SAMPLE CHAPTER

SQL Server DMVs IN ACTION

Better queries with Dynamic Management Views

Ian W. Stirk

MANNING

SQL Server DMVs in Action

by Ian W. Stirk

Chapter 3

brief contents

3 Index DMVs

This chapter covers

- Background on the importance of indexes
- Code snippets identifying aspects of suboptimal indexes
- Discussions on how to optimize indexes
- Holistic approach to index usage

Indexes are used to improve the performance of data retrieval, to order data, and sometimes to enforce uniqueness. It's the first of these uses, improving data retrieval, that I'll focus on in this chapter. We'll use the DMVs to identify indexes that may be suboptimal or unnecessary, as well as indexes the optimizer would like to use but are missing. All these aspects of index optimality can affect the performance of your SQL queries, sometimes significantly. I'll provide discussions within each code snippet on index optimization.

After we've examined various code snippets relating to index DMVs, I'll summarize the conflicting requirements of indexes with regard to data retrieval and modification and offer a holistic view of how index usage can be balanced. Let's begin by examining why indexes are important.

3.1 The importance of indexes

Because tables in production systems may easily contain many millions of rows, reading an entire table's content without using indexes would be impractical. Without an index, the entire table may need to be read to satisfy any queries run against it. Indexes typically enable queries to run quickly and efficiently.

3.1.1 Types of index

The data in an index, or indeed a table, is held as rows, and these rows belong to a page. A page (or group of pages) is typically the unit of data transfer when data is read. A brief description of the types of indexes is given in table 3.1.

Table 3.1 Types of indexes

Type of index	Description
Clustered	This index is the table itself with the physical order of the rows defined. It's typically good for range-based queries.
Nonclustered	An index that contains a subset of a table's columns. It's typically good for retrieving a small subset of easily identified rows.
Covered	An index that contains all the data required by a query.
Composite	An index that's composed of more than one column.
Filtered	An index that's based on a WHERE clause. This feature is present only in SQL Server 2008 and higher.

Indexes are data structures that duplicate part of the data already held in the underlying table the index relates to. Although this duplication can be costly for updates, it provides a quick way of accessing the table's data. This is especially true when the index contains all the information the query needs. In this case, the table isn't accessed at all; only the index is accessed. This type of index is called a *covered* index.

A table can have many indexes, but only one contains all the columns and physically orders the table's rows. This is the *clustered* index. The clustered index is typically used for retrieving a range of rows between two values (for example, to retrieve all the invoices between two invoice dates). As such, the clustered index is often determined by the most important queries, which retrieve a range of data. Having the data physically contiguous will improve query performance, especially where you want most of the data on the row, because each retrieved page will have related relevant data. Tables without a clustered index are known as heaps.

Other indexes are called *nonclustered* indexes; these are separate from the underlying table. They typically contain columns that identify a specific subset of rows (for example, you can use them to retrieve invoices that have invoice numbers 240577, 040427, 060168, 240797). Nonclustered indexes contain a pointer back to the underlying clustered index; this can be useful when other data needs to be

retrieved from the underlying table (this is shown as an index or key lookup in the cached plan).

Indexes that are made up of multiple columns are called *composite* indexes. Sometimes the optimizer will create such a composite index dynamically, combining data from two or more indexes, if it determines this will result in a faster query.

You can include some additional data columns with the index key columns. This can improve query performance because the information can be retrieved solely from the index, rather than having to visit both the index and the underlying table's data. Such columns are known as included columns.

SQL Server 2008 introduced the concept of *filtered* indexes. These allow you to create an index for a given WHERE clause. This is a powerful feature. In many cases, you're mostly concerned with either the most current data or data that has been added recently. If you create a filtered index on the relevant tables for the most recent data, you can have, in essence, the equivalent of a two-day-old database. Best of all, the statistics for the filtered indexes are more precise, giving a more complete representation of the data and hopefully giving better query performance.

3.1.2 Types of index access

Indexes can be accessed via seeks, scans, and lookups. These are explained briefly in table 3.2.

Table 3.2 Types of index access

Type of index access	Description
Seek	Selectively accesses discreet rows of data in an index
Scan	Accesses a range of index rows
Lookup	Selectively accesses discreet rows of data in index and then gets additional data from the index's underlying table via a lookup

Seeks involve selectively accessing discreet rows of the index. This is ideal for queries that can pinpoint their data relatively easily with a high degree of selectivity.

Index *scans* involve accessing the index at a given point, such as a start date, and reading the index's data until another given point is reached, such as an end date. This access mechanism is ideal for range queries, for example, selecting all the invoices within a given date range. In some cases, identifying index scans in the cached plans might indicate an index is missing; further investigation of the cached plan's SQL query should help determine whether this is the case.

Index *lookups* involve identifying part of the information you want in the index and then accessing the underlying table to obtain the rest of the data. Again, in some cases, identifying lookups in the cached plans might indicate an index is missing one or more columns. Such missing columns could be added to the index as included columns.

Ideally, when you're reviewing your SQL queries in preparation to moving them into a production environment, you should check a query's cached plan to determine if it's using the correct index access mechanism. Alternatively, you can search the available cached plans for index access mechanisms that might be inappropriate and check to see if they're valid for the underlying query. I'll provide such a code snippet to search cached plans in chapter 5, "Further query improvements."

3.1.3 Factors affecting index performance

Indexes are central to query performance. It's essential to ensure that the factors that affect the efficiency of an index are optimal. It's important to undertake regular housekeeping to ensure these factors are kept optimal.

Later, in section 3.7, I'll provide a script that identifies which indexes are used and how these indexes are used when a given SQL query or batch of SQL is run. This will allow you to optimize these known indexes for a given SQL query, hopefully resulting in faster queries. Some of the factors you can optimize to improve an index's performance are discussed in the following sections.

FILL FACTOR

Fill factor describes how full an index page is. When you're retrieving data, you want each page to have as much data on it as possible. This will allow you to fulfill a query's needs with minimum reads, less locking, and less CPU usage. Similarly, when you're identifying a group of rows that are to be subsequently updated, you want to be able to obtain as much data as possible per read operation. But when you need to insert data, which needs to be placed in sequence with related data, the index page should contain space for this. When there's no space in the index page, page splitting will occur, resulting in queries taking longer because lookups need to be performed.

Your dilemma is being able to balance the need for this additional space to cater for any inserts/updates with the need to retrieve as many rows as possible with each page read. You can reserve some space on each index page, so that new data can be inserted in the correct order. You do this by specifying the index's fill factor, which is used when the index is created or rebuilt. If the index is largely read-only, the optimal fill factor for it is 100. If the index is modified often, a fill factor of 70 may be more appropriate. These are only rough guidelines; you should test out the appropriate fill factor values on your own systems. The default fill factor value is 0; this is similar to a fill factor of 100, but some space is left in the upper levels of the index for inserted data.

Typically, reads outnumber writes on a database by a factor of at least 5 or 10, even on transaction-intensive systems. Database read performance tends to be inversely proportional to the index fill factor, so a fill factor of 50% means reads are twice as slow as when the fill factor is 100. You must take care not to overstate the role of index updates in determining the index's fill factor.

You can use the following query to see the fill factor of the indexes on the tables in the current database (the database where the query is run):

```
SELECT DB_NAME() AS DatabaseName
       , SCHEMA_NAME(o.Schema_ID) AS SchemaName
       , OBJECT_NAME(s.[object_id]) AS TableName
       , i.name AS IndexName
       , i.fill_factor
FROM   sys.dm_db_index_usage_stats s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
                    AND s.index_id = i.index_id
INNER JOIN sys.objects o ON i.object_id = O.object_id
WHERE s.database_id = DB_ID()
    AND i.name IS NOT NULL
    AND OBJECTPROPERTY(s.[object_id], 'IsMsShipped') = 0
ORDER BY fill_factor DESC
```

An example of the type of output for this query is given in figure 3.1.

Later in this chapter, in section 3.5.1, I'll provide a more comprehensive script to show the importance of fill factor in greater detail.

STATISTICS

In many ways, statistics are at the heart of the decision-making engine that is SQL Server's optimizer. *Statistics* describe the distribution and density of column values. Unless you have relevant and up-to-date statistics, which are needed to estimate the probability of retrieving a row's column value, a relevant index may not be used or may be accessed incorrectly, resulting in suboptimal query performance.

You can see the summary statistics for a given index by running a version of the following command in SSMS:

```
DBCC SHOW_STATISTICS ([schema.tableName], indexName) WITH STAT_HEADER
```

To find statistics details for an index named IX_Deal_5, on a table named Deal, belonging to the schema dbo, you'd run the following command:

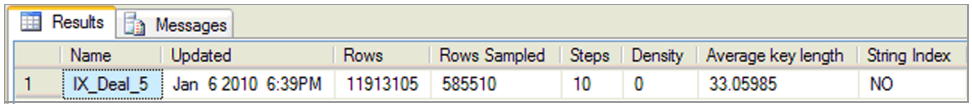
```
DBCC SHOW_STATISTICS ([dbo.deal], IX_Deal_5) WITH STAT_HEADER
```

Sample output for this query is shown in figure 3.2.

Results | Messages

	DatabaseName	SchemaName	TableName	IndexName	fill_factor
1	ParisDev	dbo	MessageLog	IX_MessageLog_1	85
2	ParisDev	dbo	MessageLog	IX_MessageLog_2	85
3	ParisDev	dbo	Deal	IX_Deal_StructuredTradeID	80
4	ParisDev	dbo	StructuredTrade	IX_StructuredTrade_StructuredTradeTypeID	80
5	ParisDev	dbo	Component	IX_Component_OrgId	50
6	ParisDev	dbo	Component	IX_Component_InstrumentId	50
7	ParisDev	dbo	Component	IX_Component_BookCode	50
8	ParisDev	dbo	Component	IX_Component_3	50
9	ParisDev	dbo	Component	IX_Component_1	50
10	ParisDev	dbo	Component	IX_Component	50
11	ParisDev	dbo	Component	IX_Component_DealId_ComponendCode	0
12	ParisDev	dbo	Party	PK_Party_1	0

Figure 3.1 Output showing the fill factor of indexes



Results | Messages

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index
1	IX_Deal_5	Jan 6 2010 6:39PM	11913105	585510	10	0	33.05985	NO

Figure 3.2 Output showing the summary statistics for the index IX_Deal_5

In the output, of particular relevance are the Updated, Rows, and Rows Sampled columns. Using these columns, you get an idea of how old the statistics are and how much data is present. The Updated column describes when the statistics for the IX_Deal_5 index were last updated. The Rows column describes the number of rows in the table, and the Rows Sampled column describes the number of rows sampled to get statistics. Because the number of rows sampled (585,510) is smaller than the number of rows (11,913,105), the detailed statistics are based on this sample (approximately 5% of the rows were sampled). Later in this chapter, in section 3.10, I'll provide a more comprehensive script to show statistics in greater detail.

