Microsoft EDW Architecture, Guidance and Deployment Best Practices

# Chapter 5: Querying, Monitoring, and Performance Tuning

**By Microsoft Corporation**

**Acknowledgements:**

**Contributing writers from Solid Quality Mentors**: James Miller

**Technical reviewers from Microsoft**: Benjamin Wright-Jones, Steve Young, Ross LoForte

**Technical reviewers from Solid Quality Mentors**: Gianluca Hotz, Larry Barnes

**Contributing editors from Solid Quality Mentors**: Kathy Blomstrom

**Published:**

**Applies to:** SQL Server 2008 R2

[Chapter 5: Querying, Monitoring, and Performance Tuning 1](#_Toc274453972)

[Introduction 4](#_Toc274453973)

[Query Optimization Overview 4](#_Toc274453974)

[Developing Queries with Attention to Both ‘What’ and ‘How’ 5](#_Toc274453975)

[Querying 5](#_Toc274453976)

[Query Execution Plans 6](#_Toc274453977)

[Sample Database Overview 6](#_Toc274453978)

[Benefits of De-normalization from a Data Warehouse Query Perspective 7](#_Toc274453979)

[Partitioned Tables 8](#_Toc274453980)

[Queries Based on the Partitioned Column 9](#_Toc274453981)

[Queries Indirectly Based on a Partitioned Column 11](#_Toc274453982)

[Querying with Partitioned Indexes 14](#_Toc274453983)

[Queries on a Partitioned Heap 17](#_Toc274453984)

[Covering Indexes 18](#_Toc274453985)

[Filtered Indexes 19](#_Toc274453986)

[Indexed Views 20](#_Toc274453987)

[Routine Aggregation Using Summary Tables in the Data Warehouse 21](#_Toc274453988)

[Conclusion 21](#_Toc274453989)

[Monitoring 23](#_Toc274453990)

[Overview 23](#_Toc274453991)

[Key Query Performance Metrics 23](#_Toc274453992)

[Options for Monitoring T-SQL Code 24](#_Toc274453993)

[Setting Up the Management Data Warehouse and Data Collection Components 25](#_Toc274453994)

[Standard Management Data Warehouse Reporting 30](#_Toc274453995)

[Querying the SQL Trace Collection in the Management Data Warehouse 35](#_Toc274453996)

[Performance Tuning 37](#_Toc274453997)

[Overview 37](#_Toc274453998)

[Targeted Performance Tuning 37](#_Toc274453999)

[Query Tuning Based on Disk I/O 38](#_Toc274454000)

[Working with Execution Plans 43](#_Toc274454001)

[Join Operators 45](#_Toc274454002)

[Isolation Level Considerations 50](#_Toc274454003)

[Best Practices Summary 52](#_Toc274454004)

[Conclusion and Resources 52](#_Toc274454005)

[Additional Resources 53](#_Toc274454006)

## Introduction

Data warehouses are read-optimized versions of line-of-business (LOB) systems and other source feeds. The sheer volume of data in a data warehouse can result in queries taking hours to complete while consuming large amounts of server resources. Worst case, queries may never complete within the required time, while negatively impacting other data warehouse workloads. Successful data warehouses require development team members to have a deeper understanding of query dynamics, as well as of best practices for monitoring, and performance tuning.

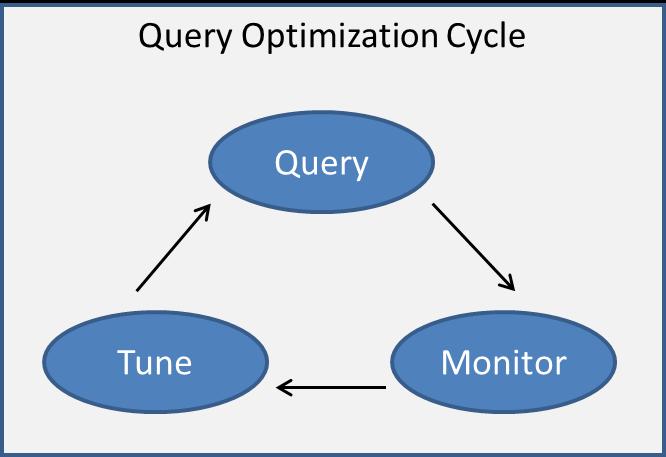
This chapter’s audience is the data warehouse team members responsible for developing and optimizing queries and monitoring data warehouse query workloads on the Microsoft SQL Server platform.

This chapter is organized into following sections:

* Querying
* Monitoring
* Performance tuning
* SQL Server best practices
* Conclusion and resources

## Query Optimization Overview

Figure 5-1 shows how querying, monitoring, and tuning are related.

****

**Figure 5-1:** Query optimization cycle

Database professionals responsible for building large data warehouse applications need to be equipped with knowledge of SQL Server techniques such as table partitioning, indexing strategies, query isolation levels, execution plans, SQL Profiler, ETL techniques, and a variety of performance-related methods for optimization.

However, this is only a first step. The data warehouse also needs to be systematically monitored for excess resource utilization and anomalous behavior. Metrics collected from monitoring processes should enable query and server base lining, expose performance trending, and help with long-term capacity planning.

A key deliverable of a monitoring solution should be the ability to pinpoint reoccurring queries that carry an exceptionally high database cost. This information should then form a “to-do” list for subsequent performance tuning efforts. This approach should provide the highest return on investment for the time devoted to optimizing data warehouse queries.

### Developing Queries with Attention to Both ‘What’ and ‘How’

There are times when it is enough to simply know the answer—the “what.” For example, when your spouse asks, “What time do you plan to be home from work?” you can generally give an ETA without delving into the method of calculation based of tasks, promises, deadlines, and commuting hazards that stand in your way.

Contrast this to the “what and how” of a common database query. Suppose your business consumers need to determine the number of orders placed for a widget during the previous day. Of paramount importance is the “what” (i.e., the correct number of orders), which you can produce by properly incorporating business rules and database logic. However, you may also need to examine how the results are derived if the query runs too long.

Now consider a large data warehouse query. In this example, let’s say you need to determine the changes to sales volume based on a correlation to Internet banner ad impressions. This scenario typically requires the analysis of large tables and filtering based on a variety of criteria. You, of course, still need accuracy, but the *query plan* may become critical, especially if the execution time expands from minutes to hours, and beyond. Bottom line: You will want to become very familiar with estimated and actual query plans when working in a data warehouse—or in other words, the “how” of your SQL queries.

**Note:** The term *query* in this chapter refers to all SELECT, INSERT, UPDATE, and DELETE statements that are often batched together in scripts, stored procedures, and ad hoc queries.

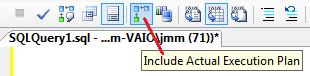
## Querying

This section covers querying SQL Server with an emphasis on the challenges encountered with large queries in a data warehouse. We’ll start with a bird’s-eye view of a query plan, and then look at the common approaches to managing query response times for very large tables: namely, optimized schemas, table partitioning, and targeted indexes.

Scripts are included for this chapter to both generate sample data and query data illustrating the techniques we cover. Due to the random nature of generating test data and the various platforms that run SQL Server, your results may vary slightly, but the underlying query fundamentals should remain the same.

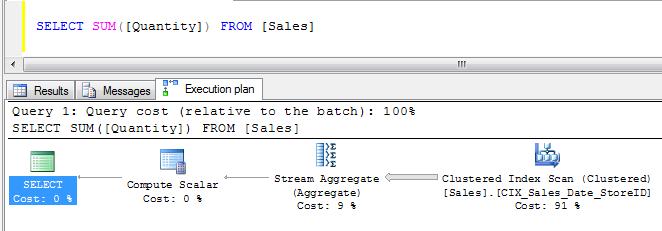
### Query Execution Plans

Throughout this chapter, we will be referring to query plans and, specifically, the *graphical* query plan, which is accessible through SQL Server Management Studio (SSMS). When querying in SSMS, you can enable the display of the query plan by clicking the icon shown in Figure 5-2.



**Figure 5-2:** Enabling the graphical execution plan

Once you’ve enabled the query plan display, and after a query completes, an additional Execution Plan tab will appear in the result pane containing the plan, which details how SQL Server actually resolved, or processed, the query. Figure 5-3 shows a simple query and its execution plan.



**Figure 5-3:** Sample graphical execution plan

In this example, the Execution Plan is contained in the last of the three tabs presented in the lower query result pane. A brief *right-to-left* reading of the plan indicates that 91% of the query cost is due to a Clustered Index Scan, followed next by a Stream Aggregate operation with a 9% cost. You can obtain details of these operations, as well as the connecting arrows (i.e., row counts), by hovering over a graphical section of the execution plan. The width of the arrows is also a visual indication of the relative number of rows that are fed into or result from an operation.

We will refer to query execution plans throughout this chapter. For more information about graphical query execution plans, see [Displaying Graphical Execution Plans (SQL Server Management Studio)](http://msdn.microsoft.com/en-us/library/ms178071.aspx).

### Sample Database Overview

This chapter includes two scripts that are used to create sample data and three scripts for querying this sample data as well as data collected by SQL Server’s Management Data Warehouse (described later in this chapter). To gain a better understanding of the concepts presented in this chapter, you are encouraged to download and run the scripts associated with this chapter.

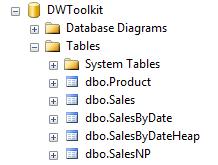
The following T-SQL scripts can be downloaded from the [Data Warehouse Toolkit area of Codeplex](file:///C:\Users\styoung\Documents\My%20Clients2\DW%20Book%20Review\Data%20Warehouse%20Toolkit%20area%20of%20Codeplex):

|  |  |
| --- | --- |
| Script | Description |
| #1 - Create DWToolkit | Script to create the DWToolkit database |
| #2 - Populate DWToolkit | Script to populate DWToolkit with randomly generated data |
| #3 - Querying Section | T-SQL code to run and test with in the Querying section |
| #4 - Monitoring Section | T-SQL code to run and test with in the Monitoring section |
| #5 - Performance Tuning Section | T-SQL code to run and test with in the Performance Tuning section |

Script #1 simply creates the database called DWToolkit. You will want to ensure that this database has been *successfully* created before running Script #2, which populates the database.

The remaining scripts (#3, #4, and #5) correspond to the sample code used in the three major sections of this chapter: Querying, Monitoring, and Performance Tuning. They are not intended to be run as a part of the DWToolkit database setup.

Figure 5-4 shows the tables you should see in the DWToolkit database after running Script #2.



**Figure 5-4:** Table container for the DWToolkit database

The four Sales tables that are created have almost identical content in terms of the number of rows and columns. What *is* different with these tables is their physical organization in SQL Server. Table 1 lists the design purposes of these tables.

|  |  |
| --- | --- |
| Table | Design |
| Sales | Partitioned by a *cyclical* month number (rotating window) |
| SalesByDate | Partitioned by month (sliding window) with a Clustered Index |
| SalesByDateHeap | Partitioned by month (sliding window) as a Heap |
| SalesNP | No partitioning |

**Table 1:** Physical organization of the Sales tables

The Product table contains the item numbers referenced by all of the Sales tables. You can rerun Script #2 at any time to regenerate the test data.

### Benefits of De-normalization from a Data Warehouse Query Perspective

Despite the storage efficiency and design elegance of a normalized database schema, the simple fact is that there are query costs associated with joining tables together. The more tables and rows you need to join, the greater this expense. For OLTP systems, the cost of joins is often mitigated by using highly selective queries. But for data warehouses, with potentially billions of rows of data and aggregations covering large data sets, the query expense for table joins can be considerable.

Other query benefits to de-normalization include a simplified, more intuitive schema; encapsulation of database rules; a reduction in foreign keys; and broader index opportunities. Another perhaps underrated feature of de-normalization is the benefit of working with more understandable query plans.

Ideally, the plan devised by SQL Server should illustrate any inefficiencies by displaying thick connecting lines and large sub-tree costs for expensive operations. However, if the plan itself contains hundreds of steps, it may be difficult and time-consuming to discern the actual bottlenecks and circumvent a performance issue.

Another consideration when dealing with highly complex query plans is that the optimizer does not perform an exhaustive search for the best query plan. Instead, it applies some heuristics to lower the number of plans to be evaluated in order to come up with a good plan in a reasonable amount of time. In other word, the optimizer will perform a *fast* search for a *good* query plan, which may preclude the *best* plan.

Although there are benefits of de-normalization, it’s also understood that this schema design technique is not always possible or practical. In the event that de-normalized techniques are not implemented within your data warehouse, you can still use strategies such as indexed (materialized, schema bound) views and summary tables. These practices can also significantly reduce the query costs associated with huge joins and table aggregations (discussed later in this chapter).

