be something like this:

Value = "19831203" – A rebind occurs, since is the first time the operator is called.

Value = "19831203" – A rewind occurs since this value was already read, and is in the spool cache.

Value = "20102206" – The value changes, so the cache is truncated and a rebind occurs, since is the value "20102206" is not in the cache.

Value = "19831203" – A rebind occurs again, since the actual value in cache is "20102206", and the value that was read in step 1 was truncated in the step 3.

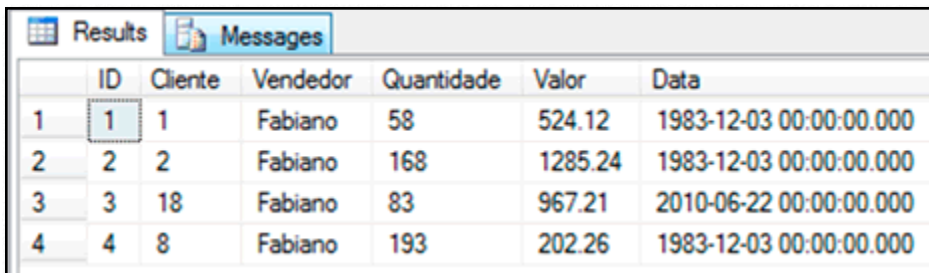
So our final numbers are **three rebinds** (steps 1, 3 and 4) and just **one rewind** (step 2). To show this in a practice, I've written a script to repopulate the table **Pedido** with four rows, exactly as I've mentioned above.

```
USE tempdb
GO
    TRUNCATE TABLE Pedido
GO
SET IDENTITY_INSERT Pedido ON
INSERT INTO Pedido(ID, Cliente, Vendedor, Quantidade, Valor, Data)
SELECT 1,
    ABS(CheckSUM(NEWID()) / 100000000),
    'Fabiano',
    ABS(CheckSUM(NEWID()) / 10000000),
    ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
```

Chapter 7: Spools – Non-Clustered Index Spool

```
'19831203'
INSERT INTO Pedido(ID, Cliente, Vendedor, Quantidade, Valor, Data)
SELECT 2,
       ABS(CheckSUM(NEWID()) / 100000000),
       'Fabiano',
       ABS(CheckSUM(NEWID()) / 100000000),
       ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
       '19831203'
INSERT INTO Pedido(ID, Cliente, Vendedor, Quantidade, Valor, Data)
SELECT 3,
       ABS(CheckSUM(NEWID()) / 100000000),
       'Fabiano',
       ABS(CheckSUM(NEWID()) / 100000000),
       ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
       '20100622'
INSERT INTO Pedido(ID, Cliente, Vendedor, Quantidade, Valor, Data)
SELECT 4,
       ABS(CheckSUM(NEWID()) / 100000000),
       'Fabiano',
       ABS(CheckSUM(NEWID()) / 100000000),
       ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
       '19831203'
SET IDENTITY_INSERT Pedido OFF
GO
```

This is what the data looks like in SSMS:



	ID	Cliente	Vendedor	Quantidade	Valor	Data
1	1	1	Fabiano	58	524.12	1983-12-03 00:00:00.000
2	2	2	Fabiano	168	1285.24	1983-12-03 00:00:00.000
3	3	18	Fabiano	83	967.21	2010-06-22 00:00:00.000
4	4	8	Fabiano	193	202.26	1983-12-03 00:00:00.000

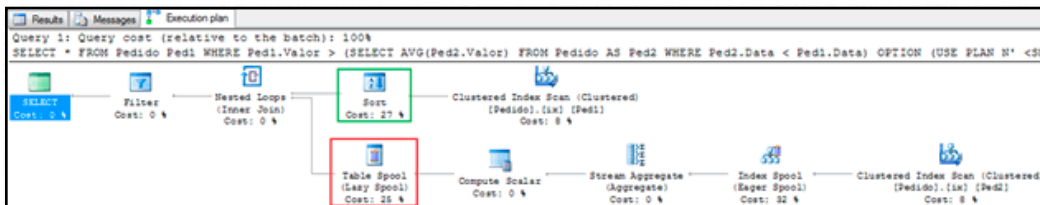
To illustrate the rebind and rewind using the Table Spool operator (which we already understand), I've written a query using the **USE PLAN** hint, to force a plan that uses the Table Spool operator. I'll omit part of the code for brevity, but you can [DOWNLOAD THE](#)

Chapter 7: Spools – Non-Clustered Index Spool

[QUERY HERE](#). Note that the following query only runs in the **TempDb** database, because the XML plan is using this database.

```
1  -- Query using an Index Spool(Eager Spool) and an Table Spool(Lazy Spool)
2  SELECT *
3      FROM Pedido Ped1
4      WHERE Ped1.Valor > (SELECT AVG(Ped2.Valor)
5                          FROM Pedido AS Ped2
6                          WHERE Ped2.Data < Ped1.Data)
7  OPTION (USE PLAN N'
8  <ShowPlanXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:
9  <BatchSequence>
10 <Batch>
11 <Statements>
12 <StmtSimple StatementCompId="1" StatementEstRows="300000" Statement
13 <StatementSetOptions ANSI_NULLS="true" ANSI_PADDING="true" ANSI
14 <QueryPlan DegreeOfParallelism="0" MemoryGrant="114232" CachedP
15 <RelOp AvgRowSize="53" EstimateCPU="0.48" EstimateIO="0" Esti
```

For the query above, we have the following execution plan:



Note that the plan above is using a Table Spool (Lazy Spool) to perform the query; for each row read in the **Pedido (Ped1)** table, SQL Server will call the table Spool to run the SubQuery. Let's look at the rebind and rewind properties, to see how many times SQL Server executes each task.

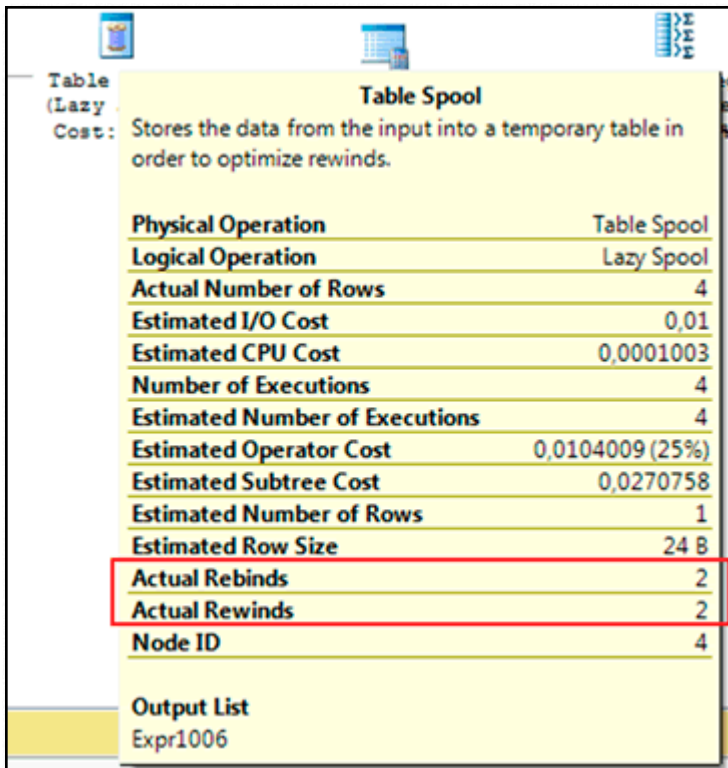


Table (Lazy)	Table Spool
Cost:	Stores the data from the input into a temporary table in order to optimize rewinds.
Physical Operation	Table Spool
Logical Operation	Lazy Spool
Actual Number of Rows	4
Estimated I/O Cost	0,01
Estimated CPU Cost	0,0001003
Number of Executions	4
Estimated Number of Executions	4
Estimated Operator Cost	0,0104009 (25%)
Estimated Subtree Cost	0,0270758
Estimated Number of Rows	1
Estimated Row Size	24 B
Actual Rebinds	2
Actual Rewinds	2
Node ID	4
Output List	Expr1006

"Hey, wait a minute, Fabiano!" I hear you exclaim. "Didn't you said that we have three rebinds and one rewind? Why SQL is showing different values?"

Sharp-eyed as ever, dear reader. Pay attention, now; do you notice that I marked the **SORT** operator with a green square? We should be asking ourselves what this is doing.

As I said earlier, to SQL Server, it is very important to distinguish between executions using the same value and executions using different values.

That means that if SQL Server reads the **Pedido** table and **Sorts** the values by Date, it will increase the chance of a rewind occurring, because the order of rows will go from:

- 19831203
- 19831203
- 20102206
- 19831203

to

- 19831203
- 19831203
- 19831203
- 20102206

That's why SQL Server only makes two rebinds and two rewinds. Pretty smart, huh?

Rebinds and rewinds with Index Spool (Lazy Spool)

Now let's see how the **Index Spool** works. Remember, index spool *doesn't* truncate its cache, even if a rebind occurs; instead it maintains a temporary index with all "rebound" rows.

So a representation of rewind and rebind in an index spool operator would be something like this:

Value = "19831203" – A rebind occurs, as this is the first time the operator is called.

Value = "19831203" – A rewind occurs, as this value was already read, and is in the spool cache.

Value = "20102206" – A rebind occurs, as the value "20102206" is not in the cache.

Value = "19831203" – A *rewind* occurs, as this value was read in step 1, and is still in the temporary index.

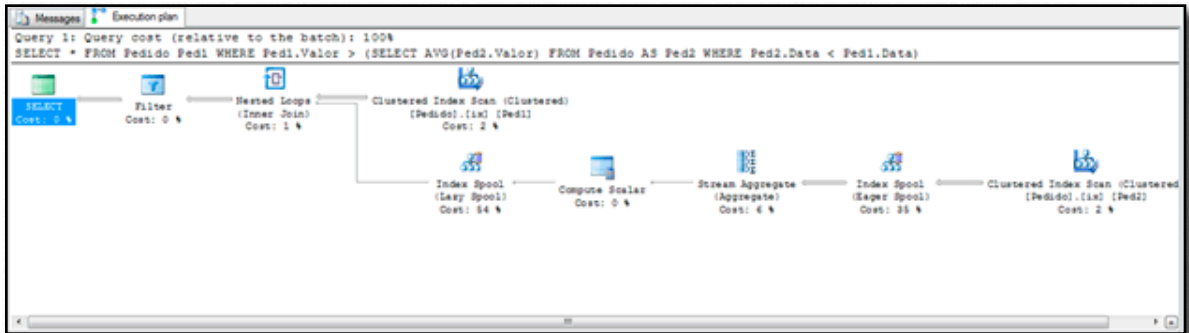
So our final numbers are **two rebinds** (steps 1 and 3) and just **two rewinds** (steps 2 and 4). The same numbers used in the plan above with the table spool operator.

To illustrate rebind and rewind using the Index Spool operator, I've written a second query using the **USE PLAN** hint to force a plan that uses the Index Spool operator. As before, I'll omit part of the code for the sake of brevity, but you can [DOWNLOAD THE QUERY HERE](#).

```
1  -- Query using an Index Spool(Eager Spool) and an Index Spool(Lazy Spool)
2  SELECT *
3      FROM Pedido Ped1
4      WHERE Ped1.Valor > (SELECT AVG(Ped2.Valor)
5                          FROM Pedido AS Ped2
6                          WHERE Ped2.Data < Ped1.Data)
7  OPTION (USE PLAN N'
8  <ShowPlanXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:
9  <BatchSequence>
10 <Batch>
11 <Statements>
12 <StmtSimple StatementCompId="1" StatementEstRows="180" StatementI
13 <StatementSetOptions ANSI_NULLS="true" ANSI_PADDING="true" ANSI
14 <QueryPlan CachedPlanSize="32" CompileTime="5" CompileCPU="5" C
15 <RelOp AvgRowSize="53" EstimateCPU="0.000288" EstimateIO="0"
```

Chapter 7: Spools – Non-Clustered Index Spool

To query above we have the following execution plan:



Note that, as intended, the plan above is using an Index Spool (Lazy Spool) to perform the query. For each row read in the **Pedido (Ped1)** table, SQL Server will call the Index Spool to run the SubQuery. Let's look at the Spool's rebind and rewind properties to see how many times SQL Server executes each task.

Index Spool	
Reformats the data from the input into a temporary index, which is then used for seeking with the supplied seek predicate.	
Index Spool (Lazy Spool) Cost:	
Physical Operation	Index Spool
Logical Operation	Lazy Spool
Actual Number of Rows	4
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0002591
Number of Executions	4
Estimated Number of Executions	4
Estimated Operator Cost	0.0041584 (17%)
Estimated Subtree Cost	0.0208333
Estimated Number of Rows	1
Estimated Row Size	24 B
Actual Rebinds	3
Actual Rewinds	1
Node ID	3
Output List	
Expr1006	
Seek Predicate	
Seek Keys[1]: Prefix: [tempdb].[dbo].[Pedido].Data = Scalar Operator([tempdb].[dbo].[Pedido].[Data] as [Ped1].[Data])	

Maybe you are wondering, "Once again, Fabiano – your prediction is different from the real values. Could you explain that please? Why is the operator's properties showing three rebinds and one rewind, when you expected two rebinds and two rewinds?"

Well, this time it's because our Microsoft friends like to confuse us. That's right, my count is correct, and the displayed properties are actually wrong. To quote from his book, [INSIDE MICROSOFT SQL SERVER 2005. QUERY TUNING AND OPTIMIZATION](#), this is what [CRAIG FREEDMAN](#) wrote about this situation:

"Note that rewinds and rebinds are counted the same way for index and nonindex spools. As described previously, a reexecution is counted as a rewind only if the correlated parameter(s) remain the same as the immediately prior execution, and is counted as a rebind if the correlated parameter(s) change from the prior execution. This is true even for reexecutions, in which the same correlated parameter(s) were encountered in an earlier, though not the immediately prior, execution. However, since lazy index spools, like the one in this example, retain results for all prior executions and all previously encountered correlated parameter values, the spool may treat some reported rebinds as rewinds. In other words, by failing to account for correlated parameter(s) that were seen prior to the most recent execution, the query plan statistics may overreport the number of rebinds for an index spool."

Summary

Generally, if you see a spool operator in your plan, you should take a closer look, because it can probably be optimized if you create the indexes properly. Doing this avoids the need for recalculations, prevents the query optimizer from having to create the indexes for you, and your query will perform better.

Chapter 8: Spools – Row Count Spool



Row Count Spool

We've now looked at the **Eager Spool**, **Lazy Spool** and **Non-Clustered Index Spool** operators, and, in this chapter, we'll be completing the set with the **Row Count Spool** Showplan operator.

Of all spool operators we've seen, I think this is the most simple. This operator just scans an input, counting how many rows are present, and returns the number of rows without any of the data they contained. It is used when it is important to check for the existence of rows, but not what data they hold. For example, if a Nested Loops operator performs a [LEFT ANTI SEMI JOIN](#) operation, and the join predicate applies to the inner input, a row count spool may be placed at the top of that input to cache the number of rows which satisfy the argument. Then, the Nested Loops operator can just use that row count information (because the actual data from the inner input is not needed) to determine whether to return the outer row or not (or rather, how many rows to return).

