

General

Tuning Options

☒ Limit tuning time

Stop at: Sunday, November 26, 20175:48 AM

Advanced Options...

Physical Design Structures (PDS) to use in database

☐ Indexes and indexed views

☒ Indexes

☐ Evaluate utilization of existing PDS only

☐ Indexed views

☐ Nonclustered indexes

☐ Include filtered indexes

☐ Recommend columnstore indexes

Partitioning strategy to employ

☒ No partitioning

☐ Aligned partitioning

☐ Full partitioning

Physical Design Structures (PDS) to keep in database

☐ Do not keep any existing PDS

☒ Keep all existing PDS

☐ Keep aligned partitioning

☐ Keep indexes only

☐ Keep clustered indexes only

Description

!

Database Engine Tuning Advisor will recommend clustered and nonclustered indexes to improve performance of your workload. No partitioning strategies will be considered. Newly recommended structures will be un-partitioned. All existing structures will remain intact in the database at the conclusion of the tuning process.

Figure 10-3. Defining options in the Database Engine Tuning Advisor

You define the length of time you want the Database Engine Tuning Advisor to run by selecting Limit Tuning Time and then defining a date and time for the tuning to stop. The longer the Database Engine Tuning Advisor runs, the better recommendations it should make. You pick the type of physical design structures to be considered for creation by the Database Engine Tuning Advisor, and you can also set the partitioning strategy so that the Tuning Advisor knows whether it should consider partitioning the tables and indexes as part of the analysis. Just remember, partitioning is foremost a data management tool, not a performance tuning mechanism. Partitioning may not necessarily be a desirable outcome if your data and structures don't warrant it. Finally, you can define the physical design structures that you want left alone within the database. Changing these options will narrow or widen the choices that the Database Engine Tuning Advisor can make to improve performance. You can optionally include filtered indexes, and the Database Engine Tuning Advisor can recommend columnstore indexes.

You can click the Advanced Options button to see even more options, as shown in Figure 10-4.

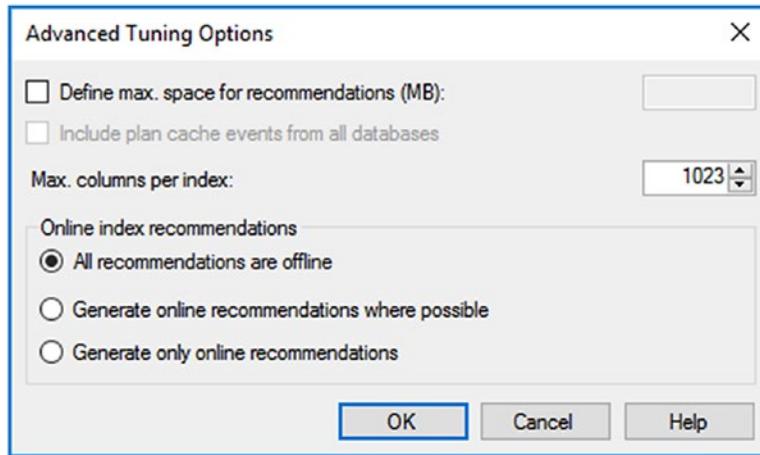


Figure 10-4. *Advanced Tuning Options dialog box*

This dialog box allows you to limit the space of the recommendations and the number of columns that can be included in an index. You decide whether you want to include plan cache events from every database on the system. Finally, you can define whether the new indexes or changes in indexes are done as an online or offline index operation.

Once you’ve appropriately defined all of these settings, you can start the Database Engine Tuning Advisor by clicking the Start Analysis button. The sessions created are kept in the msdb database for any server instance that you run the Database Engine Tuning Advisor against. It displays details about what is being analyzed and the progress that was made, which you can see in Figure 10-5.

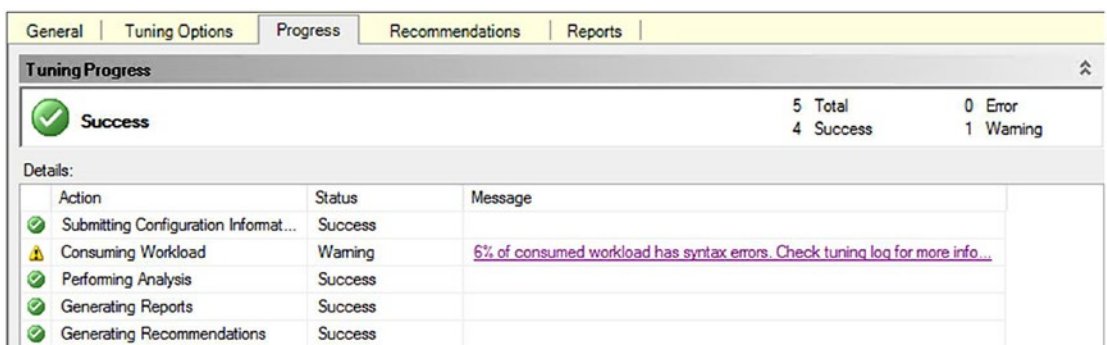


Figure 10-5. *Tuning progress*

You'll see more detailed examples of the progress displayed in the example analysis in the next session.