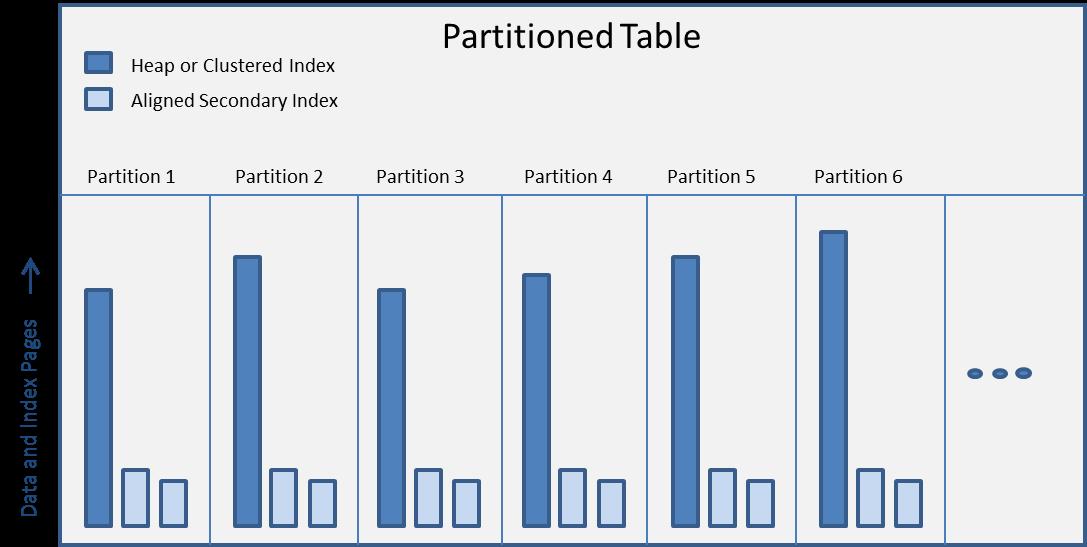
### Partitioned Tables

It’s not unusual to have tables in a data warehouse that contain billions of rows of data. With tables of this size, querying, inserting and updating, and maintenance tasks such as data archiving and re-indexing require more than casual consideration.

One approach to managing very large tables is to take advantage of table partitioning. This strategy essentially breaks a large table into smaller parts and allows independent partition access for faster queries as well as other maintenance tasks.

You will want to keep in mind that partitioned tables often require some finesse to ensure maximum query performance. The diagram in Figure 5-5 shows the logical organization of a partitioned table. A partition *function,* which is based on *range partitioning,* is used to direct table rows to a given physical partition number. Each partitioned table can have up to 1000 partitions. Index pages can be either aligned (as shown in Figure 5) or non-aligned with the data pages of the partitioned table. Aligning an index with the partitioned table is preferred since SQL Server can switch partitions quickly and efficiently while maintaining the partition structure of both the table and its indexes.

**Note:** a modulus column can also be used to implement *hash partitioning*, and will be demonstrated later in this section.



**Figure 5-5:** Partitioned table organization

### Queries Based on the Partitioned Column

To get started, let’s look at a table called SalesByDate, which has 40 partitions based on month.

**Note:** Any date division (day, week, etc.) can be used for a partition as long as the total range distributions do not exceed 1000 active partitions. These can also be based on calendar, fiscal or other date designations.

The partition function in this example uses a partition data type of Date, as illustrated in the following code snippet:

CREATE PARTITION FUNCTION [SalesByDate\_PartitionFunction](DATE)

AS RANGE RIGHT FOR VALUES

( '2008-02-01', '2008-03-01', '2008-04-01', '2008-05-01', ... )

**Note:** This scheme can also be the basis of a monthly *sliding window* partitioning strategy where, each month, the oldest partition is switched out and then merged (i.e., collapsed) and a new partition is split (created) from the most recent partition. For more information on maintaining ‘sliding window’ partitioned tables see: <http://blogs.msdn.com/b/menzos/archive/2008/06/30/table-partitioning-sliding-window-case.aspx>

Now let’s submit a query to determine the total quantity of items sold during the most recent 7 days:

-- Script Q-1

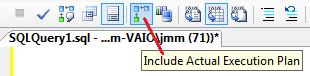
SELECT SUM([Quantity])

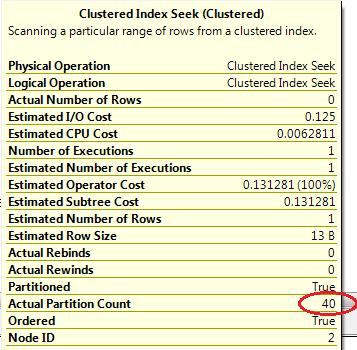
FROM [SalesByDate]

WHERE [Date] BETWEEN GETDATE()-7 AND GETDATE()

Intuitively, you know that the query should probably access just one or, at the most, two partitions. However, if the query scans each partition to satisfy the date predicate (i.e., within the previous 7 days), it’s clear that the partition design is not being leveraged for improved query performance.

You can determine the number of partitions being accessed in a query by hovering over a scan or seek operation in a graphical execution plan (discussed later in this chapter) to see if it matches your expectations, as Figure 5-6 shows.





**Figure 5-6:** Partitions used in sample Seek operation

In this example, all 40 partitions were accessed, meaning that the optimizer did not take advantage of partition elimination based on the relationship between the date-based partition scheme and the date-related predicate. There are a variety of reasons for this behavior, including:

* The use of certain functions and expressions
* Implicitly converted data types within the WHERE clause

In the above example, the problem is with the data type (datetime) returned by the GETDATE() function. The following revised code should correct this issue by using the CONVERT() function to change the predicate comparison to a DATE data type. This modification will alter the plan to access just one or two partitions:

-- Script Q-2

SELECT SUM([Quantity])

FROM [SalesByDate]

WHERE [Date] BETWEEN CONVERT(DATE, GETDATE()-7)

AND CONVERT(DATE, GETDATE())

Hover over the Clustered Index Seek operation within the resulting query plan to verify the number of partitions that are accessed.

An alternative solution for this problem is to encapsulate the logic into a stored procedure and pass the dates as parameters that explicitly use the DATE data type, as follows:

-- Script Q-3

CREATE PROCEDURE [SumQuantityByDateRange] @begDate DATE, @endDate DATE

AS

SELECT SUM([Quantity])

FROM [SalesByDate]

WHERE [Date] BETWEEN @begDate AND @endDate

Example stored procedure execution:

-- Script Q-4

DECLARE @begDate date = GETDATE()-7

DECLARE @endDate date = GETDATE()

EXEC [dbo].[SumQuantityByDateRange] @begDate, @endDate

This approach enforces the proper data type usage for the partitioned table query.

### Queries Indirectly Based on a Partitioned Column

For this example, we’ll use a table called Sales, which has 40 fixed partitions also based on month. However, in this case, the partition function uses a data type of smallint, which is derived from the following date-based expression:

(((DATEPART(yyyy,[Date])\*12)+(DATEPART(mm,[Date])))%40)+1 AS [Partition]

This *hash* formula returns a number between 1 and 40, which is then included as a column in the fact table. The associated partition function in this scenario is as follows:

CREATE PARTITION FUNCTION [Sales\_PartitionFunction](SMALLINT)

AS RANGE LEFT FOR VALUES

( 1, 2, 3, 4, 5, 6, 7, 8, 9,

10, 11, 12, 13, 14, 15, 16, 17, 18, 19,

20, 21, 22, 23, 24, 25, 26, 27, 28, 29,

30, 31, 32, 33, 34, 35, 36, 37, 38, 39 )

**Note:** This scheme is useful for adopting a monthly *rotating window* partitioning strategy where the partitions are static and eventually reused based on a date expression. Using this design, the sole maintenance requirement is to switch out the oldest partition (potentially, a very fast metadata-only operation) before the partition formula cycles back around to reuse a previously populated partition.

A consequence of having an indirect relationship between the partitioned column and, in this case, a date-based query predicate is that the optimizer will not be able to automatically use partition elimination because of the *indirect* predicate relationship. As an example, let’s look at the query we used earlier to access the minimum number of partitions, but this time, modified to use the Sales table with the above partition scheme:

-- Script Q-5

SELECT SUM([Quantity])

FROM [Sales]

WHERE [Date] BETWEEN CONVERT(DATE, GETDATE()-7)

AND CONVERT(DATE, GETDATE())

Viewing the execution plan for this query, you can see that the Clustered Index Seek operation accesses all 40 partitions, even when being sure to match the data type in the predicate. This is because the relationship of the date predicate to the derived partition column is not recognized by the optimizer. As you can see, you will need to directly code optimal partition elimination to effectively tune this query. To do so, you will need to include the partition column in the query predicate.

To set the stage for this query modification, let’s take a closer look at the physical partitions for the Sales fact table using the following query:

-- Script Q-6

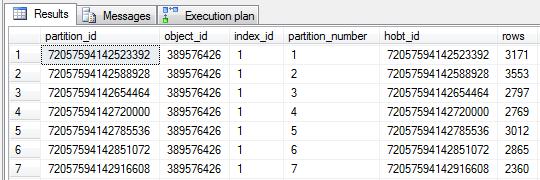
SELECT \*

FROM sys.partitions

WHERE OBJECT\_NAME(OBJECT\_ID)='Sales'

ORDER BY [index\_id], [partition\_number]

The result set in Figure 5-7 details each table partition. Notice that column 4 lists the actual partition number, and column 6 lists the number of rows for each partition.



**Figure 5-7:** Sample results from sys.partitions

Next, you can include the $Partition function in the WHERE clause to view the contents of a single partition. For example, the following query lists only the rows contained in partition number 3:

-- Script Q-7

SELECT TOP 1000 \*

FROM [dbo].[Sales]

WHERE $PARTITION.Sales\_PartitionFunction([Partition]) = 3

To assist in routinely determining the correct partition number, you will probably want to create a scalar function that is based on the partition expression used earlier. This will help you easily return the proper partition number that can be used for routine queries:

-- Script Q-8

CREATE FUNCTION [dbo].[GetSalesPartition] (@date DATE)

RETURNS SMALLINT

AS

BEGIN

RETURN ((((DATEPART(yyyy,@date)\*12)+(DATEPART(mm,@date)))%40)+1)

END

Leveraging these techniques, you can now alter the original query and benefit from partition elimination:

-- Script Q-9

SELECT SUM([Quantity])

FROM [Sales]

WHERE [Date] BETWEEN GETDATE()-7 AND GETDATE()

AND $PARTITION.Sales\_PartitionFunction([Partition])

IN ( [dbo].[GetSalesPartition] ( GETDATE()-7 )

, [dbo].[GetSalesPartition] ( GETDATE() ) )

At this point, you may be wondering why not simply use the Partition column directly, instead of the more verbose $PARTITION.Sales\_PartitionFunction([Partition]) function, as shown in the following query:

-- Script Q-9

SELECT SUM([Quantity])

FROM [Sales]

WHERE [Date] BETWEEN GETDATE()-7 AND GETDATE()

AND [Partition]

IN ( [dbo].[GetSalesPartition] ( GETDATE()-7 )

, [dbo].[GetSalesPartition] ( GETDATE() ) )

One reason, which happens with the above query, is that the optimizer misses partition elimination unless you use the $PARTITION function. Another important reason is that value of the Partition column will not necessarily equate the *physical* partition number returned by the $PARTITION function. For example, if the partitioning strategy for this table used a partition function of RANGE ‘RIGHT’ instead of RANGE ‘LEFT’, the actual value of the Partition column would be offset by 1 when compared to the number returned by the $PARTITION function.

As you can see, additional considerations are required when querying partitioned tables. Attention to the details of query plans, such as partition usage, can save many hours of run time and improve response times for all concurrently executing queries in your data warehouse environment.

### Querying with Partitioned Indexes

In the same way that partitioned tables require additional query consideration, so do the indexes on these tables. Fortunately, unless you specify a different partition scheme or a separate filegroup, indexes created on partitioned tables will automatically be aligned with the underlying table partitions. Building on the earlier example, this means that if a table has 40 partitions, so will the accompanying indexes for that table. This is a clear benefit for data management language (DML) operations, parallel execution plans, and partition maintenance (i.e., SWITCH, SPLIT, and MERGE), but it may also introduce query performance issues.

As an example, let’s compare the same aggregate query on two tables: the SalesByDate table, which is partitioned by date, and the SalesNP table, which is identical to the first table except that it is not partitioned. In the following query, our goal is to use a non-clustered, non-unique index (defined in both tables) that contains a single column (StoreID) in ascending order. The query will look for the total sale count for store number 50 during the previous month. Because we want to know the database read performance for this query, let’s begin by using the following SET command:

SET STATISTICS IO ON

Now run the following query on the non-partitioned table to get a benchmark of the read performance:

-- Script Q-11

SELECT COUNT(\*) AS [SaleCount]

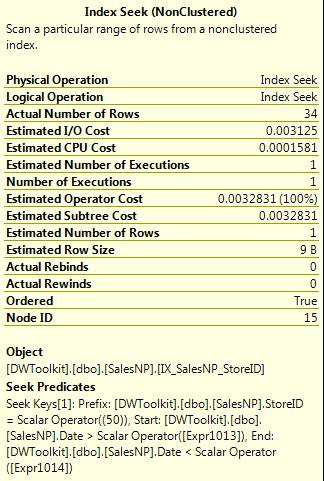
FROM [SalesNP]

WHERE [StoreID] = 50

AND [Date] BETWEEN DATEADD(m, DATEDIFF(m, 0, GETDATE()) - 1, 0)

AND DATEADD(m, DATEDIFF(m, 0, GETDATE()), 0) -1

The value returned for SalesCount will vary depending on the generated test data, but our interest here is in query cost and total disk I/O. The graphical query plan in Figure 5-8 shows a single non-clustered Index Seek operation having a .003 estimated subtree cost.



**Figure 5-8:** Cost of Index Seek operation

In this case, the index used is named IX\_SalesNP\_StoreID, which is our intention. If you click the messages tab in SSMS, it reveals that only two logical reads were required to complete the query. This is pretty impressive knowing that the table itself has 100,000 rows. One reason for this favorable performance is that SQL Server includes the clustering key for secondary indexes defined on tables with a clustered index (recall that the tables in this example are clustered on the Date column).

The benefit of having the clustering key included in each row of the secondary indexes is that it expands the *covering* effect for queries based on the index, which is arguably one of the most effective performance tuning techniques available to developers. The only downside is the additional overhead of supplementing all secondary indexes with the clustering key.