I know that's a little tricky to wrap your head around at first without a concrete example, so to illustrate this behavior I'll start, as always, by creating a table called **Pedido** (Order). The following script will create a table and populate it with some garbage data.

```
USE tempdb
GO
IF OBJECT_ID('Pedido') IS NOT NULL
    DROP TABLE Pedido
GO
CREATE TABLE Pedido (ID          Int IDENTITY(1,1),
                     Cliente      Int NOT NULL,
                     Vendedor     VarChar(30) NOT NULL,
                     Quantidade    SmallInt NOT NULL,
                     Valor         Numeric(18,2) NOT NULL,
                     Data          DateTime NOT NULL)
```

Chapter 8: Spools – Row Count Spool

```
GO
CREATE CLUSTERED INDEX ix ON Pedido(ID)
GO
DECLARE @I SmallInt
SET @I = 0

WHILE @I < 5000
BEGIN
    INSERT INTO Pedido(Cliente, Vendedor, Quantidade, Valor, Data)
    SELECT ABS(CheckSUM(NEWID()) / 100000000),
           'Fabiano',
           ABS(CheckSUM(NEWID()) / 100000000),
           ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
           GetDate() - (CheckSUM(NEWID()) / 1000000)

    INSERT INTO Pedido(Cliente, Vendedor, Quantidade, Valor, Data)
    SELECT ABS(CheckSUM(NEWID()) / 100000000),
           'Amorim',
           ABS(CheckSUM(NEWID()) / 100000000),
           ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
           GetDate() - (CheckSUM(NEWID()) / 1000000)

    INSERT INTO Pedido(Cliente, Vendedor, Quantidade, Valor, Data)
    SELECT ABS(CheckSUM(NEWID()) / 100000000),
           'Coragem',
           ABS(CheckSUM(NEWID()) / 100000000),
           ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
           GetDate() - (CheckSUM(NEWID()) / 1000000)

    SET @I = @I + 1
END
GO
```


Chapter 8: Spools – Row Count Spool

This is what the data looks like.

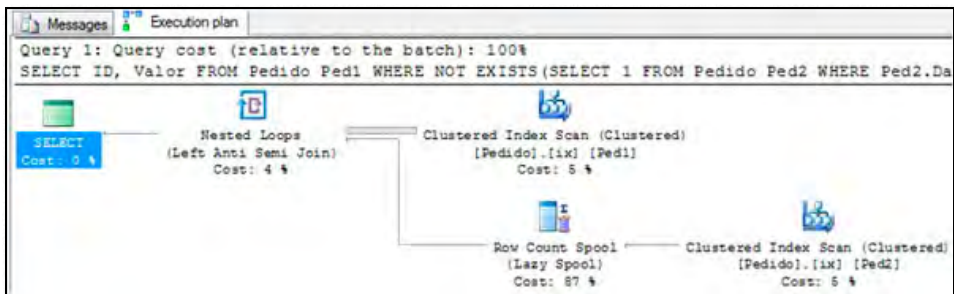
	ID	Cliente	Vendedor	Quantidade	Valor	Data
1	1	18	Fabiano	107	401.88	2005-09-07 20:46:12.290
2	2	8	Amorim	11	1133.91	2008-03-03 20:46:12.310
3	3	1	Coragem	35	1224.57	2007-12-04 20:46:12.313
4	4	3	Fabiano	158	1319.03	2007-07-01 20:46:12.317
5	5	17	Amorim	64	591.47	2013-04-27 20:46:12.317
6	6	8	Coragem	27	5.38	2006-10-10 20:46:12.327
7	7	4	Fabiano	23	443.29	2005-06-07 20:46:12.423
8	8	9	Amorim	152	543.02	2007-12-17 20:46:12.423
9	9	8	Coragem	5	965.80	2008-09-15 20:46:12.423
10	10	17	Fabiano	209	762.74	2011-07-19 20:46:12.423
11	11	18	Amorim	56	1588.25	2011-10-07 20:46:12.427

Query executed successfully.

Next, we've got a sample query which will return the ID and Value for all orders *only if* there is no order which was placed on 2009-01-01 for more than 10 items (i.e. Quantity (Quantidade) > 10).

```
SELECT ID, Valor
FROM Pedido Ped1
WHERE NOT EXISTS(SELECT 1
                  FROM Pedido Ped2
                  WHERE Ped2.Data = '20090101'
                     AND Ped2.Quantidade > 10)
OPTION (MAXDOP 1)
```

For the query above, we have the following execution plan:



As we can see, this plan is using the operator Row count Spool to check the EXISTS Sub Query, which returns just a true (some row's exist) or false (no rows exist) value, and so the actual contents of the rows doesn't matter (do you see where I'm going with this?) Let's take a closer look at what is happening.

First, the Clustered Index Scan reads all the rows in **Pedido** table using the ix index, after which all rows are passed to the Nested Loops operator to be joined with the SubQuery result. Bear in mind that this Loops operator is working as a [LEFT ANTI SEMI JOIN](#), which means that only *one side* (the outer side) of the join will be returned. In our plan, this means that only the rows read from **Ped1** will be returned to the Select operator, but for each row read in **Ped1**, the loop will look at the Row Count Spool to see if the subquery value exists (i.e. does the spool return a row or not).

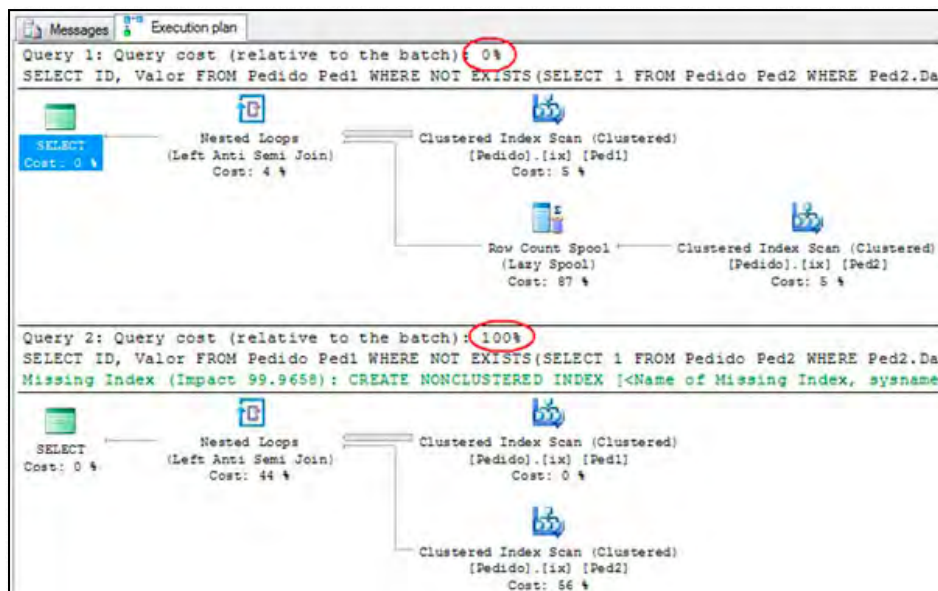
SQL Server uses the Row Count Spool operator to avoid reading the **Pedido** (Ped2) table over and over again. It calls the Clustered Index Scan at the bottom of the execution plan, which returns just one row (or not, if none satisfy the subquery). This row (or rather, it's existence) is then cached in the Spool, and this cached value is reused for each row of the **Ped1** table in the nested loop.

Why is this spool even necessary? Let's look at a comparison between a plan which uses the row count spool and a plan which doesn't. As I did in my last article, I've written an XML plan to force the Query Optimizer to use my (Row-Count-spool-less) plan, and you can download the query [here](#).

Chapter 8: Spools – Row Count Spool

```
1 SELECT ID, Valor
2   FROM Pedido Ped1
3  WHERE NOT EXISTS(SELECT 1
4                    FROM Pedido Ped2
5                     WHERE Ped2.Data = '20090101'
6                        AND Ped2.Quantidade > 10)
7 GO
8 SELECT ID, Valor
9   FROM Pedido Ped1
10  WHERE NOT EXISTS(SELECT 1
11                   FROM Pedido Ped2
12                    WHERE Ped2.Data = '20090101'
13                       AND Ped2.Quantidade > 10)
14 OPTION(USE PLAN N'<ShowPlanXML xmlns:xsi="http://www.w3.org/2003/01/XMLSchema-instance"
15         <BatchSequence>
16         <Batch>
17         <Statements>
18         <StmtSimple StatementCompId="1" StatementEstRows="1"
```

For the queries above, we have the following execution plans:



Take a look at the differences between the costs of two queries; that seems like pretty strong evidence for the usefulness of the Row Count Spool operator, doesn't it? By avoiding having to read the **Pedido** table for each row of the Inner input to the nested loop, SQL Server creates a huge performance gain. Just look at the IO results for the two queries above:

First Query using Row Count Spool

```
(15000 row(s) affected)
Table 'Pedido'. Scan count 2, logical reads 186, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

Second Query using Clustered Index Scan

```
(15000 row(s) affected)
Table 'Pedido'. Scan count 2, logical reads 1395093, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

I've nothing else to say about that, because I don't think there's much I can add to that kind of compelling evidence.

Chapter 9: Stream Aggregate



Stream Aggregate

The **Stream Aggregate** operator is very common, and it is the best way to aggregate a value, because it uses data that has been previously sorted in order to perform the aggregation quickly.

The Stream Aggregate is used to group some rows by one or more columns, and to calculate any aggregation expressions that are specified in the query. The commonest types of aggregation are: SUM, COUNT, AVG, MIN, and MAX. When you use one of these commands, you will probably see a Stream Aggregation operator being used in the query plan. The Stream Aggregation is very fast because it requires an input that has already been ordered by the columns specified in the GROUP statement. If the aggregated data is not ordered, the Query Optimizer can firstly use a Sort operator to pre-sort the data, or it can use pre-sorted data from an index seek or a scan.

We will see later in this article how a myth is born. The Stream Aggregate is a father of this myth, but don't fret, we will get there in a minute, after I've explained how the Stream Aggregate operation works.

To illustrate the Stream Aggregate behavior, I'll start as always by creating a table called **Pedido** (Order). The following script will create a table and populate it with some garbage data.

```
USE tempdb
GO

IF OBJECT_ID('Pedido') IS NOT NULL
    DROP TABLE Pedido
GO

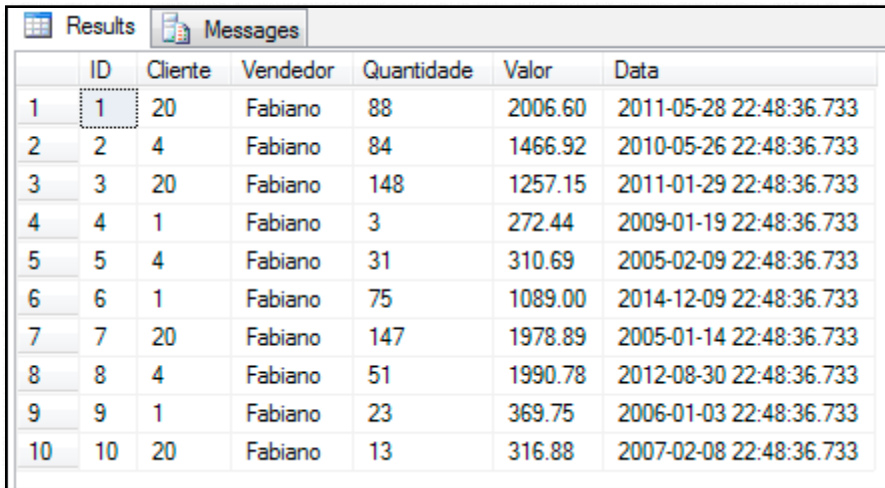
CREATE TABLE Pedido (ID INT IDENTITY(1,1) PRIMARY KEY,
    Cliente INT NOT NULL,
    Vendedor VARCHAR(30) NOT NULL,
    Quantidade SmallInt NOT NULL,
    Valor Numeric(18,2) NOT NULL,
    Data DATETIME NOT NULL)
GO

DECLARE @I SmallInt
SET @I = 0

WHILE @I < 10
BEGIN
    INSERT INTO Pedido(Cliente, Vendedor, Quantidade, Valor, Data)
        SELECT ABS(CheckSUM(NEWID()) / 100000000),
            'Fabiano',
            ABS(CheckSUM(NEWID()) / 100000000),
            ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
            GETDATE() - (CheckSUM(NEWID()) / 1000000)
    SET @I = @I + 1
END
GO

UPDATE Pedido SET Cliente = 1 WHERE ID IN (4,6,9)
UPDATE Pedido SET Cliente = 4 WHERE ID IN (8,5,2)
UPDATE Pedido SET Cliente = 20 WHERE ID IN (3,1,7,10)
GO
```

This is what the data looks like.



	ID	Cliente	Vendedor	Quantidade	Valor	Data
1	1	20	Fabiano	88	2006.60	2011-05-28 22:48:36.733
2	2	4	Fabiano	84	1466.92	2010-05-26 22:48:36.733
3	3	20	Fabiano	148	1257.15	2011-01-29 22:48:36.733
4	4	1	Fabiano	3	272.44	2009-01-19 22:48:36.733
5	5	4	Fabiano	31	310.69	2005-02-09 22:48:36.733
6	6	1	Fabiano	75	1089.00	2014-12-09 22:48:36.733
7	7	20	Fabiano	147	1978.89	2005-01-14 22:48:36.733
8	8	4	Fabiano	51	1990.78	2012-08-30 22:48:36.733
9	9	1	Fabiano	23	369.75	2006-01-03 22:48:36.733
10	10	20	Fabiano	13	316.88	2007-02-08 22:48:36.733

Let's divide the aggregations in two types, the Scalar Aggregations and the Group Aggregations.

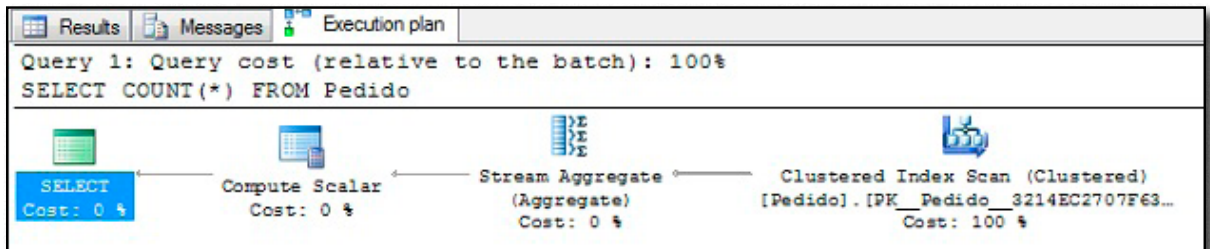
- **Scalar Aggregations** are queries that use an aggregation function, but don't have a **GROUP BY** clause, a simple sample is **SELECT COUNT(*) FROM Table**.
- **Group Aggregations** are queries that have a column specified into the **GROUP BY** clause, for instance, **SELECT COUNT(*) FROM Table GROUP BY Col1**.

Scalar aggregations

Scalar aggregations are performed using the Stream Aggregation operator. A quite simple sample is the following query that counts all rows from the **Pedido** table.

```
SELECT COUNT(*) FROM Pedido
```

For the query above, we have the following execution plan:



```
--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1004],0)))  
|--Stream Aggregate(DEFINE:([Expr1004]=Count(*)))  
|--Clustered Index Scan(OBJECT:([tempdb].[dbo].[Pedido].[PK__  
Pedido__3214EC2707F6335A]))
```

Text execution plan.

This is not a complicated plan. As we can see, the first step is to read the all rows from the Clustered Index. Then, as you can see this in the text execution plan, the Stream Aggregate performs the COUNT(*). After the COUNT, the result of the COUNT is placed into the Expr1004, and the **Compute Scalar** operator converts the Expr1004 to a Integer DataType.

You may is wondering why the Compute Scalar is needed. The answer is as follows...

The output of the Stream Aggregate is a **BigInt** value, and function COUNT is an Integer function. You may remember that, in order to count **BigInt** values, you should use the COUNT_BIG function. If you change the query above to use the COUNT_BIG, then you will see that the plan no longer uses the compute scalar operator. My great friend from SolidQ [PINAL DAVE](#) gives a very good explanation about that [HERE](#). There seems to be no performance advantage through using COUNT_BIG, though, since the casting operation of compute scalar takes very little CPU-effort.

Note that the Scalar Aggregations will always return at least one row, even if the table is empty.

Another important operation is when the Stream Aggregate is used to do two calculations; for instance, when you use the AVG function the Stream Aggregate actually computes the COUNT and the SUM, then divides the SUM by the COUNT in order to return the average.

