Execution Plans – Indexing to achieve optimal query plans

Author: Ted Krueger - LessThanDot

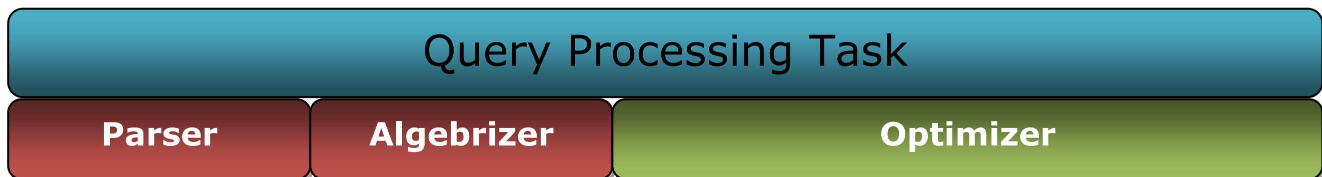
The obstacle

SQL Server and data storage itself is only as good as the means in which we can read the data. Very seldom are databases created only for the purpose of inserting and storing data without the intent of ever reading that data. While the type of structure or design of the data storage will vary depending on the type and majority of transactions that will occur on the database, the fact remains that the mechanisms in place must return the data quickly and without directly affecting other transactions from committing their own primary objectives. With SQL Server, execution plans are a foundation and starting point of the tuning process after the infrastructure phases of building a database server have been optimized.

By utilizing execution plans in SQL Server, developers and administrators have the ability to prevent non-optimal statements from entering our production systems. Implementing the steps of code reviews and initial tuning processes promotes this and removes the major risk that comes with not performing them.

Plan Cache

In order to tune a query, there is value in understanding how a query is processed and then stored for later reuse. This section will briefly discuss the flow that a statement will take when it is passed to SQL Server. For our reference, a statement will be directly related to any DML statement. There are three major steps or processes that a statement goes through. The first of these steps is called the parser. The parser name says exactly what this process accomplishes. The parser takes the statement and parses out all the steps that the statement has within it and creates these steps in what is called the parse tree. This is also the process that will dictate if the statement is well formed or not and rejected or allowed to proceed.



Following the parser, the next step is to hand the parse tree off to the Algebrizer. Within the Algebrizer, all name resolving is performed. For example, if a table name is not fully qualified, the Algebrizer will handle determining the exact location. The Algebrizer also determines data types such as, VARCHAR(10) and NUMERIC(2,2). With the Algebrizer completing successfully, a query processor tree is created. The query processor tree is the final product that is required in order for the finishing task to finally create the execution plan - The Optimizer.

The optimizer has several tasks to accomplish in order to come to a valid execution plan. During these tasks, the optimizer will determine the resources and objects available to use in order to create the most efficient execution possible. Part of this is determined on statistics that are created in your database. After everything has been calculated and the optimizer has found the most efficient execution plan possible, it is then compared to others in the plan cache. Plan cache is an allocated section in memory where SQL Server stores all of the execution plans that have been created. These plans are not estimated but the actual execution plans that have been created from running statements in SQL Server. If the optimizer finds a plan that matches the estimated plan it has created, it will use the plan that is already in plan cache. This provides a few things for SQL Server in respect to speed. It will prevent multiple plans from being cached that are identical and using memory that can be utilized for other plans.

When using execution plans to tune any query, it is a best practice to ensure the cache is clear of plans that may interfere with the true performance of the query as it is developed. This allows the processing of a statement from start to completion and the overall performance of the statement to be measured.

To clear the cache, use DBCC FREEPROCCACHE

FREEPROCCACHE accepts either sql_handle or plan_handle as a parameter. This allows for the clearing of only one cached plan over clearing the entire cache (calling FREEPROCCACHE with no parameter specified). Clearing the entire plan cache will cause performance degradation due to the need for all statements being processed to be cached again. This process involves more resources such as CPU and disk IO. Best practice tells us that development should always occur on designated development and test database servers but this will still cause other development projects to be adversely affected in a negative way.

> Note: Never develop, test or alter a statement in production or clear the cache as it will affect performance in a negative way.

Executing FREEPROCCACHE

```
DBCC FREEPROCCACHE
GO
```

We can test removing a single plan by running the following query on the AdventureWorks database

```
SELECT  SalesOrderNumber
FROM    AdventureWorks.Sales.SalesOrderHeader header
        JOIN AdventureWorks.Sales.SalesOrderDetail details ON header.salesorderid =
details.salesorderid
WHERE   customerid = 11091
```

Once this Adhoc query has been cached we can search for it with a simplified join of the DMVs (sys) dm_exec_query_stats, dm_exec_cached_plans and DMFs dm_exec_sql_text and dm_exec_query_plan.

```
SELECT  [text] ,
        cp.plan_handle ,
        sql_handle
FROM    sys.dm_exec_query_stats qs
        JOIN sys.dm_exec_cached_plans cp ON qs.plan_handle = cp.plan_handle
        CROSS APPLY sys.dm_exec_sql_text(qs.plan_handle) AS query
        OUTER APPLY sys.dm_exec_query_plan(qs.plan_handle) AS executionplan
WHERE   objtype = 'Adhoc'
```

28	select SalesOrderNumber FROM AdventureWorks.Sales.SalesOrderHeader header join AdventureW...	0x06000C00FCCD0D0FB8C0500B000000000000000000000000	0x02000000FCCD0D0FFF089

Once the plan has been identified and the decision has been made to remove it, FREEPROCCACHE can be executed with either the sql_handle or plan_handle.

Execute DBCC FREEPROCCACHE with the sql_handle as a parameter from our previous output

```
DBCC FREEPROCCACHE (0x06000C00FCCD0D0FB8C0500B000000000000000000000000)
GO
```

```
DBCC FREEPROCCACHE(0x06000C00FCCD0D0FB8C0500B000000000000000000000000)
```

Messages

```
DBCC execution completed. If DBCC printed error messages, contact your system administrator.
```

If we then search for the plan by the sql_handle, we'll see it has been removed

plan cache.sql - ...HC\tkrueger (61))* | Object Explorer Details

```
select [text],cp.plan_handle,sql_handle
from sys.dm_exec_query_stats qs
join sys.dm_exec_cached_plans cp ON qs.plan_handle = cp.plan_handle
cross apply sys.dm_exec_sql_text(qs.plan_handle) as query
outer apply sys.dm_exec_query_plan(qs.plan_handle) as executionplan
where objtype = 'Adhoc'
and sql_handle = '0x06000C00FCCD0D0FB8C0500B000000000000000000000000'
```

Results | Messages

	text	plan_handle	sql_handle

Tuning Execution Plans

To show some more common operations and tuning methods utilizing execution plans, we will create a database to work in. The database name will be PLANLAB and we will import data from AdventureWorks tables - Sales.SalesOrderHeader and Sales.SalesOrderDetails.