Now that you have a good baseline for this query, you can run it again using the SalesByDate table, which is partitioned by date:

-- Script Q-12

SELECT COUNT(\*) AS [SaleCount]

FROM [SalesByDate]

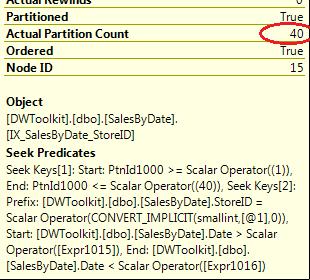
WHERE [StoreID] = 50

AND [Date] BETWEEN DATEADD(m, DATEDIFF(m, 0, GETDATE()) - 1, 0)

AND DATEADD(m, DATEDIFF(m, 0, GETDATE()), 0) -1

In this case, there are 66 logical reads and an estimated sub-tree cost of .131 for the same Index Scan operation. While it’s true that two logical reads versus 66 is not a big concern, it is 33 times greater than the baseline I/O performance of the same secondary index on a non-partitioned table. It’s also worth noting that the more partitions a table has, the greater this problem will be. When extrapolated for a fact table containing billions of rows, this cost may be significant.

So let’s dive into what happened with the partitioned side of this query. The first thing to note when viewing the query plan, shown in Figure 5-9, is that it accessed 40 partitions of a secondary index called IX\_SalesByDate\_StoreID.

****

**Figure 5-9:** Index Seek (Non-Clustered) - Seek Predicates

At first glance, this appears to make sense because the secondary index on StoreID is partition aligned, thereby creating 40 distinct sections to this index. On the other hand, the partitions are aligned by the Date column, which is included in the query predicate. This should automatically invoke the optimizer’s partition-elimination capabilities.

A closer examination of the Seek Predicates in the operation above reveals that there are two Seek Keys being used. The first, Seek Key [1], is used to qualify the specific partition, and the second, Seek Key [2], is used to filter on both StoreID and Date. This new operation in SQL Server 2008, known as Skip Scan, provides for two levels of search conditions in a single operation. One seek is used for the partition number, and the other seek is used for a secondary search condition. The Skip Scan operation is also an indication that the query may potentially access all partitions. In the case above, seven partitions are completely empty, so the 66 reads are based on two reads each of the 33 populated partitions.

To improve performance, similar to our earlier optimization with the clustered index, we need to explicitly convert the embedded GETDATE() function into the DATE data type, conforming to the data type of the partition function:

-- Script Q-13

SELECT COUNT(\*) AS [SaleCount]

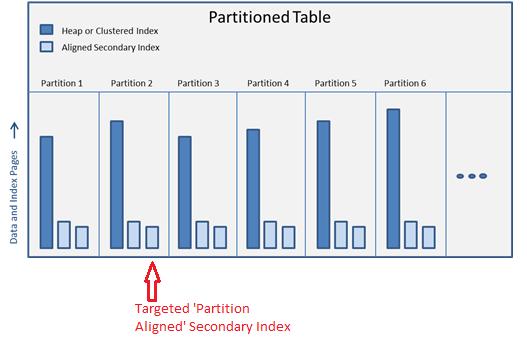
FROM [SalesByDate]

WHERE [StoreID] = 50

AND [Date] BETWEEN CONVERT(DATE, DATEADD(m, DATEDIFF(m, 0, GETDATE()) - 1, 0))

AND CONVERT(DATE, DATEADD(m, DATEDIFF(m, 0, GETDATE()), 0) -1)

This technique ensures that we achieve partition elimination and access only one partition of this secondary index, as illustrated in Figure 5-10.



**Figure 5-10:** Example of targeting a single index partition

This modification reduces the query overhead to just two reads, providing the same performance in this case as accessing a non-partitioned table index.

### Queries on a Partitioned Heap

As its name implies, there is no specified order for the data provided in a heap. However, if the table participates in a partitioning scheme, you do have some degree of selectivity available, at least to the extent of the granularity defined by your partition function. With a total of 1000 partitions available per table, you could choose to have nearly 3 years of data partitioned by day. This approach would partially simulate a clustered index on Date without the related index overhead.

To help evaluate this option, let’s use another monthly partitioned table called SalesByDateHeap, which is the same as the SalesByDate table but without a clustered index. The secondary index based on StoreID remains on the heap. However, keep in mind that the Date column is not automatically included in the secondary index, as was the case with the SalesByDate clustered table used in the last example.

Because the following query is aligned perfectly with the monthly partition scheme, the omission of the Date column from the secondary index, IX\_SalesByDateHeap\_StoreID, has no effect on query performance:

-- Script Q-14

SELECT COUNT(\*) AS [SaleCount]

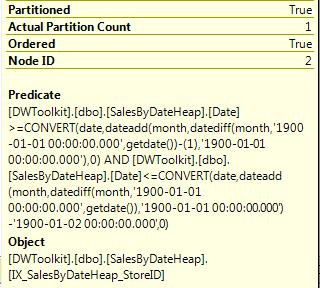
FROM [SalesByDateHeap]

WHERE [StoreID] = 50

AND [Date] BETWEEN CONVERT(DATE, DATEADD(m, DATEDIFF(m, 0, GETDATE()) - 1, 0))

AND CONVERT(DATE, DATEADD(m, DATEDIFF(m, 0, GETDATE()), 0) -1)

The Messages tab in SSMS reports that this query took only two logical reads, providing the same benefit as the clustered version of this table. By examining the execution plan, which Figure 5-11 shows, you can also confirm that the secondary index (IX\_SalesByDateHeap\_StoreID) was used and that only one partition was accessed.



**Figure 5-11:** Index Seek (Non-Clustered)

It’s worth noting here, at least for this chapter’s modest set of test data (100,000 rows), that there was close to a 20% decrease in overall disk I/O when populating the SalesByDateHeap table compared with the disk I/O required to populate the SalesByDate clustered index table. This isn’t to imply that heaps are always faster or that you should avoid using a clustered index with partitioned tables. However, you may want to consider using a heap, along with supporting secondary indexes, especially if the granularity of the partitions partially imitates the ordered sequence of the clustered index. In the case of heap fragmentation (less likely with fact tables), SQL Server 2008 now also provides a method to rebuild heaps.

### Covering Indexes

As mentioned earlier, covering indexes are a proven technique for boosting SQL query performance, and this holds true in both OLTP and data warehouse environments. As a quick review, a covered index is an index that contains all required columns from an underlying table needed for all or part of a query. The primary reasons for the performance improvement with covering indexes are:

* Row look-ups for additional columns to the primary table are avoided
* The order and selectivity of the secondary index may benefit the query
* The index row width is usually much narrower than the actual table (clustered or heap), resulting in fewer database reads

**Include Column Option**

Beginning with SQL Server 2005, secondary indexes can take advantage of the INCLUDE clause to add non-key columns to the leaf level of an index. The good news is that this technique increases the likelihood that your index will cover a query and has less maintenance overhead than appending the column(s) as a larger, composite key. It also provides for including LOB data in an index, such as varchar (max). On the downside, this approach increases the width of the index, resulting in fewer index rows included with each database read.

Taken to the extreme, a covered index could consume the same disk space and overhead of the table itself. In a data warehouse environment, this could be significant. Consequently, you shouldn’t add columns to an index without first evaluating the proportion and relative cost of queries that would actually benefit from the broader index.

### Filtered Indexes

SQL Server 2008 has introduced a new filtered index feature that can dramatically optimize queries that have an aligned predicate. For example, let’s say that a sales quantity of less than zero indicates a product return, and let’s further assume that there are several common queries used to manage these returns. In this scenario, we can anticipate that queries may often have the predicate of ‘[Quantity] < 0’. To understand the potential query impact, let’s start by benchmarking two queries: one based on a clustered index and one based on a heap:

-- Script Q-15

SELECT COUNT(\*) AS [ReturnCount]

FROM [SalesByDate]

WHERE [Quantity] < 0

SELECT COUNT(\*) AS [ReturnCount]

FROM [SalesByDateHeap]

WHERE [Quantity] < 0

Using feedback from STATISTICS IO, you can see that the first query, based on a clustered index, took 501 reads and the second query, based on a heap, took 363 reads (SQL Server Profiler reported 501 and 427, respectively).

Now let’s add to both tables a partitioned-aligned filtered index, which will use the predicate of ‘[Quantity] < 0’:

-- Script Q-16

CREATE NONCLUSTERED INDEX [FIX\_SalesByDate\_StoreID]

ON [dbo].[SalesByDate] ([StoreID])

WHERE [Quantity] < 0

CREATE NONCLUSTERED INDEX [FIX\_SalesByDateHeap\_StoreID]

ON [dbo].[SalesByDateHeap] ([StoreID], [Quantity])

WHERE [Quantity] < 0

Now let’s rerun the original two queries and see if there was an associated performance benefit. Having the filtered indexes in place, both of the abovementioned queries resulted in a total of 99 logical reads, or up to an 80% improvement in disk I/O. Another benefit to the filtered index is that, in this case, it only occupies about 22% of the space of a similar non-filtered index (i.e., without using the WHERE clause).

**Note:** The Quantity column was added to the filtered index for the heap. This is a workaround because the optimizer bypasses this filtered index on a heap when this column is excluded.

Filtered indexes are also beneficial to use on columns with a high percentage of null values, letting you exclude those rows from the index. This of course would need to align with queries filtering on the same column and also excluding nulls.

### Indexed Views

As mentioned earlier, join costs tend to be very high in a data warehouse, which is why this expense is often offset by using a de-normalized schema design. Another way to avoid this cost, especially if the underlying tables are relatively static, is to use indexed views. This approach *materializes* the result set of a view that:

* Uses SCHEMABINDING and
* Is supplemented by a unique clustered index

This technique also works well for views that aggregate data because the materialized storage overhead is typically much smaller than that for the original base tables.

**Note:** When SCHEMABINDING is specified, the base table(s) cannot be changed in a way that would alter the definition of the Indexed View.

To see an indexed view in action, let’s benchmark a query that performs an aggregation after inner-joining two tables together:

-- Script Q-17

SELECT DATEPART(yyyy,[Date]) AS [Year]

, SUM([ListPrice]) AS [TotalListPrice]

FROM [dbo].[SalesByDate] s

JOIN [dbo].[Product] p

ON s.[Item] = p.[Item]

GROUP BY DATEPART(yyyy,[Date])

According to Profiler, this query runs with our limited test data in .3 second and takes 746 logical reads. You can improve this performance by first creating an indexed view based on this same query:

-- Script Q-18

CREATE VIEW [dbo].[IV\_TotalListByYear] WITH SCHEMABINDING AS

SELECT DATEPART(yyyy,[Date]) AS [Year]

, SUM([ListPrice]) AS [TotalListPrice]

, COUNT\_BIG(\*) AS [CountBig] -- required to index the view

FROM [dbo].[SalesByDate] s

JOIN [dbo].[Product] p

ON s.[Item] = p.[Item]

GROUP BY DATEPART(yyyy,[Date])

You can then add a unique clustered index on this schema-bound view:

-- Script Q-19

CREATE UNIQUE CLUSTERED INDEX CIX\_TotalListByYear

ON [dbo].[IV\_TotalListByYear] ([Year])

Now the cost for running the original “join and aggregate” query, or alternatively the cost of simply selecting directly from the new indexed view, is reduced to only two reads and zero measured duration. Obviously, that’s a big improvement, but keep in mind there is a cost to creating and maintaining indexed views. The more static your data (which may be the case between during the day for some data warehouses), the more practical it is to use indexed views. On the other hand, constant updates to the underlying base tables may preclude the use of this method for improving query performance.

**Note:** If you are not using Enterprise, Data Center or Developer editions of SQL Server, there are two caveats to consider when using indexed views:

1. The query hint NOEXPAND must be included to ensure that the indexed view is not expanded to the underlying tables by the query optimizer.
2. The indexed view must be directly referenced *by name* in the query. If not, the optimizer will not consider using the indexed view as demonstrated in the example above.

### Routine Aggregation Using Summary Tables in the Data Warehouse

A related strategy to using indexed views is the creation of summary tables to feed reports or Key Performance Indicators (KPIs) that are routinely displayed, perhaps on dashboards or organizational Web sites. This process would typically be included as a late step of a daily ETL process and/or updated on a scheduled basis.

Of course, the use of summary tables will not provide the same currency as live data, but in a data warehouse environment, you typically have some built-in scheduled latency due to the timing of various data feeds and ETL packages. To provide clarity on the timeliness of downstream reporting, you can add a refreshed date/time column to a summary table, which could then be used to qualify your reports, KPIs, and other data visualizations.

### Conclusion

De-normalization, table partitioning, and a variety of indexing tactics can all help significantly improve query performance in a data warehouse. Table partitioning can also streamline bulk maintenance tasks such as switching out old partitions via fast metadata-only operations. However, because these techniques may require additional overhead, either for the database developer (object creation and maintenance) or for SQL Server itself, take care not to overemploy these strategies.

One way to determine the appropriate use of performance tuning techniques is to monitor your SQL Server environment for spikes in server resource utilization and correlate this information with reoccurring query patterns. We’ll cover this topic in the next section.

## Monitoring

### Overview

Paying close attention to actual execution plans during the development process is not a guarantee that query plans will stay predictable and efficient for deployed code over time, especially if a variety of parameters are available to your users. In addition, a data warehouse is characteristically a work in progress, which usually spawns a mixture of new, deployed, and ad hoc queries. This can lead to uncertainty in run times, heavy resource utilization, and general application unpredictability. Spikes to database I/O, CPU utilization, and run times are all possible based on a variety of these activities within your data warehouse environment.