We can illustrate this in practice. This query performs a simple AVG into the **Pedido** table.

```
SELECT AVG(Valor) FROM Pedido
```

```
|--Compute Scalar(DEFINE:([Expr1003]=CASE WHEN [Expr1004]=(0) THEN NULL ELSE  
[Expr1005]/CONVERT_IMPLICIT(numeric(19,0),[Expr1004],0) END))  
|--Stream Aggregate(DEFINE:([Expr1004]=Count(*), [Expr1005]=SUM([tempdb].  
[dbo].[Pedido].[Valor])))  
|--Clustered Index Scan(OBJECT:([tempdb].[dbo].[Pedido].[PK__  
Pedido__3214EC2707F6335A]))
```

Text Execution plan.

As we can see, the Stream Aggregate calculates the COUNT and the SUM, and then the Compute Scalar divides one value by the other. You'll notice that a CASE is used to avoid a division by zero.

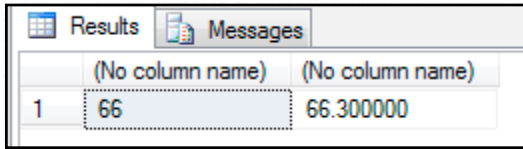
Tip

Pay very close attention that the result of the AVG will be the same type as the result of the expression contained in its argument, in our case the column specified by the query. For instance, look at the difference of the two columns of the following query.

Chapter 9: Stream Aggregate

```
SELECT AVG(Quantidade), AVG(CONVERT(Numeric(18,2),Quantidade)) FROM Pedido
```

The result of the query is the following:



	(No column name)	(No column name)
1	66	66.300000

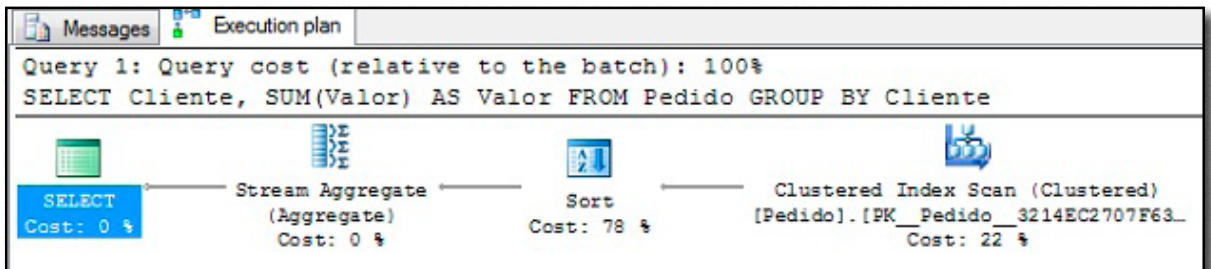
The AVG function can return **int**, **bigint**, **Decimal/Numeric**, **money** and **float** data, depending on the result of the expression.

Group Aggregations

Group Aggregations are queries that use the GROUP BY column. Let's take a quite simple query to start with. The following query is aggregating all orders by each customer.

```
SELECT Cliente,  
       SUM(Valor) AS Valor  
FROM Pedido  
GROUP BY Cliente
```

For the query above, we have the following execution plan:



Chapter 9: Stream Aggregate

```
--Stream Aggregate(GROUP BY: ([tempdb].[dbo].[Pedido].[Cliente])
DEFINE: ([Expr1003]=SUM([tempdb].[dbo].[Pedido].[Valor])))
|--Sort(ORDER BY: ([tempdb].[dbo].[Pedido].[Cliente] ASC))
|--Clustered Index Scan(OBJECT: ([tempdb].[dbo].[Pedido].[PK__
Pedido__3214EC2707F6335A]))
```

Text Execution plan.

The plan here is not too complicated: firstly, SQL Server reads all the rows from the **Pedido** table via the Clustered Index, then sorts the rows by Customer (Cliente), with the table ordered by Customers, SQL Server then starts the aggregation. For each Customer, all rows are read, row by row, computing the value of the orders. When the Customer changes, the operator returns the actual requested row (the first Customer) and starts to aggregate this new Customer. This process is repeated until all rows are read.

The following picture shows the groups by each Customer.

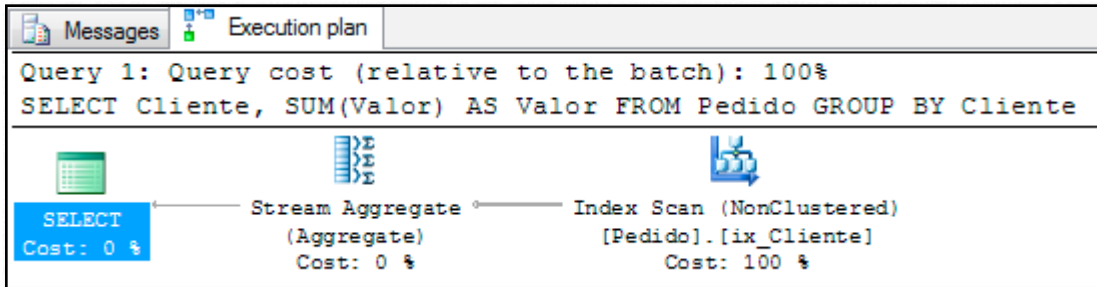
	ID	Cliente	Vendedor	Quantidade	Valor	Data
1	4	1	Fabiano	129	1716.69	2010-05-28 02:17:16.453
2	6	1	Fabiano	46	1955.55	2015-10-07 02:17:16.457
3	9	1	Fabiano	179	1147.34	2016-04-05 02:17:16.457
4	8	4	Fabiano	71	415.21	2016-03-03 02:17:16.457
5	5	4	Fabiano	17	1229.06	2011-11-27 02:17:16.453
6	2	4	Fabiano	167	1299.60	2015-01-23 02:17:16.453
7	3	20	Fabiano	17	474.06	2010-03-05 02:17:16.453
8	1	20	Fabiano	97	1833.34	2013-08-16 02:17:16.423
9	7	20	Fabiano	101	1075.61	2005-03-20 02:17:16.457
10	10	20	Fabiano	149	199.96	2012-03-10 02:17:16.457

As we can see from the execution plan, the sort operator costs 78% of the entire cost of the plan, which implies that, if we can avoid this step, we'll have a considerable gain in performance.

Let's create the index properly to see what will happen.

```
CREATE INDEX ix_Cliente ON Pedido(Cliente) INCLUDE(Valor)
```

Now let's see the execution plan.



Great. It's faster. As we can see, now the SQL Server can now take advantage of the index. It uses the ordered index by Cliente (Customer) to perform the aggregation, because the rows already are ordered by Customer, SQL doesn't need to sort the data.

A myth is born

You've probably heard from someone that you don't need to use the ORDER BY clause if you've already put the columns into the GROUP BY. For instance, in our sample, if I write:

```
SELECT Cliente,  
       SUM(Valor) AS Valor  
FROM Pedido  
GROUP BY Cliente
```

And if I want the data be returned by **Cliente**, I'll be told that I don't need to put **ORDER BY Cliente** clause into the query:

```
SELECT Cliente,  
        SUM(Valor) AS Valor  
FROM Pedido  
GROUP BY Cliente  
ORDER BY Cliente
```

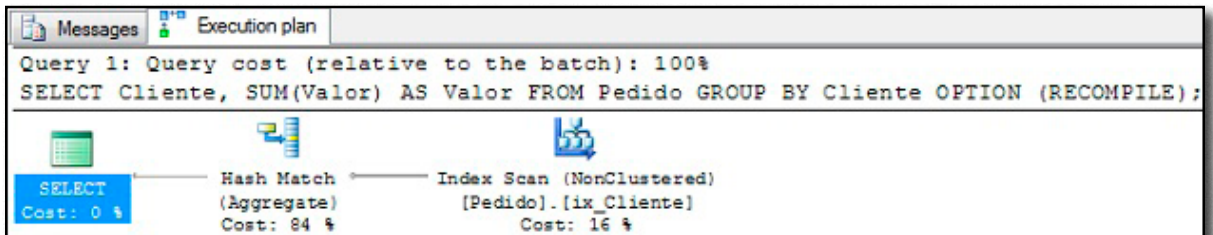
Because Stream Aggregate needs to have the data ordered by the columns specified into the GROUP BY, the data will generally be returned into the GROUP BY order. But this is **not always true**.

Since SQL Server 7.0, the Query Optimizer has two options to perform an aggregation: either by using the Stream Aggregate or a Hash Aggregate.

I'll cover the Hash Aggregate in a next opportunity, but by now, to see the Hash Aggregate in action, we could use a very good tip from [PAUL WITHE](#) (you should read all his blog posts), to disable the rule used by Query Optimizer to create the execution plan using the Stream Aggregate. In other words, I'll tell the Query Optimizer that the Stream Aggregation is not an option to create the execution plan.

```
DBCC TRACEON (3604);  
DBCC RULEOFF('GbAggToStrm');  
GO  
SELECT Cliente,  
        SUM(Valor) AS Valor  
FROM Pedido  
GROUP BY Cliente  
OPTION (RECOMPILE);  
GO  
DBCC RULEON('GbAggToStrm');
```

For the query above, we have the following execution plan:



We can now see that the Query Optimizer creates a plan that uses the Hash Match (Aggregate) operator to perform the aggregation and it doesn't require any SORT operator; but that means that the rows will be returned in a random order.

So the truth is that if you need the data ordered by some column, you should, please, **always** put the column in the ORDER BY clause.

If the Query Optimizer chooses to use the Hash Match instead the Stream Aggregate, the data may not be returned in the expected order.

Chapter 10: SORT

The **SORT** operator is quite simple, and my intention here is to explain the use of this operator, and demonstrate how you can improve the performance of your queries by avoiding the sorting operation.

As we can see by its name, the **SORT** operator sorts all rows received by the operator into order. It's important to you keep in mind that, in some cases, the SORT operation is performed in the temporary database **TempDb**. Because **TempDb** is used for all databases within the SQL Server Instance, this can lead to **TempDb** becoming a bottle-neck and thereby affecting performance.

It is surprising how often developers and DBAs take the SORT operation for granted, even though it is can be expensive in terms of CPU and I/O. You should always pay due attention to this process, and check to make sure that it does not appear in the query plan unless it is necessary.

SORT into execution plans

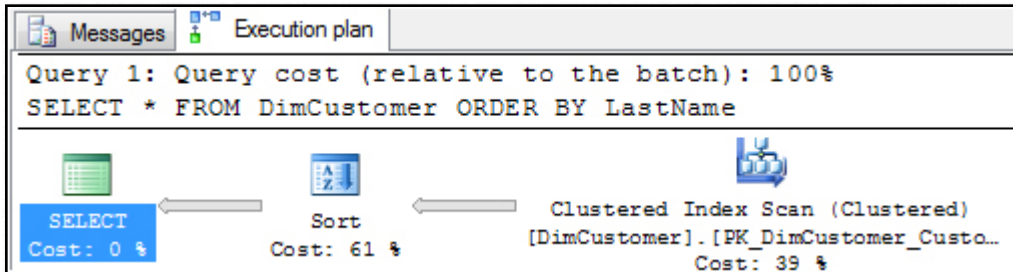
To see the SORT in practice, let's start with a simple query that uses the table **DimCustomer** from the Microsoft Sample **AdventureWorksDW** database.

The following query does a simple **SELECT** operation, ordering the result by **LastName**. Because the table is not so large, the Sort will be performed in memory. That means that the data doesn't need to be written to disk.

```
SELECT *  
FROM DimCustomer  
ORDER BY LastName
```

Chapter 10: SORT

For the query above, we have the following execution plan:

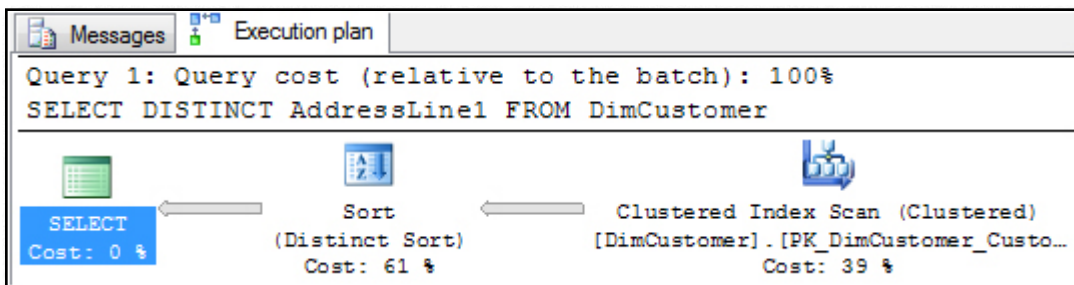


Here we can see that the Sort is performed so as to order the data by **LastName**. Because I didn't specify the word **ASC** or **DESC**, the result will be ordered ascending; which is the default option.

SORT can also be used to remove duplicate rows, in other words, perform a **DISTINCT** operation. For instance, consider the following query:

```
SELECT DISTINCT AddressLine1
FROM DimCustomer
```

From this, we have the following execution plan:

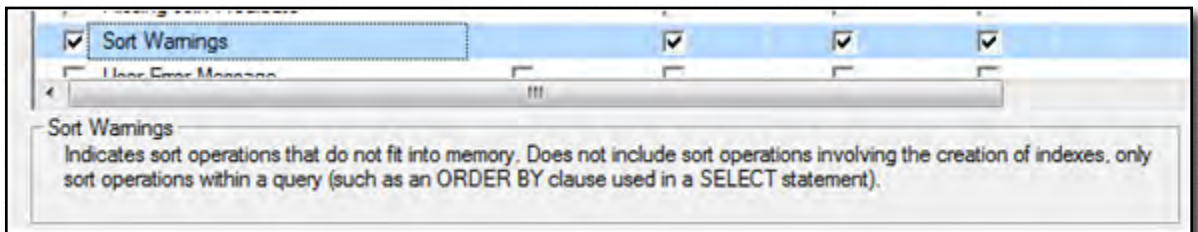


Here the SORT is a **Distinct SORT**. You'll notice that I didn't specify anything in an **ORDER BY** clause; I'm just getting a distinct list of **AddressLine1**.

SORT in memory/disk

It's important to know that the SORT operation is a very expensive task, and it usually requires a lot of memory. When the query plan is created, SQL Server reserves the memory to perform the SORT in a "grant;" but sometimes this sort is written to disk to run in the **TempDb** database. To understand more about SQL Server memory grant, look at [THIS ARTICLE](#).

To see when the SORT operation is being done in **TempDb**, we can use the SQL Server Profiler to capture an event called "Sort Warnings."



Many of these warnings are an indication that you need more memory. If your application requires the use of many disk-based SORT operations, you can also check physical location of the **TempDb** database so as to be sure that it is using the best storage subsystem available.

Note

Just adding more memory is not the only option, and maybe not the easiest. If you want to know more about the sort warning you should read [THE COMMENTS BELOW THE SIMPLE-TALK WEB VERSION OF THIS ARTICLE](#). Thanks a lot to Holger, Chris, and Celko for all comments.

How to avoid SORT operations

The easiest way to avoid a SORT is by creating an Index. As we know, indexes are ordered by the columns so that , if you create an index covering your query, the Query Optimizer identifies this index and uses it to avoid a SORT operation. Let's look at a sample using the same query used before.

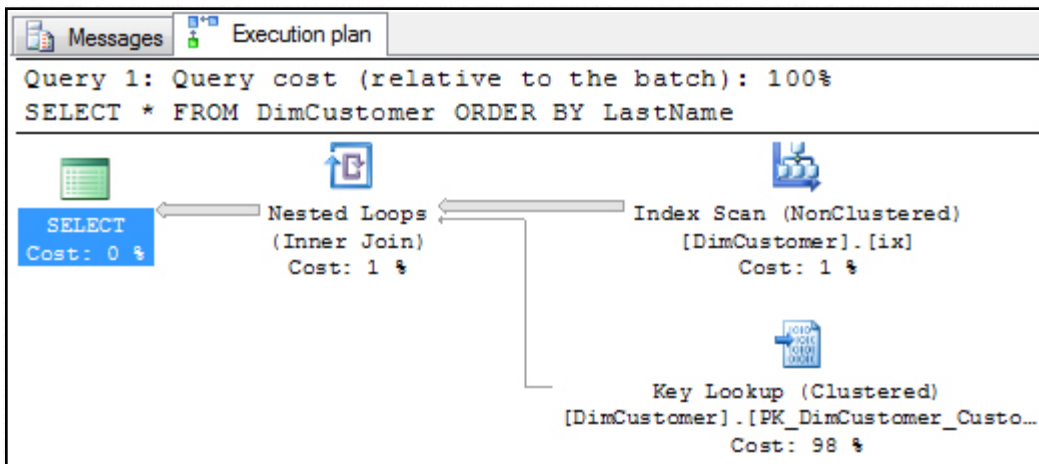
The following command creates an index using the column **LastName**.

```
CREATE INDEX ix ON DimCustomer (LastName)
```

Now, let's see the execution plan of the same query that we've already used.

```
SELECT *  
FROM DimCustomer  
ORDER BY LastName
```

For this query, we have the following execution plan:



This time, the Query Optimizer has chosen to read the data from the **ix** index and then uses a Key Lookup to read the other columns to the clustered index. The query optimizer is smart enough to understand that the index is ordered by the column **LastName** and there is no reason to order the data again.