After the analysis completes, you'll get a list of recommendations (visible in Figure 10-6), and a number of reports become available. Table 10-1 describes the reports.

Table 10-1. *Database Engine Tuning Advisor Reports*

| Report Name | Report Description |
|---------------------------------|--|
| Column Access | Lists the columns and tables referenced in the workload |
| Database Access | Lists each database referenced in the workload and percentage of workload statements for each database |
| Event Frequency | Lists all events in the workload ordered by frequency of occurrence |
| Index Detail (Current) | Defines indexes and their properties referenced by the workload |
| Index Detail (Recommended) | Is the same as the Index Detail (Current) report but shows the information about the indexes recommended by the Database Engine Tuning Advisor |
| Index Usage (Current) | Lists the indexes and the percentage of their use referenced by the workload |
| Index Usage (Recommended) | Is the same as the Index Usage (Current) report but from the recommended indexes |
| Statement Cost | Lists the performance improvements for each statement if the recommendations are implemented |
| Statement Cost Range | Breaks down the cost improvements by percentiles to show how much benefit you can achieve for any given set of changes; these costs are estimated values provided by the optimizer |
| Statement Detail | Lists the statements in the workload, their cost, and the reduced cost if the recommendations are implemented |
| Statement-to-Index Relationship | Lists the indexes referenced by individual statements; current and recommended versions of the report are available |
| Table Access | Lists the tables referenced by the workload |
| View-to-Table Relationship | Lists the tables referenced by materialized views |
| Workload Analysis | Gives details about the workload, including the number of statements, the number of statements whose cost is decreased, and the number where the cost remains the same |

Database Engine Tuning Advisor Examples

The best way to learn how to use the Database Engine Tuning Advisor is to use it. It's not a terribly difficult tool to master, so I recommend opening it and getting started.

Tuning a Query

You can use the Database Engine Tuning Advisor to recommend indexes for a complete database by using a workload that fairly represents all SQL activities. You can also use it to recommend indexes for a set of problematic queries.

To learn how you can use the Database Engine Tuning Advisor to get index recommendations on a set of problematic queries, say you have a simple query that is called rather frequently. Because of the frequency, you want a quick turnaround for some tuning. This is the query:

```
SELECT soh.DueDate,  
       soh.CustomerID,  
       soh.Status  
FROM Sales.SalesOrderHeader AS soh  
WHERE soh.DueDate  
BETWEEN '1/1/2008' AND '2/1/2008';
```

To analyze the query, right-click it in the query window and select Analyze Query in the Database Engine Tuning Advisor. The advisor opens with a window where you can change the session name to something meaningful. In this case, I chose Report Query Round 1 – 1/16/2014. The database and tables don't need to be edited. The first tab, General, will look like Figure 10-6 when you're done.