To prepare the database, tables and import the data from an already existing AdventureWorks database local to the instance, run the following statements in the order they appear. Modify the file paths to match your own development file system structure.

```
USE Master
GO

CREATE DATABASE [PLANLAB] ON  PRIMARY
( NAME = N'PLANLAB_DATA_1', FILENAME = N'C:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\PLANLAB_DATA_1.mdf' ,
SIZE = 100MB , MAXSIZE = UNLIMITED, FILEGROWTH = 10MB )
 LOG ON
( NAME = N'PLANLAB_TRAN_1_log', FILENAME = N'C:\Program Files\Microsoft SQL
Server\MSSQL10.MSSQLSERVER\MSSQL\DATA\PLANLAB_TRAN_1_log.ldf' ,
SIZE = 100MB , MAXSIZE = 2GB , FILEGROWTH = 3%)
GO
ALTER DATABASE [PLANLAB] SET RECOVERY SIMPLE
GO
USE [PLANLAB]
GO
CREATE SCHEMA [WISSUG] AUTHORIZATION [dbo]
GO


CREATE TABLE WISSUG.[SalesOrderHeader](
      [SalesOrderID] [int] IDENTITY(1,1) NOT FOR REPLICATION NOT NULL,
      [RevisionNumber] [tinyint] NOT NULL,
      [OrderDate] [datetime] NOT NULL,
```

```
	[DueDate] [datetime] NOT NULL,
	[ShipDate] [datetime] NULL,
	[Status] [tinyint] NOT NULL,
	[OnlineOrderFlag] smallint NOT NULL,
	[SalesOrderNumber]  AS (isnull(N'SO'+CONVERT([nvarchar](23),[SalesOrderID],0),N'***
ERROR ***')),
	[PurchaseOrderNumber] varchar(30) NULL,
	[AccountNumber] varchar(30) NULL,
	[CustomerID] [int] NOT NULL,
	[ContactID] [int] NOT NULL,
	[SalesPersonID] [int] NULL,
	[TerritoryID] [int] NULL,
	[BillToAddressID] [int] NOT NULL,
	[ShipToAddressID] [int] NOT NULL,
	[ShipMethodID] [int] NOT NULL,
	[CreditCardID] [int] NULL,
	[CreditCardApprovalCode] [varchar](15) NULL,
	[CurrencyRateID] [int] NULL,
	[SubTotal] [money] NOT NULL,
	[TaxAmt] [money] NOT NULL,
	[Freight] [money] NOT NULL,
	[TotalDue]  AS (isnull(([SubTotal]+[TaxAmt])+[Freight],(0))),
	[Comment] [nvarchar](128) NULL,
	[rowguid] [uniqueidentifier] ROWGUIDCOL  NOT NULL,
	[ModifiedDate] [datetime] NOT NULL)
GO


CREATE TABLE WISSUG.[SalesOrderDetail](
	[SalesOrderID] [int] NOT NULL,
	[SalesOrderDetailID] [int] IDENTITY(1,1) NOT NULL,
	[CarrierTrackingNumber] [nvarchar](25) NULL,
	[OrderQty] [smallint] NOT NULL,
	[ProductID] [int] NOT NULL,
	[SpecialOfferID] [int] NOT NULL,
	[UnitPrice] [money] NOT NULL,
	[UnitPriceDiscount] [money] NOT NULL,
	[LineTotal]  AS (isnull(([UnitPrice]*((1.0)-
[UnitPriceDiscount]))*[OrderQty],(0.0))),
	[rowguid] [uniqueidentifier] ROWGUIDCOL  NOT NULL,
	[ModifiedDate] [datetime] NOT NULL)

GO

USE PLANLAB
GO

SET IDENTITY_INSERT WISSUG.SalesOrderHeader ON
INSERT INTO WISSUG.SalesOrderHeader
([SalesOrderID]
	,[RevisionNumber]
	,[OrderDate]
	,[DueDate]
	,[ShipDate]
	,[Status]
	,[OnlineOrderFlag]
	,[PurchaseOrderNumber]
	,[AccountNumber]
	,[CustomerID]
	,[ContactID]
	,[SalesPersonID]
	,[TerritoryID]
	,[BillToAddressID]
```

```
      ,[ShipToAddressID]
      ,[ShipMethodID]
      ,[CreditCardID]
      ,[CreditCardApprovalCode]
      ,[CurrencyRateID]
      ,[SubTotal]
      ,[TaxAmt]
      ,[Freight]
      ,[Comment]
      ,[rowguid]
      ,[ModifiedDate])
SELECT
         [SalesOrderID]
      ,[RevisionNumber]
      ,[OrderDate]
      ,[DueDate]
      ,[ShipDate]
      ,[Status]
      ,[OnlineOrderFlag]
      ,[PurchaseOrderNumber]
      ,[AccountNumber]
      ,[CustomerID]
      ,[ContactID]
      ,[SalesPersonID]
      ,[TerritoryID]
      ,[BillToAddressID]
      ,[ShipToAddressID]
      ,[ShipMethodID]
      ,[CreditCardID]
      ,[CreditCardApprovalCode]
      ,[CurrencyRateID]
      ,[SubTotal]
      ,[TaxAmt]
      ,[Freight]
      ,[Comment]
      ,[rowguid]
      ,[ModifiedDate]
FROM AdventureWorks.Sales.SalesOrderHeader
SET IDENTITY_INSERT WISSUG.SalesOrderHeader OFF

SET IDENTITY_INSERT WISSUG.SalesOrderDetail ON
INSERT INTO WISSUG.SalesOrderDetail
([SalesOrderID]
      ,[SalesOrderDetailID]
      ,[CarrierTrackingNumber]
      ,[OrderQty]
      ,[ProductID]
      ,[SpecialOfferID]
      ,[UnitPrice]
      ,[UnitPriceDiscount]
      ,[rowguid]
      ,[ModifiedDate])
SELECT
[SalesOrderID]
      ,[SalesOrderDetailID]
      ,[CarrierTrackingNumber]
      ,[OrderQty]
      ,[ProductID]
      ,[SpecialOfferID]
      ,[UnitPrice]
      ,[UnitPriceDiscount]
      ,[rowguid]
      ,[ModifiedDate]
```

```
FROM AdventureWorks.Sales.SalesOrderDetail
SET IDENTITY_INSERT WISSUG.SalesOrderDetail OFF
```

Starting with a query to tune

The below query requests five columns from the tables we created. Within the query, the SUM() function is used on the column, LineTotal. Note that neither of these tables have any indexes as of yet. There are also no statistics created due to nothing being captured in the selection from any statements being processed.