Another very common usage of a sorting operation is when developers create reports with a lot of options to allow the end-users to choose how to sort the result. In this case, the **ORDER BY** clause can often be avoided in your query. It is usually better to sort the data within the client application, not the server.

SQL Server also can use the SORT operator for other operations, for instance, to be able to use the **Merge** algorithm, and you can also avoid this by creating the proper index.

Chapter 11: Merges – Merge Join

For quite a while, I wanted to talk about the **Join** operators (**Loop**, **Merge** and **Hash**), but I always wondered whether, if I wrote on this subject, you'd find it interesting, since there are already a lot of blog posts and articles about them. Despite that, I just couldn't miss having those operators in my series, so I really hope that you like my approach on this topic. I'll start by featuring the **Merge Join** operator. I don't have to follow the usual order, Loop, Merge and Hash, so I'll start with the Merge.

Introduction

Once when I was presenting a lecture on this subject, I asked the audience if there was anyone in the room who had a database with only one table. To my surprise, one person raised his hand – I was stunned into silence, to the amusement of the audience.

Simply put, a join is an operation that links one table to another, and SQL Server can use three algorithms to perform this operation, Loop, Merge and Hash.

The **Merge Join** performs an inner join, an outer join or, in some cases, even a union operation. The **Merge Join** is very fast because it requires that both inputs are already sorted by the respective key columns. To do an analogy with the joins I'll use the same example that I like to show in my presentations.

To illustrate the Merge Join behavior, I'll start by creating two tables, one called **Cursos** ("Courses" in Portuguese) and one table called **Alunos** (Students). The following script will create the tables and populate them with some garbage data.

Chapter 11: Merges – Merge Join

```
USE tempdb
GO
IF OBJECT_ID('Alunos') IS NOT NULL
BEGIN
    DROP TABLE Alunos
    DROP TABLE Cursos
END
GO
CREATE TABLE Cursos (ID_Cursos INT PRIMARY KEY, Nome_Curso VARCHAR(80))
CREATE TABLE Alunos (ID_Alunos INT PRIMARY KEY, Nome_Aluno VARCHAR(80), ID_Cursos
INT)
Y, Nome_Curso VARCHAR(80))
GO

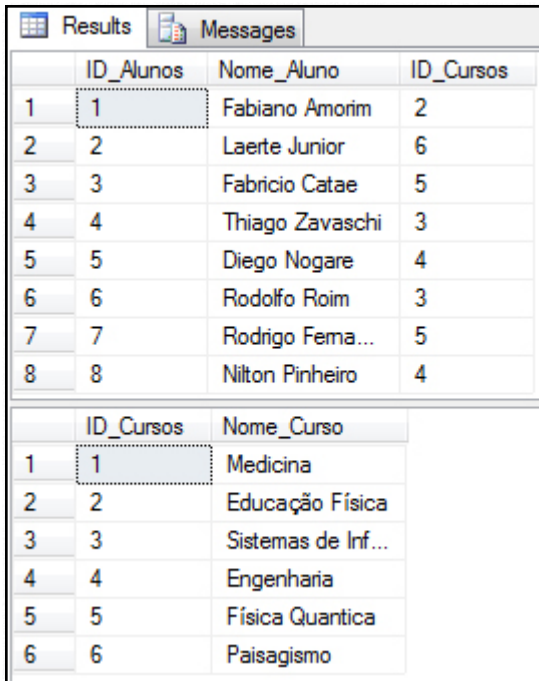
INSERT INTO Cursos (ID_Cursos, Nome_Curso) VALUES (1, 'Medicina')
INSERT INTO Cursos (ID_Cursos, Nome_Curso) VALUES (2, 'Educação Física')
INSERT INTO Cursos (ID_Cursos, Nome_Curso) VALUES (3, 'Sistemas de Informação')
INSERT INTO Cursos (ID_Cursos, Nome_Curso) VALUES (4, 'Engenharia')
INSERT INTO Cursos (ID_Cursos, Nome_Curso) VALUES (5, 'Física Quântica')
INSERT INTO Cursos (ID_Cursos, Nome_Curso) VALUES (6, 'Paisagismo')
GO

INSERT INTO Alunos (ID_Alunos, Nome_Aluno, ID_Cursos) VALUES (1, 'Fabiano Amorim',
2)
INSERT INTO Alunos (ID_Alunos, Nome_Aluno, ID_Cursos) VALUES (2, 'Laerte Junior',
6)
INSERT INTO Alunos (ID_Alunos, Nome_Aluno, ID_Cursos) VALUES (3, 'Fabricio Catae',
5)
INSERT INTO Alunos (ID_Alunos, Nome_Aluno, ID_Cursos) VALUES (4, 'Thiago
Zavaschi', 3)
INSERT INTO Alunos (ID_Alunos, Nome_Aluno, ID_Cursos) VALUES (5, 'Diego Nogare',
4)
INSERT INTO Alunos (ID_Alunos, Nome_Aluno, ID_Cursos) VALUES (6, 'Rodolfo Roim',
3)
INSERT INTO Alunos (ID_Alunos, Nome_Aluno, ID_Cursos) VALUES (7, 'Rodrigo
Fernandes', 5)
INSERT INTO Alunos (ID_Alunos, Nome_Aluno, ID_Cursos) VALUES (8, 'Nilton
Pinheiro', 4)
```

Chapter 11: Merges – Merge Join

This is what the data looks like:

```
SELECT * FROM Alunos
SELECT * FROM Cursos
```



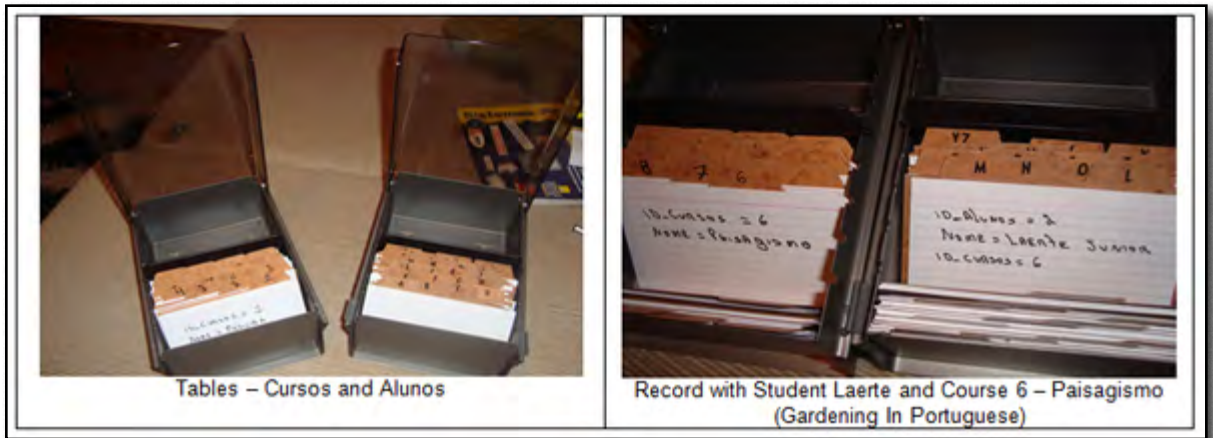
The screenshot shows a database interface with two tabs: 'Results' and 'Messages'. The 'Results' tab is active, displaying two tables. The first table, 'Alunos', has columns 'ID_Alunos', 'Nome_Aluno', and 'ID_Cursos'. The second table, 'Cursos', has columns 'ID_Cursos' and 'Nome_Curso'. The data is as follows:

	ID_Alunos	Nome_Aluno	ID_Cursos
1	1	Fabiano Amorim	2
2	2	Laerte Junior	6
3	3	Fabricio Catae	5
4	4	Thiago Zavaschi	3
5	5	Diego Nogare	4
6	6	Rodolfo Roim	3
7	7	Rodrigo Fema...	5
8	8	Nilton Pinheiro	4

	ID_Cursos	Nome_Curso
1	1	Medicina
2	2	Educação Física
3	3	Sistemas de Inf...
4	4	Engenharia
5	5	Física Quântica
6	6	Paisagismo

As you can see, in the table **Alunos** I've a column, **ID_Cursos**, that links to the course that the student is signed up for.

Now you know about the tables and the data. Before we go on with the script stuff, I'd like to show you a picture with the same tables but using something that we are more used to seeing in real life.



I'm sure you already saw one of these card indexes in a movie rental store, a library, your dental clinic, or even in your college. Before databases became commonplace, this kind of material was very popular and very useful, but in the past few years we have evolved IT systems (thank goodness) and now we don't need to store our customers' info in this kind of table since we have SQL Server to do that job.

You'll notice that the card index **Cursos** is sorted by **ID_Cursos** and the card index **Alunos** is ordered by **Name**.

If you are using the card index and I ask you what Fabiano's course is, what is the first thing you have to do? First of all you have to scan the Aluno's card index to find the record with the info about the student, Fabiano, then you have to look at the field **ID_Cursos** to know the number of the course. Finally you have to search in the card index **Cursos** to find the course relative to the ID you found in the Aluno's card.

You just did a join!

Now, what if I change my question to, "Please give me the details of all students and all their courses"? What would you have to do then?

One method would entail starting to scan the card index of courses and, for each course you read, you scan the card index for the students that relate to the course, and you would repeat this until all the courses are read. That means that, for each course, you would have to scan the whole card index with the students' info.

Now, can you tell me what will happen if, before we start the process of reading the students card index, I manually re-order the student card index from the Student Name to **ID_Cursos** order?

If I do this, I can just read one Course (which is already sorted by **ID_Cursos**) and read just a small part of the Students card index, because now the **ID_Cursos = 1** will be the first occurrence I'll read in the Course card, and the **ID_Course = 1** will also be the first card in the Students card index. Did you understand that you don't need to scan the Students card index any more?

This is exactly what SQL Server does with the Merge Join algorithm, it takes advantage of the column orders and performs a very fast join. It will read the rows and, if one of inputs get to the end, it just stops the read, because that means that all possible rows have been joined.

SORT Merge Join

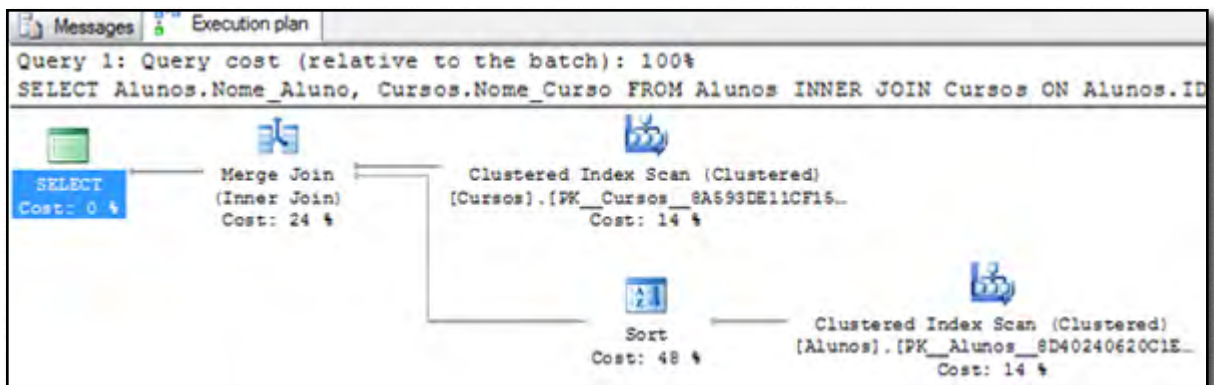
As you can see, both sides of the join must be sorted by the key columns before you can use the Merge Join. Sometimes the Query Optimizer can explicitly force a Sort operator to precede the Merge Join operator, or it can read the rows from an index that is already sorted in the expected order.

Chapter 11: Merges – Merge Join

The following query is using a hint to force the use of the Merge Join operator.

```
SELECT Alunos.Nome_Aluno, Cursos.Nome_Curso
FROM Alunos
INNER JOIN Cursos
ON Alunos.ID_Cursos = Cursos.ID_Cursos
OPTION (MERGE JOIN)
```

For the query above, we have the following execution plan:



You can see in the execution plan that the Query Optimizer explicitly sorted the table **Alunos** by the **ID_Cursos** column so as to use the Merge Join, and the highest cost of the query is the SORT operator.

We could avoid this step by just creating an index into the table **Alunos** by the **ID_Cursos** column. The following creates the index:

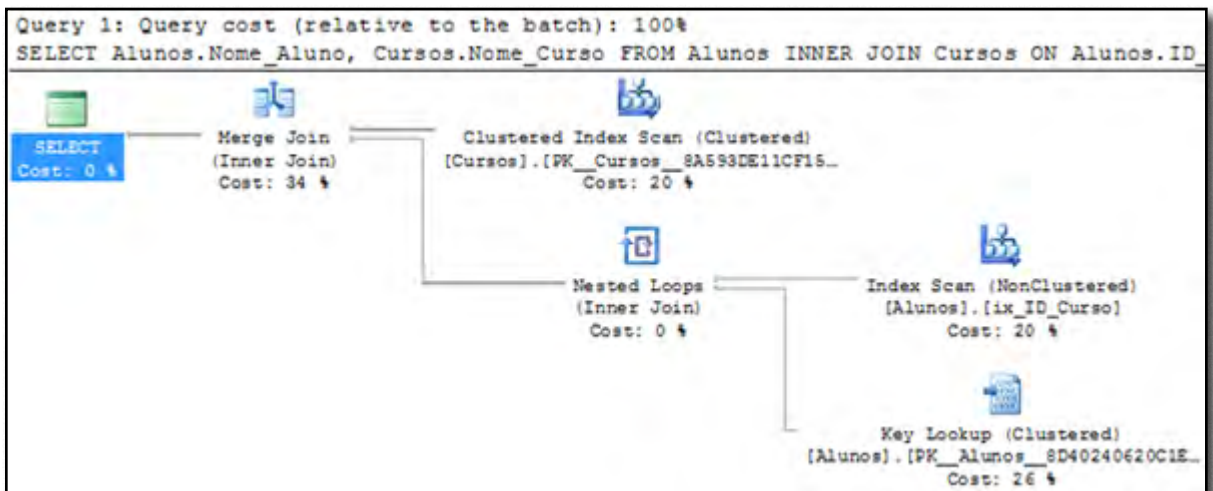
```
CREATE NONCLUSTERED INDEX ix_ID_Curso ON Alunos (ID_Cursos)
```

Now let's look at the execution plan again.

Chapter 11: Merges – Merge Join

```
SELECT Alunos.Nome_Aluno, Cursos.Nome_Curso
FROM Alunos
INNER JOIN Cursos
ON Alunos.ID_Cursos = Cursos.ID_Cursos
OPTION (MERGE JOIN)
```

For the query above we have the following execution plan:



Wow, no! Now the plan is using the index and we don't need the Sort any more, but we got a plan using a Key Lookup operator. Yes, that's happened because we are asking for the **Cursos.Nome_Curso** column, and we didn't include the column into the index, so let's try creating a new index including the column **Nome_Curso**.

```
CREATE NONCLUSTERED INDEX ix_ID_Curso_Nome_Aluno ON Alunos(ID_Cursos) INCLUDE
(Nome_Aluno)
```

Let's look at the execution plan again.

```
SELECT Alunos.Nome_Aluno, Cursos.Nome_Curso
FROM Alunos
INNER JOIN Cursos
ON Alunos.ID_Cursos = Cursos.ID_Cursos
OPTION (MERGE JOIN)
```

Now, everything is perfect.