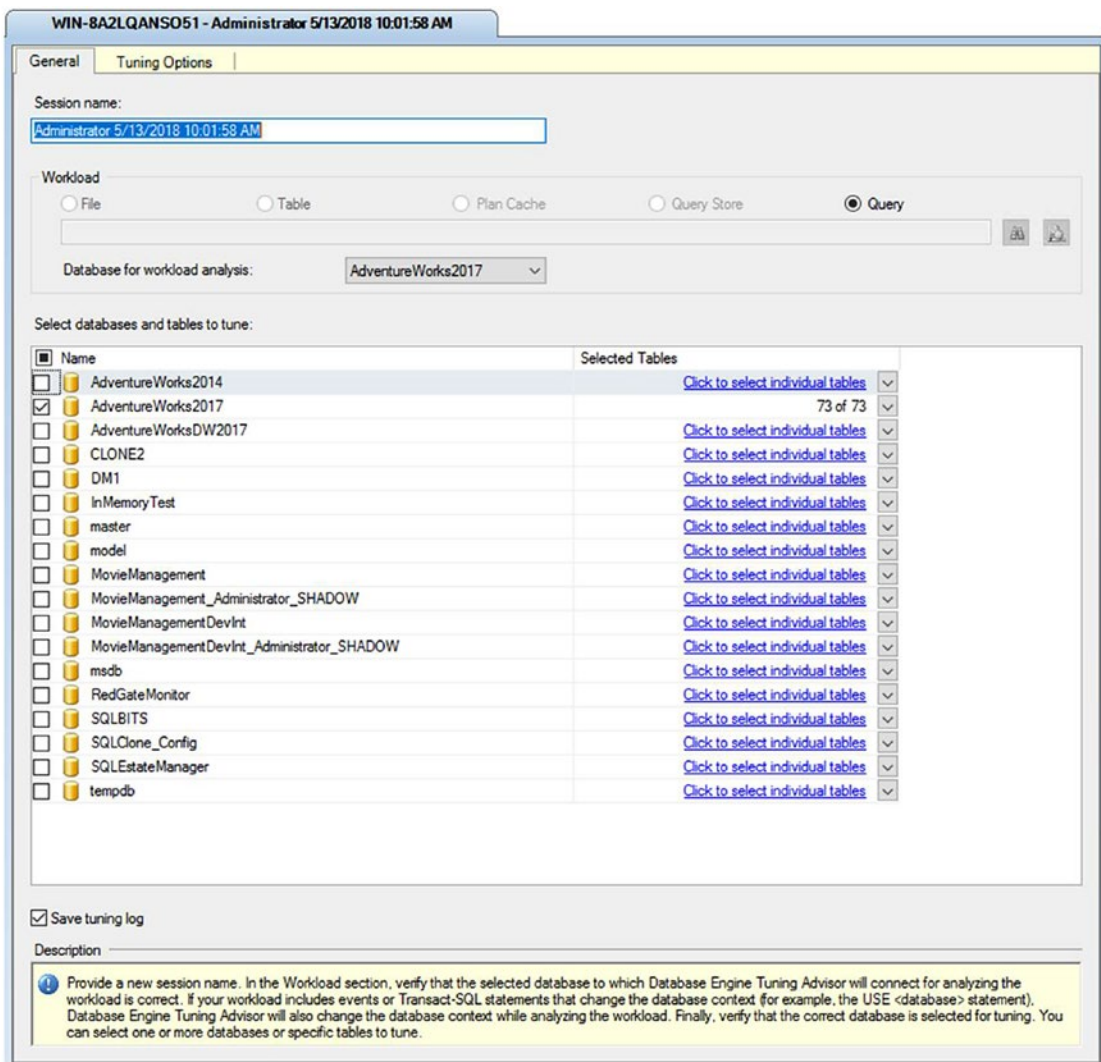
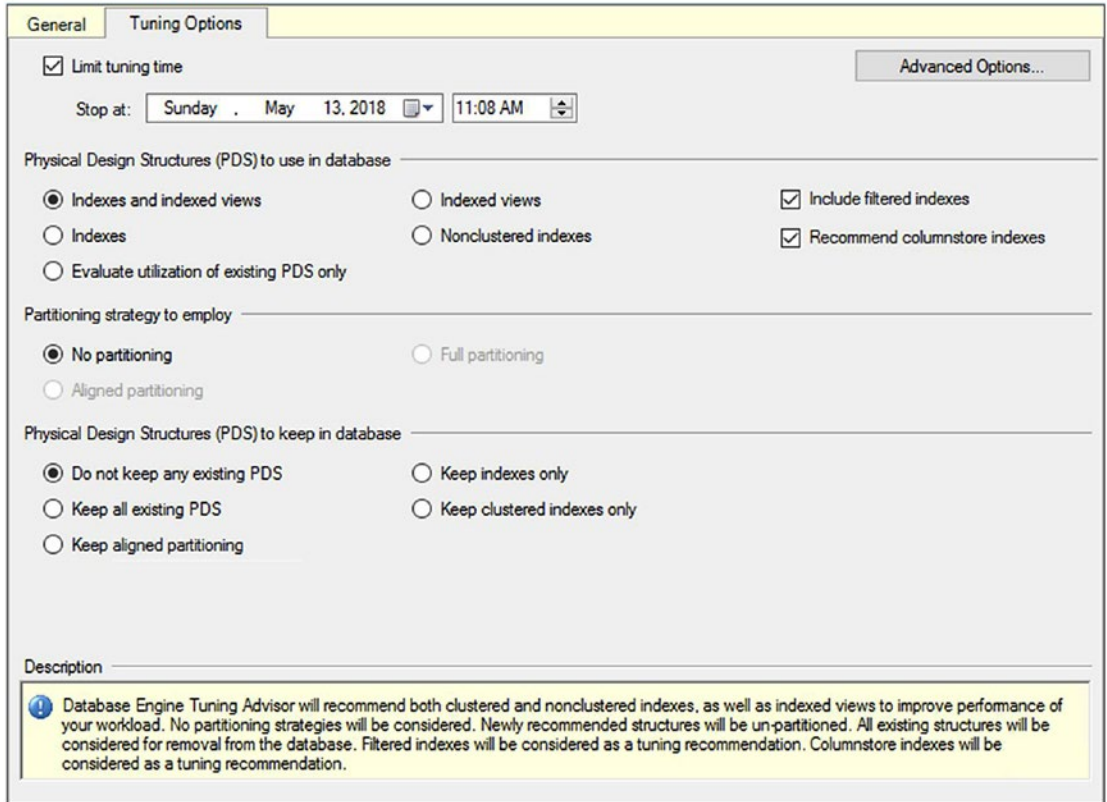