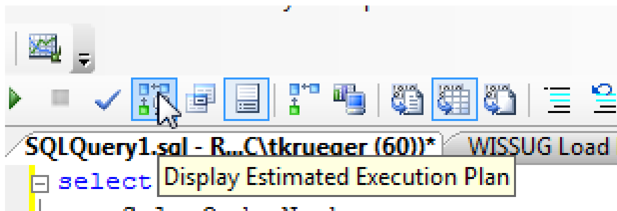
We can see statistics by viewing them in SSMS by expanding the database/tables/table in question and then statistics node. The system view, sys.stats, can also be queried to view statistics information on the tables. The following query shows an example filtered on the tables we created

```
SELECT  *
FROM    sys.STATS
WHERE   OBJECT_ID = OBJECT_ID(N'WISSUG.SalesOrderDetail')
        OR OBJECT_ID = OBJECT_ID(N'WISSUG.SalesOrderHeader')
```

At this time we do not need to run the query because our interest is in the estimated execution plan only.

Note: Statistics are created when processing statements on indexes and column. This is also true for viewing estimated execution plans.

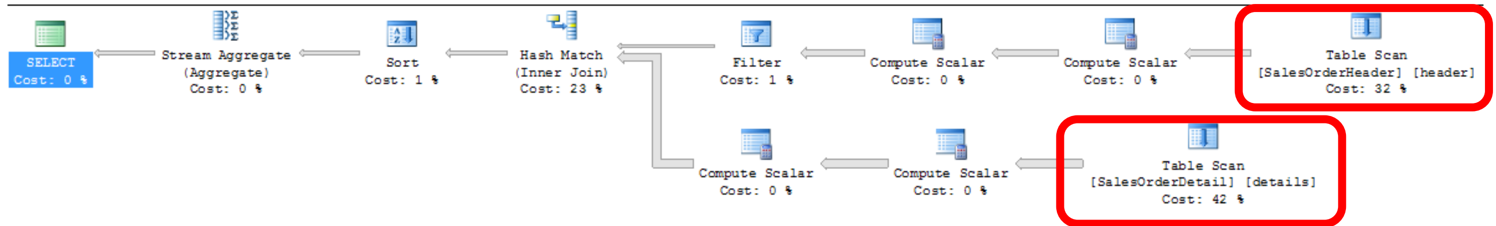
To view the estimated execution plan, click the button located in the menu strip in SSMS.



Place the query below into a new query window while under the context of the PLANLAB database and execute the estimated execution plan.

```
SELECT  SalesOrderNumber ,
        OrderDate ,
        ShipDate ,
        AccountNumber ,
        UnitPriceDiscount ,
        SUM(LineTotal) Total
FROM    WISSUG.SalesOrderHeader header
        JOIN WISSUG.SalesOrderDetail details ON header.salesorderid =
details.salesorderid
WHERE   customerid = 11091
GROUP BY SalesOrderNumber ,
        OrderDate ,
        ShipDate ,
        UnitPriceDiscount ,
        AccountNumber
```

The plan below that was generated from the query shows two very distinct and problematic operations. These operations are the table scans both on SalesOrderHeader and SalesOrderDetail. Essentially, when a table scan is performed, all the records of a table are scanned for matches. These operations tell us that our query will cause significant disk IO due to no supporting indexes to help easily find the location of the records we want returned or aggregated.



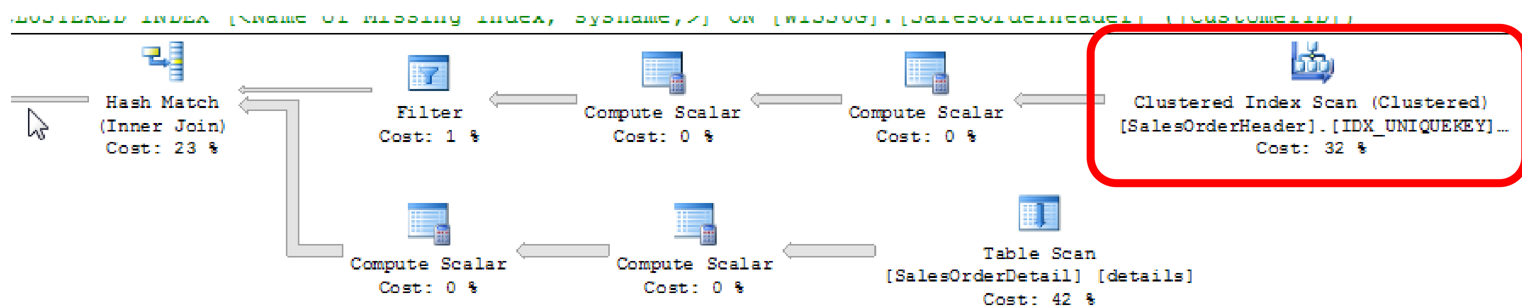
The process to start tuning a query like this can begin in many places. The columns being requested, the filter values (WHERE clause) sorting operations such as ORDER BY or GROUP BY and JOIN values.

Note: Each step in tuning any operation may or may not have direct effects on the total plan itself. When tuning, start with one operation while keeping the entire plan under consideration for these effects. In many cases, one index created with the proper columns allows for the entire plan to be optimized. This will be shown at the end of this article.

In the JOIN condition, SalesOrderID is being used in order to bring SalesOrderDetail into our results in order to fulfill the needs of the query. This will require us to make sure that the join between SalesOrderHeader and SalesOrderDetail does not have an exstensive table scans to fulfill the JOIN. SalesOrderID is a unique value in our table so we have the ability to create a CLUSTERED INDEX on it. A clustered index differs from other indexes by actually ordering the data physically. This is also why only one clustered index is allowed per table.