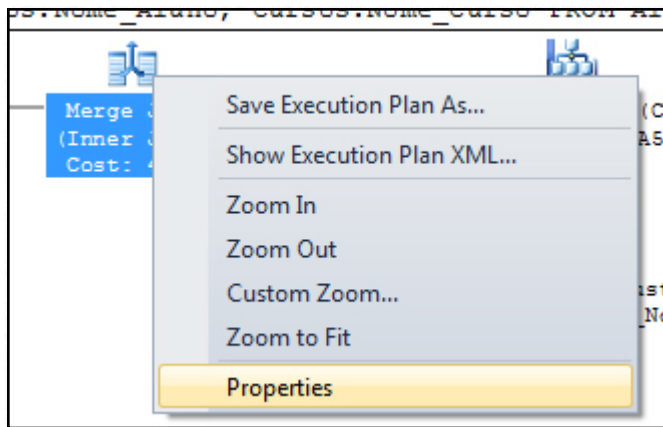
Residual predicate

Another thing that the Query Optimizer can do, using the **Merge Join** operator, is to filter a new predicate into the Merge that is not part of the join of the tables.

For instance:

```
SELECT Alunos.Nome_Aluno, Cursos.Nome_Curso
FROM Alunos
LEFT OUTER JOIN Cursos
ON Alunos.ID_Cursos = Cursos.ID_Cursos
AND Alunos.Nome_Aluno LIKE 'F%'
OPTION (MERGE JOIN)
```

After checking whether rows from **Alunos** join with **Cursos**, SQL Server can also check whether the **Nome_Aluno** starts with the letter "F." To see the residual predicate you can click with the right button on the Merge Join operator and then select **Properties**.



Physical Operation	Merge Join
Residual	s].[Nome_Aluno] like 'F%
Where (join columns)	([tempdb].[dbo].[Alunos]

One to Many and Many to Many Merge Join

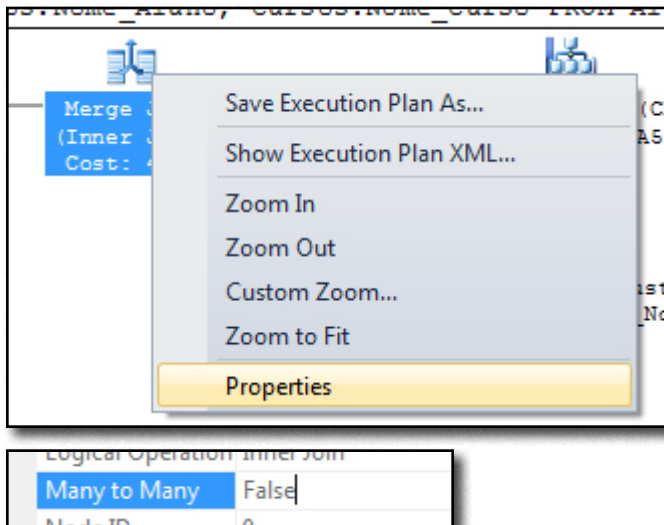
To show you this concept of One to Many, I would like to present a pseudo code of the merge join algorithm. All credits to Craig Freedman who wrote this code into his book.

The code is the following:

```
Get first row Cursos from input 1
Get first row Alunos from input 2
While not at the end of either input
Begin
    If Cursos joins with Alunos
    Begin
        Output(Cursos, Alunos)
        Get next row Alunos from input 2
    end
    else if Cursos < Alunos
        get next row Cursos from input 1
    else
        get next row Alunos form input 2
    end
end
```

This is a very interesting code and this code works just for the One to Many joins (read more about it [HERE](#)).

When the Merge Join operator was executed using the Many to One algorithm, you can see what happened when the property **Many to Many** of the operator is false. To see the **Many To Many** property, click with the right button into the Merge Join operator and then select **Properties**.



To illustrate the pseudo code, I don't know of any better way than show you step by step, so I created the following video (sorry about my accent) so as to explain the code.

Alternatively, download the video here: [SQL SERVER MERGE JOIN \(.WMV VERSION\)](#).

Chapter 12: Merges – Merge Interval



Merge Interval

In the previous chapter, I wrote about the **Merge Join** operator. It is now time to feature another kind of merge, the **Merge Interval** operator.

I was recently working with a customer in Finland to optimize some queries, when I saw this operator in the execution plan. Because this is not very well documented, I'll try to cover all aspects and bugs related to this operator (a.k.a. **iterators**).

In short, this is used to remove duplicated predicates in a query, and to find possible overlapping intervals in order to optimize these filters so as to avoid scanning the same data more than once.

As always, I completely understand that this is not as simple as I've just stated. Don't worry if you have to read what I wrote more than three times to understand what I mean – I'll be going deep into this subject, step by step, so as to make it easier to understand.

Creating sample data

To illustrate the **Merge Interval** behavior, I'll start by creating one table called **Pedidos** (Orders). The following script will create the tables and populate them with some garbage data.

Chapter 12: Merges – Merge Interval

```
USE tempdb
GO

IF OBJECT_ID('Pedidos') IS NOT NULL
    DROP TABLE Pedidos
GO

CREATE TABLE Pedidos (ID INT IDENTITY(1,1) PRIMARY KEY,
    ID_Cliente INT NOT NULL,
    Quantidade SmallInt NOT NULL,
    Valor Numeric(18,2) NOT NULL,
    Data DATETIME NOT NULL)
GO

DECLARE @I SmallInt
SET @I = 0

WHILE @I < 10000
BEGIN
    INSERT INTO Pedidos(ID_Cliente, Quantidade, Valor, Data)
        SELECT ABS(CheckSUM(NEWID()) / 100000000),
            ABS(CheckSUM(NEWID()) / 10000000),
            ABS(CONVERT(Numeric(18,2), (CheckSUM(NEWID()) / 1000000.5))),
            GETDATE() - (CheckSUM(NEWID()) / 1000000)
    SET @I = @I + 1
END
GO
```

Now that we have the table, we have to create two non-clustered indexes. The first uses the column **ID_Cliente** as a Key, including the column **Valor** to create a covered index to our query. And another using the column **Data** as a Key and including the column **Valor**.

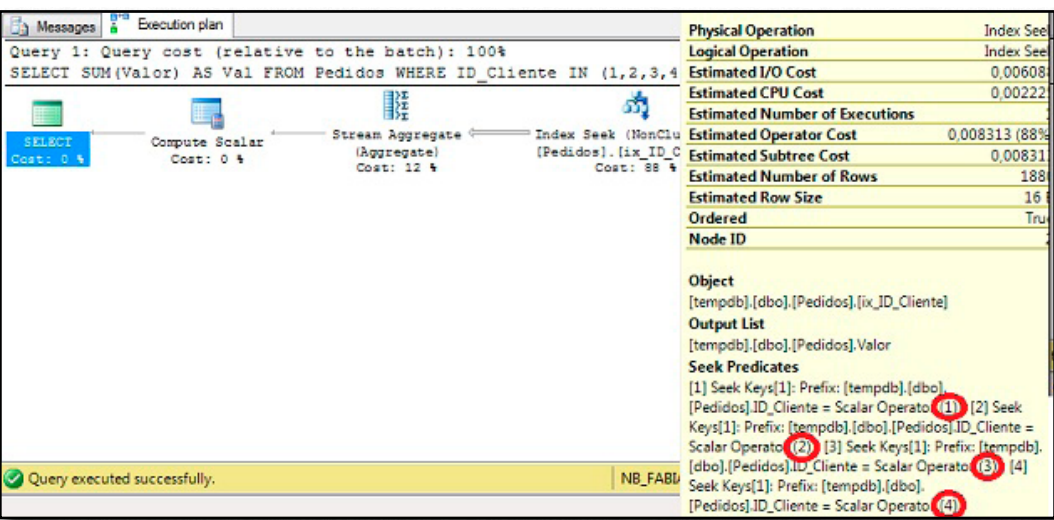
```
CREATE NONCLUSTERED INDEX ix_ID_Cliente ON Pedidos(ID_Cliente) INCLUDE (Valor)
GO
CREATE NONCLUSTERED INDEX ix_Data ON Pedidos(Data) INCLUDE (Valor)
GO
```


Merge Interval

Now that we have the data, we can write a query to see the merge interval. The following query is selecting the amount of sales for four customers:

```
SELECT SUM(Valor) AS Val
FROM Pedidos
WHERE ID_Cliente IN (1,2,3,4)
GO
```

For the query above, we have the following execution plan:



In the execution plan above we can see that QO chose to use the index **ix_ID_Cliente** to seek the data for each **ID_Cliente** specified in the IN clause, and then uses the **Stream Aggregate** to perform the sum by each **ID_Cliente**.

Chapter 12: Merges – Merge Interval

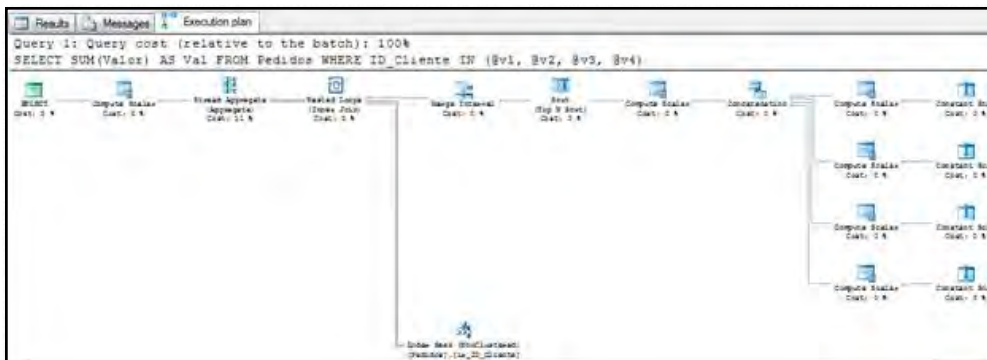
This is a classic Index Seek task. For each value, SQL Server will read the data through the balanced index tree searching for the **ID_Cliente**. For now, it doesn't require the Merge Interval.

Now let's look at a similar query:

```
DECLARE @v1 Int = 1,
        @v2 Int = 2,
        @v3 Int = 3,
        @v4 Int = 4

SELECT SUM(Valor) AS Val
FROM Pedidos
WHERE ID_Cliente IN (@v1, @v2, @v3, @v4)
GO
```

For the query above, we have the following execution plan:



As you can see, the only difference between the queries is that now we are using variables instead of constant values, but the Query Optimizer creates a very different execution plan for this query. So the question is, "What do you think? Do you think that SQL should have used the same execution plan for this query?"

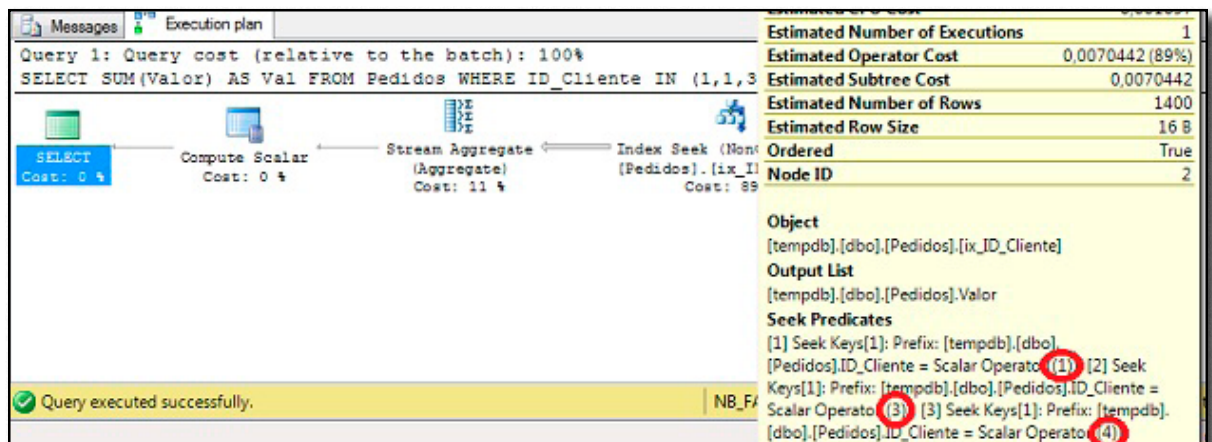
The right answer is No. Why not? Because at the compile time SQL Server doesn't know the values of the constants, and if the values turn out to be duplicates, then it will read the same data twice. Suppose that the value of the `@v2` is also "I", SQL will read the ID 1 twice, one for variable `@v1` and another for variable `@v2`, something that we don't expect to see since we expect performance, reading the same data twice is not good. So it has to use the Merge Interval to remove the duplicate occurrences.

Let's wait a minute, Fabiano! Are you saying that for the first query, QO automatically removes the duplicated occurrences in the IN clause?

Yes. Do want to see it?

```
SELECT SUM(Valor) AS Val
FROM Pedidos
WHERE ID_Cliente IN (1,1,3,4)
GO
```

For the query above we have the following execution plan:



You will see that now we only have three Seek Predicates. Perfect.

Let's go back to Merge Interval plan.

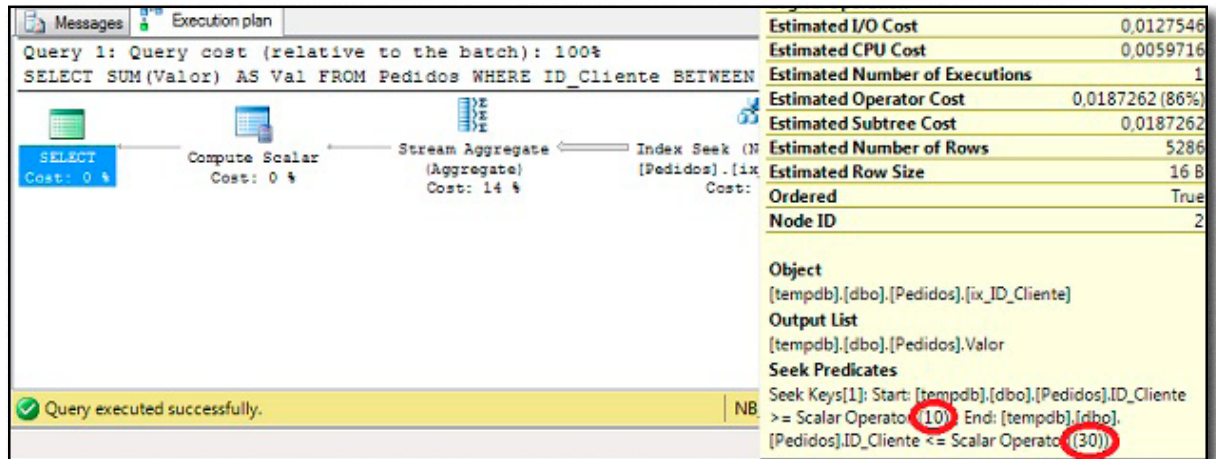
The plan is using the operators **Compute Scalar**, **Concatenation**, **Sort** and **Merge Interval** to eliminate the duplicated values at the execution plan phase.

At this time, maybe some questions are rising in your mind. First: Why doesn't SQL Server don't just use a DISTINCT in the IN variables to remove the joins? Second: Why is this called a Merge, when I can't see anything related to a merge here?

The answer is that the Query Optimizer (QO) uses this operator to perform the DISTINCT because, with this code, the QO also recognize overlapping intervals and will potentially merge these to non-overlapping intervals that will then be used to seek the values. To understand this better, let's suppose that we have the following query that doesn't use variables.

```
SELECT SUM(Valor) AS Val
FROM Pedidos
WHERE ID_Cliente BETWEEN 10 AND 25
      OR ID_Cliente BETWEEN 20 AND 30
GO
```

Now, let's look at the execution plan:



Notice how smart the Query Optimizer was. (That's why I love it!) It recognizes the overlap between the predicates, and instead of doing two seeks in the index (one for each between filter), it creates a plan that performs just one seek.