Figure 10-6. Query tuning general settings

Because this query is important and tuning it is extremely critical to the business, I’m going to change some settings on the Tuning Options tab to try to maximize the possible suggestions. For the purposes of the example, I’m going to let the Database Engine Tuning Advisor run for the default of one hour, but for bigger loads or more complex queries, you might want to consider giving the system more time. I’m going to select the Include Filtered Indexes check box so that if a filtered index will help, it can be considered. I’m also going to let it recommend columnstore indexes. Finally, I’m going

to allow the Database Engine Tuning Advisor to come up with structural changes if it can find any that will help by switching from Keep All Existing PDS to Do Not Keep Any Existing PDS. Once completed, the Tuning Options tab will look like Figure 10-7.



The screenshot shows the 'Tuning Options' tab in the Database Engine Tuning Advisor. The 'Limit tuning time' checkbox is checked, and the 'Stop at' field is set to Sunday, May 13, 2018 at 11:08 AM. The 'Physical Design Structures (PDS) to use in database' section has 'Indexes and indexed views' selected, with 'Include filtered indexes' and 'Recommend columnstore indexes' checked. The 'Partitioning strategy to employ' section has 'No partitioning' selected. The 'Physical Design Structures (PDS) to keep in database' section has 'Do not keep any existing PDS' selected. A description box at the bottom states: 'Database Engine Tuning Advisor will recommend both clustered and nonclustered indexes, as well as indexed views to improve performance of your workload. No partitioning strategies will be considered. Newly recommended structures will be un-partitioned. All existing structures will be considered for removal from the database. Filtered indexes will be considered as a tuning recommendation. Columnstore indexes will be considered as a tuning recommendation.'

Figure 10-7. *Tuning Options tab adjusted*

Notice that the description at the bottom of the screen changes as you change the definitions in the selections made above. After starting the analysis, the progress screen should appear. Although the settings were for one hour of evaluations, it took only about a minute for the DTA to evaluate this query. The initial recommendations were not a good set of choices. As you can see in Figure 10-8, the Database Engine Tuning Advisor has recommended dropping a huge swath of indexes in the database. This is not the type of recommendation you want when running the tool.