To create a clustered index on SalesOrderID in the SalesOrderHeader table, run the following CREATE INDEX statement.

```
CREATE CLUSTERED INDEX IDX_UNIQUEKEY ON WISSUG.SalesOrderHeader(SalesOrderID)
GO
```

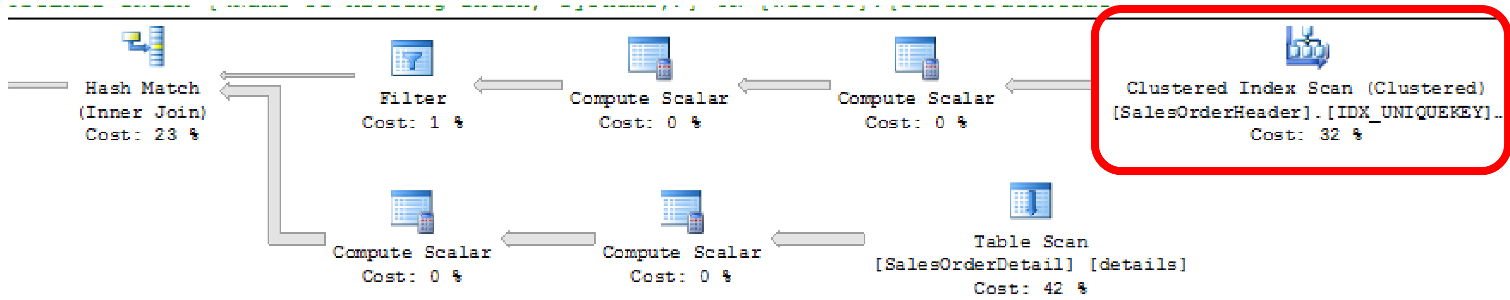


Once the clustered index is created successfully, execute the estimated execution plan again. The table scan has been replaced by an index scan now for SalesOrderHeader. An index scan is more optimal than a table scan. Index scans will only scan the entire contents of an index to determine the records to return. However, this operation is still not the most optimal operation to have in the plan. An Index Seek operation would be optimal in both lower resource utilization and more selective abilities.

Looking further at the query and SalesOrderHeader, we can see we are requesting SalesOrderNumber, OrderDate and AccountNumber. These columns must be returned from the SalesOrderHeader table as well as SalesOrderID. We can now bring in a nonclustered index to fulfill these new requirements in order to further tune the plan.

To create the nonclustered index on SalesOrderHeader, execute the following statement.

```
CREATE NONCLUSTERED INDEX IDX_SalesNumOrderDate_Ship_ASC ON
WISSUG.SalesOrderHeader(SalesOrderNumber,OrderDate,AccountNumber)
GO
```



As shown in the new estimated plan, the index scan on the clustered index remains even after covering the other columns we require from the table. Covering index refers to an index that contains all of the required columns to fully cover all the paths a query is taking to utilize them. This includes the columns that are being returned, the WHERE clause and JOIN conditions.

A set of properties exists with with all operations. These properties can be seen by hovering over the operation or right clicking the operation and selecting properties.

Hovering over the index scan with the mouse pointer shows that we are covering everything but CustomerID which is in our where clause and ShipDate. ShipDate was left out previously when we thought we had covered the returned columns.

Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated I/O Cost	0.483866
Estimated CPU Cost	0.0347685
Estimated Number of Executions	1
Estimated Operator Cost	0.518634 (24%)
Estimated Subtree Cost	0.518634
Estimated Number of Rows	31465
Estimated Row Size	53 B
Ordered	False
Node ID	6

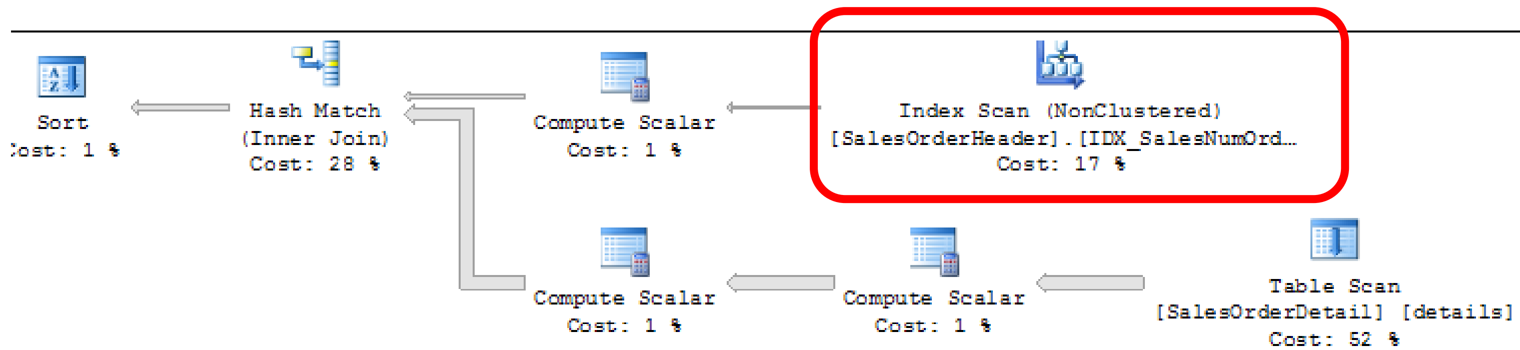
Object
[DUMP].[WISSUG].[SalesOrderHeader].[IDX_UNIQUEKEY] [header]

Output List
[DUMP].[WISSUG].[SalesOrderHeader].SalesOrderID, [DUMP].[WISSUG].[SalesOrderHeader].OrderDate, [DUMP].[WISSUG].[SalesOrderHeader].ShipDate, [DUMP].[WISSUG].[SalesOrderHeader].AccountNumber, [DUMP].[WISSUG].[SalesOrderHeader].CustomerID

To tune further, we now need to change our indexing so that the two columns shown in the details are included in the index. Using the CREATE INDEX statement, use the WITH(DROP_EXISTING=ON) to make this task easier. The DROP_EXISTING still performs the same operations of dropping the index in place already and recreating it, but does it in one statement versus multiple statements. In the new index, we introduce another option called INCLUDE. INCLUDE adds columns to the leaf level of the index and does not require any sorting which makes them more efficient when we are only interested in covering the column in the index. This benefits us by reducing overhead on the index itself.

```
CREATE NONCLUSTERED INDEX IDX_SalesNumOrderDate_Ship_ASC ON
WISSUG.SalesOrderHeader(SalesOrderNumber,OrderDate,AccountNumber)
INCLUDE (customerid,shipdate)
WITH (DROP_EXISTING=ON)
```

```
GO
```



The plan that is created from the index change shows that an index scan persists on the SalesOrderHeader table. This is the case even after covering all of the columns. There is one more tuning task that is needed; structuring the index. Recall that INCLUDE does not sort. In terms of our query, this means CustomerID is forcing a scan due to the use in the WHERE clause.