Typically, you want indexes on those columns that are used in JOIN and WHERE clauses (the joins often imply that you should implement indexes on foreign keys). When an index is created, statistics about the columns in the index are also created. Statistics are updated when the data in the underlying tables changes. Typically, when a table's data changes by 20% (since the statistics were last updated), its statistics are automatically recalculated. Changes in the statistics also cause any queries that use the underlying table to be recompiled, using the new statistics to produce (hopefully) a better cached plan and a better SQL query.

Sometimes, especially for larger tables, waiting for the table to change by 20% takes too long, resulting in the usage of stale and suboptimal plans. I've known of many queries that could be improved almost immediately by updating the table's statistics. I'll provide a script later in this chapter, in section 3.7, that will identify the indexes used by a given routine or SQL batch. This will allow you to update the statistics of those given indexes before the queries are run, rather than doing a blanket table update. This will allow the statistics update to run faster or have a large sampling percentage that should help improve query performance.

Similarly, in section 3.10, I'll provide a script that will describe the current state of your index statistics. Investigating the percentage of rows changed and the last updated columns should help in determining whether the statistics should be updated more often than the default.

In chapter 10, "The self-healing database," I'll provide a script that automatically updates the index statistics in an intelligent manner. The script updates only the statistics of the indexes whose data has changed and does so using an intelligent sampling algorithm.

FRAGMENTATION

Logical index *fragmentation* describes the percentage of index entries that are out of sequence. This has an impact on indexes that are involved in scans, increasing the

amount of work they have to do because the index data isn't contiguous. Where possible, you should remove this type of fragmentation by reorganizing or rebuilding the index. Typically, for indexes that have more than 30% fragmentation, an index rebuild is recommended. If the fragmentation percentage is between 10% and 30%, index reorganization is recommended.

You can use the following query to see the fragmentation percentage of the indexes on a table named currency, within a database named parisdev:

```
SELECT i.name AS IndexName
       , ROUND(s.avg_fragmentation_in_percent,2) AS [Fragmentation %]
FROM sys.dm_db_index_physical_stats(DB_ID('parisdev'),
OBJECT_ID('currency'), NULL, NULL, NULL) s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
    AND s.index_id = i.index_id
```

Figure 3.3 shows an example of the type of output for this query.

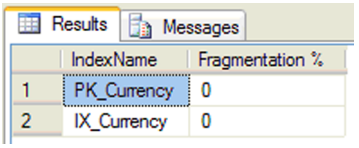


Figure 3.3 Output showing the fragmentation percentage of indexes

Later in this chapter, in section 3.6, I'll provide a more comprehensive script to show the fragmentation in greater detail. In chapter 10 I'll provide a script that automatically defragments indexes in an intelligent manner. The script defragments only the indexes whose data has fragmented significantly, and it decides between an index rebuild or a reorganization based on the degree of fragmentation.

I/O SUBSYSTEM

If you know that certain indexes are typically used together, for example, in JOINs or WHERE clauses, it might be prudent to put them on a different physical disk with their own disk controllers. This will allow a higher degree of parallelism and a corresponding performance improvement. It may be worthwhile putting the most-frequently used indexes on their own disks for similar reasons; a script later in this chapter, in section 3.5, will identify these indexes.

This applies to frequently used tables as well as indexes, and where possible, tables that are used together should be placed on different disks.

COMPRESSION (2008)

SQL Server 2008 provides a facility to compress data, including data in indexes. By default, indexes aren't compressed when the underlying table is compressed; they have to be done separately. Compression ratios of 40% or more are common, allowing you to retrieve almost twice as much data per read operation.

If data is compressed, you can retrieve more data per page you read, so the performance of many queries should increase. You do need additional time to uncompress the data. You need to balance the positive aspect of retrieving more data per page with the negative impact of the time taken to uncompress the data. Luckily a GUI tool and associated stored procedures are provided that allow you to estimate

the saving compression will provide. You can access the GUI tool from within SSMS by right-clicking a table, selecting Storage, and then selecting Manage Compression. You can use this tool to determine whether index compression is advantageous for your indexes.

Having discussed why indexes are important and factors that affect their importance, let's now look at which indexes could significantly improve the query performance but are missing.

3.2 Costly missing indexes

Indexes are typically the most important factor in identifying the relevant data rows quickly. They're used both for data retrieval and to identify data for subsequent modification. As much as an index has a big impact on quickly identifying rows of data, a missing index can have a corresponding detrimental impact on performance.

When SQL Server runs queries, it examines the query and tables/views and determines which indexes it would like to use. If these indexes are present, it typically uses them. But if these indexes aren't present, it makes a note of them with the cached plan in internal data structures that you can view via the DMVs.

> **Example of impact of implementing a missing index**
>
> A missing index can have a huge effect on performance. I've seen an example in a production system where a stored procedure was taking more than four hours to run, but when the missing indexes were applied, it ran in under five minutes.

3.2.1 Finding the most important missing indexes

Indexes are a principal means of improving the performance of SQL queries. But for various reasons, for example, changing systems, useful indexes may not always have been created. Running the SQL query given in the following listing will identify the top 20 indexes, ordered by impact (Total Cost), that are missing from your system.

Listing 3.1 Identifying the most important missing indexes

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT TOP 20
    ROUND(s.avg_total_user_cost *                                    (1) Calculate total cost
            s.avg_user_impact                                            of missing index
                    * (s.user_seeks + s.user_scans),0)
                                    AS [Total Cost]
    , d.[statement] AS [Table Name]
    , equality_columns
    , inequality_columns
    , included_columns
FROM sys.dm_db_missing_index_groups g
INNER JOIN sys.dm_db_missing_index_group_stats s
    ON s.group_handle = g.index_group_handle
```

```
INNER JOIN sys.dm_db_missing_index_details d
    ON d.index_handle = g.index_handle
ORDER BY [Total Cost] DESC
```

In the listing, you can see that three DMVs are involved in identifying missing indexes; a brief description of each one is given in table 3.3.

Table 3.3 DMVs used to find missing indexes

DMV	Description
sys.dm_db_missing_index_details	Contains details of the database/schema/table the missing index relates to, together with how the index usage has been identified in queries (such as equality/inequality).
sys.dm_db_missing_index_group_stats	Contains details of how often the index would have been used, how it would be used (seek or scan), and a measure of the effectiveness of the index.
sys.dm_db_missing_index_groups	This is a linking DMV, linking the previous two DMVs together.

The joining of these three DMVs provides you with enough information to fully describe the missing indexes across all the databases on the server and assign an importance weighting (called Total Cost) to their usefulness. There are various ways of calculating the importance of an index. In this example, I combine the frequency of index usage (user_seeks and user_scans) with the measures of query improvement (avg_total_user_cost and avg_user_impact) ❶.

SQL Server Books Online defines the column avg_total_user_cost this way: Average cost of the user queries that could be reduced by the index in the group. Similarly, avg_user_impact is defined: Average percentage benefit that user queries could experience if this missing index group was implemented. The value means that the query cost would on average drop by this percentage if this missing index group was implemented. Combining these columns gives us a measure of expected query improvement.

The column user_seeks represents the number of SQL queries that would have used the index to seek data; it's not the number of times the index has been accessed. Similarly, user_scans represents the number of SQL queries that would have used the index to scan for data.

The results are ordered by the calculated total cost column, in descending order, so that the most important indexes are listed first. The T-SQL ROUND function is used to ensure the value of the total cost column is rounded up to remove any decimal places. The TOP command is used to restrict the output to the 20 most important missing indexes.

The column named TableName identifies the database/schema and table the index relates to. The columns equality_columns and inequality_columns contain

Results | Messages

	Total Cost	TableName	equality_columns	inequality_columns	included_columns
1	35090665	[Paris].[dbo].[PositionGrid...	[DomainId]	[COB], [SourceId]	[PositionGridCellId], [Positi...
2	22405884	[Paris].[dbo].[Request]	[RequestDefinit...	NULL	[StatusCode], [COB]
3	17681267	[Paris].[dbo].[Deal]	[DealVersion]	[DealCode]	[DealId], [SourceId], [Part...
4	17186169	[Paris].[dbo].[PNLAdjustm...	NULL	[Status]	[PNLAdjustmentQueueId],...
5	9087210	[Paris].[dbo].[Mapping]	[MappingTypeI...	NULL	[FromValue], [ToValue]
6	8026926	[Paris].[dbo].[PNLValue]	NULL	[BaseValue]	[PNLValueId], [PositionGri...
7	6950941	[Paris].[dbo].[PNLAdjustm...	NULL	[Status]	[PNLAdjustmentQueueId],...
8	6906903	[Paris].[dbo].[RequestDeal...	[BatchNbr]	NULL	[RequestDealComponent...
9	6024897	[Paris].[dbo].[PNLAdjustm...	NULL	[Status]	[PNLAdjustmentQueueId],...
10	5988629	[Paris].[dbo].[PositionGrid...	[DomainId]	[SourceId]	[PositionGridCellId], [COB]...
11	5963443	[Paris].[dbo].[PNLAdjustm...	NULL	[BaseValue]	[PNLAdjustmentId], [Positi...
12	5944962	[Paris].[dbo].[RequestPNL...	[BatchNbr]	NULL	NULL

Figure 3.4 Output showing the most important missing indexes

the names of columns that should be used to create the index. The column named included_columns identifies the columns that should be defined as included columns on the index.

An example of the type of output for this query is shown in figure 3.4.

3.2.2 The impact of missing indexes

Although you can use the output from the missing-indexes script to create the missing indexes, and perhaps automate this, a word of caution is needed. The output doesn't take into account the potentially conflicting requirements of all the queries that run against the identified tables. The output relates to individual queries rather than their sum cumulative actions. With this in mind, you need to balance the detrimental effects that any index may have on any updates (INSERT/UPDATE/DELETE) with the improved performance an index can bring. When you update a table's data, any relevant index will also need to be updated. This can add to the duration of the query, transaction length, and locks, leading to potential problems with blocking (and associated client timeouts).

You'll need to test to determine whether the added missing indexes, on balance, add value. The Database Tuning Advisor (DTA) is a good tool for amalgamating the sum total effects of all the queries, to determine if an index should be added. That said, on many occasions I've found the DTA to be both relatively conservative in its recommendations and time consuming to run.

The sys.dm_db_missing_index_group_stats DMV includes data for both user and system usage. *System* relates to administration-like queries, whereas *user* queries relate to application queries. Because we tend to be more interested in our own user queries, we ignore the system usage details in our script.

It's possible to create a DMV snapshot of queries (relating to duration, CPU, I/O, and the like) and another relating to missing indexes. (How to create DMV snapshots is shown in chapter 2, "Common patterns.") It should then be possible to correlate

the two snapshots, apply the missing indexes, and rerun the query to determine if the added indexes have improved performance.

You need to be careful with missing indexes that have a large number of columns in the included_columns column; this may be because someone is accessing the table via a SELECT * FROM tableName query. This in itself may be interesting, because it suggests the user may not know what data columns they want, or maybe they're being lazy in specifying the columns they want.

It's probably better, initially, to focus on those missing indexes that have a NULL value in the included_columns column. If the inequality_columns column is populated, it might be worthwhile searching your code base, which represents all your SQL queries taken as a whole, for the column identified. Often it's a good indicator of a SQL query that can be written more efficiently. For example, instead of specifying the column is *not* equal to something, it may be better to rewrite the query so it contains just the possible values.

If you want to focus your performance improvements on a given database, schema, or table, you could amend the missing-indexes query to look only at a given database, schema, or table you're interested in. For example, to retrieve missing index information for a database/schema/table called '[Paris].[dbo].[Component]', you could add the following to the query:

```
WHERE [statement] = '[Paris].[dbo].[Component]'
```

It's also possible to search the cached plans for missing indexes. I'll show a script for this later in the chapter on improving poor query performance, in section 5.2 ("Finding queries that have missing indexes"). Looking at these cached plans before and after the indexes have been implemented should provide valuable insight into the effectiveness of the index. In addition, when you open the cached plan in SSMS 2008, you can easily extract the missing index definition.

In chapter 10 I'll provide a script that automatically creates the SQL to build (and optionally implement) these missing indexes.

Having looked at useful indexes that are missing, we'll now look at the opposite view, indexes that exist but aren't being used at all for data retrieval.

3.3 Unused indexes

Indexes are great for improving the performance of retrieval-based queries. In addition, if an update query has a WHERE clause or a JOIN condition, it may use an index to identify the subset of rows to update, and this will improve the performance of the query.

But indexes can have a detrimental effect on updates. This occurs when a table is updated (via UPDATE, DELETE, or INSERT) and the index isn't used. In these cases, the index can have a detrimental effect on query performance, because the index may need to be updated too. This will add to the query duration, length of transaction, and locks, leading to blocking and potential client timeouts.

In essence, costly unused indexes force SQL Server to do unnecessary work. In addition to queries taking longer to execute, administrative functions like backups and restores will take longer to complete, and there's an additional cost associated with the storage of unnecessary data.

3.3.1 Finding the most-costly unused indexes

Superfluous indexes have a detrimental effect on the performance of your SQL queries, because they cause SQL Server to do unnecessary work. Running the SQL script given in the next listing will identify the top 20 most-costly unused indexes, ordered by the number of updates that have been applied to them.

Listing 3.2 The most-costly unused indexes

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT
    DB_NAME() AS DatabaseName
    , SCHEMA_NAME(o.Schema_ID) AS SchemaName
    , OBJECT_NAME(s.[object_id]) AS TableName
    , i.name AS IndexName
    , s.user_updates
    , s.system_seeks + s.system_scans + s.system_lookups
                          AS [System usage]
INTO #TempUnusedIndexes
FROM   sys.dm_db_index_usage_stats s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
    AND s.index_id = i.index_id
INNER JOIN sys.objects o ON i.object_id = O.object_id
WHERE 1=2

EXEC sp_MSForEachDB 'USE [?];
INSERT INTO #TempUnusedIndexes
SELECT TOP 20
    DB_NAME() AS DatabaseName
    , SCHEMA_NAME(o.Schema_ID) AS SchemaName
    , OBJECT_NAME(s.[object_id]) AS TableName
    , i.name AS IndexName
    , s.user_updates
    , s.system_seeks + s.system_scans + s.system_lookups
                                        AS [System usage]
FROM   sys.dm_db_index_usage_stats s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
    AND s.index_id = i.index_id
INNER JOIN sys.objects o ON i.object_id = O.object_id
WHERE s.database_id = DB_ID()
AND OBJECTPROPERTY(s.[object_id], ''IsMsShipped'') = 0
AND s.user_seeks = 0
    AND s.user_scans = 0
    AND s.user_lookups = 0
AND i.name IS NOT NULL
ORDER BY s.user_updates DESC'

SELECT TOP 20 * FROM #TempUnusedIndexes ORDER BY [user_updates] DESC

DROP TABLE #TempUnusedIndexes
```

1 Temp table to hold results

2 Loop around all databases

3 Identify most-costly unused indexes

Here you can see that a single DMV and two system tables are involved in identifying the most-costly unused indexes; a brief description of each is shown in table 3.4.

Table 3.4 DMV/system tables to identify the most-costly unused indexes

DMV/tables	Description
sys.dm_db_index_usage_stats	Contains details of the different types of index operations, for example, number of updates by user queries
sys.indexes	Contains details for each index, for example, name and type
sys.objects	Contains details for each object, for example, schema name

By joining the DMV and two system tables, we have enough information to identify the most-costly unused indexes across all the databases on the server. The DMV and the sys.indexes system table are joined on their common key columns: object_id and index_id. The system tables sys.objects and sys.indexes are joined on the object_id key. The system table sys.objects is used to provide information about the schema the index relates to.

We use a common pattern to create the temporary table to hold the transient results. Again, we use another common pattern to loop over all the databases on the server. For more detail on these common patterns, see chapter 2.

The first part of the script creates an empty temporary table (named #Temp-UnusedIndexes) with the required structure of column names and data types ❶. We use the Microsoft-supplied stored procedure sp_MSForEachDB to execute a query on each database on the server ❷. The query we execute selects the 20 most-costly unused indexes on each database ❸. We put the results of each execution into the temporary table. Finally, we select the top 20 most-costly indexes across all the databases on the server.

The query we execute on each database identifies the top 20 most-costly unused indexes by selecting those indexes that haven't been used in any user queries to retrieve data (there are no seeks, scans, or lookups) but have been updated when the relevant columns in the underlying table have been updated. The results are sorted by the number of updates that user queries have caused to be applied to the index, in descending order.

> **NOTE** We ignore any indexes whose name column is set to NULL. This is because they aren't indexes; they're heaps. Similarly, because we're interested in only our own user-created indexes, we filter out any indexes that relate to tables created by the SQL Server installation process (the column IsMsShipped has a value of 1). We include the calculated sum of any system usage columns in the output; this will allow us to determine if the index is necessary for any system processing and if further investigation is needed.

Sample output for this query is shown in figure 3.5.

Results | Messages

	DatabaseName	SchemaName	TableName	IndexName	user_updates	System usage
1	ParisMini	dbo	MessageLog	IX_MessageLog_1	1559	0
2	ParisMini	dbo	MessageLog	IX_MessageLog_2	1559	0
3	ParisDev	dbo	Request	IX_Request_3	512	0
4	ParisDev	dbo	Request	IX_Request_1	252	0
5	ParisDev	dbo	Request	IX_Request_2	252	0
6	ParisDev	dbo	Mapping	IX_Mapping_4	168	0
7	ParisDev	dbo	Deal	IX_Deal_OrgId	132	0
8	ParisDev	dbo	Deal	IX_Deal_ProductGroupCode	132	0
9	ParisDev	dbo	Deal	IX_Deal_StructuredTradeID	132	0
10	ParisDev	Logging	ProcessLog	IX_Logging.ProcessLog_UniqueReference	130	0

Figure 3.5 Output showing the most-costly unused indexes

3.3.2 The impact of unused indexes

There are various reasons why you might have indexes that aren't used for data retrieval. These include changing application functionality, changing database usage, inexperienced developers, and the lack of a project follow-up phase.

Often when a project starts, there's limited knowledge about the queries you want to run on the database and the indexes you need to fulfill these queries. Indexes may be created with a best-guess approach. As the project progresses, the needs of the project become more fully understood, and the queries and indexes become more stable. Indexes that were created early on may no longer be appropriate but are forgotten about or ignored, resulting in potential impedance on query performance.

Similarly, an application may change significantly, running new queries and requiring new indexes, but the old indexes are left in place. Perhaps a key player in the development team leaves, and the remaining developers are unsure of query and index usage. Maybe a combination of increased data volume and a change in the type of data results in another index being used for data retrieval.

For a variety of reasons, some projects are implemented by inexperienced staff. They may implement indexes without sufficient thought as to the queries that run against the tables. For example, they might assume an index is being used but have insufficient knowledge to inspect the cached plans, which show the index isn't being used (maybe a table scan is being used or a data type conversion is taking place; both can result in unused indexes).

All these reasons could potentially produce costly unused indexes. Because you know that costly unused indexes have a detrimental impact on performance, you should try to remove them where possible. I'd suggest running the query given in the previous script (listing 3.2) as part of the user acceptance testing plan and also when changes are made to the application, in order to determine if indexes are still required. It would also be prudent to run the script as a regular part of any database housekeeping, to ensure you identify any potential superfluous indexes and to remove them.

Removing indexes

Since SQL Server 2005, it has been possible to disable indexes rather than delete them. Disabling the index removes the index and its data but allows you to keep the definition of the index tied with its table, without having the complication of storing the definition elsewhere, should you wish to reapply it at a later date.

If you want to focus your performance improvements on a given database, schema, or table, you could amend the most-costly indexes script to look only at a given database, schema, or table you're interested in.

A note of caution is needed here. The DMVs contain data that has accumulated since the last SQL Server restart. You need to ensure that you have enough data, from all the queries that have run on the SQL Server, to ensure the indexes aren't used for data retrieval.

The DMV sys.dm_db_index_usage_stats contains entries only for indexes that have been used. Unused indexes are ones that have never been updated or used for retrieval. Although these unused indexes do no real harm, removing these indexes will simplify the schema. An OUTER JOIN between sys.indexes and sys.dm_db_index_usage_stats will identify these indexes. A script later in this chapter (see section 3.9) will identify these indexes.

3.4 High-maintenance indexes

High-maintenance indexes are indexes that are rarely used to retrieve data for user queries but may be updated when the underlying table's data is modified. In many ways they're similar to the most-costly unused indexes; they're relatively expensive and can have a negative effect on query performance, potentially increasing blocking and client timeouts.

3.4.1 Finding the top high-maintenance indexes

High-maintenance indexes, like unused indexes, can have a detrimental impact on SQL performance because they cause SQL Server to perform unnecessary work. Running the SQL script given in the following listing will identify the top 20 high-maintenance indexes, ordered by maintenance cost.