Now let's change the query to use the variables.

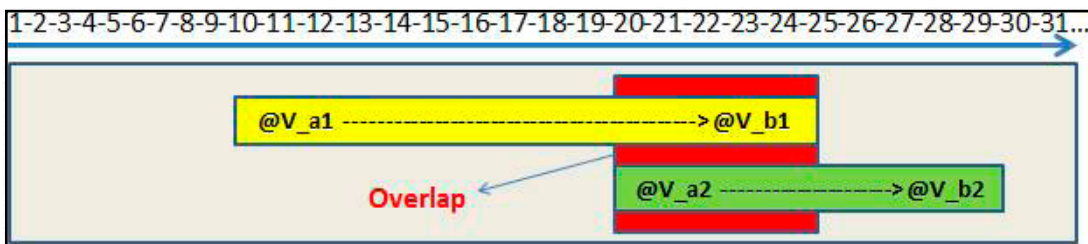
```
DECLARE @v_a1 Int = 10,  
        @v_b1 Int = 20,  
        @v_a2 Int = 25,  
        @v_b2 Int = 30  
  
SELECT SUM(Valor) AS Val  
FROM Pedidos  
WHERE ID_Cliente BETWEEN @v_a1 AND @v_a2  
OR ID_Cliente BETWEEN @v_b1 AND @v_b2  
GO
```

For this query we have the following execution plan:

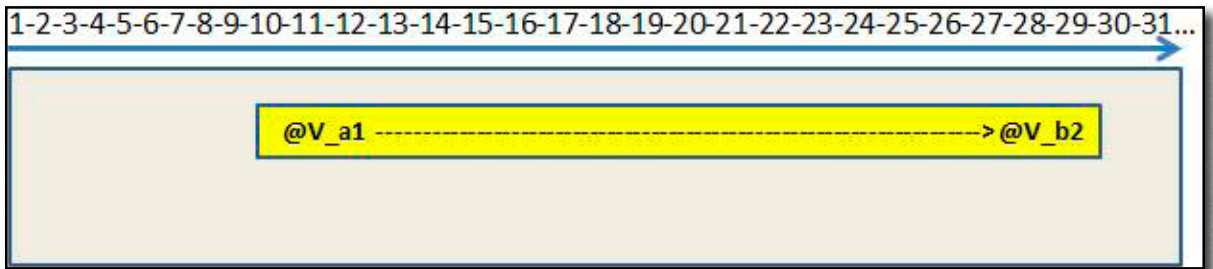
.....



Let's check what the plan is doing using a different perspective. First let's understand the overlap.



In the figure above, we can see that if SQL Server reads the ranges separately, it will read the range from 20 to 25 twice. I've used a small range to test with, but think in terms of a very large scan that we'd see in a production database; if we can avoid this step, then we'll see a great performance improvement.



After the Merge Interval runs, SQL Server can seek only the final range. It knows that is possible to go to **@v_a1** to **@vb_2** directly.

Finally

To finish this subject I would recommend that you read about a bug in SQL Server 2005 caused by a mistake in this process. Just take a look at the blog of [MLADEN PRAJDIC](#), a SQL Server MVP from Slovenia.

I wouldn't want to miss the opportunity to congratulate the Microsoft guys that build icons in SQL Server/Windows. I once read a book called *The Icon Book*; it was amazing how beautiful and meaningful the icons in the graphical query plan are. The Merge Interval icon is perfect. If you look at the icon you will see exactly what it is doing. Brilliant, it's incredible how they can express something in a small picture. Well done!

Chapter 13: Split, Sort, Collapse

Introduction

In this chapter, we'll feature two new operators, **Split** and **Collapse**, and describe how the **SORT** operator can be used to help with the validation of a query. The **SORT** operator has already been covered in an earlier chapter.

In short, these operators, **Split** and **Collapse**, are used to identify a phantom unique key violation. To help to understand this better, I'll start by explaining the **Unique Index** and how it is used by the query optimizer, then I'll go on to present a update command that uses these operators.

Unique Index

The **Unique Index** is an index that guarantees that no duplicate values are allowed in a specified key column. In other words, if you want to be sure that some value can't be duplicated in your table, then a unique index can be made responsible for enforcing this. A classic use for a unique index is as the primary key of your table, where no duplicated rows are allowed.

Every time that you create a new index, decide whether this can be a unique index. Why? Because the Query Optimizer can then use this index to simplify a query plan: If it knows that the rows are always going to be unique, it knows that the selectivity is always one. The more selective an index has, the greater the likelihood that it will be used by a query.

Creating sample data

To illustrate the operators that are the subject of this article, I'll start by creating one table called **TabTest**. The following script will create the table and populate it with some meaningless data:

```
USE tempdb
GO
IF OBJECT_ID('TabTest') IS NOT NULL
    DROP TABLE TabTest
GO
CREATE TABLE TabTest (ID      Int IDENTITY(1,1) PRIMARY KEY,
                        Name    VarChar(250) NULL,
                        Name2   VarChar(250) NULL,
                        Val     Int NULL,
                        Val2    Int NULL)
GO
CREATE UNIQUE INDEX ix_Name_Unique ON TabTest(Name)
CREATE INDEX ix_Name2_NonUnique ON TabTest(Name2)
CREATE UNIQUE INDEX ix_Val_Unique ON TabTest(Val)
CREATE INDEX ix_Val2_NonUnique ON TabTest(Val2)
GO
INSERT INTO TabTest(Name, Val) VALUES(NEWID(), 1)
INSERT INTO TabTest(Name, Val) VALUES(NEWID(), 2)
INSERT INTO TabTest(Name, Val) VALUES(NEWID(), 3)
INSERT INTO TabTest(Name, Val) VALUES(NEWID(), 4)
INSERT INTO TabTest(Name, Val) VALUES(NEWID(), 5)
GO
UPDATE TabTest SET Name2 = Name, Val2 = Val
GO
SELECT * FROM TabTest
```

You'll have noticed that the table has four indexes, one for each column. The difference between the columns **Name** and **Name2** is that the indexes on the columns with the prefix "2" are non-unique indexes, and the index on the columns **Name** and **Val** are unique.

The value of the columns **Name2** and **Val2** are the same as the columns **Name** and **Val**.

Here is what the data looks like:

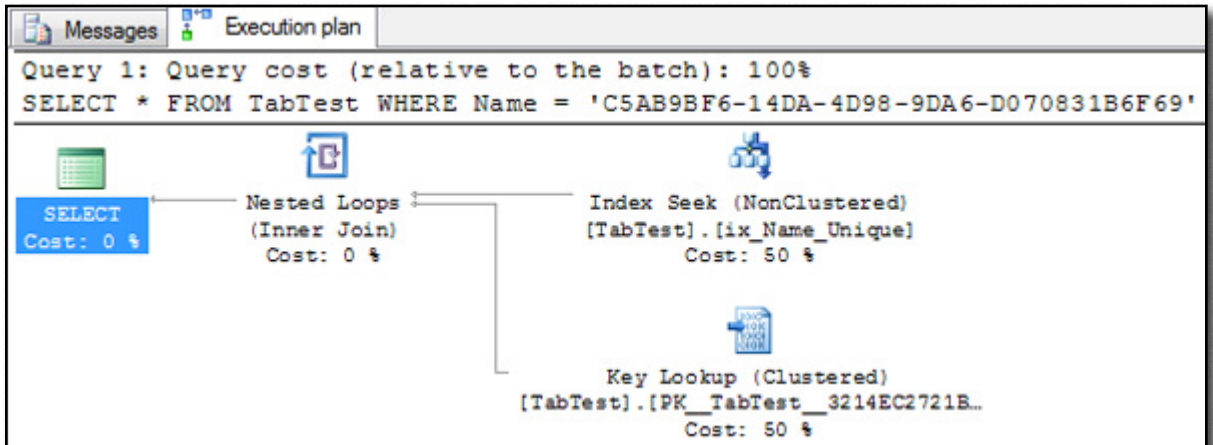
Results		Messages				
	ID	Name	Name2	Val	Val2	
1	1	C5AB9BF6-14DA-4D98-9DA6-D070831B6F69	C5AB9BF6-14DA-4D98-9DA6-D070831B6F69	1	1	
2	2	5162C34F-DF57-422D-8EF1-54A874021AFE	5162C34F-DF57-422D-8EF1-54A874021AFE	2	2	
3	3	FC975066-420A-4C59-BDB5-E56DD6B44CB3	FC975066-420A-4C59-BDB5-E56DD6B44CB3	3	3	
4	4	1312A997-C6A3-45DD-8619-654781D7FCF4	1312A997-C6A3-45DD-8619-654781D7FCF4	4	4	
5	5	46A594F8-3D0B-46BA-9031-7802E3BB72D9	46A594F8-3D0B-46BA-9031-7802E3BB72D9	5	5	

Querying a Unique Index

Now that we have created and populated the table, let's try two queries: first, a query that selects all the rows using a filter on the column with the unique index:

```
SELECT *
FROM TabTest
WHERE Name = '1C9629D2-593A-4C42-8EEA-CFA289AE060F'
GO
```

For the query above, we have the following execution plan:

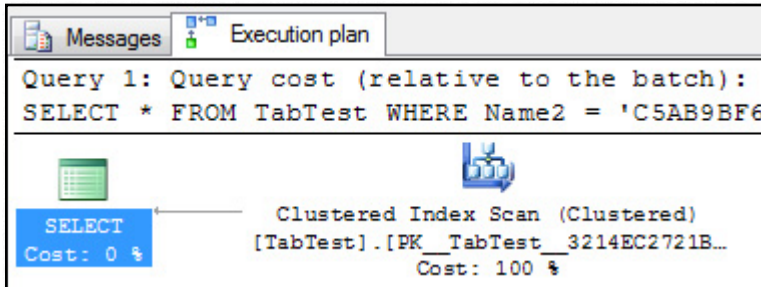


As we can see from the execution plan, SQL Server chose to perform an Index Seek on the `ix_Name_Unique` index and a **Key Lookup** to read the data on the clustered index.

Now let's try to run the same query using the column with the **non-unique** index:

```
SELECT *  
FROM TabTest  
WHERE Name2 = 'C5AB9BF6-14DA-4D98-9DA6-D070831B6F69'  
GO
```

For the query above, we have the following execution plan:



As we can see, now SQL Server chose to run a scan on the clustered index.

Because the first query is using a Unique Index, SQL Server decides to simplify matters by merely creating a Trivial Plan. Oops! I have a feeling that I haven't yet described what a trivial plan is, so let me do it now.

Trivial plan

An execution plan is created during the optimization process. Generally, SQL Server has to use a variety of techniques to try to optimize your query in order to create the best-possible plan. Sometimes, however, the Query Optimizer can decide that it does not need to do the full optimization process to find the best plan, Because it has determined that there is a plan that is good enough.

Within the execution plan, we can see if the query optimizer has decided that the Plan is trivial by looking at the properties of the plan. For instance, let's look at the execution plan of the first query:

Logical Operation	
Optimization Level	TRIVIAL
Parameter List	@0

Apart from that, we also can see if the plan is trivial by looking at the XML plan:

```
id="1" StatementOptmLevel="TRIVIAL" Statem
```

A trivial plan is created when the Query Optimizer finds an optimal way of read the data that you are querying. For instance a **select * from table** will probably create a trivial plan that merely reads the data from the clustered index. When a trivial plan is selected, the Query Optimizer doesn't then need to expend resources in trying to figure out the best plan because it knows that this trivial plan is the best option.

If a plan is determined to be trivial, then the query isn't recompiled when an **Update Statistics** occurs on the statistics, and an **Auto Update Statistics** isn't fired. Let's see this in practice.

Let's try the first query. If we look at the cache plan, we can see that we have the trivial plan and that it has been reused:

```
USE tempdb
GO
DBCC FREEPROCCACHE
GO
SELECT *
  FROM TabTest
 WHERE Name = 'D1397278-67CA-4EE6-B383-2E278018DC8F'
GO
SELECT *
  FROM TabTest
 WHERE Name = '7462AD1E-3335-44BE-AFC4-28F44FCA4F90'
GO
SELECT cp.objType,
```

```

cp.Usecounts,
st.Text AS Query,
qp.query_plan.value('declare default element
                    namespace "http://schemas.microsoft.com/
sqlserver/2004/07/ Showplan";
                    (//StmtSimple/@StatementOptmLevel)[1]',
                    'varchar(20)') AS StatementOptmLevel,
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
WHERE st.Text like '%from TabTest%'
AND st.text not like '%sys.%'
AND cp.ObjType = 'Prepared'

```

Results					
Messages					
ID	Name	Name2	Val	Val2	
1	1	D1397278-67CA-4EE6-B383-2E278018DC8F	D1397278-67CA-4EE6-B383-2E278018DC8F	1	1
ID	Name	Name2	Val	Val2	
1	2	7462AD1E-3335-44BE-AFC4-28F44FCA4F90	7462AD1E-3335-44BE-AFC4-28F44FCA4F90	2	2
objType	Usecounts	Query	StatementOptmLevel	query_pla	
1	Prepared	2	(@0 varchar(8000))select * from TabTest where Na...	TRIVIAL	<ShowPl

In this query, I start by freeing the plan cache; then I run the query twice. Finally, I query the DMVs to find out how many times the plan was reused, and to see the **StatementOptmLevel** within the XML plan.

As we can see in the picture (column **UseCounts**), the query was executed twice and the plan was reused.

Now, let's insert some data in the table in order to force an automatic **update statistics**. If you want to know how much data you will need to change so as to trigger an **update statistics**, then read Item 13 of [THIS ARTICLE](#).

```
INSERT INTO TabTest(Name, Val)
SELECT NEWID(), ABS(CHECKSUM(NEWID())) / 1000
GO 500
```

After inserting 500 rows into the table, I'll run the query again and check whether the plan was reused or if a new plan was created as a result of the **auto update statistics**. Optionally, you can check the event **SP:StmtCompleted** in the profiler to see if the update statistics was executed.

```
SELECT *
FROM TabTest
WHERE Name = '74DD0A57-56CF-4331-A324-7A7E83C6043E'
GO
SELECT cp.objType,
       cp.Usecounts,
       st.Text AS Query,
       qp.query_plan.value('declare default element
                           namespace "http://schemas.microsoft.com/
sqlserver/2004/07/ Showplan";
                           (//StmtSimple/@StatementOptmLevel)[1]',
                           'varchar(20)') AS StatementOptmLevel,
       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
WHERE st.Text like '%from TabTest%'
      AND st.text not like '%sys.%'
      AND cp.ObjType = 'Prepared'
```

Results					
Messages					
ID	Name	Name2	Val	Val2	
1	5	74DD0A57-56CF-4331-A324-7A7E83C6043E	74DD0A57-56CF-4331-A324-7A7E83C6043E	5	5
objType	Usecounts	Query	StatementOptmLevel	query_pla	
1	Prepared	(@0 varchar(8000))select * from TabTest where Na...	TRIVIAL	<ShowPl	

Now we can see that the plan was reused, even after we had inserted sufficient data to fire an update statistics and thereby cause the creation of a new plan.