Even with a more static T-SQL code base, there are changes within databases, connections, and the server that may alter existing query plans. These include:

* Data modifications and the impact on data distribution
* New, revised, or updated indexes and statistics
* Schema changes
* SQL Server configuration changes and SET options

User behavior can also affect query performance over time. A common example is varying parameters used by dynamic queries, stored procedures, and reports. User options can translate into alternate execution paths within your T-SQL code, which may result in extended run times. This situation leads to the need to monitor the queries running in your data warehouse environment so that you can identify, profile, and manage poorly performing code.

SQL Server’s MAXDOP (Maximum Degree of Parallelism) setting can have a significant effect on query performance, especially when working with partitioned tables. For a helpful discussion on this topic see: <http://blogs.msdn.com/b/sqlcat/archive/2005/11/30/498415.aspx>

### Key Query Performance Metrics

Regardless of the method you use for monitoring data warehouse queries, there are three important measures to evaluate when determining query performance:

* Query Duration
* Database I/O (typically reads for a data warehouse)
* CPU Time

Extended run times for a query are usually the first indicator of a performance problem with ad hoc or deployed code. Just as rush hour is not the best indication of how fast your car is able to go, query duration, while important, is not definitive in the sense that it may simply reflect excessive server activity. Conversely, database reads and CPU time are measures that are directly attributed to a SQL database query and not influenced by other processes running concurrently in your data warehouse.

### Options for Monitoring T-SQL Code

SQL Server provides several built-in options for monitoring database performance that you can configure to provide the key measures discussed above:

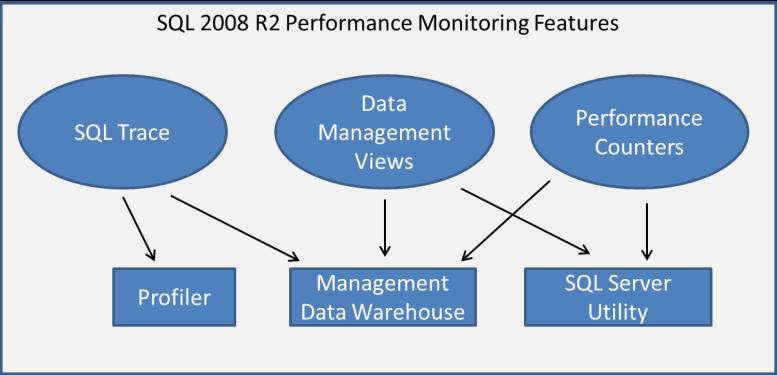
* SQL Server Profiler
* sp\_trace\_create and related system stored procedures
* Management Data Warehouse
* SQL Server Utility
* Activity Monitor
* SQL Server Management Studio Reports

**Note:** for a discussion comparing client-side with server-side traces, see: <http://sqlserverpedia.com/wiki/The_Server-side_Trace:_What,_Why,_and_How>

**SQL Server Profiler**

Profiler is an interactive, front-end application that uses SQL Server Management Objects (SMO) and system stored procedures such as sp\_trace\_create to provide near real-time query metrics. The data captured is based on the occurrence of specified events such as SQL Batch, SQL Statements, RPC completed, and others depending on your configuration.

Profiler is most useful to quickly investigate a performance issue or to filter and spot-check certain queries. However, because Profiler is not the most convenient option for daily or routine monitoring, SQL Server provides other database monitoring options, as Figure 5-12 illustrates.



**Figure 5-12:** SQL Server 2008 R2 Performance monitoring Features

**Sp\_trace\_create**

The sp\_trace\_create procedure is actually one of a series of system stored procedures that can be used for automating the capture of query performance data. You can specify which events and columns to collect (sp\_trace\_setevent), set a variety of filters to eliminate unneeded data (sp\_trace\_setfilter), and set limits on the size and/or length of time for capturing the trace.

A monitoring solution based sp\_trace\_create involves a fair amount of custom development for data capture, processing, and reporting. For more information about these system stored procedures and how to use them to build a custom monitoring system, see [SQL Server Profiler Stored Procedures (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms187346.aspx).

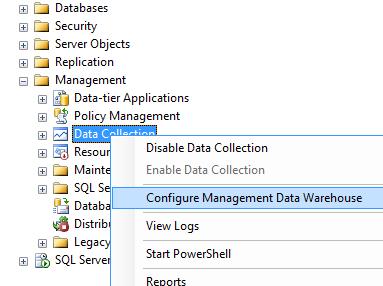
**Management Data Warehouse and SQL Server Utility**

Starting with SQL Server 2008, two new monitoring features and components are available for performance management: Data Collections and the Management Data Warehouse (MDW). SQL Server 2008 R2 further builds upon these features with the new SQL Server Utility.

In the next section, we explore using Data Collections and the MDW for setting up a SQL Server monitoring solution.

### Setting Up the Management Data Warehouse and Data Collection Components

The MDW is designed to hold performance-related statistics stored in a SQL Server 2008 database. SQL Server 2008 has a Configure Management Data Warehouse wizard that helps you get started with this newer SQL Server monitoring extension. Figure 5-13 shows how to launch the wizard.



**Figure 5-13:** Launching the Configure Management Data Warehouse Wizard

The wizard steps you through creating the MDW database and configuring the standard System Data Collections Sets (Disk Usage, Query Activity, and Server Activity). You should start the MDW configuration process on the SQL Server instance that you want to use as the Management Data Warehouse repository. This instance would ideally be separate from the SQL Server instance(s) that are being monitored (i.e., *collecting* the data).

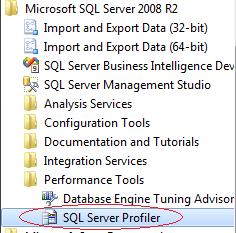
For step-by-step assistance on using the wizard, see the article “[SQL Server 2008 Data Collections and the Management Data Warehouse](http://www.databasejournal.com/features/mssql/article.php/3771871/SQL-Server-2008-Data-Collections-and-the-Management-Data-Warehouse.htm).”

**Note:** Any name can be provided for the resulting Management Data Warehouse database. As a matter of practice, MDW is often used, and is assumed in the following examples. Also be aware that once you configure and enable the MDW, there is no way to remove the corresponding SQL Server Agent jobs that are created. You can disable data collection, which will also disable the SQL Agent job, but the jobs will continue to reside in SQL Agent’s list of jobs.

Once you have configured the MDW for your environment, you will want to manually add the “SQL Trace Collection Set”, which is not automatically set up by the wizard. The SQL Trace Collection Set provides key metrics such as Query Duration, Database Reads, and CPU Time that will then be populated within the MDW database.

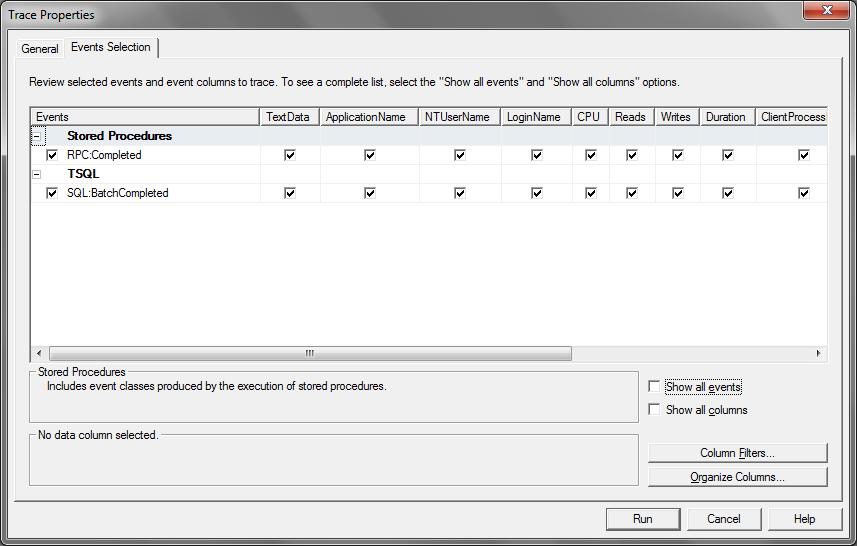
**Note:** the SQL Trace Collection Set can run independent of the other *collection sets*; however the ‘Management Data Warehouse’ must first be configured as referenced above.

Albeit a slight detour, the best way to set up the SQL Trace Collection Set for the MDW is to start Profiler, as Figure 5-14 shows, and create an active trace.



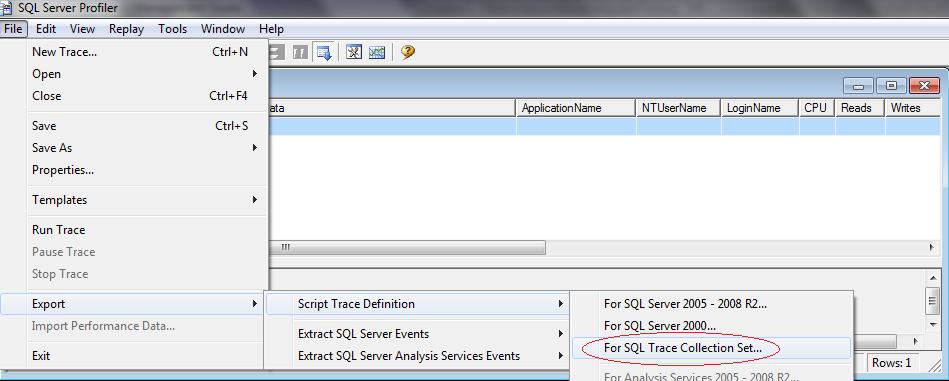
**Figure 5-14:** Starting SQL Server Profiler

Once in Profiler, click File, New Trace… . After connecting to SQL Server, you can set the desired columns and events to capture. The standard (default) template automatically sets most of the events and columns of interest for query monitoring. To minimize server overhead for data collection, you will likely want to remove all events with the exception of RPC:Completed and SQL:BatchCompleted. You can click the Events Selection tab to specifically choose the items to collect, as Figure 5-15 shows.



**Figure 5-15:** Setting the events and columns to collect in Profiler

After clicking Run, you can immediately stop the trace because you only need to extract the trace definition. You can export the trace definition script for a SQL Trace Collection Set, as Figure 5-16 shows.



**Figure 5-16:** Using Profiler to script a SQL Trace Collection Set

The resulting script can then be loaded into SSMS and executed. However, before executing the script, you will want to provide a friendlier name for the collection set, such as SQL Trace, which is shown in the code snippet below:

...

-- \*\*\* with the name you want to use for the collection set.

-- \*\*\*

DECLARE @collection\_set\_id int;

EXEC [dbo].[sp\_syscollector\_create\_collection\_set]

@name = N'SQL Trace',

@schedule\_name = N'CollectorSchedule\_Every\_15min',

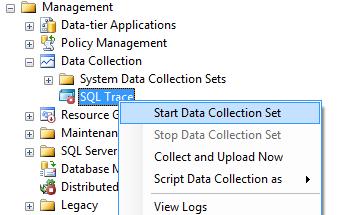
@collection\_mode = 0, -- cached mode needed for Trace collections

@logging\_level = 0, -- minimum logging

@days\_until\_expiration = 5,   
 ...

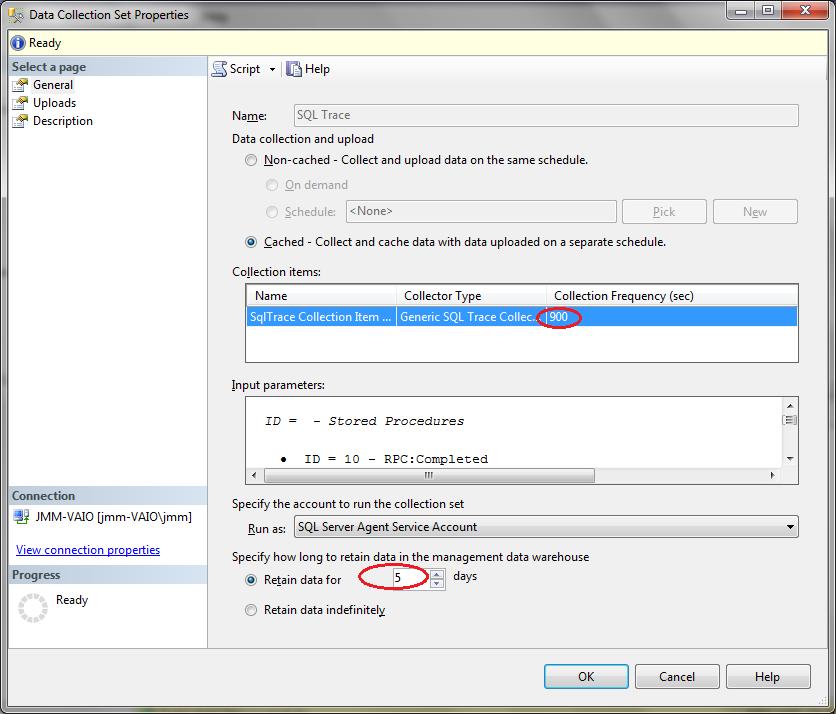
**Note:** The parameter option @days\_until\_expiration = 5 determines how many days the trace data will be stored in the MDW database before it is automatically deleted. This value can be increased for longer data retention, but remember that this also increases the monitoring overhead of the MDW database.

After running this script, you will see a new Data Collection set called SQL Trace (you may need to right-click and refresh the Data Collection container). To start the collection set, right-click the SQL Trace collection and then click Start Data Collection Set, as Figure 5-17 shows.