Index Scan (NonClustered)

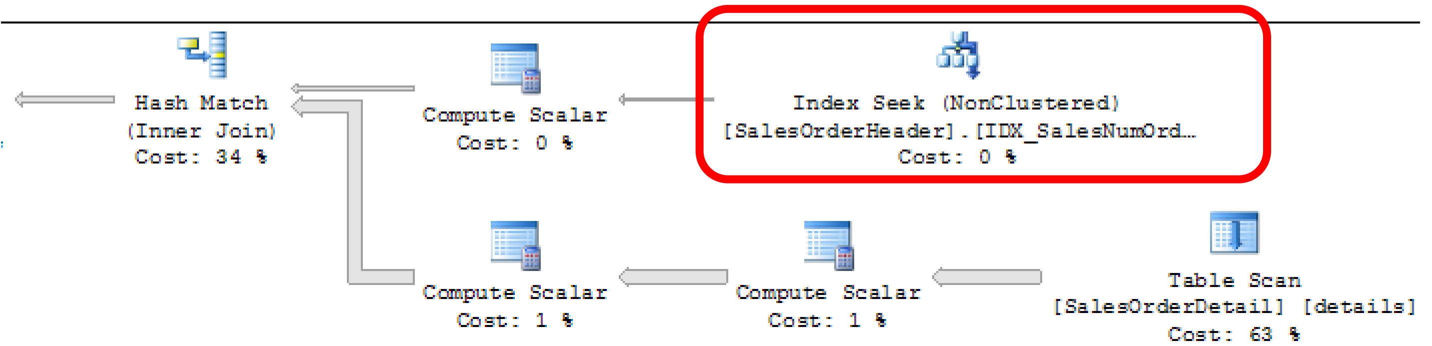
Scan a nonclustered index, entirely or only a range.

Physical Operation	Index Scan
Logical Operation	Index Scan
Estimated I/O Cost	0.187569
Estimated CPU Cost	0.0347685
Estimated Number of Executions	1
Estimated Operator Cost	0.222338 (12%)
Estimated Subtree Cost	0.222338
Estimated Number of Rows	28
Estimated Row Size	65 B
Ordered	False
Node ID	4

Predicate
[DUMP].[WISSUG].[SalesOrderHeader].[CustomerID] as [header].[CustomerID]=(11091)
Object
[DUMP].[WISSUG].[SalesOrderHeader].[IDX_SalesNumOrderDate_Ship_ASC] [header]
Output List
[DUMP].[WISSUG].[SalesOrderHeader].SalesOrderID, [DUMP].[WISSUG].[SalesOrderHeader].OrderDate, [DUMP].[WISSUG].[SalesOrderHeader].ShipDate, [DUMP].[WISSUG].[SalesOrderHeader].SalesOrderNumber, [DUMP].[WISSUG].[SalesOrderHeader].AccountNumber

To restructure the index to handle this specific query, there is a need to move the predicate of CustomerID to the column list. The above image shows the predicate value listed in the tooltip properties. This will force the covering index concept to the include as the ordering is not a requirement for returning those columns.

```
CREATE NONCLUSTERED INDEX IDX_SalesNumOrderDate_Ship_ASC ON WISSUG.SalesOrderHeader(CustomerID)
INCLUDE (SalesOrderNumber,OrderDate,AccountNumber,shipdate)
WITH (DROP_EXISTING=ON)
GO
```



The Index Seek is now present and performed on the SalesOrderHeader table. This is the optimal operation to have here and the index structure accomplishes this.

So far, we've learned about the need for covering indexes and also how the structure of the index is important to accomplishing the needed index seeks. With this information we can move to the second table scan in our plan.

Hover over the table scan on SalesOrderDetail.

Table Scan

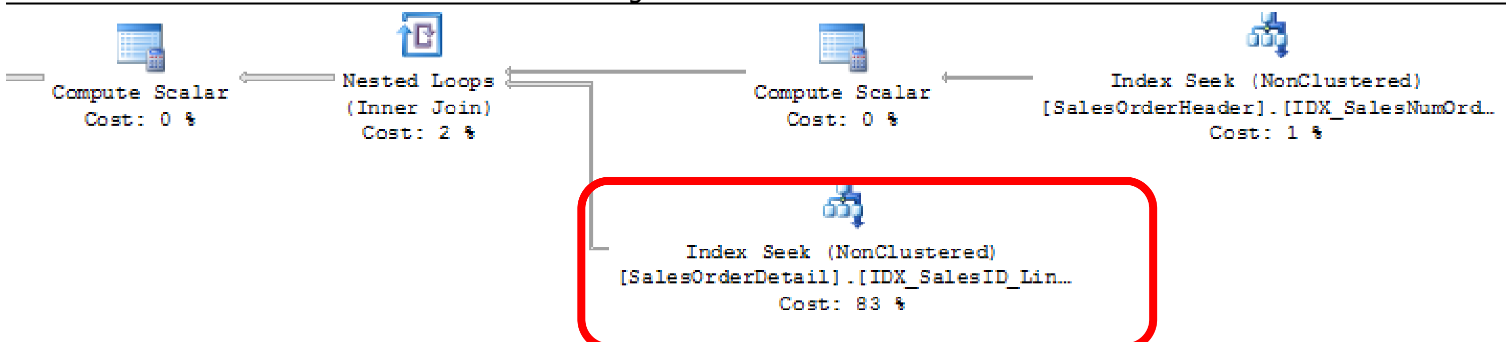
Scan rows from a table.