Let's try the same exercise with the **non-trivial** plan:

```
USE tempdb
GO
DBCC FREEPROCCACHE
GO
SELECT *
  FROM TabTest
 WHERE Name2 = '47B12B84-6B81-4D54-A2ED-3F4BAE31835E'
GO
SELECT *
  FROM TabTest
 WHERE Name2 = '3901385F-2481-4126-BAF7-15B9C656952A'
GO
SELECT cp.objType,
       cp.Usecounts,
       st.Text AS Query,
       qp.query_plan.value('declare default element
                           namespace "http://schemas.microsoft.com/
sqlserver/2004/07/ Showplan";
                           (//StmtSimple/@StatementOptmLevel)[1]',
                           'varchar(20)') AS StatementOptmLevel,
       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
 WHERE st.Text like '%from TabTest%'
       AND st.text not like '%sys.%'
       AND cp.ObjType = 'Prepared'
```


Results					
	ID	Name	Name2	Val	Val2
1	1	47B12B84-6B81-4D54-A2ED-3F4BAE31835E	47B12B84-6B81-4D54-A2ED-3F4BAE31835E	1	1
	ID	Name	Name2	Val	Val2
1	2	3901385F-2481-4126-BAF7-15B9C656952A	3901385F-2481-4126-BAF7-15B9C656952A	2	2
	objType	Usecounts	Query	StatementOptmLevel	query_plan
1	Prepared	2	(@0 varchar(8000))select * from TabTest where Na	FULL	ShowPlanXML

Again the plan was reused. Now let's see what happens after we insert the new data.

```

INSERT INTO TabTest(Name, Val)
SELECT NEWID(), ABS(CHECKSUM(NEWID())) / 1000
GO 500

UPDATE TabTest SET Name2 = Name, Val2 = Val
GO
SELECT *
FROM TabTest
WHERE Name2 = 'AC7E0C8E-1240-4537-817E-D20819971542'
GO
SELECT cp.objType,
       cp.Usecounts,
       st.Text AS Query,
       qp.query_plan.value('declare default element
                           namespace "http://schemas.microsoft.com/
sqlserver/2004/07/ Showplan";
                           (//StmtSimple/@StatementOptmLevel)[1]',
                           'varchar(20)') AS StatementOptmLevel
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
WHERE st.Text like '%from TabTest%'
      AND st.text not like '%sys.%'
      AND cp.ObjType = 'Prepared'

```

Results					
Messages					
	ID	Name	Name2	Val	Val2
1	5	AC7E0C8E-1240-4537-817E-D20819971542	AC7E0C8E-1240-4537-817E-D20819971542	5	5
	objType	Usecounts	Query	StatementOptmLevel	
1	Prepared	1	(@0 varchar(8000))select * from TabTest where Na.	FULL	

As we can see, the auto update statistics was now triggered and a new plan was created.

Full Optimization

An interesting thing is that, in the execution plans, we can only see two levels of optimization, TRIVIAL and FULL. But behind the scenes we can see that the FULL optimization is divided into three steps. These steps are called Search 0, Search 1, and Search 2. (I don't like these names either, I expected something more intuitive.)

If we query the DMV `sys.dm_exec_query_optimizer_info`, we can see which phase was executed in a FULL optimization.

```

SELECT Counter, Occurrence
  FROM sys.dm_exec_query_optimizer_info
 WHERE Counter IN (N'trivial plan', N'search 0', N'search 1', N'search 2')
GO
SELECT *
  FROM TabTest
 WHERE Name2 = 'AC7E0C8E-1240-4537-817E-D20819971542'
OPTION (RECOMPILE)
GO
SELECT Counter, Occurrence
  FROM sys.dm_exec_query_optimizer_info
 WHERE Counter IN (N'trivial plan', N'search 0', N'search 1', N'search 2')

```

Results		Messages	
Counter	Occurrence		
1	trivial plan	457	
2	search 0	113	
3	search 1	424	
4	search 2	0	

ID	Name	Name2	Val	Val2
1	5	AC7E0C8E-1240-4537-817E-D20819971542	AC7E0C8E-1240-4537-817E-D20819971542	5

Counter	Occurrence		
1	trivial plan	457	
2	search 0	113	
3	search 1	425	
4	search 2	0	

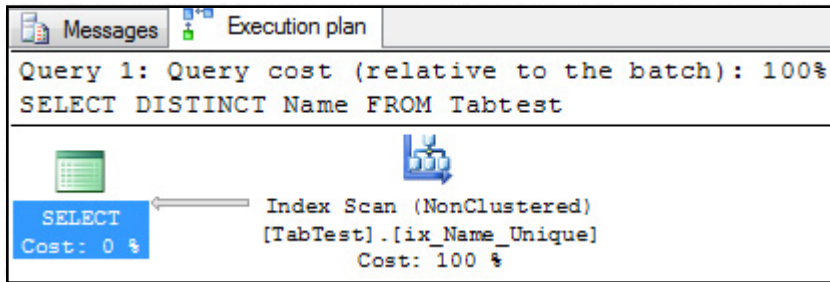
In the picture above, we can see that the **Search 1** counter has increased by one. If you run the query using the unique index, you will see the trivial plan counter increasing.

Some queries require a full optimization. This means that, if you have a heavy query with lot of joins, it may use **Search 1** or if you have a query that maybe uses a indexed view it will use **Search 2**. You can play with the DMV to see the level of optimization that you got in your query.

More about querying a Unique Index

After explaining both trivial and full optimization, let's see more samples that demonstrate the benefit of having a unique index. The Query Optimizer can avoid an unnecessary **DISTINCT**:

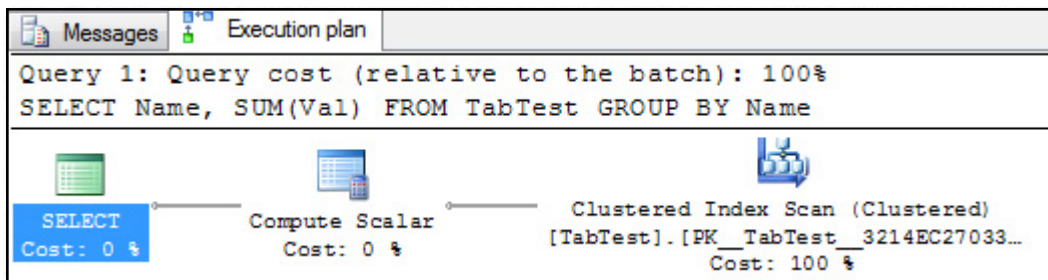
```
SELECT DISTINCT Name
FROM Tabtest
GO
```



In this execution plan, the Query Optimizer knows that the `DISTINCT` clause is totally unnecessary, and it doesn't waste time trying to remove the duplicated occurrences because it knows that, if the column is unique, then it is sufficient to just read the data from the index.

The Query Optimizer can avoid an unnecessary aggregation:

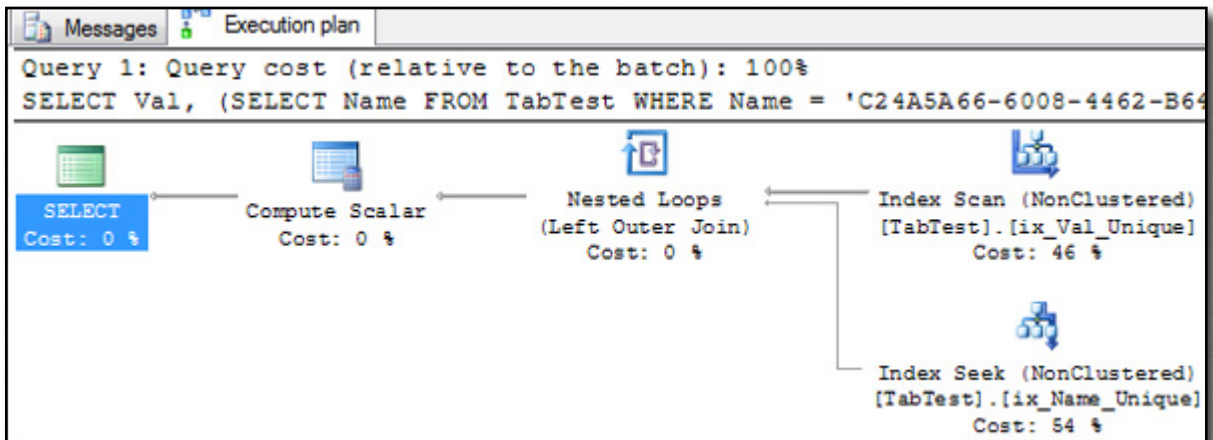
```
SELECT Name, SUM(Val)
FROM TabTest
GROUP BY Name
```



As we can see, there is no aggregation needed to compute the `SUM` because SQL Server knows that there is only one row per **Name**. SQL is reading the data from the clustered index. The mere presence of the unique index is enough to enable a optimization, even though SQL Server is not actually reading the data from that index.

The Query Optimizer can avoid an assert validation:

```
SELECT Val, (SELECT Name
              FROM TabTest
              WHERE Name = 'C24A5A66-6008-4462-B64D-EB3F76D0A420')
FROM TabTest
```



Here we can see that SQL Server is not using the **Assert** operator to validate the Sub-Query expression.

Non-Unique Index and updates

Every update is executed in two main steps. Firstly, SQL Server has to read the data that will be updated, and then it must update these values.

Let's analyze a plan that updates a non-unique indexed column. Here is a query that updates the column **Val2** that doesn't have a unique index.

```
UPDATE TabTest SET Val2 = Val2 + 1
```



```
--Clustered Index Update(OBJECT:([TabTest].[PK__TabTest__3214EC2703317E3D]),  
OBJECT:([TabTest].[ix_Val2_NonUnique]), SET:([TabTest].[Val2] = [Expr1003]))  
    |--Compute Scalar(DEFINE:([Expr1016]=[Expr1016]))  
        |--Compute Scalar(DEFINE:([Expr1016]=CASE WHEN [Expr1007] THEN (0) ELSE  
(1) END))  
            |--Compute Scalar(DEFINE:([Expr1003]=[TabTest].[Val2]+(1),  
[Expr1007]=CASE WHEN [TabTest].[Val2] = ([TabTest].[Val2]+(1)) THEN (1) ELSE (0)  
END))  
                |--Top(ROWCOUNT est 0)  
                    |--Clustered Index Scan(OBJECT:([TabTest].[PK__  
TabTest__3214EC2703317E3D]))
```

Text Execution Plan.

As usual, let's go through this execution plan step by step, analyzing what is happening with some operators of the plan.

Clustered Index Scan

Here, the SQL Server is reading the data that will be updated on the clustered index.

TOP

The TOP operator is necessary in order to perform the SET ROWCOUNT. I know that sounds a little weird the first time that you hear it, because we aren't executing the SET ROWCOUNT. What happens here is that, if you change the value of the ROWCOUNT, it will not trigger a recompilation of the plan, which means that SQL has to find some way to execute the query using a cached plan, plus updating just those rows specified in the ROWCOUNT.

Compute Scalar

In this operation, we can see the new optimization that is created on SQL Server 2005 in order to avoid the update of a value that hadn't changed. For more details, see Item 11 of [THIS ARTICLE](#).

If we analyze the text execution plan, we can see that there is a case expression that checks whether the value has changed:

```
[Expr1007] = Scalar Operator(CASE
                                WHEN [TabTest].[Val2] = ([TabTest].[Val2]+(1)) THEN (1)
                                ELSE (0)
                                END)
```

As we can see, if the value is the same, the value 1 (one) will be returned to the [Expr1007] and if not, then 0 (zero) will be returned.

Clustered Index Update

Here, the optimizer will execute the Update. An interesting thing here is that we can call this plan a **Narrow Plan** or a **Per-Row Plan**.

You'll probably have noticed from the text execution plan that there are two indexes being updated by this operator, the clustered index and the index **ix_Val2_NonUnique**.

Every time that you update a column that has a non-clustered index, SQL Server has to maintain the data in the clustered and the non-clustered indexes by updating it. Usually this update is executed in the clustered index key order, which means that the non-clustered updates are executed in a random order (because it's on the clustered index order) and not in the non-clustered key order.

Sometimes, the Query Optimizer can create a **Wide Plan** that reads the data from each updated index and executes the update in the index order.

To show you a sample of a wide plan, I'll create some new indexes, and change the number of rows and pages on the statistics of the table **TabTest**:

```
CREATE INDEX ix1 ON TabTest (Val2)
CREATE INDEX ix2 ON TabTest (Val2)
UPDATE STATISTICS TabTest WITH ROWCOUNT = 50000, PAGECOUNT = 180
UPDATE STATISTICS TabTest ix_Val2_NonUnique WITH ROWCOUNT = 50000, PAGECOUNT = 180
UPDATE STATISTICS TabTest ix1 WITH ROWCOUNT = 50000, PAGECOUNT = 180
UPDATE STATISTICS TabTest ix2 WITH ROWCOUNT = 50000, PAGECOUNT = 180
GO
```

Note

If you want to read more about this UPDATE STATISTICS command you can read [THIS POST IN MY BLOG](#) or you can read [THIS POST](#) from Benjamin Nevarez.

After messing with the statistics, let's try the same update:

```
UPDATE TabTest SET Val2 = Val2 + 1
```


Chapter 13: Split, Sort, Collapse



```
--Sequence
|--Index Update(OBJECT:([TabTest].[ix_Val2_NonUnique]), SET:([ID1031]
= [TabTest].[ID],[Val21032] = [TabTest].[Val2]) WITH ORDERED PREFETCH
ACTION:([Act1030]))
|--Sort(ORDER BY:([TabTest].[Val2] ASC, [TabTest].[ID] ASC, [Act1030]
ASC))
|--Filter(WHERE:(NOT [Expr1024]))
|--Table Spool
|--Split
|--Clustered Index Update(OBJECT:([TabTest].[PK__
TabTest__3214EC27440B1D61]), SET:([TabTest].[Val2] = [Expr1003]))
|--Compute Scalar(DEFINE:([Expr1024]=[Expr1024],
[Expr1025]=[Expr1025], [Expr1026]=[Expr1026]))
|--Compute Scalar(DEFINE:([Expr1024]=CASE
WHEN [Expr1007] THEN (1) ELSE (0) END, [Expr1025]=CASE WHEN [Expr1007] THEN (1)
ELSE (0) END, [Expr1026]=CASE WHEN [Expr1007] THEN (1) ELSE (0) END))
|--Compute Scalar(DEFINE:([Expr1003]=[
TabTest].[Val2]+(1), [Expr1007]=CASE WHEN [TabTest].[Val2] = ([TabTest].[Val2]+(1))
THEN (1) ELSE (0) END))
|--Top(ROWCOUNT est 0)
|--Clustered Index
Scan(OBJECT:([TabTest].[PK__TabTest__3214EC27440B1D61]), ORDERED FORWARD)
|--Index Update(OBJECT:([TabTest].[ix1]), SET:([ID1033] = [TabTest].
[ID],[Val21034] = [TabTest].[Val2]) WITH ORDERED PREFETCH ACTION:([Act1030]))
|--Sort(ORDER BY:([TabTest].[Val2] ASC, [TabTest].[ID] ASC, [Act1030]
```

```

ASC))
      |
      |--Filter(WHERE:(NOT [Expr1025]))
      |
      |--Table Spool
      |
      |--Index Update(OBJECT:([TabTest].[ix2]), SET:([ID1035] = [TabTest].
[ID],[Val21036] = [TabTest].[Val2]) WITH ORDERED PREFETCH ACTION:([Act1030]))
      |--Sort(ORDER BY:([TabTest].[Val2] ASC, [TabTest].[ID] ASC, [Act1030]
ASC))
      |
      |--Filter(WHERE:(NOT [Expr1026]))
      |
      |--Table Spool

```

Text Execution Plan..

Now we have a bigger plan and we can see that SQL is updating one index per time, using the data sorted by the index key.

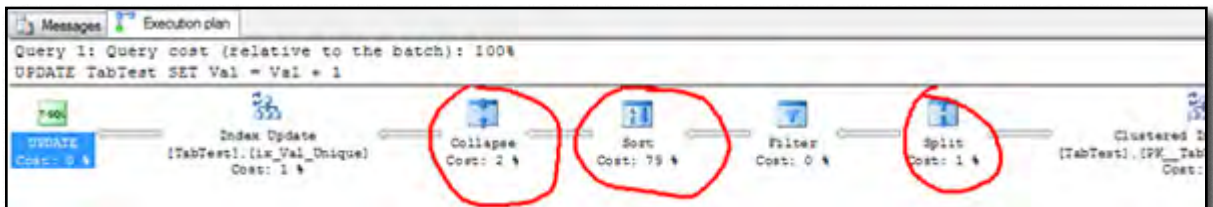
Unique Index and updates

After some explanation about unique indexes, it is now time to show you, in action, the three operators that are the subject of this article. Let's run the update in the column with the unique index and analyze the plan.

```

UPDATE TabTest SET Val = Val + 1

```

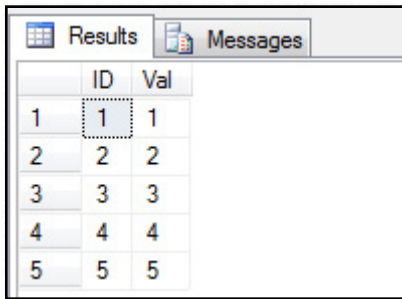


Part of the plan was omitted so as to concentrate on the topic.