**Figure 5-17:** Starting the SQL Trace Collection Set

You can double-click the SQL Trace Data Collection to view its configured properties, which Figure 5-18 shows.



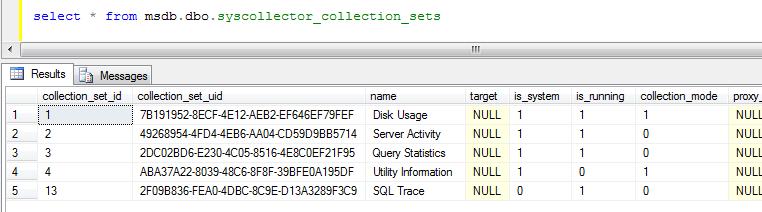
**Figure 5-18:** SQL Trace Collection Set properties

Be aware of the configurable collection frequency, which in this case is every 900 seconds (15 min), as well as how long the data is retained in the MDW (the default script is set to 5 days). Because the resulting trace table ([snapshots].[trace\_data]) can be very large, especially in high volume environments, you may prefer to reduce the number of days for data retention. This, however, lessens the ability to trend reoccurring query patterns.

To retain the ability to trend, you could consider an alternative approach of creating a simple ETL routine to export (and perhaps summarize) only the trace rows of interest. This is especially beneficial when trending stored procedures or query patterns over a long period of time.

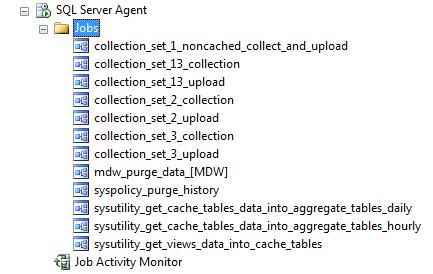
**Note:** You can also reduce the size of the resulting trace table by adding filters to the original trace you created in Profiler to create the collection set.

When you start data collection sets, corresponding SQL Agents jobs are created to schedule and manage the data collection in the MDW. The query in Figure 5-19 will return details of collection sets, including the collection\_set\_id, which can be used to correlate with the actual SQL Agent jobs.



**Figure 5-19:** Querying Data Collection Sets

Figure 5-20 shows a comprehensive list of the SQL Agent jobs that are created to collect, upload, aggregate, and purge data stored in the MDW database.

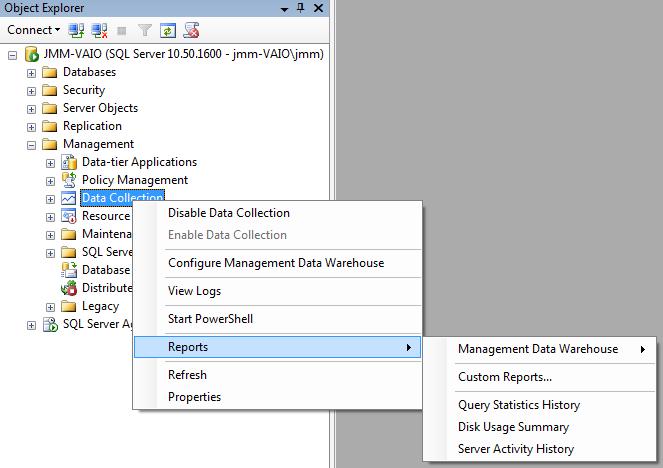


**Figure 5-20:** SQL Agent jobs used by the MDW

This completes the setup of the MDW and standard data collectors. Next, we’ll look at the standard reports available in the MDW and how to leverage the SQL Trace data collections through queries and customized reports.

### Standard Management Data Warehouse Reporting

There are three standard reports that are available out of the box for the MDW. These can be accessed by right-clicking the Data Collection container in SSMS, as in Figure 5-21.

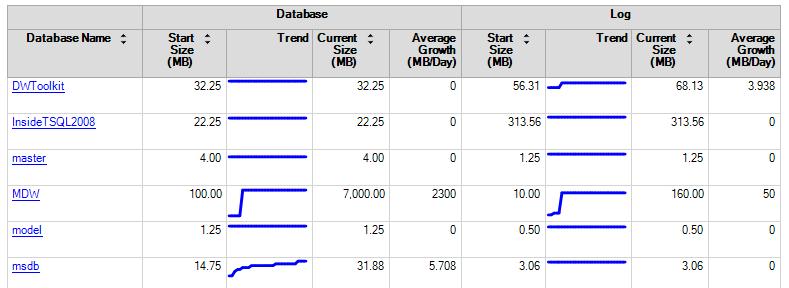


**Figure 5-21:** Viewing Management Data Warehouse reports

These reports are designed with a clean dashboard presentation style and include many interactive features for sort orders, time frame selections, and other report links. These links provide drill-down as well as drill-through capabilities to reports, some of which are normally accessed via other containers in SSMS. Let’s look at an overview of the details and analysis provided by each report.

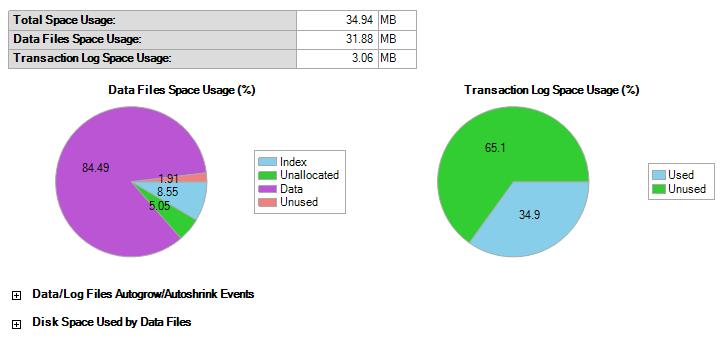
**Disk Usage Summary**

The Disk Usage Summary report, shown in Figure 5-22, lists each database contained in the SQL Server instance and provides a spark line to indicate the database growth trend. The Start Size is determined by the time of the first snapshot (i.e., when the MDW Disk Usage Collection Set was configured to start collecting data).



**Figure 5-22:** Disk Usage Summary Report

The link beside each database name allows for drill-through to a more familiar SSMS report, shown in Figure 5-23, which provides a breakdown of how the database space is being used.



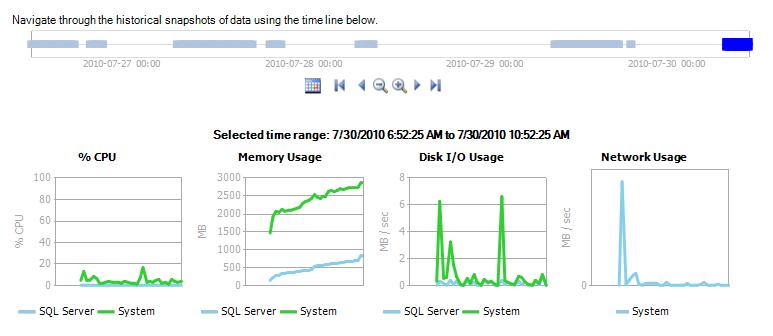
**Figure 5-23:** Database Disk Usage Report

These reports provide important insight into database disk consumption and database growth trends. In the context of a data warehouse, this information may be helpful for:

* Identifying larger-than-anticipated growth rates
* Detecting problems with database maintenance plans
* Justifying architectural changes for data archiving and the creation of summary tables
* Factoring disk usage into long-term disk storage capacity planning

**Server Activity History**

The Server Activity History report provides an overview of SQL Server activity and resource utilization and can indicate resource contention. Figure 5-24 shows a partial view of the report.



**Figure 5-24:** Partial view of Server Activity History Report

This is a highly interactive report that lets you alter time frames and drill into many report components. You can see historical resource trends in different time ranges, and you can go back to specific points in time to correlate resource consumption to specific events. This allows you to both analyze trends and troubleshoot specific problems that have been reported to you after a certain amount of time passed.

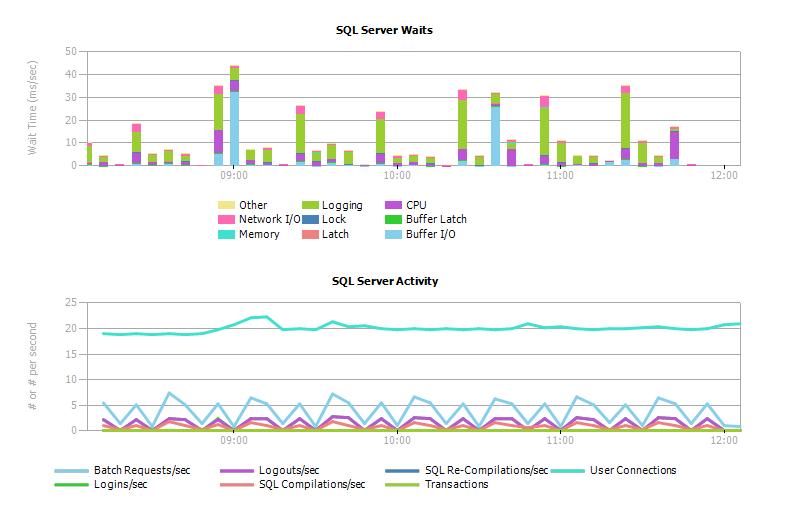
A considerable amount of data is gathered from the underlying collection set to make this report possible. The sources include the following dynamic management views:

* sys.dm\_os\_wait\_stats
* sys.dm\_os\_latch\_stats
* sys.dm\_os\_schedulers
* sys.dm\_exec\_sessions
* sys.dm\_exec\_requests
* sys.dm\_os\_waiting\_tasks
* sys.dm\_os\_process\_memory
* sys.dm\_os\_memory\_nodes

The report also pulls from more than 75 SQL Server performance counters in the following categories:

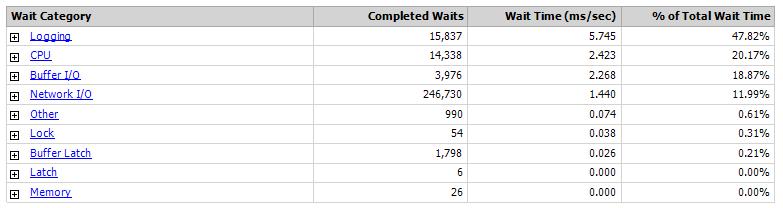
* CPU
* Disk I/O
* Memory
* Network

In addition, the Server Activity History report includes interactive SQL Server-specific charts, which let you track SQL wait states along with key counters for trending overall SQL activity, as Figure 5-25 shows.



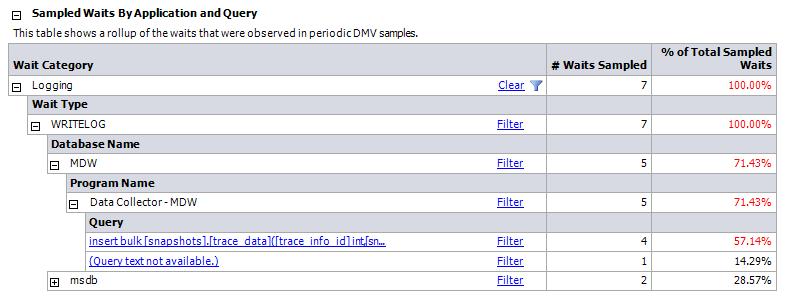
**Figure 5-25:** Server Activity History Report (continued)

When viewing the above report, you can click a vertical bar in the SQL Server Waits chart to see the Wait Category table that Figure 5-26 shows.



**Figure 5-26:** Wait Category Table

From there, you can click a link under the Wait Category column to go to several other reports related to the category selected. For example, the Logging link renders a context-sensitive Wait Category report with a drill-down into the executing queries, as Figure 5-27 shows.

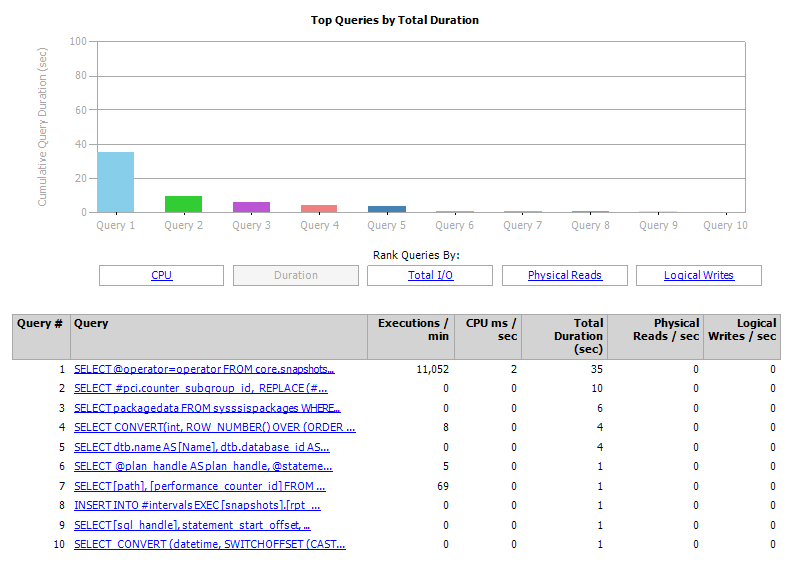


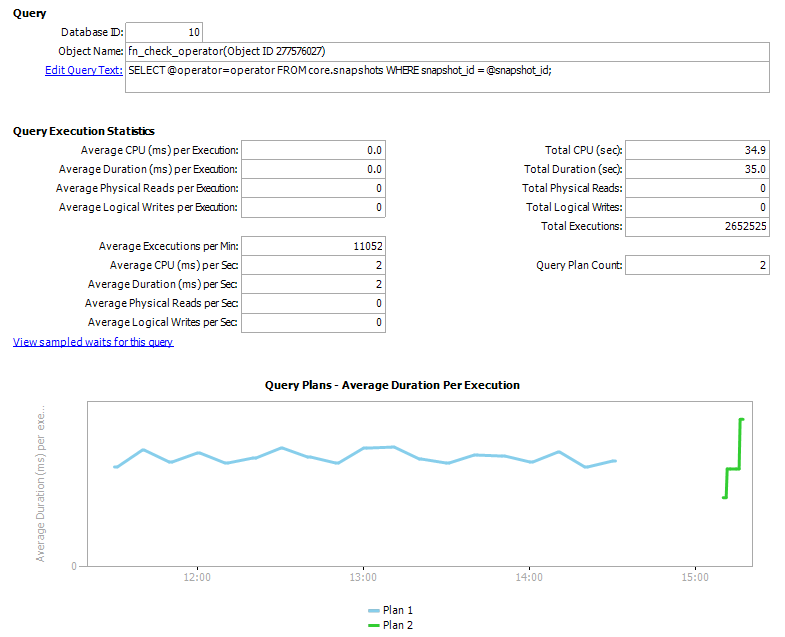
**Figure 5-27:** Sampled Waits by Application and Query

The CPU link branches over to the next section that we cover, which details query statistics.