Physical Operation	Table Scan
Logical Operation	Table Scan
Estimated I/O Cost	0.915718
Estimated CPU Cost	0.133606
Estimated Number of Executions	1
Estimated Operator Cost	1.04932 (63%)
Estimated Subtree Cost	1.04932
Estimated Number of Rows	121317
Estimated Row Size	29 B
Ordered	False
Node ID	10

Object
[DUMP].[WISSUG].[SalesOrderDetail] [details]
Output List
[DUMP].[WISSUG].[SalesOrderDetail].SalesOrderID,
[DUMP].[WISSUG].[SalesOrderDetail].OrderQty,
[DUMP].[WISSUG].[SalesOrderDetail].UnitPrice,
[DUMP].[WISSUG].
[SalesOrderDetail].UnitPriceDiscount

From the details of the table scan, the output of this operation results in the columns SalesOrderID, OrderQty, UnitPrice and UnitPriceDiscount. Looking at the statement, the results required back are UnitPriceDiscount and LineTotal. This tells us that we can use the INCLUDE to capture these columns. In the JOIN condition, SalesOrderID is the next important column. This column requires the index to be sorted so the JOIN can be accomplished without a scan.

With this information we can write the following nonclustered index to cover the SalesOrderDetail table.

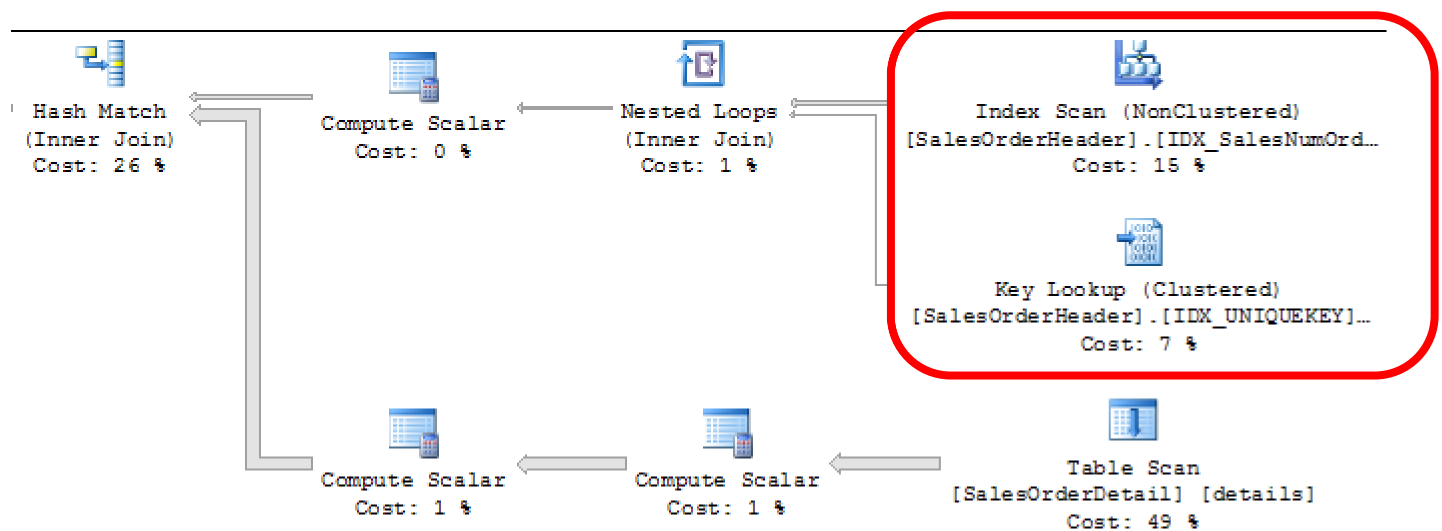


Key Lookup

A Key Lookup operation is performed when a supporting index operation is utilized. However, to fulfill the requirements of the statement, a bookmark lookup must be performed in order to return the remaining output required. This operation can cause a statement's performance to be very poor given the added need to retrieve what is needed to satisfy the requirements.

To show a Key Lookup, alter the index previously created to restructure the columns in order SalesOrderNumber, OrderDate and AccountNumber. Place the CustomerID column in the index by means of INCLUDE.

```
CREATE NONCLUSTERED INDEX IDX_SalesNumOrderDate_Ship_ASC ON
WISSUG.SalesOrderHeader(SalesOrderNumber,OrderDate,AccountNumber)
INCLUDE (customerid)
WITH (DROP_EXISTING=ON)
GO
```



Running the execution plan will show the new index is being used and the Key Lookup operation has been introduced. The lookup in this case is going to return ShipDate. ShipDate is missing from the index so there is missing support to cover this column. Removing the Key Lookup requires the structure of the index we created earlier in order to seek on the index.