Listing 3.3 The top high-maintenance indexes

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT
    DB_NAME() AS DatabaseName
    , SCHEMA_NAME(o.Schema_ID) AS SchemaName
    , OBJECT_NAME(s.[object_id]) AS TableName
    , i.name AS IndexName
    , (s.user_updates ) AS [update usage]
    , (s.user_seeks + s.user_scans + s.user_lookups)
                                        AS [Retrieval usage]
```

```
    , (s.user_updates) -
      (s.user_seeks + s.user_scans + s.user_lookups) AS [Maintenance cost]
    , s.system_seeks + s.system_scans + s.system_lookups AS [System usage]
    , s.last_user_seek
    , s.last_user_scan
    , s.last_user_lookup
INTO #TempMaintenanceCost
FROM   sys.dm_db_index_usage_stats s
INNER JOIN sys.indexes i ON  s.[object_id] = i.[object_id]
    AND s.index_id = i.index_id
INNER JOIN sys.objects o ON i.object_id = O.object_id
WHERE 1=2

EXEC sp_MSForEachDB 'USE [?];                                    <-- 2 Loop around all databases
INSERT INTO #TempMaintenanceCost
SELECT TOP 20
    DB_NAME() AS DatabaseName
    , SCHEMA_NAME(o.Schema_ID) AS SchemaName
    , OBJECT_NAME(s.[object_id]) AS TableName
    , i.name AS IndexName
    , (s.user_updates ) AS [update usage]
    , (s.user_seeks + s.user_scans + s.user_lookups)
                   AS [Retrieval usage]
    , (s.user_updates) -
(s.user_seeks + user_scans +
                         s.user_lookups) AS [Maintenance cost]
    , s.system_seeks + s.system_scans + s.system_lookups AS [System usage]
    , s.last_user_seek
    , s.last_user_scan
    , s.last_user_lookup
FROM   sys.dm_db_index_usage_stats s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
    AND s.index_id = i.index_id
INNER JOIN sys.objects o ON i.object_id = O.object_id
WHERE s.database_id = DB_ID()
    AND i.name IS NOT NULL
    AND OBJECTPROPERTY(s.[object_id], ''IsMsShipped'') = 0       <-- 3 Identify top high-maintenance indexes
    AND (s.user_seeks + s.user_scans + s.user_lookups) > 0
ORDER BY [Maintenance cost] DESC'

SELECT top 20 * FROM #TempMaintenanceCost ORDER BY [Maintenance cost] DESC

DROP TABLE #TempMaintenanceCost
```

In the script a single DMV and two system tables are involved in identifying the most-unused indexes. Table 3.5 gives a brief description of each.

Table 3.5 DMVs/system tables to identify the top high-maintenance indexes

DMV/table	Description
sys.dm_db_index_usage_stats	Contains details of the different types of index operations, for example, number of updates by user queries
sys.indexes	Contains details for each index, for example, name and type
sys.objects	Contains details for each object, for example, schema name

The joining of the DMV and system tables provides us with enough information to identify the top high-maintenance indexes across all the databases on the server. The DMV and the sys.indexes system table are joined on their common key columns, object_id and index_id. The system tables sys.objects and sys.indexes are joined on the object_id key. The system table sys.objects is used to provide information about the schema the index relates to.

We use a common pattern to create the temporary table to hold the transient results. Then we use another common pattern to loop over all the databases on the server. For more detail on these common patterns, see chapter 2.

The first part of the script creates an empty temporary table (named #TempMaintenanceCost) with the required structure of column names and data types ❶. We use the Microsoft-supplied stored procedure, sp_MSForEachDB, to execute a query on each database on the server ❷. The query we execute selects the top 20 high-maintenance indexes on each database ❸. We put the results of each execution into the temporary table. Finally we select the top 20 high-maintenance indexes across all the databases on the server.

The query we execute on each database identifies the top 20 high-maintenance indexes by subtracting the retrieval usage from the update usage. The update usage is given by the column user_updates, and the retrieval usage is calculated as the sum of the various user index access types (user_seeks + user_scans + user_lookups). The dates of the last seek, scan, and lookup are also included. These can be used to determine if the index lookup was a long time ago, increasing the probability that it can be disabled or removed. The results are sorted by the maintenance cost in descending order.

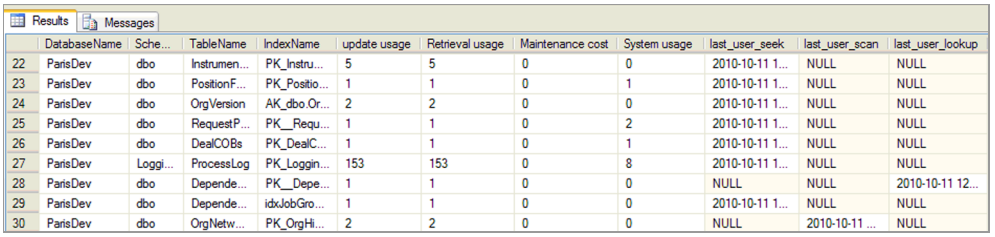
> **NOTE** We ignore any indexes whose name column is set to NULL. This is because they aren't indexes; they're heaps. Similarly, we're interested in only our own user-created indexes, so we filter out any indexes that relate to tables created by the SQL Server installation process (the column IsMsShipped has a value of 1).

Because we're interested in indexes that have at least some usage, we exclude indexes that haven't been used for data retrieval (we've already identified these, in the most-costly unused indexes script). We include the calculated sum of any system usage columns in the output; this will allow us to determine if the index is necessary for any system processing and if further investigation is needed.

Figure 3.6 shows an example of the type of output for this query.

3.4.2 The impact of high-maintenance indexes

In this section I've identified indexes that are used relatively infrequently compared to the number of index updates (which reflect updates to the underlying table's data). What you need to determine now is whether the cost of the index is too expensive compared with its usage and whether the index should be removed. If the



Results | Messages

	DatabaseName	Sche...	TableName	IndexName	update usage	Retrieval usage	Maintenance cost	System usage	last_user_seek	last_user_scan	last_user_lookup
22	ParisDev	dbo	Instrumen...	PK_Instru...	5	5	0	0	2010-10-11 1...	NULL	NULL
23	ParisDev	dbo	PositionF...	PK_Positio...	1	1	0	1	2010-10-11 1...	NULL	NULL
24	ParisDev	dbo	OrgVersion	AK_dbo.Or...	2	2	0	0	2010-10-11 1...	NULL	NULL
25	ParisDev	dbo	RequestP...	PK__Requ...	1	1	0	2	2010-10-11 1...	NULL	NULL
26	ParisDev	dbo	DealCOBs	PK_DealC...	1	1	0	1	2010-10-11 1...	NULL	NULL
27	ParisDev	Loggi...	ProcessLog	PK_Loggin...	153	153	0	8	2010-10-11 1...	NULL	NULL
28	ParisDev	dbo	Depende...	PK__Depe...	1	1	0	0	NULL	NULL	2010-10-11 12...
29	ParisDev	dbo	Depende...	idxJobGro...	1	1	0	0	2010-10-11 1...	NULL	NULL
30	ParisDev	dbo	OrgNetw...	PK_OrgHi...	2	2	0	0	NULL	2010-10-11 ...	NULL

Figure 3.6 Output showing the top high-maintenance indexes

index is required to obtain a subset of the underlying data, it may be advisable to create a filtered index.

If you examine the column retrieval_usage, you can see how often the index is used. A small value might reflect a rare ad hoc query or even a query that previously used the index but no longer does, perhaps because of changes in the volume and/or type of data. You could use DMV snapshots here to determine if the indexes are still used for data retrieval. For more information on the use of DMV snapshots, please see the DMV snapshot section in chapter 2.

If you want to focus your performance improvements on a given database, schema, or table, you could amend the top high-maintenance indexes script to look only at a given database, schema, or table you're interested in.

It might be sensible to run the script regularly and see if its retrieval usage changes. If it doesn't, this suggests the index is no longer being used, and the index is eligible for removal.

3.5 Most-frequently used indexes

If you know which indexes are used most often, you can target these indexes for further optimization. This should have a positive impact on the performance of those queries that use these indexes. These index optimizations include ensuring that the index statistics are up to date and have a good sampling percentage, the fill factor is optimal for the type of index usage, and logical fragmentation is low.

3.5.1 Finding the most-used indexes

Optimizing the indexes that are used most often will have a proportionally better impact on SQL query performance than optimizing other indexes. Running the SQL script given in the following listing will identify the top 20 most-used indexes, ordered by usage.

Listing 3.4 The most-used indexes

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT
    DB_NAME() AS DatabaseName
```

1 Temp table to hold results

```
    , SCHEMA_NAME(o.Schema_ID) AS SchemaName
    , OBJECT_NAME(s.[object_id]) AS TableName
    , i.name AS IndexName
    , (s.user_seeks + s.user_scans + s.user_lookups) AS [Usage]
    , s.user_updates
    , i.fill_factor
INTO #TempUsage
FROM sys.dm_db_index_usage_stats s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
    AND s.index_id = i.index_id
INNER JOIN sys.objects o ON i.object_id = O.object_id
WHERE 1=2

EXEC sp_MSForEachDB 'USE [?];
INSERT INTO #TempUsage
SELECT TOP 20
    DB_NAME() AS DatabaseName
    , SCHEMA_NAME(o.Schema_ID) AS SchemaName
    , OBJECT_NAME(s.[object_id]) AS TableName
    , i.name AS IndexName
    , (s.user_seeks + s.user_scans + s.user_lookups) AS [Usage]
    , s.user_updates
    , i.fill_factor
FROM   sys.dm_db_index_usage_stats s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
            AND s.index_id = i.index_id
INNER JOIN sys.objects o ON i.object_id = O.object_id
WHERE s.database_id = DB_ID()
    AND i.name IS NOT NULL
    AND OBJECTPROPERTY(s.[object_id], ''IsMsShipped'') = 0
ORDER BY [Usage] DESC'

SELECT TOP 20 * FROM #TempUsage ORDER BY [Usage] DESC

DROP TABLE #TempUsage
```

❷ **Loop around all databases**

❸ **Identify most-used indexes**

A single DMV and two system tables are used to identify the most-used indexes. A brief description of each is given in table 3.6.

Table 3.6 DMV/system tables to identify the most-used indexes

DMV/tables	Description
sys.dm_db_index_usage_stats	Contains details of the different types of index operations, for example, number of updates by user queries
sys.indexes	Contains details for ea.ch index, for example, name and type
sys.objects	Contains details for each object, for example, schema name

Joining the DMV and two system tables provides us with enough information to identify the most-used indexes across all the databases on the server. The DMV and the sys.indexes system table are joined on their common key columns, object_id and index_id. The system tables sys.objects and sys.indexes are joined on the object_id key.

The system table sys.objects is used to provide information about the schema the index relates to.