**Query Statistics History**

The Query Statistics History report pulls its source data primarily from the sys.dm\_exec\_requests, sys.dm\_exec\_sessions, and sys.dm\_exec\_query\_stats dynamic management views. The report provides a top 10 listing based on resource utilization of CPU, Duration, Total I/O, Physical Reads, and Logical Writes by second. The queries are ranked and linked to a detail report, as Figure 5-28 shows.





**Figure 5-28:** Query Statistics History with drill-through

### Querying the SQL Trace Collection in the Management Data Warehouse

The SQL Trace Collection Set provides additional source data that can be used to analyze the resource utilization and performance of SQL queries. Unlike the built-in reports we just saw, which are based on snapshots of performance counters and data management views, a SQL Trace is configurable to capture key metrics at the time a query actually completes. This provides start-to-finish measurements of individual queries and can lead to very specific analysis (e.g., pattern matching) of the collected performance data.

Based on the earlier SQL Trace configuration, we are collecting the following measures: CPU Time, Disk I/O, and Query Duration. These were also set to be recorded based on the completion of the following events:

* RPC:Completed
* SQL:BatchCompleted

Based on this configuration, completed query metrics are placed in the MDW table called [snapshots].[trace\_data]. Using this table, you can view the top 10 queries for any of our key metrics. For example, you can retrieve the top 10 queries based on duration by using the following code:

-- Script M-1

SELECT TOP 10 \*

FROM [snapshots].[trace\_data]

ORDER BY [Duration] DESC -- or BY [Reads], or [CPU]

This result of this query should indicate the 10 longest-running queries that completed based on data collected from all the SQL Server instances monitored by this MDW (scroll to the right to see the Server Name column).

You can obtain more information about the snapshots used to gather this trace information by joining to the [core].[snapshots] view on the snapshot\_id column:

-- Script M-2

SELECT TOP 10 s.\*, t.\*

FROM [snapshots].[trace\_data] t

JOIN [core].[snapshots] s

ON s.[snapshot\_id] = t.[snapshot\_id]

ORDER BY [Duration] DESC

You may find it helpful to group by a substring of the TextData column. This may better illustrate recurring queries that perhaps vary within the WHERE clause but are based on the same table joins and columns. An example of this query, using just the first 80 characters of the Text Data column and based on disk reads, is displayed below:

-- Script M-3

SELECT TOP 10

COUNT(\*) AS [RunCount]

, SUM([Reads]) AS [SumReads]

, LEFT([TextData],80) AS [TextShort]

FROM [snapshots].[trace\_data]

GROUP BY LEFT([TextData],80)

ORDER BY [SumReads] DESC

**Note:** The technique of grouping by the first 80 characters is used here in an attempt to find recurring code. You may need to alter this method to better suit the SQL code patterns in your database environment. The objective is to find recurring code that carries a high disk I/O (read) cost and then to be able to identify the origin/source of the query for subsequent tuning efforts.

Another option is to trend SUM and AVG query metrics over time. The following query again groups by the first 80 characters of the TextData column to spot a recurring query pattern and then uses a secondary GROUP BY on date to show a performance trend. A LIKE predicate is also placed on the TextData column to narrow the queries to those referencing the Sales table:

-- Script M-4

SELECT CONVERT(VARCHAR(10), [EndTime], 120) AS [DisplayDate]

, LEFT([TextData],80) AS [TextShort]

, COUNT(\*) AS [TimesRun]

, SUM(Duration) AS [SumDuration]

, SUM(Reads) AS [SumReads]

, SUM(CPU) AS [SumCPU]

, AVG(Duration) AS [AvgDuration]

, AVG(Reads) AS [AvgReads]

, AVG(CPU) AS [AvgCPU]

FROM [snapshots].[trace\_data]

WHERE [TextData] LIKE '%Sales%'

GROUP BY LEFT([TextData],80), CONVERT(VARCHAR(10), [EndTime], 120)

ORDER BY [TextShort], [DisplayDate]

As noted earlier, due to the typical size of trace tables, it is recommended that the most notable or problematic queries (rows) be moved to a history table. This allows for a broader period of time for trending, while keeping the base trace\_data table in the MDW to a minimum size. You may also choose to use an aggregation technique, similar to that in the query above, to further lessen the storage and processing overhead for your historical table(s).

The examples in this section really just scratch the surface of the types of queries that can be used with the MDW trace\_data table. For routine use of this information, you may want to create parameter-driven SQL Server Reporting Services (SSRS) reports based on similar queries or maybe use the trace\_data table (or a derivation thereof) as a fact table within a PowerPivot solution.

## Performance Tuning

### Overview

After ensuring accuracy and reliability, performance is the most critical objective for queries running in a data warehouse. Due to very large data volumes and unpredictable query patterns, it is common for performance complaints to surface in the data warehouse server environment.

Because we like easy solutions, a common response to performance issues involves acquiring a larger server. While this usually does provide faster response times, it’s not the most economical or greenest solution. It also tends to mask core issues with the database schema, indexing strategy, reusable database objects, and/or query design—all problems that tend to resurface even after procuring bigger hardware. Additionally, the expense and migration headache associated with server and SAN upgrades can be considerable, especially when code and schema optimization can often bring dramatic improvements in query performance.

The concepts and techniques presented in this section are geared more toward query optimization, rather than overall server optimization. The emphasis is on identifying recurring problematic SQL code and applying a variety of query tuning techniques. See Chapter 4 for recommendations about overall server- optimization practices.

### Targeted Performance Tuning

A fairly high percentage of SQL code submitted for execution to a database server could be optimized, at least to some degree. This doesn’t mean, however, that it makes sense to performance-tune all code that is developed for the database. Shaving a mere 4 milliseconds off a query that runs once a day is an obvious waste of time. However, taking 30 seconds off a stored procedure that runs 200 times a day is significant, both in terms of server capacity management and user wait times.

The best way to ensure that you have targeted the most problematic code is to base your efforts on the analysis from your monitoring applications, as discussed in the last section. The objective is to identify SQL code that is repeatedly run, either as a stored procedure or a batch script, and that consumes the most server resources. This approach should provide the highest return on investment (ROI) for your tuning efforts.

The three most common measures to examine for this exercise are:

* Disk I/O
* CPU Time
* Run Times (Duration)

The Disk I/O and CPU Time are metrics that can be directly attributed to a single query. Duration, on the other hand, may indicate a problem with a given query or perhaps a more complex problem involving resource contention on the database server. In a data warehouse environment, it’s not unusual for there to be tremendous pressure on Disk I/O simply due to the volume of data needed for ad hoc and deployed queries. A good practice is to use one of the monitoring techniques discussed in the last section and look for recurring queries with a high number of database reads (or overall disk I/O).

The CPU Time measure in a data warehouse often parallels the Disk I/O measure in that a reduction in reads will typically have a corresponding drop in CPU. However, certain techniques have an inverse relationship involving these two measures. An example of this is table compression, where you would most likely have a positive effect on Disk I/O with more data per page but a negative effect on CPU due to handling the compression routine. Another sometimes counterintuitive scenario is when you upgrade to faster disk storage and then find your system has an increase in CPU usage. This of course is due to faster I/O, and you should still see a net gain unless your CPUs were already under great pressure.

In the situation where a trade-off exists, you should consider overall server performance counters (available in MDW reporting,) and reduce demand for the resource that is usually under the greatest pressure.

Query run times should of course be monitored and targeted for performance tuning, but with an eye toward concurrently running queries. For example, a large UPDATE statement causing an extended run time for one query may reflect poorly on the duration of another query that tries to concurrently SELECT from the same table. Diagnosing the second query outside of the context of the first would likely hide the real problem of locking and blocking. In addition, heavy activity due to one or more very large queries (deployed or ad hoc) can make all queries appear to run poorly.

**Note:** One way to spot this type of locking and blocking is to query the [snapshots].[trace\_data] table, looking for a large number of queries that have a similar EndTime—perhaps within 500 milliseconds of each other. A close-to-common end time often indicates that a large blocking query finally completed and the corresponding blocked queries were then able to quickly complete.

### Query Tuning Based on Disk I/O

To demonstrate examples of performance tuning, let’s take advantage of the MDW, which if configured as illustrated in the previous section, should contain a substantial amount of performance-related data for the instances being tracked. Our first target will be SQL code with a large number of (logical) reads as recorded by the SQL Trace data collector. To determine this, we will group the SQL text by the first 80 characters as a simple (albeit imperfect) attempt to spot recurring dynamic SQL code:

-- Script PT-1

SELECT TOP 10 LEFT([TextData],80) AS [TextShort]

, COUNT(\*) AS [TimesRun]

, AVG([Reads]) AS [AvgReads]

, SUM([Reads]) AS [SumReads]

, MIN([Reads]) AS [MinReads]

, MAX([Reads]) AS [MaxReads]

FROM [snapshots].[trace\_data]

GROUP BY LEFT([TextData],80)

HAVING COUNT(\*) > 4

ORDER BY [SumReads] DESC

**Note:** You may want to vary the number of characters used with the LEFT character function to better match your most common T-SQL query patterns.

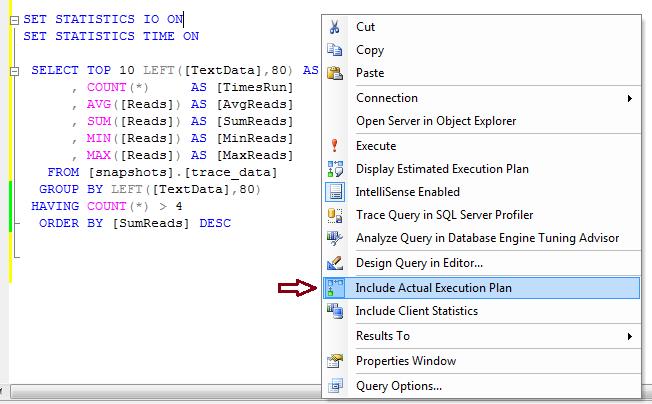
The above code should help identify the most expensive queries based on database reads. If you need to see the full TextData column for the abbreviated T-SQL code (i.e., text beyond 80 characters), you can re-query the trace\_data table with a predicate based on the returned MinReads and/or MaxReads values. This method is required because there is no unique row identifier in the [snapshots].[trace\_data] table.

To continue with tuning based on disk I/O, the query above can double as a query that itself needs to be tuned, since it takes a fairly long time to run. As in earlier examples, you want a get a benchmark of the query targeted for tuning so that you can track your optimization progress. By specifying the following SET options, you will be able to see the needed query feedback metrics under the Messages tab in SSMS:

SET STATISTICS IO ON

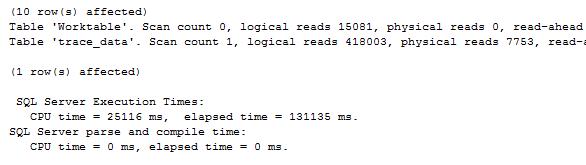
SET STATISTICS TIME ON

Also make sure you right-click in the design surface and choose Include Actual Execution Plan before rerunning the query, as Figure 5-29 shows, so that you see a graphical view of the query plan.



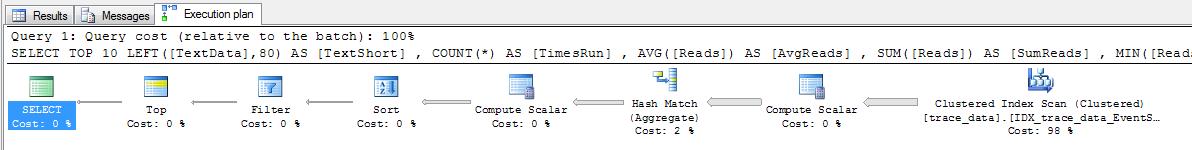
**Figure 5-29:** Including Actual Execution Plan in SSMS

After rerunning the query above, you can refer to the statistical feedback in the SSMS Messages tab (shown in Figure 5-30) to see that the query used a total of about 440,000 reads, with a CPU time of roughly 25,000 ms (milliseconds).



**Figure 5-30:** Results Pane - Query Messages

By clicking the Execution plan tab in SSMS, you can also see a graphical representation of the query plan, which Figure 5-31 shows.