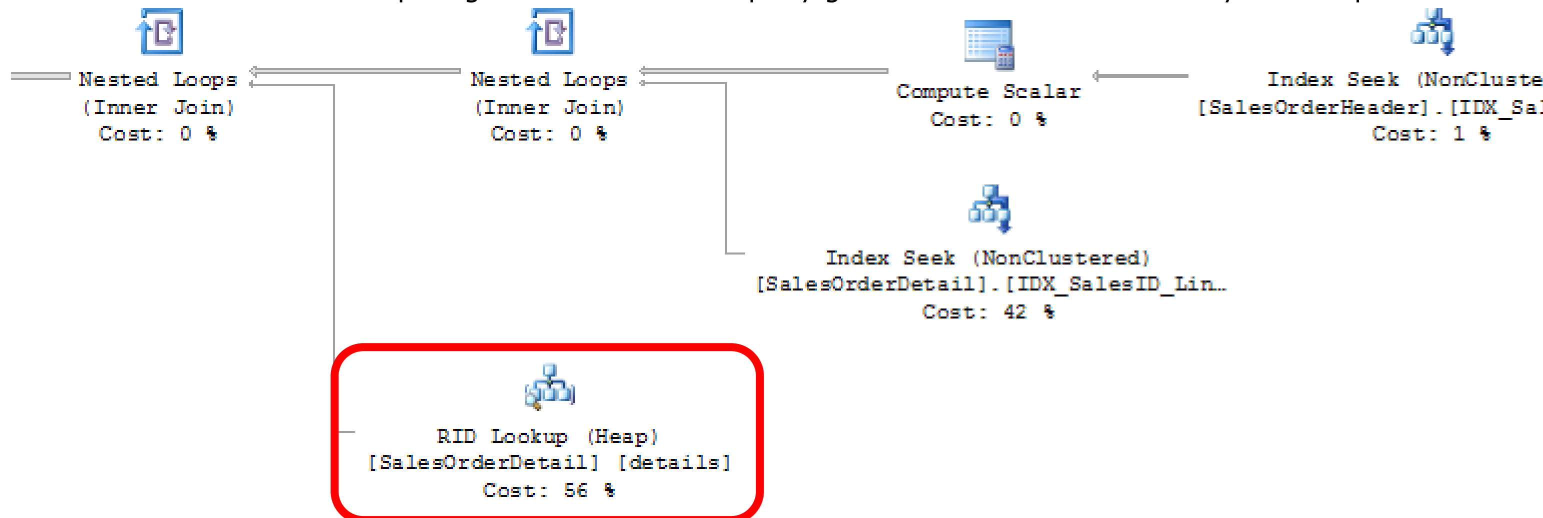
RID Lookup

Recall earlier that SalesOrderDetail did not have a clustered index created on it. A table that does not have a clustered index is referred to as a HEAP. HEAP tables play an important role in the performance of any query that is using them. Without the physical ordering of the table from the creation of a clustered index, the table is not read optimally. In some cases HEAP tables are beneficial and nonclustered indexes can be created on them in order to optimize statements that are running against them.

One of the side effects of a HEAP table is an operation called a RID Lookup. Like a Key Lookup, an index scan or seek is not able to completely satisfy the request so it must do a lookup in order to retrieve the remaining records. In most cases, creating clustered indexes is the best solution to resolve RID Lookup operations. This allows for better plan generation and also helps with maintenance of the table itself in regards to fragmentation. Below we will show how to resolve a RID Lookup by changing our query to additionally check SalesOrderDetail.UnitPriceDiscount where the data is greater than 0.00. This will also be added to the output CarrierTrackingNumber

```
SELECT  SalesOrderNumber ,
        OrderDate ,
        ShipDate ,
        AccountNumber ,
        UnitPriceDiscount ,
        CarrierTrackingNumber ,
        SUM(LineTotal) Total
FROM    WISSUG.SalesOrderHeader header
        JOIN WISSUG.SalesOrderDetail details ON header.salesorderid =
details.salesorderid
WHERE   customerid = 11091
        AND UnitPriceDiscount > 0.00
GROUP BY SalesOrderNumber ,
        OrderDate ,
        ShipDate ,
        UnitPriceDiscount ,
        AccountNumber ,
        CarrierTrackingNumber
```

Take a look at the execution plan generated from this query given the indexes we currently have in place.



If we look at the properties of the RID Lookup, we can see what the RID Lookup output list generates the following:

[PLANLAB].[WISSUG].[SalesOrderDetail].CarrierTrackingNumber.

Also, looking at the nested loop operation, the output list results in the following:

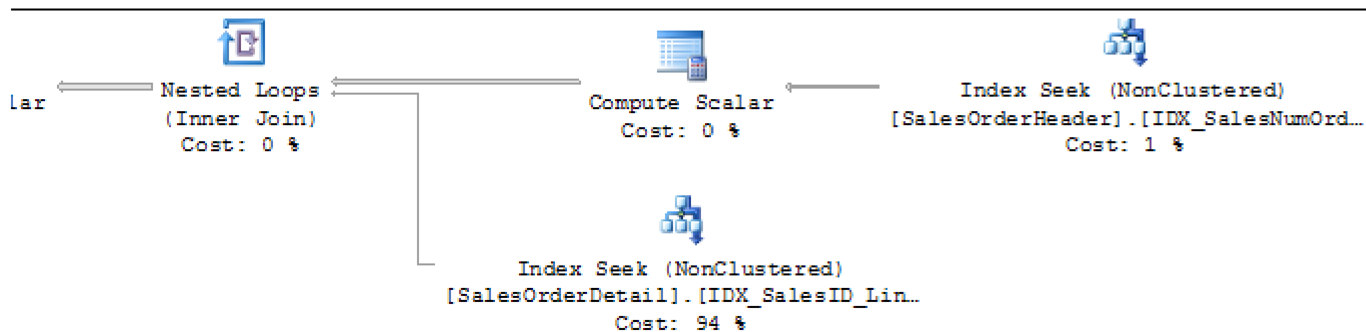
[PLANLAB].[WISSUG].[SalesOrderDetail].CarrierTrackingNumber
[PLANLAB].[WISSUG].[SalesOrderDetail].UnitPriceDiscount
[PLANLAB].[WISSUG].[SalesOrderDetail].LineTotal

Moving to the index seek on SalesOrderDetail, the output list results in

Bmk1003,
[PLANLAB].[WISSUG].[SalesOrderDetail].UnitPriceDiscount
[PLANLAB].[WISSUG].[SalesOrderDetail].LineTotal

Between these three operations we can see that the nested loop is joining the seek output to the lookup output to obtain CarrierTrackingNumber. This tells us that there is a covering index problem. To fix the RID Lookup, we could alter the index IDX_SalesID_LineTotal_ASC or add a new index to support the query. Adding new indexes may not be optimal either if the index still fulfills the needs of the other statements that it was created for. In this case, altering the index to add CarrierTrackingNumber in the INCLUDE section makes more sense given the covering of all the statements we have tested so far.

```
CREATE NONCLUSTERED INDEX IDX_SalesID_LineTotal_ASC ON WISSUG.SalesOrderDetail(salesorderid)
INCLUDE (LineTotal,UnitPriceDiscount,CarrierTrackingNumber)
WITH (DROP_EXISTING=ON)
GO
```



Reviewing the execution plan, the RID Lookup has been resolved.