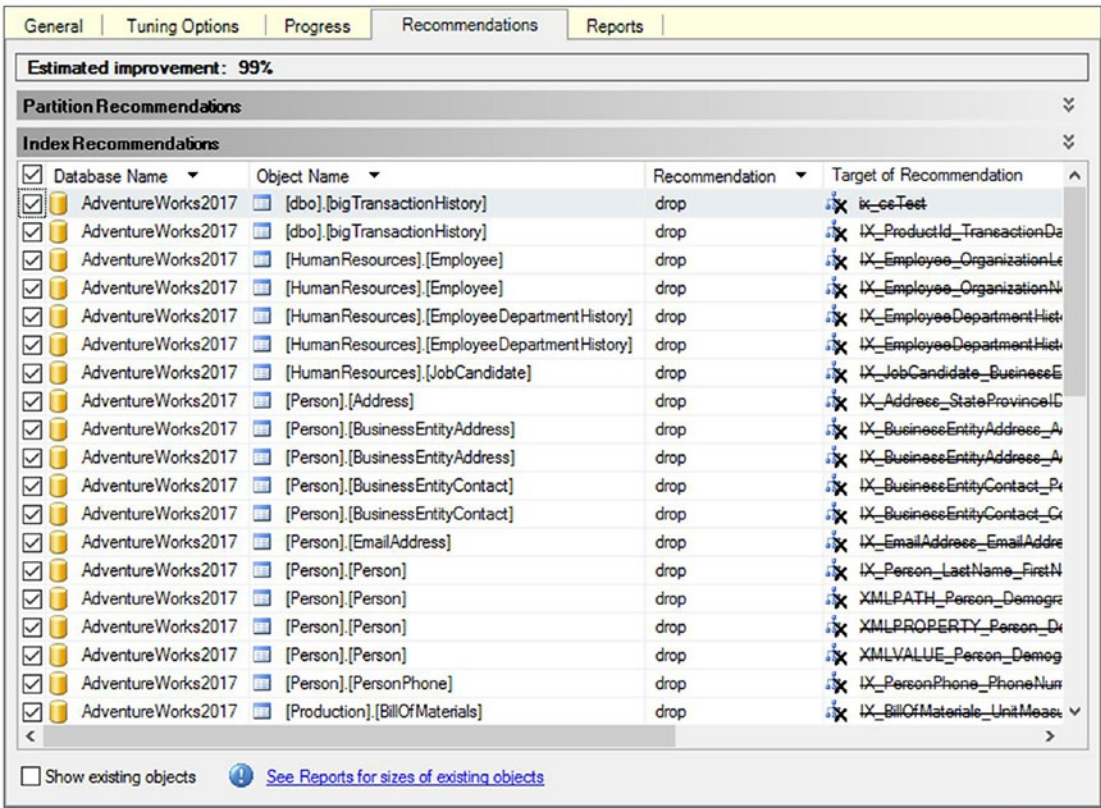


Figure 10-8. Query tuning initial recommendations

The Database Engine Tuning Advisor assumes that the load being tested is the full load of the database. For every test, every time. If the test you are running is not your representative workload, you could have serious issues with the suggested changes.

If there are indexes not being used, then they should be removed. This is a best practice and one that should be implemented on any database. However, in this case, this is a single query, not a full load of the system. To see whether the advisor can come up with a more meaningful set of recommendations, you must start a new session.

This time, I'll adjust the options so that the Database Engine Tuning Advisor will not be able to drop any of the existing structure. This is set on the Tuning Options tab (shown earlier in Figure 10-7). There I'll change the Physical Design Structure (PDS) to Keep in Database setting from Do Not Keep Any Existing PDS to Keep All Existing PDS. I'll keep

the running time the same because the evaluation worked well within the time frame. After running the Database Engine Tuning Advisor again, it finishes in less than a minute and displays the recommendations shown in Figure 10-9.

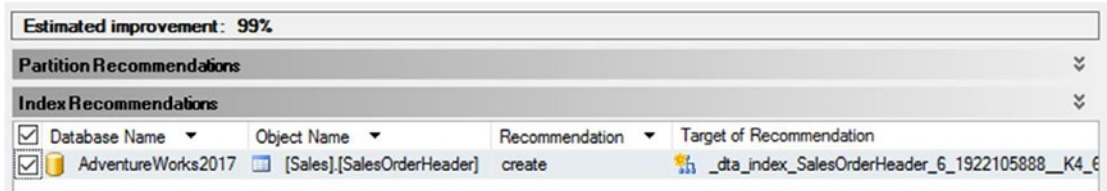


Figure 10-9. Query tuning recommendations

The first time through, the Database Engine Tuning Advisor suggested dropping most of the indexes on the tables being tested and a bunch of the related tables. This time it suggests creating a covering index on the columns referenced in the query. As outlined in Chapter 9, a covering index can be one of the best-performing methods of retrieving data. The Database Engine Tuning Advisor was able to recognize that an index with all the columns referenced by the query, a covering index, would perform best.

Once you've received a recommendation, you should closely examine the proposed T-SQL command. The suggestions are not always helpful, so you need to evaluate and test them to be sure. Assuming the examined recommendation looks good, you'll want to apply it. Select **Actions ► Evaluate Recommendations**. This opens a new Database Engine Tuning Advisor session and allows you to evaluate whether the recommendations will work using the same measures that made the recommendations in the first place. All of this is to validate that the original recommendation has the effect that it claims it will have. The new session looks just like a regular evaluation report. If you're still happy with the recommendations, select **Actions ► Apply Recommendation**. This opens a dialog box that allows you to apply the recommendation immediately or schedule the application (see Figure 10-10).

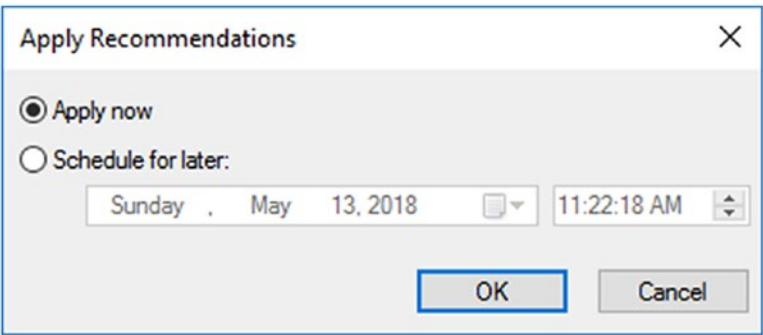


Figure 10-10. *Apply Recommendations dialog box*

If you click the OK button, the Database Engine Tuning Advisor will apply the index to the database where you’ve been testing queries (see Figure 10-11).