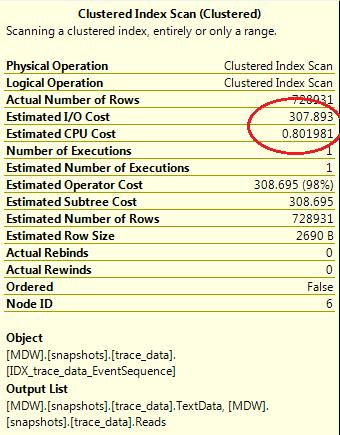


**Figure 5-31:** Graphical execution plan in SSMS

**Note:** Later in this section, we’ll look at other options for obtaining details about execution plans. SSMS also lets you view an estimated execution plan (using the same drop-down menu as displayed above), which provides a very similar plan without the need run the actual query.

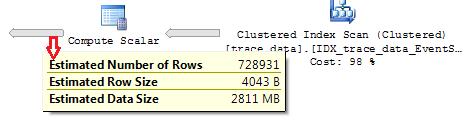
By scanning the above execution plan from right to left, you can see a steady decrease in arrow width between operations. This indicates a gradually reduced number of rows handled from step to step. The right-most operation, a Clustered Index Scan, also happens to carry the largest operational cost, which is 98% of the overall query.

You can hover over the icons in the execution plan to obtain more details about the operation. For example, hovering over the Clustered Index Scan icon displays the metrics for the operation, as Figure 5-32 shows.



**Figure 5-32:** Details for Clustered Index Scan operation

In addition, hovering over an arrow in the execution plan provides the number of rows going into or out of a given step, as Figure 5-33 shows. Note how the arrows grow wider as the rows involved in the specific operation increase.



**Figure 5-33:** Number of rows between operations

The best place to start when considering strategies for optimization is to focus on operations with a high percentage cost as well as wide arrows between query steps. The greatest expense by operation in this case, as with many data warehouse queries, is in scanning a large table or index. With this particular query, most of the cost occurs with scanning the clustered index (IDX\_trace\_data \_EventSequence) for the specified query columns. Because this operation carries virtually all the cost of this query (98%), you would need to address this trace\_data clustered scan operation to have any real effect on the performance of this query.

**Optimization Strategy**

Following are a few options to consider for optimizing a large clustered index scan:

* Modify an existing index
* Create a new index
* Create an indexed view
* Create a summary table and schedule periodic updates

However before pursuing a specific option, make sure you consider the objectives of the query itself. In some cases, clarifying the purpose of a targeted query may provide additional tuning opportunities. For example, in this case, you are looking for queries in need of performance tuning from an I/O (read) perspective. However, a casual browsing of the trace\_data table shows that most rows have a very small value in the Reads column. These rows have no meaningful consequence to the original intent of this query, which is to find recurring queries with a very high amount I/O.

This insight and clarification of the query objective suggests two potential refinements to the original optimization strategies:

1. A new filtered index where the Reads column is greater than a certain value
2. A new or modified index with the Reads column as the first column

Based on the data distribution of the Reads column, a filtered index is probably the best choice, especially if you set the filter threshold fairly high. This will ensure that the index will always have far fewer rows than a comparable non-filtered index. Another decision to make before creating the filtered index is to determine which columns to house within the index, either as a part of a composite index that begins with the Reads column or simply as an included column for the index.

Another key consideration for the filtered index is the Reads threshold to use in order to be included in the index. The lower the threshold, the more rows from the base trace\_data table will be placed in the index. A lower threshold also means more pages required for facilitating this index as well as greater overall maintenance costs. Fortunately for our purpose, we are looking for queries with an exceptionally large number of reads, which argues for a higher threshold. This will also translate to fewer index rows, providing higher selectivity when using the index and lower maintenance costs. For this exercise, let’s restrict the new filtered index to include only those rows with an excess of 50,000 reads for the queries being traced.

For the query in question, the only two columns you need to make this a covering index are Reads and TextData. Because TextData has a data type of NVARCHAR(MAX), it cannot be used as part of a composite index. It can, however, be placed after the INCLUDE clause, providing a covering dynamic with queries referencing the TextData column:

-- Script PT-2

CREATE NONCLUSTERED INDEX [FIDX\_trace\_data\_Reads\_incl\_TextData]

ON [snapshots].[trace\_data]

( [Reads] ) INCLUDE ( [TextData] )

WHERE ([Reads] > 50000)

This index can now be automatically leveraged for any queries on the trace\_data table with a predicate based on Reads greater than 50,000. Once you add this predicate to the original query, it’s almost guaranteed to reduce the number of returned rows.

**Note:** With the exception of MaxReads, the aggregate values will probably differ because the query will likely be based on a much smaller dataset. Despite this, we are still gaining the most actionable data from the query and staying true to our intent of finding queries that have a very high number of database reads and, therefore, a high impact on disk I/O.

Now let’s rerun the query, being sure to align the WHERE clause with our new filtered index of Reads > 50000:

-- Script PT-3

SELECT TOP 10 [TextDataShort]

, COUNT(\*) AS [TimesRun]

, AVG([Reads]) AS [AvgReads]

, SUM([Reads]) AS [SumReads]

, MIN([Reads]) AS [MinReads]

, MAX([Reads]) AS [MaxReads]

FROM [snapshots].[trace\_data]

WHERE ([Reads] > 50000) -- 🡨 add to align with filtered index

GROUP BY [TextDataShort]

HAVING COUNT(\*) > 4

ORDER BY [SumReads] DESC

This revised query now requires 85 reads (compared to 440,000) and takes only 31 ms of CPU time (compared to 25,000 ms). This represents a major performance improvement, with the only additional overhead being the maintenance of the new filtered index, which should hold a small fraction of the total rows in the trace\_data table.

To see how many rows are included in the new filtered index, you can use the following query:

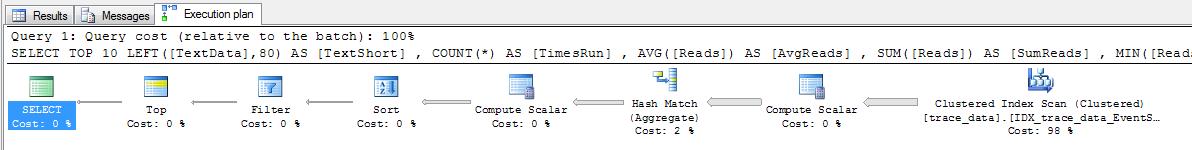
-- Script PT-4

SELECT COUNT(\*) FROM [snapshots].[trace\_data] WHERE ([Reads] > 50000)

Ideally, this number would be less than a 1,000 because you are looking for worst-case queries. You will want to adjust the filtered index threshold placed on Reads up or down from 50,000 to best fit the size of your large data warehouse queries. For large databases, you may find 500,000 is a better number to use in the filtered index. In any case, the objective is to keep enough rows to obtain actionable information while adding the least amount of database overhead.

### Working with Execution Plans

While you are optimizing queries, you may find it beneficial to store query plans that can be used for later reference. For example, it may be helpful to compare execution plans before and after optimizing a query. This may help you become more familiar with the operations SQL Server uses and how they can affect performance. For problematic SQL code, it’s also a handy to have a well-performing query plan stored. Then in the event that the same query starts to run poorly, you can see what has changed from a query plan perspective. Figure 5-34 shows a sample graphical query plan, as we’ve seen earlier in this chapter.

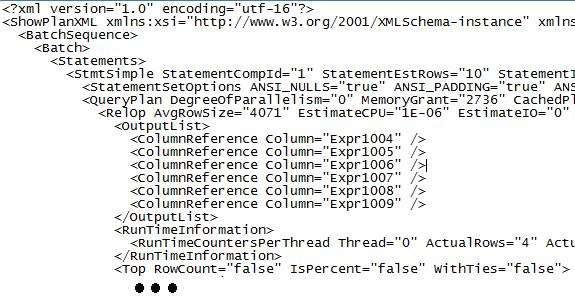


**Figure 5-34:** Graphical query plan

Execution plans are also available in XML and other text modes. You can use the SET option in SSMS to obtain text-based versions of execution plans, as follows:

* SET SHOWPLAN\_XML ON
* SET SHOWPLAN\_TEXT ON
* SET SHOWPLAN\_ALL ON

Figure 5-35 shows a limited screen shot of an XML-based query plan.



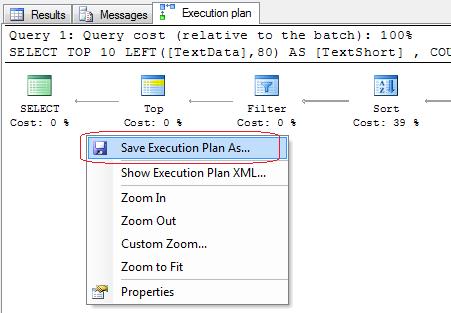
**Figure 5-35:** Text-based (XML) query plan

For more information about these and other Showplan SET options, see [Displaying Execution Plans by Using the Showplan SET Options (Transact-SQL)](http://msdn.microsoft.com/en-us/library/ms180765.aspx).

SSMS also has a nice feature to let you save a graphical execution plan in the XML file format. You are able to do this while viewing a plan graphically under the estimated or actual *Execution plan* tab. The saved XML file can later be reloaded into SSMS and displayed graphically as it was originally rendered.

**Note:** You can also collect and store SQL Plan data using Profiler. For more information see: <http://blogs.techrepublic.com.com/datacenter/?p=269>

To store an execution plan in XML format, simply right-click anywhere on the graphical display pane and provide a name and location for the .sqlplan file, as Figure 5-36 shows.



**Figure 5-36:** Saving an XML-formatted execution plan

Because these file reside in the file system, you have the option to query or transform these plans via a variety of XML script techniques, including XPath, XQuery, and XSLT.

For more details about XML Showplans in general, see [XML Showplans](http://msdn.microsoft.com/en-us/library/ms189298.aspx).

### Join Operators

The join operator is a key element of most query plans. There are three primary join operator types that SQL Server may choose to use when joining together tables:

* Nested Loops
* Merge
* Hash Match

Using complex mathematical algorithms based on schema design, actual data, keys, indexes, and metadata, the SQL Server optimizer will normally choose the best join operator to use in a query plan. However, the optimizer is not perfect in making this determination, and in the event of a performance problem, you may want to identify the join operator chosen for a given query step and know how to alter if necessary.

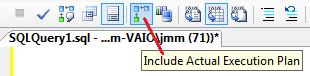
**Nested Loops Join Operator**

To get started with our look at the Nested Loops join operator, be sure your SET STATISTICS statements are on:

SET STATISTICS IO ON

SET STATISTICS TIME ON

And also make sure to include the actual execution plan for the current connection, as Figure 5-37 shows.



**Figure 5-37:** Including the query execution plan

You’re now ready to run the following query, which illustrates a join operator:

-- Script PT-5

SELECT pci.performance\_counter\_id

, pcv.collection\_time

, pci.counter\_name

, pci.instance\_name

, pcv.formatted\_value

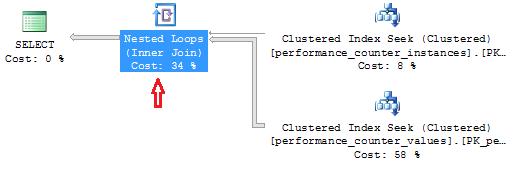
FROM snapshots.performance\_counter\_instances pci

INNER JOIN snapshots.performance\_counter\_values pcv

ON pci.performance\_counter\_id = pcv.performance\_counter\_instance\_id

WHERE pci.[performance\_counter\_id] = 2 -- Avg. Disk Queue Length

The results of this query show the values that have been captured in the MDW for the system performance counter of Avg. Disk Queue Length. We will refine this query later to be a little more helpful, but for now, let’s stay on track and examine the join operator. In the results pane, click the Execution Plan tab to see the join operator selected by the optimizer, as Figure 5-38 shows.



**Figure 5-38:** Nested Loops join operator

In this case, you can see that the optimizer chose the Nested Loops join operator. This operator is driven first by a top or outer input of qualifying rows, which in this example is just one row:

WHERE pci.[performance\_counter\_id] = 2

It is then followed by a bottom or inner scan of matching rows to satisfy the join predicate:

INNER JOIN snapshots.performance\_counter\_values pcv

ON pci.performance\_counter\_id = pcv.performance\_counter\_instance\_id

Because each qualifying row in the outer table requires an inner scan, the Nested Loops join works best with a small number of rows in the outer, or top, loop. Now let’s try another query—one that pulls back two performance counters:

-- Script PT-6

SELECT pci.performance\_counter\_id

, pcv.collection\_time

, pci.counter\_name

, pci.instance\_name

, pcv.formatted\_value

FROM snapshots.performance\_counter\_instances pci

INNER JOIN snapshots.performance\_counter\_values pcv

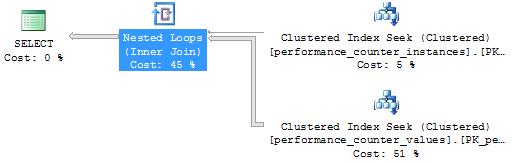
ON pci.performance\_counter\_id = pcv.performance\_counter\_instance\_id

WHERE pci.[performance\_counter\_id] IN (3,7) -- Avg. Read and Write Queue

The plan for this query should still use the Nested Loops join operator. However, your results could vary because this query appears to border on using the Merge join operator. For purposes of this exercise, you can force the Nested Loops operator (if needed) by including the following hint inside the INNER JOIN clause:

INNER LOOP JOIN snapshots.performance\_counter\_values pcv

The updated query execution plan should also show the Nested Loops join operator. However in this case, as Figure 5-39 shows, this join operator has a higher overall operation percentage cost (45%) than it did in the previous query, which was 34%. This result indicates that as the number of rows in the outer loop increases, the less efficient the Nested Loops operator may become due to the increased inner (bottom) loops that are required:



**Figure 5-39:** Nested Loops join operator (continued)

To verify the number of inner loops, you can also look on the Messages tab in SSMS to see that two scans are now required to satisfy the join operation:

Table 'performance\_counter\_values'. Scan count 2, logical reads 61, physical reads 0 …

Table 'performance\_counter\_instances'. Scan count 2, logical reads 4, physical reads 0 …

**Merge Join Operator**

To demonstrate the Merge Join operator, let’s alter the query slightly, this time looking for all performance counters with an ID less than 5:

-- Script PT-7

SELECT pci.performance\_counter\_id

, pcv.collection\_time

, pci.counter\_name

, pci.instance\_name

, pcv.formatted\_value

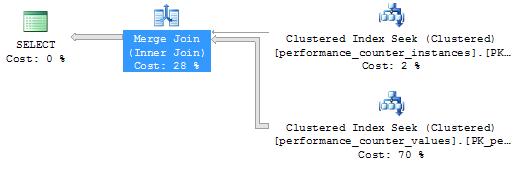
FROM snapshots.performance\_counter\_instances pci

INNER JOIN snapshots.performance\_counter\_values pcv

ON pci.performance\_counter\_id = pcv.performance\_counter\_instance\_id

WHERE pci.[performance\_counter\_id] < 5

When you examine the new execution plan, you’ll see that the optimizer has switched to the Merge Join operator, as Figure 5-40 shows.