We use a common pattern to create the temporary table to hold the transient results. We use another common pattern to loop over all the databases on the server. For more details on these common patterns, see chapter 2.

The first part of the script creates an empty temporary table (named #TempUsage) with the required structure of column names and data types ❶. We use the Microsoft-supplied stored procedure, sp_MSForEachDB, to execute a query on each database on the server ❷. The query we execute selects the 20 most-used indexes on each database ❸. We put the results of each execution into the temporary table. Finally, we select the top 20 most-used indexes across all the databases on the server.

The query we execute on each database identifies the top 20 most-used indexes by calculating the sum of all the user index access counts (user_seeks + user_scans + user_lookups) and sorting by this calculated sum in descending order. We also report on the index's fill_factor value; this will help us decide if we're using the appropriate fill factor in light of the amount of index data being read/updated.

Note that we ignore any heaps because they aren't indexes. Also, we're interested in only user-created indexes.

An example of the type of output for this query is shown in figure 3.7.

3.5.2 The importance of the most-used indexes

Knowing the most-commonly used indexes allows you to target your optimizations, confident in the knowledge that your changes should have a positive effect on the performance of those queries that use these most-popular indexes.

Indexes are used for data retrieval or to identify a subset of rows to modify. For the identified most-used indexes, you should ensure that the index's statistics are up to date and have a good sampling percentage. This should ensure that any queries that use the index have access to valid information about the probability of data values based on a column's data values. This is especially important for large tables where the automatic updating of statistics information may be suboptimal.

Results | Messages

	DatabaseName	SchemaName	TableName	IndexName	Usage	user_updates	fill_factor
1	ParisDev	dbo	OrgNetwork	PK_OrgHierarchy	7993236	0	0
2	ParisDev	dbo	OrgNetwork	AK1_dbo.OrgNetwork_OrgHierarchyGroupId_ChildOrgId	7993213	0	0
3	ParisDev	dbo	Org	PK_Org	1486184	0	0
4	ParisMini	dbo	Source	IX_Source_1	160162	0	0
5	ParisMini	dbo	Currency	IX_Currency	109007	0	0
6	ParisMini	dbo	MappingType	IX_MappingType	77560	0	0
7	ParisDev	dbo	Product	PK_Product	8348	0	0
8	ParisDev	dbo	BusinessRegion	PK_BusinessRegion	6769	0	0
9	ParisDev	dbo	Bucket	PK_Bucket	3992	0	0
10	ParisDev	dbo	Source	IX_Source_1	3664	0	0
11	ParisDev	dbo	BucketGroupLink	IX_BucketGroupLink_1	3405	0	0
12	ParisDev	dbo	BucketType	PK_BucketType	2971	0	0

Figure 3.7 Output showing the most-used indexes

Similarly, you should look at the index's fill factor with a view to optimizing it, by ensuring you can get the most data per read, taking into account the number of updates the index is involved in. This should be easy to apply to tables and indexes that are relatively static, such as currency or country tables.

You might also consider placing the indexes on their own disk drive with their own disk controller, allowing queries to take advantage of concurrent access. If you're using SQL Server 2008, you might also consider these indexes for compression, because this will allow you to obtain more data for each read.

If you want to focus your performance improvements on a given database, schema, or table, you could amend the most-used-indexes query to look only at a given database, schema, or table you're interested in.

3.6 Fragmented indexes

Fragmentation relates to index entries that are out of sequence. For queries that access data sequentially, typically index scans, additional work is needed to retrieve the index's data. This additional work can result in longer-running queries, with potentially more blocking and client timeouts. Where possible, you should remove this fragmentation so you don't perform any unnecessary work.

3.6.1 Finding the most-fragmented indexes

The crux of this script is the DMV sys.dm_db_index_physical_stats. This DMV accepts various parameters, allowing fragmentation to be reported on at various levels of granularity, such as for a given database, table, or index. Under the hood, this DMV calls database console commands (DBCC), which can take a long time to execute. Consider this and its impact on resources when running this script. On my 4.5 terabyte database, with 128 GB of RAM, 16 CPUs, 600 indexes, and containing 255 tables, this script took more than an hour to execute.

The script we use to identify the most-fragmented indexes is shown here.

Listing 3.5 The most-fragmented indexes

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT
    DB_NAME() AS DatbaseName
    , SCHEMA_NAME(o.Schema_ID) AS SchemaName
    , OBJECT_NAME(s.[object_id]) AS TableName
    , i.name AS IndexName
    , ROUND(s.avg_fragmentation_in_percent,2) AS [Fragmentation %]
INTO #TempFragmentation
FROM sys.dm_db_index_physical_stats(db_id(),null, null, null, null) s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
    AND s.index_id = i.index_id
INNER JOIN sys.objects o ON i.object_id = O.object_id
WHERE 1=2

EXEC sp_MSForEachDB 'USE [?];
INSERT INTO #TempFragmentation
```

❶ Temp table to hold results

❷ Loop around all databases

```
SELECT TOP 20
    DB_NAME() AS DatbaseName
    , SCHEMA_NAME(o.Schema_ID) AS SchemaName
    , OBJECT_NAME(s.[object_id]) AS TableName
    , i.name AS IndexName
    , ROUND(s.avg_fragmentation_in_percent,2) AS [Fragmentation %]
FROM sys.dm_db_index_physical_stats(db_id(),null, null, null, null) s
INNER JOIN sys.indexes i ON s.[object_id] = i.[object_id]
    AND s.index_id = i.index_id
INNER JOIN sys.objects o ON i.object_id = O.object_id
WHERE s.database_id = DB_ID()
  AND i.name IS NOT NULL
  AND OBJECTPROPERTY(s.[object_id], ''IsMsShipped'') = 0
ORDER BY [Fragmentation %] DESC'

SELECT top 20 * FROM #TempFragmentation ORDER BY [Fragmentation %] DESC

DROP TABLE #TempFragmentation
```

❸ **Identify most-fragmented indexes**

In the listing, a single DMV and two system tables are involved in identifying the most-fragmented indexes; a brief description of each is given in table 3.7.

Table 3.7 DMV/system tables to identify the most-fragmented indexes

DMV/tables	Description
sys.dm_db_index_physical_stats	Contains size and fragmentation information for the data and indexes for tables or views
sys.indexes	Contains details for each index, for example, name and type
sys.objects	Contains details for each object, for example, schema name

By joining the DMV and system tables, we have enough information to identify the most-fragmented indexes across all the databases on the server. The DMV and system tables are joined on their common key columns, object_id and index_id. The system tables sys.objects and sys.indexes are joined on the object_id key. The system table sys.objects is used to provide information about the schema the index relates to.

Again, we use a common pattern to create the temporary table to hold the transient results and another common pattern to loop over all the databases on the server. For more details on these common patterns, see chapter 2.

The first part of the script creates an empty temporary table (named #TempFragmentation) with the required structure of column names and data types ❶. We use the Microsoft-supplied stored procedure, sp_MSForEachDB, to execute a query on each database on the server ❷. The query we execute selects the 20 most-fragmented indexes on each database ❸. We put the results of each execution into the temporary table. Finally, we select the top 20 most-fragmented indexes across all the databases on the server.

Results | Messages

	DatbaseName	SchemaName	TableName	IndexName	Fragmentation %
1	ParisPhil	TWS	SwapsDiaryError	IX_TWS.SwapsDiaryError_AllMandatoryQueriableColu...	97.24
2	ParisPhil	TWS	SwapsDiary	IX_TWS.SwapsDiary_AllMandatoryQueriableColumns	96.85
3	ParisPhil	Load	StagingRisk	idxGDIRisk_RequestId	95.87
4	ParisPhil	dbo	AuthorisationAuditPNL	IX_AuthorisationAuditPNL_OrgId	94.12
5	ParisPhil	Legacy	StagingPnl	idxLegacyPNL_RequestId	92.58
6	ParisPhil	ACBS	LL_RF_FAC_DETAIL	IDX_PORTFOLOI_ID	92.31
7	ParisPhil	Legacy	StagingRisk	idxLegacyRisk_RequestId	91.81
8	ParisPhil	dbo	Bucket	IX_Bucket_3	90.91
9	ParisPhil	dbo	Org	IX_Org	90.91
10	ParisPhil	dbo	BucketGroup	IX_BucketGroup	89.34
11	ParisPhil	Legacy	StagingRiskLog	idxLegacyRiskLog_RequestId	88.93
12	ParisPhil	dbo	OrgNetwork	PK_OrgHierarchy	87.5

Figure 3.8 Output showing the most-fragmented indexes

The query we execute on each database identifies the top 20 most-fragmented indexes using the column avg_fragmentation_in_percent and sorts by this column in descending order.

Note that we ignore any heaps because they aren't indexes. In addition, we're interested in only user-created indexes.

Figure 3.8 contains an example of the type of output for this query.

3.6.2 The impact of fragmented indexes

Having a low level of fragmentation is especially important for those queries that involve ranges, which retrieve data between two points. Fragmentation results in additional work being done. Where possible, you should remove fragmentation.

Typically, Microsoft recommends that indexes that have a fragmentation percentage in excess of 30% be rebuilt. Similarly, indexes with a fragmentation percentage between 10% and 30% should be reorganized.

It's possible to rebuild/reorganize indexes individually from within SSMS, by right-clicking the relevant index and selecting Rebuild. Although this is okay for selected indexes, for a more encompassing approach you should create a script and run the output automatically. You should run this script at regular intervals as part of the regular database housekeeping jobs. Note that you can perform these operations when the database is online, but it may have a negative impact on performance, so be sure to test a small change before it's applied more aggressively.

In chapter 10 I'll provide a script that automatically defragments indexes in an intelligent manner. The script defragments only the indexes whose data has fragmented significantly, and it decides between an index rebuild or a reorganization based on the degree of fragmentation.

It's possible to concentrate your efforts on a given database, table, or index by supplying relevant parameters to the sys.dm_db_index_physical_stats DMV.

We'll now look at which specific indexes are used by a given routine. This provides you with an opportunity to pre-optimize before the query is run.

3.7 Indexes used by a given routine

When you run your SQL queries or batches, some queries are more important than others. If you know which queries (stored procedure or a batch of one or more SQL statements) are your important ones and need to perform optimally, you can pre-optimize the indexes these queries use. If you can identify which indexes are used by a given batch of SQL, you can target these indexes for optimization. This should give you better performance where and when it matters.

When you run a SQL query, information about which indexes it uses and how it uses them (for example, updates, seeks, scans, or lookups) is stored. In addition, information about the number of rows affected by the running code is recorded. Using this information, you can target your performance improvements to those specific indexes, leading to better-performing code.

The purpose of the script described in this section is to identify the name and type of usage of indexes a SQL query uses and then suggest ways in which these targeted indexes can be improved.

Indexes are one of the main tools for improving SQL query performance. But information associated with indexes can become stale over time. Such information includes statistics, degree of logical fragmentation, and the fill factor. We'll discuss how these can be improved later in this section, but first we need to create the script to identify which indexes are used by a given SQL query.

3.7.1 Finding the indexes used by a given routine

If you know which indexes are used by a given routine, you can pre-optimize these indexes before the next time the routine is run, ensuring optimal performance. The script we use to identify the indexes used by a given routine is shown here.

Listing 3.6 Identifying indexes used by a given routine

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT
SchemaName = ss.name
    , TableName = st.name
    , IndexName = ISNULL(si.name, '')
    , IndexType = si.type_desc
    , user_updates = ISNULL(ius.user_updates, 0)
    , user_seeks = ISNULL(ius.user_seeks, 0)
    , user_scans = ISNULL(ius.user_scans, 0)
    , user_lookups = ISNULL(ius.user_lookups, 0)
    , ssi.rowcnt
    , ssi.rowmodctr
    , si.fill_factor
INTO #IndexStatsPre
FROM   sys.dm_db_index_usage_stats ius
```