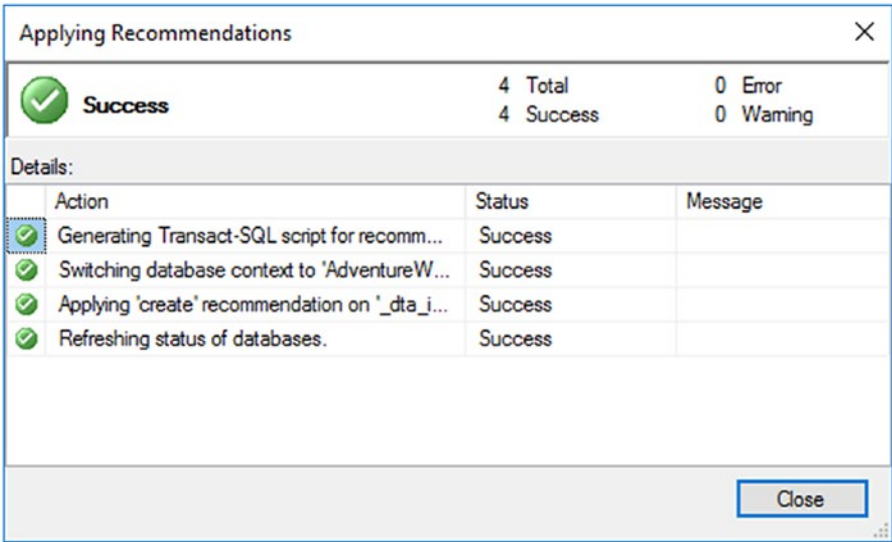


Figure 10-11. *A successful tuning session applied*

After you generate recommendations, you may want to, instead of applying them on the spot, save the T-SQL statements to a file and accumulate a series of changes for release to your production environment during scheduled deployment windows. Also, just taking the defaults, you’ll end up with a lot of indexes named something like this: `_dta_index_SalesOrderHeader_5_1266103551__K4_6_11`. That’s not terribly clear, so saving the changes to T-SQL will also allow you to make your changes more human

readable. Remember that applying indexes to tables, especially large tables, can cause a performance impact to processes actively running on the system while the index is being created.

Although getting index suggestions one at a time is nice, it would be better to be able to get large swaths of the database checked all at once. That's where tuning a trace workload comes in.

Tuning a Trace Workload

Capturing a trace from the real-world queries that are running against a production server is a way to feed meaningful data to the Database Engine Tuning Advisor. (Capturing traces will be covered in Chapter 18.) The easiest way to define a trace for use in the Database Engine Tuning Advisor is to implement the trace using the Tuning template. Start the trace on the system you need to tune. I generated an artificial load by running queries in a loop from the PowerShell `sqlps.exe` command prompt. This is the PowerShell command prompt with the SQL Server configuration settings. It gets installed with SQL Server.

To find something interesting, I'm going to create one stored procedure with an obvious tuning issue.

```
CREATE PROCEDURE dbo.uspProductSize
AS
SELECT  p.ProductID,
        p.Size
FROM    Production.Product AS p
WHERE   p.Size = '62';
```

Here is the simple PowerShell script I used. You'll need to adjust the connection string for your environment. After you have downloaded the file to a location, you'll be able to run it by simply referencing the file and the full path through the command prompt. You may run into security issues since this is an unsigned, raw script. Follow the help guidance provided in that error message if you need to (`queryload.ps1`).

```

$SqlConnection = New-Object System.Data.SqlClient.SqlConnection
$SqlConnection.ConnectionString = 'Server=WIN-8A2LQANSO51;Database=AdventureWorks2017;trusted_connection=true'

# Load Product data
$ProdCmd = New-Object System.Data.SqlClient.SqlCommand
$ProdCmd.CommandText = "SELECT ProductID FROM Production.Product"
$ProdCmd.Connection = $SqlConnection
$SqlAdapter = New-Object System.Data.SqlClient.SqlDataAdapter
$SqlAdapter.SelectCommand = $ProdCmd
$ProdDataSet = New-Object System.Data.DataSet
$SqlAdapter.Fill($ProdDataSet)

# Set up the procedure to be run
$WhereCmd = New-Object System.Data.SqlClient.SqlCommand
$WhereCmd.CommandText = "dbo.uspGetWhereUsedProductID @StartProductID = @
ProductId, @CheckDate=NULL"
$WhereCmd.Parameters.Add("@ProductID",[System.Data.SqlDbType]"Int")
$WhereCmd.Connection = $SqlConnection

# And another one
$BomCmd = New-Object System.Data.SqlClient.SqlCommand
$BomCmd.CommandText = "dbo.uspGetBillOfMaterials @StartProductID = @
ProductId, @CheckDate=NULL"
$BomCmd.Parameters.Add("@ProductID",[System.Data.SqlDbType]"Int")
$BomCmd.Connection = $SqlConnection

# Bad Query
$BadQuerycmd = New-Object System.Data.SqlClient.SqlCommand
$BadQuerycmd.CommandText = "dbo.uspProductSize"
$BadQuerycmd.Connection = $SqlConnection

while(1 -ne 0)
{
    $RefID = $row[0]
    $SqlConnection.Open()
    $BadQuerycmd.ExecuteNonQuery() | Out-Null
    $SqlConnection.Close()
}