Earlier in this chapter, I made the statement that these operators are used to avoid a phantom unique key violation. Let's try to understand that statement a bit better.

The data on the column **Val** is the following:

```
SELECT ID, Val FROM TabTest
```



	ID	Val
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5

We already know that an update is executed by a delete operation followed by an insert operation. You can see an explanation about it in Item 11 of [THIS ARTICLE](#). We can also detect that the update has been executed by the clustered key order.

Following this logic, let's mimic what will happen.

1. Read the first row from the clustered index to be updated. Result, ID = 1
2. Find the row ID = 1 in the unique index. Result, Val = 1
3. Delete the row with the value Val = 1
4. Insert the new value Val = 1 + 1 (2).

Here we have a phantom unique key violation because the value 2 already exists in the Unique Index. Because this value will eventually also be updated, this is a false violation.

To avoid it, SQL Server uses the operators **Split**, **Sort** and **Collapse** in order to reorganize the Deletes and the Inserts.

In our table, we have the following rows to be updated:

ID	Val	New_Val
1	1	2
2	2	3
3	3	4
4	4	5
5	5	6

The column **Val** is the key column that we need to update, and the column **New_Val** is the value of the column after the update.

After the Split occurs, SQL Server gets the deletes and the inserts of the values, and a new list is created:

Action	Val
Delete	1
Insert	2
Delete	2
Insert	3
Delete	3
Insert	4
Delete	4
Insert	5
Delete	5
Insert	6

The **Split** creates a list with all the deletes and the inserts, but this list is not properly ordered because the insertion of the value 2 is done before the delete, so we still have the phantom violation.

The **Sort** operator is used to reorder the list by **Action** plus the Key Value, and the Deletes then appear before the Inserts. So, after the sort operation, we have the following list:

Action	Val
Delete	1
Delete	2
Insert	2
Delete	3
Insert	3
Delete	4
Insert	4
Delete	5
Insert	5
Insert	6

Now the delete appears before the insert. If SQL Server runs the deletes and inserts in that order then all goes well and no phantom read occurs. But we still have the **Collapse** operator.

The **Collapse** operator combines those adjacent delete and insert pairs that share the same key value into a single "update." **Collapse** finds deletes and inserts for the same key and changes them into just one "update."

For instance, why should I have to delete and insert a new row with the value 2? If I can just ignore this step, then it's a good improvement. Right?

After the **collapse** we have the following actions to be executed:

Action	Old_Val	New_Val
Delete	1	
Update	2	2
Update	3	3
Update	4	4
Update	5	5
Insert		6

Because I know you are clever, you are probably thinking, "Ummm. Why update the value 2 to 2? Wouldn't it be better if we got a list just like the following?"

Action	Old_Val	New_Val
Delete	1	
Insert		6

If we perform these two actions, we have the expected final result.

I have to confess that I didn't know why SQL Server executed these updates until I watched the [CONOR CUNNINGHAM](#) session at the PASS 2010 talking about updates.

According to him, SQL Server needs to update the same row to guarantee that the stored engine puts the locks on the rows that are being updated. If SQL Server doesn't lock the row, another transaction can change the row and the update would consequently fail.

And finally

I want to say "Thank you" to the people from the Query Optimization community (yes, there are some guys out there just talking about QO) that indirectly help me to understand it better. [PAUL WHITE](#) (BTW congrats for the MVP award), [CONOR CUNNINGHAM](#), [BENJAMIN NEVAREZ](#) (BTW congrats on [YOUR BOOK](#)) and [CRAIG FREEDMAN](#).

If you don't follow these guys, you are missing a lot of good stuff about query optimization.

If you want to know more about a Showplan operator that I haven't covered yet, please leave a comment [HERE](#) with your suggestion.

That's all folks, I hope you've enjoyed learning about these operators. Keep an eye on [SIMPLE-TALK.COM](#) for more Showplan operators.

.NET and SQL Server Tools from Red Gate Software

Pricing and information about Red Gate tools are correct at the time of going to print. For the latest information and pricing on all Red Gate's tools, visit www.red-gate.com

redgate®
ingeniously simple tools

ANTS Memory Profiler

\$495

Find memory leaks and optimize memory usage

- Find memory leaks within minutes
- Jump straight to the heart of the problem with intelligent summary information, filtering options and visualizations
- Optimize the memory usage of your C# and VB.NET code

"Freaking sweet! We have a known memory leak that took me about four hours to find using our current tool, so I fired up ANTS Memory Profiler and went at it like I didn't know the leak existed. Not only did I come to the conclusion much faster, but I found another one!"

Aaron Smith IT Manager, R.C. Systems Inc.

ANTS Performance Profiler

from **\$395**

Profile your .NET code and boost the performance of your application

- Identify performance bottlenecks within minutes
- Drill down to slow lines of code thanks to line-level code timings
- Boost the performance of your .NET code
- Get the most complete picture of your application's performance with integrated SQL and File I/O profiling

"Thanks to ANTS Performance Profiler, we were able to discover a performance hit in our serialization of XML that was fixed for a 10x performance increase."

Garret Spargo Product Manager, AFHCAN

"ANTS Performance Profiler took us straight to the specific areas of our code which were the cause of our performance issues."

Terry Phillips Sr Developer, Harley-Davidson Dealer Systems

Visit www.red-gate.com for a 14-day, free trial

.NET Reflector[®]

From \$35

Browse, compile, analyze and decompile .NET code

- View, navigate and search through the class hierarchies of .NET assemblies, even if you don't have access to the source code for them
- Decompile and analyze .NET assemblies in C#, Visual Basic and IL
- Step into decompiled assemblies whilst debugging in Visual Studio, with all the debugging techniques you would use on your own code

"One of the most useful, practical debugging tools that I have ever worked with in .NET! It provides complete browsing and debugging features for .NET assemblies, and has clean integration with Visual Studio."

Tom Baker Consultant Software Engineer, EMC Corporation

SmartAssembly[®]

from \$795

.NET obfuscator and automated error reporting

Obfuscate your .NET code and protect your IP

Let your end-users report errors in your software with one click

Receive a comprehensive report containing a stack trace and values of all the local variables

Identify the most recurrent bugs and prioritize fixing those first

Gather feature usage data to understand how your software is being used and make better product development decisions

"I've deployed Automated Error Reporting now for one release and I'm already seeing the benefits. I can fix bugs which might never have got my attention before. I really like it a lot!"

Stefal Koell MVP

Visit **www.red-gate.com** for a 14-day, free trial

SQL Compare® Pro

\$595

Compare and synchronize SQL Server database schemas

- Eliminate mistakes migrating database changes from dev, to test, to production
- Speed up the deployment of new database schema updates
- Find and fix errors caused by differences between databases
- Compare and synchronize within SSMS

"Just purchased SQL Compare. With the productivity I'll get out of this tool, it's like buying time."

Robert Sondles Blueberry Island Media Ltd

SQL Data Compare Pro

\$595

Compares and synchronizes SQL Server database contents

- Save time by automatically comparing and synchronizing your data
- Copy lookup data from development databases to staging or production
- Quickly fix problems by restoring damaged or missing data to a single row
- Compare and synchronize data within SSMS

"We use SQL Data Compare daily and it has become an indispensable part of delivering our service to our customers. It has also streamlined our daily update process and cut back literally a good solid hour per day."

George Pantela GPAnalysis.com

Visit www.red-gate.com for a 14-day, free trial

SQL Prompt Pro

\$295

Write, edit, and explore SQL effortlessly

- Write SQL smoothly, with code-completion and SQL snippets
- Reformat SQL to a preferred style
- Keep databases tidy by finding invalid objects automatically
- Save time and effort with script summaries, smart object renaming and more

"SQL Prompt is hands-down one of the coolest applications I've used. Makes querying/developing so much easier and faster."

Jorge Segarra University Community Hospital

SQL Source Control

\$295

Connect your existing source control system to SQL Server

- Bring all the benefits of source control to your database
- Source control schemas and data within SSMS, not with offline scripts
- Connect your databases to TFS, SVN, SourceGear Vault, Vault Pro, Mercurial, Perforce, Git, Bazaar, and any source control system with a capable command line
- Work with shared development databases, or individual copies
- Track changes to follow who changed what, when, and why
- Keep teams in sync with easy access to the latest database version
- View database development history for easy retrieval of specific versions

"After using SQL Source Control for several months, I wondered how I got by before. Highly recommended, it has paid for itself several times over"

Ben Ashley Fast Floor

Visit **www.red-gate.com** for a 28-day, free trial

SQL Backup Pro

\$795

Compress, encrypt, and strengthen SQL Server backups

- Compress SQL Server database backups by up to 95% for faster, smaller backups
- Protect your data with up to 256-bit AES encryption
- Strengthen your backups with network resilience to enable a fault-tolerant transfer of backups across flaky networks
- Control your backup activities through an intuitive interface, with powerful job management and an interactive timeline

"SQL Backup is an amazing tool that lets us manage and monitor our backups in real time. Red Gate's SQL tools have saved us so much time and work that I am afraid my director will decide that we don't need a DBA anymore!"

Mike Poole Database Administrator, Human Kinetics

Visit www.red-gate.com for a 14-day, free trial

SQL Monitor

SQL Server performance monitoring and alerting

Intuitive overviews at global, cluster, machine, SQL Server, and database levels for up-to-the-minute performance data

Use SQL Monitor's web UI to keep an eye on server performance in real time on desktop machines and mobile devices

Intelligent SQL Server alerts via email and an alert inbox in the UI, so you know about problems first

Comprehensive historical data, so you can go back in time to identify the source of a problem

Generate reports via the UI or with Red Gate's free SSRS Reporting Pack

View the top 10 expensive queries for an instance or database based on CPU usage, duration and reads and writes

PagerDuty integration for phone and SMS alerting

Fast, simple installation and administration

"Being web based, SQL Monitor is readily available to you, wherever you may be on your network. You can check on your servers from almost any location, via most mobile devices that support a web browser."

Jonathan Allen Senior DBA, Careers South West Ltd

SQL Virtual Restore

\$495

Rapidly mount live, fully functional databases direct from backups

Virtually restoring a backup requires significantly less time and space than a regular physical restore

Databases mounted with SQL Virtual Restore are fully functional and support both read/write operations

SQL Virtual Restore is ACID compliant and gives you access to full, transactionally consistent data, with all objects visible and available

Use SQL Virtual Restore to recover objects, verify your backups with DBCC CHECKDB, create a storage-efficient copy of your production database, and more.

"We find occasions where someone has deleted data accidentally or dropped an index etc., and with SQL Virtual Restore we can mount last night's backup quickly and easily to get access to the data or the original schema. It even works with all our backups being encrypted. This takes any extra load off our production server. SQL Virtual Restore is a great product."

Brent McCracken Senior Database Administrator/Architect, Kiwibank Limited

SQL Storage Compress

\$1,595

Silent data compression to optimize SQL Server storage

- Reduce the storage footprint of live SQL Server databases by up to 90% to save on space and hardware costs
- Databases compressed with SQL Storage Compress are fully functional
- Prevent unauthorized access to your live databases with 256-bit AES encryption
- Integrates seamlessly with SQL Server and does not require any configuration changes

Visit www.red-gate.com for a 14-day, free trial

SQL Toolbelt

\$1,995

The essential SQL Server tools for database professionals

You can buy our acclaimed SQL Server tools individually or bundled. Our most popular deal is the SQL Toolbelt: fourteen of our SQL Server tools in a single installer, with **a combined value of \$5,930 but an actual price of \$1,995**, a saving of 66%.

Fully compatible with SQL Server 2000, 2005, and 2008.

SQL Toolbelt contains:

- | | |
|-------------------------------|---|
| ➤ SQL Compare Pro | ➤ SQL Doc |
| ➤ SQL Data Compare Pro | ➤ SQL Dependency Tracker |
| ➤ SQL Source Control | ➤ SQL Packager |
| ➤ SQL Backup Pro | ➤ SQL Multi Script Unlimited |
| ➤ SQL Monitor | ➤ SQL Search |
| ➤ SQL Prompt Pro | ➤ SQL Comparison SDK |
| ➤ SQL Data Generator | ➤ SQL Object Level Recovery Native |

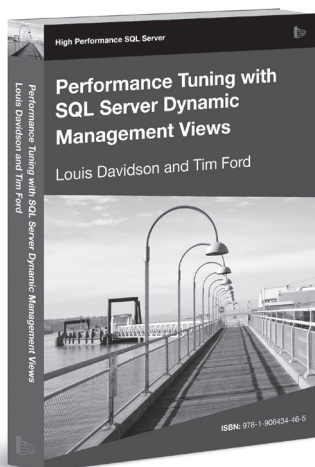
"The SQL Toolbelt provides tools that database developers, as well as DBAs, should not live without."

William Van Orden Senior Database Developer,
Lockheed Martin

Visit **www.red-gate.com** for a 14-day, free trial

Performance Tuning with SQL Server Dynamic Management Views

Louis Davidson and Tim Ford



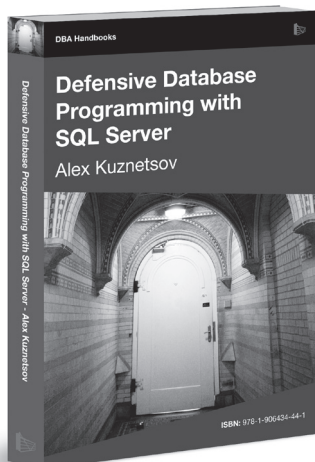
This is the book that will de-mystify the process of using Dynamic Management Views to collect the information you need to troubleshoot SQL Server problems. It will highlight the core techniques and "patterns" that you need to master, and will provide a core set of scripts that you can use and adapt for your own requirements.

ISBN: 978-1-906434-47-2

Published: October 2010

Defensive Database Programming

Alex Kuznetsov



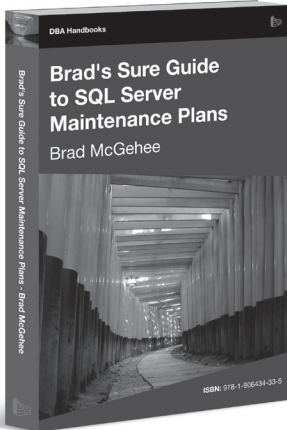
Inside this book, you will find dozens of practical, defensive programming techniques that will improve the quality of your T-SQL code and increase its resilience and robustness.

ISBN: 978-1-906434-49-6

Published: June 2010

Brad's Sure Guide to SQL Server Maintenance Plans

Brad McGehee



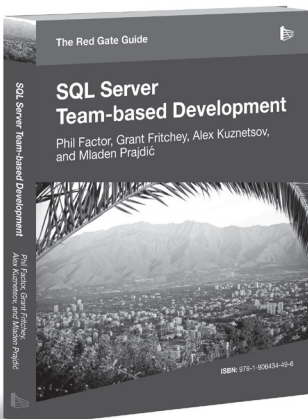
Brad's Sure Guide to Maintenance Plans shows you how to use the Maintenance Plan Wizard and Designer to configure and schedule eleven core database maintenance tasks, ranging from integrity checks, to database backups, to index reorganizations and rebuilds.

ISBN: 78-1-906434-34-2

Published: December 2009

The Red Gate Guide to SQL Server Team-based Development

Phil Factor, Grant Fritchey, Alex Kuznetsov, and Mladen Prajdić



This book shows how to use a mixture of home-grown scripts, native SQL Server tools, and tools from the Red Gate SQL Toolbelt, to successfully develop database applications in a team environment, and make database development as similar as possible to "normal" development.

ISBN: 978-1-906434-59-5

Published: November 2010