❶ Get pre-index counter values

```
RIGHT OUTER JOIN sys.indexes si ON  ius.[object_id] = si.[object_id]
          AND ius.index_id = si.index_id
INNER JOIN sys.sysindexes ssi ON si.object_id = ssi.id
          AND si.name = ssi.name
INNER JOIN sys.tables st ON st.[object_id] = si.[object_id]
INNER JOIN sys.schemas ss ON ss.[schema_id] = st.[schema_id]
WHERE ius.database_id = DB_ID()
    AND OBJECTPROPERTY(ius.[object_id], 'IsMsShipped') = 0

SELECT COB, COUNT(*) FROM dbo.request GROUP BY COB
```

❷ **Run routine or native SQL**

```
SELECT
SchemaName = ss.name
    , TableName = st.name
    , IndexName = ISNULL(si.name, '')
    , IndexType = si.type_desc
    , user_updates = ISNULL(ius.user_updates, 0)
    , user_seeks = ISNULL(ius.user_seeks, 0)
    , user_scans = ISNULL(ius.user_scans, 0)
    , user_lookups = ISNULL(ius.user_lookups, 0)
    , ssi.rowcnt
    , ssi.rowmodctr
    , si.fill_factor
INTO #IndexStatsPost
FROM   sys.dm_db_index_usage_stats ius
RIGHT OUTER JOIN sys.indexes si ON ius.[object_id] = si.[object_id]
          AND ius.index_id = si.index_id
INNER JOIN sys.sysindexes ssi ON si.object_id = ssi.id
          AND si.name = ssi.name
INNER JOIN sys.tables st ON st.[object_id] = si.[object_id]
INNER JOIN sys.schemas ss ON ss.[schema_id] = st.[schema_id]
WHERE ius.database_id = DB_ID()
    AND OBJECTPROPERTY(ius.[object_id], 'IsMsShipped') = 0
```

❸ **Get post-index counter values**

```
SELECT
DB_NAME() AS DatabaseName
    , po.[SchemaName]
    , po.[TableName]
    , po.[IndexName]
    , po.[IndexType]
    , po.user_updates - ISNULL(pr.user_updates, 0) AS [User Updates]
    , po.user_seeks - ISNULL(pr.user_seeks, 0) AS [User Seeks]
    , po.user_scans - ISNULL(pr.user_scans, 0) AS [User Scans]
    , po.user_lookups - ISNULL(pr.user_lookups , 0) AS [User Lookups]
    , po.rowcnt - pr.rowcnt AS [Rows Inserted]
    , po.rowmodctr - pr.rowmodctr AS [Updates I/U/D]
    , po.fill_factor
FROM #IndexStatsPost po LEFT OUTER JOIN #IndexStatsPre pr
        ON pr.SchemaName = po.SchemaName
            AND pr.TableName = po.TableName
            AND pr.IndexName = po.IndexName
            AND pr.IndexType = po.IndexType
WHERE ISNULL(pr.user_updates, 0) != po.user_updates
OR          ISNULL(pr.user_seeks, 0) != po.user_seeks
OR          ISNULL(pr.user_scans, 0) != po.user_scans
```

❹ **Determine which index counters have changed**

```
OR          ISNULL(pr.user_lookups, 0) != po.user_lookups
ORDER BY po.[SchemaName], po.[TableName], po.[IndexName];

DROP TABLE #IndexStatsPre
DROP TABLE #IndexStatsPost
```

Here you can see that a single DMV and four system tables are involved in identifying the indexes used by a given routine; a brief description of each is given in table 3.8.

Table 3.8 DMV/system tables to identify the index usage

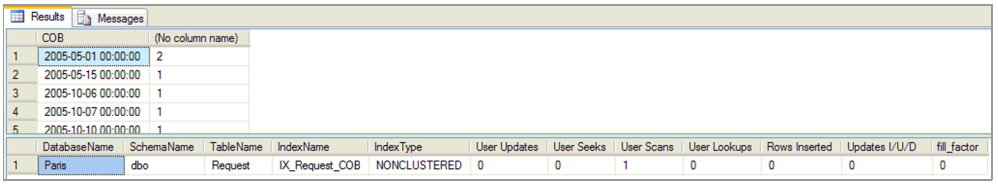
DMV/tables	Description
sys.dm_db_index_usage_stats	Contains details of the different types of index operations, for example, number of updates by user queries
sys.indexes	Contains details for each index, for example, name and type
sys.sysindexes	Contains details of row counts and row changes (since last update statistics run)
sys.tables	Contains details of table objects, for example, name
sys.schemas	Contains details of schema objects, for example, name

By joining the DMV with the system tables, we have enough information to identify the indexes used by a given routine and how they're used. The DMV joins to the system table sys.indexes on their common key columns, object_id and index_id. The other system tables typically provide descriptive information.

The first part of the script stores the current value of various index counters into a temporary table (named #IndexStatsPre) ❶. We then run the SQL query or routine about which we want to discover indexes usage ❷. Next, we store the new value of various index counters into another temporary table (named #IndexStatsPost) ❸. Finally, we compare the temporary tables to determine which indexes have been used, how they've been used—for example, scan, seek, or lookup—and how much they have been used ❹. The results are sorted by schema name, table name, and index name.

The User column usage counts relate to the number of times the index was accessed by the running SQL query, not the number of rows within the index that were accessed. As an example, if a SQL query updates 10 rows (that are part of an index), it will have a User Updates value of 1 (because one UPDATE statement has been run) and will have an Update I/U/D value of 10 (because 10 rows have been modified—inserted, updated, or deleted) if the column updated is the leading column of the index. The Rows Inserted column will have a value of 0 because the number of rows hasn't changed.

Depending on how the query uses the index, it updates the relevant usage counters. A seek is a keyed access and is typically the most efficient method of retrieving a small number of selective rows of data. A scan occurs when an index is examined to



Results | Messages

	COB	(No column name)
1	2005-05-01 00:00:00	2
2	2005-05-15 00:00:00	1
3	2005-10-06 00:00:00	1
4	2005-10-07 00:00:00	1
5	2005-10-10 00:00:00	1

	DatabaseName	SchemaName	TableName	IndexName	IndexType	User Updates	User Seeks	User Scans	User Lookups	Rows Inserted	Updates I/U/D	fill_factor
1	Paris	dbo	Request	IX_Request_COB	NONCLUSTERED	0	0	1	0	0	0	0

Figure 3.9 Output showing the indexes used by a given routine

retrieve a range of rows. A lookup occurs when this index is used to look up data in another index.

The WHERE clause ensures only nonsystem indexes in the current database are examined. The results are sorted by schema name, table name, and then index name.

> **NOTE** When determining which indexes have been used, we use a RIGHT OUTER JOIN between the DMV sys.dm_db_index_usage_stats and the system table sys.indexes; this is necessary because the index may not have been used since the last reboot and may not be present in the DMV. Because of this, when you compare the temporary tables, you need to take into account any potential NULL values.

In addition, in the final step, there is a LEFT OUTER JOIN when calculating which indexes have changed their counter values. This is needed because indexes that may have been used by the routine may not have been used before.

Figure 3.9 shows sample output for this type of query.

3.7.2 The importance of knowing which indexes are used

This script allows you to determine which indexes are used, how they're used, and the number of rows affected when a given stored procedure or batch of SQL code is run. This information can be useful in targeting improvements to your T-SQL with a view to improving its performance.

Each index has a statistics object associated with it. This object includes information about the distribution and density of the index's columns, which is used by the optimizer to determine whether an index is used and how it's used (seek, scan, or lookup). For large tables, updating these statistics can be time consuming, so a smaller sample of rows is typically taken. If you know the specific indexes involved with a query, you can provide a greater sampling size and a better representation of the data, in the same amount of time, compared with a blanket statistics update for the table as a whole.

Large tables have an additional problem concerning statistics. Typically, a table's statistics are updated (automatically) only when 20% of its rows change. For large tables, this means their statistics can be stale for quite a while before they're updated. Using the previous targeted method of improvement should help ensure that the relevant statistics are kept up to date, and this should help improve query performance.

Logical index fragmentation indicates the percentage of entries in the index that are out of sequence. This isn't the same as the page-fullness type of fragmentation. Logical fragmentation has an impact on any order scans that use an index. Where possible, you should remove this fragmentation. You can achieve this by rebuilding or reorganizing the index. You can see the degree of fragmentation by examining the DMV sys.dm_db_index_physical_stats. Typically, if an index has over 30% fragmentation, a rebuild is recommended; if it's between 10% and 30%, a reorganization is recommended.

It's also advisable to ensure that the degree of physical fragmentation isn't excessive, because this will result in greater I/O with a corresponding decrease in performance.

Fill factor describes the number of entries on the page. For an index that's mostly read-only, you'd want the page to be relatively full, so you can get more data per read. Whereas for an index that has many update/inserts, you'd prefer a less-full page to prevent fragmentation that could hurt performance. You can use the output from this script to determine whether an index has mostly read access or not and then set the fill factor accordingly.

The type of index access can provide you with some interesting information. A large number of user lookups could indicate that additional columns should be added to the index via the INCLUDE keyword, because the index is being used to get data from the underlying table. User scans sometimes are indicative of a missing index; inspecting the underlying SQL query code will show whether a scan was intended or whether a more appropriate index needs to be created.

Indexes can cause updates to run more slowly. Because the updates may need to be applied to both the table and its indexes, this can have a significant impact when you make a large number of updates. If the indexes have many updates (see the User Updates column in the output) but few or no reads (see the other User columns), you can investigate whether you'd get better performance by removing the index. In some cases, it may be advisable to disable the index during updating and then enable it afterward. In essence, you should consider the combined total SQL query code that runs on the underlying tables before you consider adding or removing indexes.