```

```

foreach($row in $ProdDataSet.Tables[0])
{
    $SqlConnection.Open()
    $BomCmd.Parameters["@ProductID"].Value = $ProductId
    $BomCmd.ExecuteNonQuery() | Out-Null
    $SqlConnection.Close()

    $SqlConnection.Open()
    $ProductId = $row[0]
    $WhereCmd.Parameters["@ProductID"].Value = $ProductId
    $WhereCmd.ExecuteNonQuery() | Out-Null
    $SqlConnection.Close()
}
}

```

Note For more information on PowerShell, check out *PowerShell in a Month of Lunches* by Don Jones and Jeffrey Hicks (Manning, 2016).

Once you've created the trace file, open the Database Engine Tuning Advisor. It defaults to a file type under the Workload section, so you'll only have to browse to the trace file location. As before, you'll want to select the AdventureWorks2017 database as the database for workload analysis from the drop-down list. To limit the suggestions, also select AdventureWorks2012 from the list of databases at the bottom of the screen. Set the appropriate tuning options and start the analysis. This time, it will take more than a minute to run (see Figure 10-12).

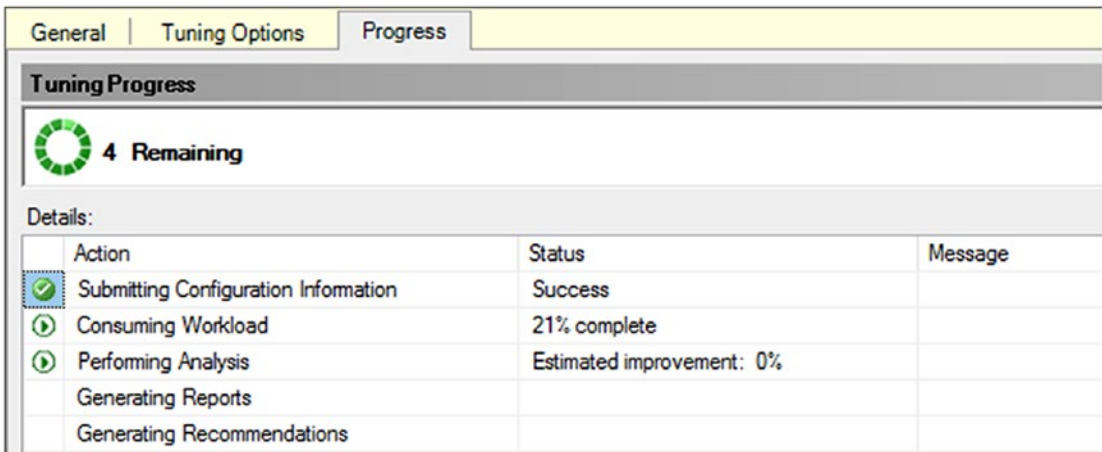


Figure 10-12. Database tuning engine in progress

The processing runs for about 15 minutes on my machine. Then it generates output, shown in Figure 10-13.

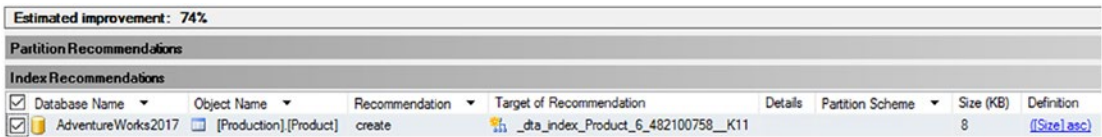


Figure 10-13. Recommendation for a manual statistic

After running all the queries through the Database Engine Tuning Advisor, the advisor came up with a suggestion for a new index for the Product table that would improve the performance the query. Now I just need to save that to a T-SQL file so that I can edit the name prior to applying it to my database.

Tuning from the Procedure Cache

You can take advantage of the query plans that are stored in the cache as a source for tuning recommendations. The process is simple. There’s a choice on the General page that lets you choose the plan cache as a source for the tuning effort, as shown in Figure 10-14.