**Figure 5-40:** Merge Join operator

If your query plan results differ from the above, you can force the Merge Join operator by including the following hint in the INNER JOIN clause:

INNER MERGE JOIN snapshots.performance\_counter\_values pcv

Unlike the Nested Loops join operator, where the inner table may need to be scanned multiple times, the Merge Join operator scans data just once. This is clearly an advantage for the Merge Join operator, but it also requires that the tables involved be sorted by the join predicate. After closer examination, you can find that these tables both have a clustered index that begins with the columns named the JOIN clause:

FROM snapshots.performance\_counter\_instances pci

JOIN snapshots.performance\_counter\_values pcv

ON pci.performance\_counter\_id = pcv.performance\_counter\_instance\_id

This makes the Merge Join a natural choice for this particular query.

**Note:** The optimizer may choose to first create a worktable that is sorted based on the join predicate in order to take advantage of the Merge Join operator.

Because the sample queries above operate very close to the optimizer’s threshold between the Nested Loops and Merge Join operators, you shouldn’t see much difference in the I/O or CPU costs by tweaking these queries with a table join hint. The important thing to understand is the characteristics of the join type being used and the technique used to override the join choice, if necessary.

**Hash Match Join Operator**

To demonstrate the Hash Match join operator, we will change our earlier query by substituting a table named [core].[snapshots\_internal]. With this code revision, the optimizer will not have a convenient clustered or secondary index to use when considering the join predicate for this query:

-- Script PT-8

SELECT pcv.collection\_time

, pcv.formatted\_value

, si.snapshot\_time\_id

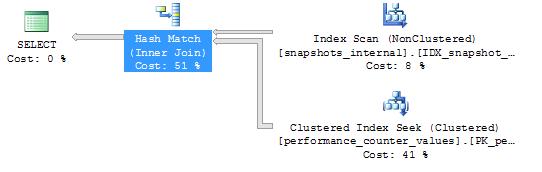
FROM [core].[snapshots\_internal] si

INNER JOIN snapshots.performance\_counter\_values pcv

ON si.[snapshot\_id] = pcv.[snapshot\_id]

WHERE pcv.performance\_counter\_instance\_id < 5

After running this query, you should see that the execution plan contains the Hash Match join operator, shown in Figure 5-41.



**Figure 5-41:** Hash Match join operator

For this join exercise, if your plan does not reflect the Hash Match operator, you can force it by including the following hint in the JOIN clause:

INNER HASH JOIN snapshots.performance\_counter\_values pcv

The Hash Match join operator is the most complex of the three join operators and works based on a hash function that is applied to each row in both the outer (top) and inner (bottom) inputs. For the very large tables in a data warehouse, the Hash Match operator may put pressure the CPU to perform the hash function and may place considerable pressure on memory depending on the cardinality of the hash output. This is because all the distinct hash values must be held in memory or paged to tempdb while the Hash Match operation completes.

The good news for the Hash Match operator is that the tables are scanned only once, providing a similar benefit to the Merge Join operator.

For a helpful article on the internals of the Hash Match join operator, see [Physical Join Operators in SQL Server - Hash Operator](http://www.sql-server-performance.com/articles/dba/physical_join_operators_hash_p1.aspx).

As noted earlier, the SQL Server optimizer normally selects the most efficient join operator. However, if you have a query that is in need of attention due to performance issues, it’s helpful to verify that the query plan is using the best join method for the operation. If the optimizer chooses a less-efficient join operator, you may decide to supplement the query with a table join hint. If you choose this approach, make sure you document where you have embedded query hints. This will allow you to more easily reexamine the performance effects of these supplemental hints with the release of new versions of SQL Server.

Now that you’re familiar with more of the tables maintained in the MDW database, you may find the next query helpful in trending one or more of the performance counters that are of interest. The following example provides a daily trend for the system performance counter of Average Disk Queue Length:

-- Script PT-9

SELECT pci.performance\_counter\_id

, CAST(pcv.collection\_time AS DATE) AS [Date]

, pci.counter\_name

, pci.instance\_name

, AVG(pcv.formatted\_value) AS [Avg Value]

FROM snapshots.performance\_counter\_instances pci

INNER JOIN snapshots.performance\_counter\_values pcv

ON pci.performance\_counter\_id = pcv.performance\_counter\_instance\_id

WHERE pci.[performance\_counter\_id] = 2 -- Avg. Disk Queue Length

GROUP BY pci.performance\_counter\_id

, CAST(pcv.collection\_time AS DATE)

, pci.counter\_name

, pci.instance\_name

ORDER BY pci.performance\_counter\_id, [Date]

You may want to enhance this query to filter out rows for non-peak times in your data warehouse. By the way, according to the execution plan for this last query, the optimizer decided on using the Nested Loops join operator.

### Isolation Level Considerations

SQL Server has these basic transaction isolation levels, listed below in order of lowest to highest isolation level:

1. Read Uncommitted
2. Read Committed (default level), and Read Committed Snapshot
3. Repeatable Read
4. Serializable

The two highest levels, Repeatable Read and Serializable, are typically used for more sensitive operational database (OLTP) queries, where data integrity is a greater concern than query performance or user concurrency. Conversely, many data warehouse applications are able to take advantage of a more liberal, or lower-level, isolation strategy. This is because:

* Data warehouses are typically just copies of underlying operational source systems, where critical transactional consistency is maintained.
* Daily updates to the data warehouse are often batched together within ETL packages that are sequenced to run before the reporting processes and queries.
* Many queries that are run in a data warehouse feed aggregate-level reports that are used more for rendering KPIs, sparklines, charts, and other trends, which are not necessarily sensitive to small anomalies in the underlying data.

The Read Committed isolation level is the default used by the SQL Server Database Engine. This level ensures that all pages being altered in the database are also committed before they can be read by another process. This isolation level can block a database read operation while waiting for a database update process to commit. This I/O constraint is a reasonable default, but at the same time, may be too pessimistic for certain data warehouse queries. SQL Server provides mechanisms to override this behavior, which we’ll discuss next.

Because Read Committed is the default isolation level, no explicit coding is required when querying directly on SQL Server. A variation of this default isolation level is Read Committed SNAPSHOT, which uses the most recent committed version of the data when there is lock contention, but it also requires greater resource consumption. For more information on row versioning based isolation levels see: <http://msdn.microsoft.com/en-us/library/ms191190.aspx>

**Note:** An upper layer within a multi-tier database application may choose to override SQL Server’s default Read Committed transaction isolation level.

The lowest isolation level of Read Uncommitted is considered highly optimistic, ignoring all locks that other queries and transactions may have temporarily placed in the database. As the name suggests, this level may return data that has not been formally committed to the database. Uncommitted data could occur during logical database operations such as a row update or a physical operation such as a page split.

Accessing these database pages is sometimes referred to as *dirty reads*. However, it is worth noting that dirty reads should not necessarily be equated to bad data since uncommitted pages have a reasonable chance of being swiftly committed to the database. Having said that, and because there are no guarantees when using Read Uncommitted, you should consider if the query in question can weather a small degree of uncertainty.

The Read Uncommitted isolation level must be explicitly coded and can be indicated for a given database connection by using a SET command:

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

Alternatively, you can embed a NOLOCK table hint within a query, as demonstrated below:

-- Script PT-10

SELECT COUNT(\*)

FROM [snapshots].[trace\_data] (NOLOCK)

WHERE ([Reads] > 50000)

As mentioned earlier, using Read Uncommitted or the NOLOCK hint in a query does carry a higher risk of returning inaccurate data than using the default level of Read Committed. This is especially true if there is a high volume of data modifications occurring while your query is running. On the other hand, if the tables in your data warehouse are mostly static during query execution or if there is an allowance for imperfections within some of your aggregate queries, Read Uncommitted may be a valid option to consider.

For more information about isolation levels, see [Isolation Levels in the Database Engine](http://msdn.microsoft.com/en-us/library/ms189122.aspx).

## Best Practices Summary

Here is a summary of this chapter’s best practices for querying, monitoring, and performance tuning in your data warehouse:

* Pay attention to both the results of a query and the execution plan designed to return those results. Make it a routine exercise to scan query plans to become more familiar with how SQL Server resolves queries.
* De-normalize your data warehouse schema where possible to reduce the need to join very large tables.
* Use table partitioning for your largest tables, and be sure when querying that the optimizer uses partition elimination whenever possible.
* Align secondary indexes with the same table partition function to reduce partitioned table maintenance.
* Consider using a heap instead of a clustered index, especially with a table that partially mimics a clustered index via its partition function.
* Design indexes to be covering for larger recurring queries whenever possible, especially when creating a filtered (i.e., partial) index.
* Use an indexed view as an alternative to a denormalized table or to materialize an aggregated dataset. The more static the underlying data, the less overhead will be required to maintain the indexed views.
* Use summary tables with automatic (scheduled) updates as an alternative to indexed views, especially when there is less significance in the currency of the data.
* For query monitoring, focus on the quantitative measures of Disk I/O, CPU Time, and Query Duration.
* Use the features of the Management Data Warehouse for overall server monitoring, and use the SQL Trace data collection set for query monitoring.
* Focus your tuning efforts on SQL code that is repeatedly run, either as a stored procedure or a SQL script, and that consumes the most server resources. Data warehouse applications tend to place the greatest pressure on Disk I/O.
* For tuning and management of large queries, consider saving historical copies of XML execution plans to disk for later reference.

## Conclusion and Resources

Querying in a data warehouse introduces complexities beyond those typically encountered in a standard OLTP database. The sheer volume of data to be loaded, processed, analyzed, and managed requires greater understanding of some of the internals and capabilities of SQL Server. Any database server, regardless of size, deployed in a data warehouse environment runs the risk of getting bogged down and even becoming unresponsive without the benefit of these fundamental data warehouse management practices.

This chapter only begins to frame the discipline of data warehouse querying, monitoring, and performance tuning. With this foundation, however, you can dig deeper into these topics, deepen your understanding with further research and testing, and verify and share the best practices you discover as you deploy your SQL Server data warehouse applications.

## Additional Resources

De-normalization Concepts: <http://en.wikipedia.org/wiki/Denormalization>

Partitioned Tables: <http://technet.microsoft.com/en-us/library/dd578580(SQL.100).aspx>

Partitioned Indexes: <http://msdn.microsoft.com/en-us/library/ms187526.aspx>

Partition Table Maintenance (Sliding Window’):  
<http://blogs.msdn.com/b/menzos/archive/2008/06/30/table-partitioning-sliding-window-case.aspx>

SQL Server’s MAXDOP setting and partitioned tables: <http://blogs.msdn.com/b/sqlcat/archive/2005/11/30/498415.aspx>

Covering Indexes: <http://www.simple-talk.com/sql/learn-sql-server/using-covering-indexes-to-improve-query-performance/>

Filtered Indexes: <http://www.sql-server-performance.com/articles/dba/Filtered_Indexes_in_SQL_Server_2008_p1.aspx>

Indexed Views: <http://msdn.microsoft.com/en-us/library/dd171921(SQL.100).aspx>

Client side versus server side traces: <http://sqlserverpedia.com/wiki/The_Server-side_Trace:_What,_Why,_and_How>

The Management Data Warehouse: <http://msdn.microsoft.com/en-us/library/bb677306.aspx>

SQL Server Utility: <http://msdn.microsoft.com/en-us/library/ee210548.aspx>

Query Execution Plans: <http://www.simple-talk.com/sql/performance/execution-plan-basics>

Join Operators:  
<http://www.sql-server-performance.com/articles/dba/pysical_join_operators_p1.aspx>  
<http://www.sql-server-performance.com/articles/dba/physical_join_operators_merge_p1.aspx>  
<http://www.sql-server-performance.com/articles/dba/physical_join_operators_hash_p1.aspx>

Isolation Levels: <http://msdn.microsoft.com/en-us/library/ms189122.aspx>

Row versioning based isolation levels: <http://msdn.microsoft.com/en-us/library/ms191190.aspx>

SQL 2008 Data Compression: <http://msdn.microsoft.com/en-us/library/dd894051.aspx>