If you know that certain indexes are used more than others, you might want to put them on different physical disks; this should give better concurrent access and improved data retrieval times. This is particularly relevant to indexes that are used repeatedly to join the same tables.

> **NOTE** If a table is relatively small, none of its indexes might be used. This is because it's cheaper for the optimizer to get the data directly via the table rather than from an index.

You can also use this script to confirm that certain indexes aren't being used by a SQL query. Investigating why an index isn't being used might result in the SQL query

being rewritten or changing the index definition or even the deletion of an unnecessary index.

It may be worthwhile running the Database Tuning Advisor on the SQL query to ensure that the indexes you have are still appropriate or if additional ones should be added. Additionally, the missing indexes script described earlier can be useful in tracking down missing indexes.

Sometimes the values in the Rows Inserted and Updates I/U/D columns may not match what you might expect in the other User columns. This can have various causes including the following:

- For the Updates I/U/D column to be updated, the SQL query must change an entry in an index where the leading column is updated.
- The combined effect of updates/deletes/inserts needs to be considered.
- If statistics are updated when the utility is run, the Rows Inserted and Updates I/U/D columns get reset (sometimes leading to a negative value).
- Although an update statement may run (so it shows up in the User Updates column), it may not update any rows of data (so it doesn't show up in the Updates I/U/D column).
- A rolled-back transaction seems to affect the Rows Inserted and Updates I/U/D columns, whereas a committed transaction may not.

It might be advisable to run the SQL query under investigation twice because if the indexes haven't been loaded before, they'll give a NULL value in the Rows Inserted and Updates I/U/D columns.

A potential caveat of the method described is that it doesn't limit the index changes to only the SQL query under investigation. If any other code is running concurrently on the same database, its index accesses will also be recorded. One way around this problem is by running the SQL query on a standalone database or at a time when you know nothing else is running. That said, you can turn this caveat into an advantage because you may want to know about all index access on a given database.

3.8 Databases with most missing indexes

It's often the case that you have several different databases running on the same server. A consequence of this is that no matter how optimal your individual database may be, another database on the server, running suboptimally, may affect the server's resources, and this may impact the performance of your database. Remember, CPU, memory, and tempdb are shared across all the databases on the server. Now that you know about the importance of indexes on query performance, it makes sense to report on those databases with the most missing indexes, because they may be indirectly affecting the performance of your database.

It should be possible to amend the other queries in this chapter to provide counts of other aspects of indexing, to give you an indication of databases that perhaps need further attention.

3.8.1 Finding which databases have the most missing indexes

Excessive I/O is often reported as a root cause of poor system performance; often this relates to, and is exacerbated by, missing indexes. Identifying the databases that have the most missing indexes should help alleviate this problem. Running the SQL query given in the following listing will identify the databases with the most missing indexes, ordered by the number of missing indexes.

Listing 3.7 The databases with the most missing indexes

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT
    DB_NAME(database_id) AS DatabaseName
    , COUNT(*) AS [Missing Index Count]
FROM sys.dm_db_missing_index_details
GROUP BY DB_NAME(database_id)
ORDER BY [Missing Index Count] DESC
```

Here a single DMV is used to identify the databases with the most missing indexes; a brief description of it is given in table 3.9.

Table 3.9 DMV to identify the databases with the most missing indexes

DMV	Description
sys.dm_db_missing_index_details	Contains details of the database/schema/table the missing index relates to, together with how the index usage has been identified in queries (for example, equality/inequality)

The sole DMV sys.dm_db_missing_index_details provides you with enough information to determine which databases are missing the most indexes, across all the databases on the server. The query counts the number of missing indexes per database and sorts the results by the number of missing indexes in descending order.

An example of the type of output for this query is shown in figure 3.10.

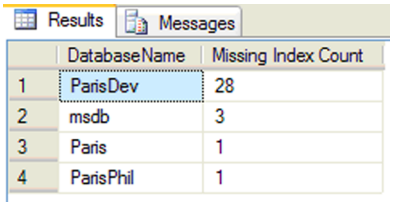


Figure 3.10 Output showing the databases with the most missing indexes

3.8.2 The importance of other databases

As mentioned earlier, your database may be highly optimized, but because you often have multiple databases sharing the same server, a suboptimal database running on the same server may impact the performance of the server and your database.

Often, one thing leads to another. A database that has a high number of missing indexes might also be indicative of a poorly designed database, undertaken by inexperienced staff, using poor-quality hardware. If this is the case, you could use the

databases with the most-missing-indexes script as an indicator of general database quality and to identify those in need of further investigation.

3.9 Completely unused indexes

Earlier I presented a script that identified the most-costly unused indexes. These indexes aren't used for data retrieval, and they're expensive because they may need to be updated when the underlying table is updated. The script given here is different. It identifies indexes that haven't been used at all, neither for retrieval nor for update. These indexes don't have any effect on performance because they aren't used. They do, however, have an effect on the complexity of your database model, because you have additional indexes to understand. This is unnecessary, and you should try to remove them where possible.

> **A word of caution**
>
> The indexes identified as unused should be treated with caution. The algorithm used here to determine if an index is unused compares what's in the system table sys.indexes with the DMV sys.dm_db_index_usage_stats. The latter has entries only if a query accesses the index. It may be that the queries that would use these indexes have not yet been run (since the last SQL Server reboot); perhaps they're run on a quarterly or annual basis.

3.9.1 Finding which indexes aren't used at all

Unused indexes increase the complexity of your database model, resulting in longer and more complex analysis when you undertake maintenance work. Running the following SQL script will identify all the indexes that are unused on your server instance.

Listing 3.8 Indexes that aren't used at all

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT
    DB_NAME() AS DatbaseName
    , SCHEMA_NAME(O.Schema_ID) AS SchemaName
    , OBJECT_NAME(I.object_id) AS TableName
    , I.name AS IndexName
INTO #TempNeverUsedIndexes
FROM sys.indexes I INNER JOIN sys.objects O ON I.object_id = O.object_id
WHERE 1=2

EXEC sp_MSForEachDB 'USE [?];
INSERT INTO #TempNeverUsedIndexes
SELECT
    DB_NAME() AS DatbaseName
    , SCHEMA_NAME(O.Schema_ID) AS SchemaName
    , OBJECT_NAME(I.object_id) AS TableName
    , I.NAME AS IndexName
FROM sys.indexes I INNER JOIN sys.objects O ON I.object_id = O.object_id
```

1 Temp table to hold results

2 Loop around all databases

```
LEFT OUTER JOIN sys.dm_db_index_usage_stats S ON S.object_id = I.object_id
        AND I.index_id = S.index_id
        AND DATABASE_ID = DB_ID()
WHERE OBJECTPROPERTY(O.object_id,''IsMsShipped'') = 0
  AND I.name IS NOT NULL
  AND S.object_id IS NULL'

SELECT * FROM #TempNeverUsedIndexes
ORDER BY DatbaseName, SchemaName, TableName, IndexName

DROP TABLE #TempNeverUsedIndexes
```

❸ **Identify unused indexes**

In the listing, you can see that a single DMV and two system tables are involved in identifying the databases that are unused. They're briefly described in table 3.10.

Table 3.10 DMV/tables to identify unused indexes

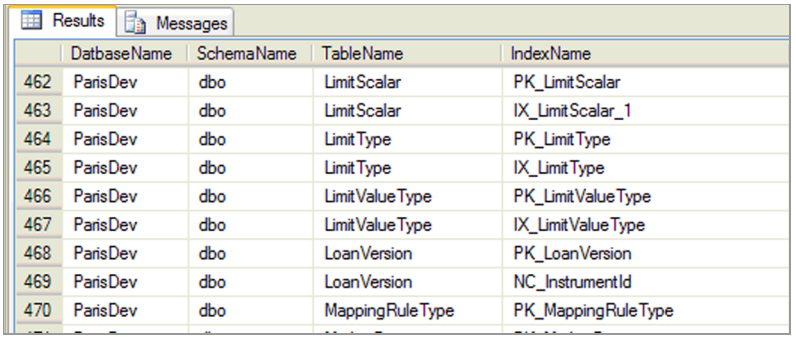
DMV/tables	Description
sys.dm_db_index_usage_stats	Contains details of the different types of index operations, for example, number of updates by user queries
sys.indexes	Contains details for each index, for example, name and type
sys.objects	Contains details for each object, for example, schema name

The first part of the script creates an empty temporary table (named #TempNeverUsedIndexes) with the required structure of column names and data types ❶. We use the Microsoft-supplied stored procedure, sp_MSForEachDB, to execute a query on each database on the server ❷. The query we execute selects all the unused indexes on each database. We put the results of each execution into the temporary table. Finally, we select the unused indexes across all the databases on the server ❸.

The DMV sys.dm_db_index_usage_stats is populated with details of any indexes that have been used in running our queries. If an index has not been used, it's not present in this DMV. The system table sys.indexes contains details of all the indexes present on a given database. The system table sys.objects is used to provide information about the schema the index relates to. You can determine which indexes are unused by comparing what's in the system table sys.indexes but not in the DMV sys.dm_db_index_usage_stats. We use the system table sys.objects to display the schema name of the index. The results are sorted by database name, schema name, table name, and index name.

> **NOTE** When determining which indexes haven't been used, there's a LEFT OUTER JOIN between the system table sys.indexes and the DMV sys.dm_db_index_usage_stats. This is necessary because the index may not have been used since the last reboot and may not be present in the DMV. Checking for a NULL object_id in the DMV will ensure you obtain all the unused indexes.

Sample output for this query is shown in figure 3.11.



Results | Messages

	DatbaseName	SchemaName	TableName	IndexName
462	ParisDev	dbo	LimitScalar	PK_LimitScalar
463	ParisDev	dbo	LimitScalar	IX_LimitScalar_1
464	ParisDev	dbo	LimitType	PK_LimitType
465	ParisDev	dbo	LimitType	IX_LimitType
466	ParisDev	dbo	LimitValueType	PK_LimitValueType
467	ParisDev	dbo	LimitValueType	IX_LimitValueType
468	ParisDev	dbo	LoanVersion	PK_LoanVersion
469	ParisDev	dbo	LoanVersion	NC_InstrumentId
470	ParisDev	dbo	MappingRuleType	PK_MappingRuleType

Figure 3.11 Output showing indexes that aren't used at all

3.9.2 The importance of unused indexes

The indexes identified as unused here should be treated with caution. It may be that the queries that would use these indexes haven't yet been run (since the last SQL Server reboot) and don't have an entry in the DMV sys.dm_db_index_usage_stats. As the time since the last reboot increases and the amount of data in the DMV increases, you can be more confident that the indexes aren't required. As a compromise, you could disable the identified used indexes, and if they subsequently appear in the list of missing indexes, you could reinstate them.