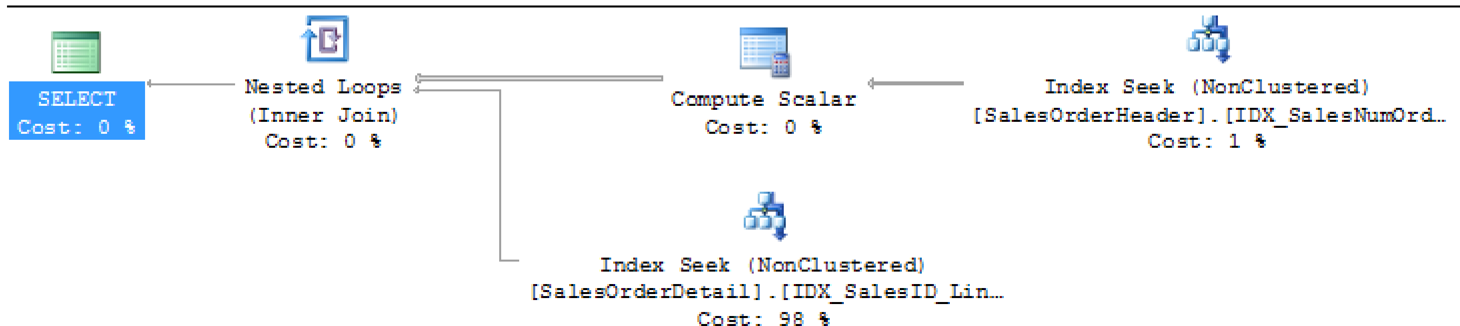
Sorting

Sorting, in general, is an expensive operation as it relates to resources. Order By and Group By forces a sort of the data. Sorting can be ok in execution plans but can also cause a severe performance problem. Since sorting is done in memory, the operation can be fast. However, if memory is not available, the sorting operation can go to the tempdb to accomplish the task which would affect the performance greatly – in a negative way.

In some cases, sorting may not be needed at all in statements. Even in cases when data is required to be sorted, letting front end applications and reporting do this operation may be a better option. With the queries that were used in this article, we can't fix the sort operation because of the SUM() on LineTotal is required forcing us to use a GROUP BY.

Running the example below will show how we could return the results without the SUM(LineTotal) while utilizing something like SSRS functions and expressions to add the SUM(LineTotal) on a grouped result set directly to the dataset at the time the report is rendered.

```
SELECT  SalesOrderNumber ,
        OrderDate ,
        ShipDate ,
        AccountNumber
FROM    WISSUG.SalesOrderHeader header
        JOIN WISSUG.SalesOrderDetail details ON header.salesorderid =
details.salesorderid
WHERE   customerid = 11091
        AND UnitPriceDiscount > 1.00
```



Wrapping up

In the tuning exercises so far, we've created 3 indexes to tune the query. However, we could accomplish this with two indexes. The clustered index could be removed and create another HEAP table situation. This situation is also common when the primary key (or clustered index) is another column on the table that isn't part of the query and thus does not assist in improving performance of the statement.

To show this, drop the clustered index SalesOrderHeader.IDX_UNIQUEKEY

```
DROP INDEX WISSUG.SalesOrderHeader.IDX_UNIQUEKEY
GO
```

To fix this without a clsutered index on SalesOrderID, change the final nonclustered index to add SalesOrderID

```
CREATE NONCLUSTERED INDEX IDX_SalesNumOrderDate_Ship_ASC ON WISSUG.SalesOrderHeader(CustomerID,SalesOrderID)
INCLUDE (SalesOrderNumber,OrderDate,AccountNumber,shipdate)
WITH (DROP_EXISTING=ON)
GO
```

Running the plan again will output the same plan as we accomplished earlier with three separate indexes.

Looking closer at the JOIN conditions, index scans may be seeks. If our bottom join is returning a very limited number of rows, the optimizer will not need to scan on the top join in order to join the operations. This would be relevant when a clustered index is joined from the top.

Example: Prepare this condition by dropping the nonclustered index on SalesOrderHeader we created earlier.

In our previous examples, this would force a scan on SalesOrderHeader if UnitPriceDiscount = 0.00 was filtered. This is due to our estimated rows being 21 causing the results to be very small in size. If we change to UnitPriceDiscount > 1.00 and rerun the estimated plan, the resulting scan turns into a seek and estimated number of rows of 1.

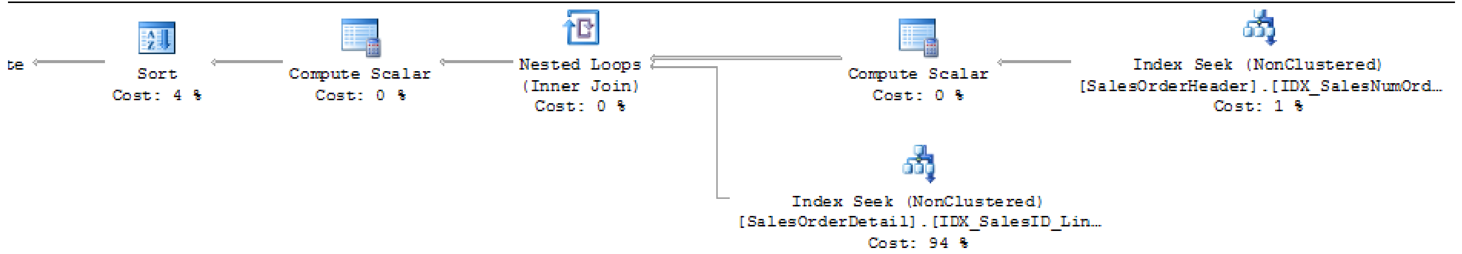
Note: This is for example purposes only and to show the resulting joins make a difference in the operations that are shown by the optmizer. You should always tune by the concept that data grows in size and prepare for that in your indexing and development of T-SQL.

```
DROP INDEX WISSUG.SalesOrderHeader.IDX_SalesNumOrderDate_Ship_ASC
GO
```

And now rerun the execution plan with the change to UnitPriceDiscount

```
SELECT  SalesOrderNumber ,
        OrderDate ,
        ShipDate ,
        AccountNumber ,
        UnitPriceDiscount ,
        SUM(LineTotal) Total
FROM    WISSUG.SalesOrderHeader header
        JOIN WISSUG.SalesOrderDetail details ON header.salesorderid =
details.salesorderid
WHERE   customerid = 11091
        AND UnitPriceDiscount > 1.00
GROUP BY SalesOrderNumber ,
        OrderDate ,
        ShipDate ,
        UnitPriceDiscount ,
```

AccountNumber



Take away

Execution plan analysis is a critical aspect to keeping SQL Server running optimally. With any amount of hardware, the risk factor of a poorly written or unsupported statement being executed in a production environment has the ability to bring service availability to a halt. Testing on a futuristic sizing of tables and indexes is also a part of tuning that should be taken into account. Data will grow and, with that growth, queries will react differently if not tuned to their full potential.

Special thanks to Janice Lee, George Mastros, Howard Churchill, Denis Gobo and Jes Borland for the reviews of this paper. The SQL Community is simply amazing!