As noted earlier, these indexes haven't been used for either retrieval or update, so they have no effect on query performance. They do, however, have an effect on schema complexity. For example, you need to take them into account when analyzing the impact of changes. They result in unnecessary work and should be removed.

In chapter 10, I'll provide a script that automatically disables or deletes indexes that aren't used at all.

One of the major factors that affects whether an index is used or not is the index's statistics; this is discussed next.

3.10 Your statistics

Queries typically use indexes for WHERE clauses and JOIN conditions. Whether or not an index is used and how it's used are typically determined by the statistics on the columns in the index. If you know how often a given data value is likely to occur and its distribution in relation to other data values, you can provide an estimate to the optimizer that's used to determine which indexes are used by queries.

In many ways, statistical information is at least as important as the indexes it relates to, and so you should ensure it's up to date and representative of the underlying index data.

Statistics are typically automatically updated when 20% of the rows in a table have changed since the statistics were last updated. For small- to medium-size tables, the frequency of that statistics update may be adequate. But for larger tables, this

automatic update may be insufficient. I've experienced many occasions where a query has taken many minutes to run only to be canceled because of its bad performance. Updating the relevant statistics has allowed the same query to subsequently run in a few seconds.

Although this script doesn't involve DMVs, I've included it here because knowledge of the current state of index statistics can have a profound effect on the use of indexes and the performance of SQL queries.

3.10.1 Finding the state of your statistics

Up-to-date statistics help ensure that the most appropriate index is chosen to obtain the underlying data. They also help ensure that the correct index access mechanism is chosen, for example, seek or lookup. Running the SQL script given in the following listing will identify the current state of your statistics.

Listing 3.9 What is the state of your statistics?

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT
    ss.name AS SchemaName
    , st.name AS TableName
    , s.name AS IndexName
    , STATS_DATE(s.id,s.indid) AS 'Statistics Last Updated'
    , s.rowcnt AS 'Row Count'
    , s.rowmodctr AS 'Number Of Changes'
    , CAST((CAST(s.rowmodctr AS DECIMAL(28,8))/CAST(s.rowcnt AS
     DECIMAL(28,2)) * 100.0)
                              AS DECIMAL(28,2)) AS '% Rows Changed'
FROM sys.sysindexes s
INNER JOIN sys.tables st ON st.[object_id] = s.[id]
INNER JOIN sys.schemas ss ON ss.[schema_id] = st.[schema_id]
WHERE s.id > 100
  AND s.indid > 0
  AND s.rowcnt >= 500
ORDER BY SchemaName, TableName, IndexName
```

In the listing, you can see that three system tables are involved in identifying the current state of index statistics; a brief description of each is given in table 3.11.

Table 3.11 The state of your statistics

Tables	Description
sys.indexes	Contains details for each index, for example, name, type, row count, number of rows changed since statistics last updated
sys.tables	Contains table information, for example, name
sys.schemas	Contains details of schema objects, for example, name

The joining of the system tables provides enough information to identify when the index statistics were last updated and the percentage of rows that have changed since the last update of the statistics. The system table sys.sysindexes is joined to sys.tables on their key column id/object_id, and sys.tables is joined to sys.schemas on the schema_id column.

The script retrieves schema name, table name, index name, current row count, and number of changes. It calculates the number of rows changed as a percentage and uses the SQL function STATS_DATE to determine when the statistics for the index were last updated.

You filter out indexes with an index id (indId) of 0 because these aren't indexes; they're heaps. You also filter out indexes with fewer than 500 rows, because statistics are more important for larger tables. You also filter out system objects. The results are sorted by schema name, table name, and index name.

An example of the type of output for this query is shown in figure 3.12.

3.10.2 The importance of statistics

When the optimizer looks at how it will fulfill a user's query, it inspects the JOIN and WHERE clause and determines whether an index should be used. Statistics have an impact on both whether an index is used and how it would be used. Statistics describe the distribution and density of data values. Knowing the statistics for a given index column, you can estimate the probability of a given data value being used in a WHERE clause. Knowing this, the optimizer can choose a relevant index and decide how that index should be used, be it a seek, scan, or lookup.

You can see that statistics can have a profound effect on the performance of a query. Unfortunately, statistics can become stale; this is especially true of larger tables. An index's statistics tend to be updated automatically when 20% of its rows have changed. For large tables, this might take a considerable time, during which queries may run more slowly because of having stale statistics.

Results | Messages

	SchemaName	TableName	IndexName	Statistics Last Updated	Row Count	Number Of Changes	% Rows Changed
1	ACBS	LL_RF_FAC_DE...	IDX_PORTFOLOI_ID	2010-01-26 10:51:04.853	11140	0	0.00
2	ACBS	LL_RF_FAC_DE...	IXD_UNQ_LL_RF_F...	2010-01-26 10:51:10.400	11140	0	0.00
3	ACBS	LL_RF_FAC_LIM...	IXD_UNQ_LL_RF_F...	2010-01-26 10:51:10.613	11170	0	0.00
4	ACBS	LL_RF_LOAN_I...	IDX_CLUST_LL_RF...	2010-01-26 10:51:11.730	17159	0	0.00
5	ACBS	LL_RF_LOAN_LI...	IXD_UNQ_LL_RF_L...	2010-01-26 10:51:11.423	20532	0	0.00
6	ACBS	StagingPNL	idxRequestId_ASP	2010-01-26 14:01:57.010	633975	0	0.00
7	ACBS	StagingPNLLog	idxRequestId_ASPL	2010-01-26 14:01:55.503	633975	0	0.00
8	dbo	ACBSLoan	PK_ACBSLoan	2010-01-25 18:00:06.420	1553	0	0.00
9	dbo	AuthorisationAudi...	IX_AuthorisationAudit...	2010-01-25 18:00:06.640	8338	0	0.00
10	dbo	AuthorisationAudi...	PK_PNLAuthorisatio...	2010-01-25 18:00:06.467	8338	0	0.00
11	dbo	BondVersion	IX_BondVersion_2	2010-01-25 18:00:06.793	96146	15815	16.45
12	dbo	BondVersion	PK_BondVersion	2010-01-25 18:00:06.717	96146	15815	16.45
13	dbo	Bucket	IX_Bucket	2010-01-25 18:00:13.730	1925	10	0.52

Figure 3.12 Output showing the current state of your statistics

To determine whether the statistics should be updated, you should look at the column % Rows Changed together with the Statistics Last Updated column. For a large table, perhaps the statistics need to be updated on a daily basis. I've found this to be the case where the current date is part of the index key.

You could use the output to automatically update the statistics of individual indexes. Because you have the individual index names, you could target your updates to the relevant indexes. This will allow you to have a faster statistics update or a higher sampling percentage within the same time period as a blanket table update.

In chapter 10 I'll provide a script that automatically updates the index statistics in an intelligent manner. The script updates only the statistics of the indexes whose data has changed and does so using an intelligent sampling algorithm.

We've discussed various aspects of indexes in detail in this chapter. One of the themes that comes through is the conflict between the usefulness of indexes in retrieving data and their cost in terms of unnecessary updates. We'll examine this theme next.

3.11 A holistic approach to managing indexes

Indexes have a significant contribution to make to the discussion relating to balancing transactional and reporting systems. Typically, queries in transactional systems update a small number of rows relatively quickly. If they need to update additional indexes (required for reporting), the update will add time to the query duration, transaction time, and resource locks. By contrast, queries in reporting systems typically retrieve a large number of rows, run for long time periods, and often require many indexes. If both of these conflicting systems are present on the same database, you should try to balance these two contradictory requirements.

Ideally, you'd have the two systems (transactional and reporting systems) on different databases and preferably on different servers. The transactional system could feed the reporting system with periodic updates.

If system separation isn't feasible, you should take steps to minimize the adverse effects of indexes. This would include ensuring that unused or high-maintenance indexes are removed (see the scripts given earlier for this). In addition, indexes that are used heavily or for important queries should be optimized, with reference to their statistics, fill factor, and fragmentation (again, you have scripts to discover these things).

Another solution might be to disable reporting indexes during the transactional processing and reenable them during the reporting processing. You could also see how indexes are used in terms of reads/writes and compare this with the number of I/O reads/writes.

Databases tend to have a bias toward either reporting or transactional processing. Even with transactional systems, the database typically has more reads than writes, often by a factor of at least 5 or 10, so you need to be careful not to overestimate the cost of index updates (unless you have lots of indexes on a table!). Database read

performance tends to be inversely proportional to the index fill factor, so a fill factor of 50% means reads are twice as slow as when the fill factor is 100.

3.12 Summary

Indexes are critical to query performance. In this chapter we've discussed the different types of indexes along with the different index access mechanisms.

I've provided a variety of scripts that use DMVs to identify indexes that may be suboptimal, unnecessary, or even missing. We've discussed several factors that relate to indexes that can be used to optimize the identified indexes, resulting in faster-performing queries.

Indexes are a vital element in determining how a table's data is accessed, thus impacting query performance. Having looked at various useful aspects of indexes, we'll now move on to looking at the execution of SQL queries in the next chapter.

SQL SERVER ADMINISTRATION

SQL Server DMVs IN ACTION

Ian W. Stirk

Free ebook
SEE INSERT

Every action in SQL Server leaves a set of tiny footprints. SQL Server records that valuable data and makes it visible through Dynamic Management Views, or DMVs. You can use this incredibly detailed information to significantly improve the performance of your queries and better understand what's going on inside your SQL Server system.

SQL Server DMVs in Action shows you how to obtain, interpret, and act on the information captured by DMVs to keep your system in top shape. The over 100 code examples help you master DMVs and give you an instantly reusable SQL library. You'll also learn to use Dynamic Management Functions (DMFs), which provide further details that enable you to improve your system's performance and health.

What's Inside

- Many practical solutions
- How to correct missing indexes
- What's slowing down your queries
- What's compromising concurrency
- Much more

This book is written for DBAs and developers.

Ian Stirk is a freelance consultant based in London. He's an expert in SQL Server performance and a fierce advocate for DMVs.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/SQLServerDMVsinAction

"Essential reference for SQL Server Administrators."
—Dave Corun, Avanade

"Arm yourself with an arsenal of DMV knowledge."
—Tariq Ahmed
Amcom Technology

"Lifts the hood on SQL Server performance."
—Richard Siddaway, Serco

"The examples alone are worth *twice* the price of the book!"
—Nikander and Margriet Bruggeman
Lois & Clark IT Services

ISBN 13: 978-1-935182-73-3
ISBN 10: 1-935182-73-0
56499
9 781935 182733

MANNING $64.99 / Can $74.99 [INCLUDING eBOOK]