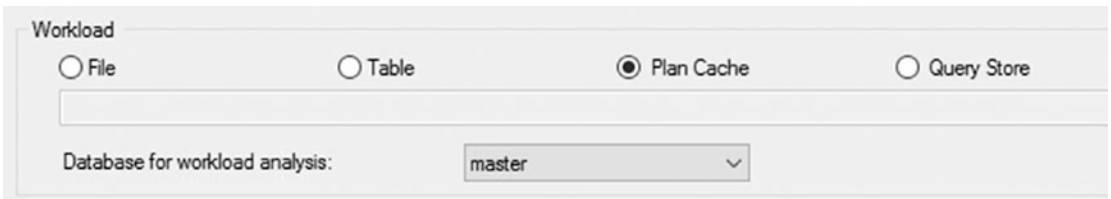


Figure 10-14. Selecting Plan Cache as the source for the DTA

All other options behave exactly the same way as previously outlined in this chapter. The processing time is radically less than when the advisor processes a workload. It has only the queries in cache to process so, depending on the amount of memory in your system, may be a short list. The results from processing my cache suggested several indexes and some individual statistics, as you can see in Figure 10-15.

| Database Name | Object Name | Recommendation | Target of Recommendation | Details | Partition Scheme | Size (KB) | Definition |
|--------------------|-----------------------------|----------------|---|---------|------------------|-----------|--|
| AdventureWorks2017 | [Production].[BIOMaterials] | create | idx_index_BIOMaterials_6_1157579162_K2_K1_K5_K3_K8_K7 | | | 194 | (ProductSearchGL.asl, UserData.asl, Inventory.asl, ComponentClass, PartSearchGL.asl, BOMLevel) |
| AdventureWorks2017 | [Production].[BIOMaterials] | create | idx_index_BIOMaterials_6_1157579162_K3_K4_K5_K2_K8_K7 | | | 194 | (ComponentGL.asl, PartData.asl, BOMLevel.asl, ProductAssemblyGL.asl, PartSearchGL.asl, BOMLevel) |
| AdventureWorks2017 | [Production].[BIOMaterials] | create | idx_index_1157579162_2_3_3_2_3_4 | | | | (ProductSearchGL, ComponentGL, PartSearchGL, BOMLevel, PartData, PartData) |
| AdventureWorks2017 | [Production].[BIOMaterials] | create | idx_index_1157579162_2_3_2_3_3_5 | | | | (ComponentGL, ProductSearchGL, PartData, BOMLevel, PartSearchGL) |
| AdventureWorks2017 | [Production].[Product] | create | idx_index_Product_6_48252558_K1_K2_K8_K10 | | | 32 | (ProductGL.asl, BOMLevel.asl, PartSearchGL.asl, PartData.asl) |
| AdventureWorks2017 | [Production].[Product] | create | idx_index_48252558_2_3_3_10_1 | | | | (Steel, ShownPart, PartData, ProductID) |

Figure 10-15. Recommendations from the plan cache

This gives you one more mechanism to try to tune your system in an automated fashion. But it is limited to the queries that are currently in cache. Depending on the volatility of your cache (the speed at which plans age out or are replaced by new plans), this may or may not prove useful.

Tuning from the Query Store

We're going to cover the Query Store in Chapter 11. However, we can take advantage of the information that the Query Store gathers in an attempt to get tuning suggestions from the Tuning Advisor. You select the Query Store workload from the list shown in Figure 10-14. Then you have to select a database because the Query Store is turned on only for individual databases. However, from there, the tuning options and behavior are the same. From my system there were several more suggestions than what were pulled from the plan cache, as shown in Figure 10-16.

Estimated improvement: 55%

| Partition Recommendations | | | | | | |
|---------------------------|-----------------------------------|----------------|---|---------|------------------|-----------|
| Database Name | Object Name | Recommendation | Target of Recommendation | Details | Partition Scheme | Size (KB) |
| AdventureWorks2017 | [Production].[BIO1Material] | create | idx_index_BIO1Material_K1157579162_K2_K3_K4_K5_K6_K7 | | | 164 |
| AdventureWorks2017 | [Production].[BIO1Material] | create | idx_index_BIO1Material_K1157579162_K3_K4_K5_K6_K7_K8_K9 | | | 164 |
| AdventureWorks2017 | [Production].[BIO1Material] | create | idx_index_1157579162_2_3_4_5_6_7_8_9 | | | 32 |
| AdventureWorks2017 | [Production].[Product] | create | idx_index_Product_K442100758_K1_K2_K3_K10 | | | 200 |
| AdventureWorks2017 | [Production].[Product] | create | idx_index_442100758_2_3_10_... | | | 216 |
| AdventureWorks2017 | [Production].[TransactionHistory] | create | idx_index_TransactionHistory_K1230627427_K3_K1_K2 | | | 216 |
| AdventureWorks2017 | [Production].[TransactionHistory] | create | idx_index_1230627427_2_1 | | | |

Figure 10-16. Recommendations from the Query Store

The reason there were even more suggestions is because the Query Store contains more plans than those that are simply in the plan cache or those that are captured during an individual trace run. That improved data makes the Query Store an excellent resource to use for tuning recommendations.

Database Engine Tuning Advisor Limitations

The Database Engine Tuning Advisor recommendations are based on the input workload. If the input workload is not a true representation of the actual workload, then the recommended indexes may sometimes have a *negative* effect on some queries that are missing in the workload. But most important, in many cases, the Database Engine Tuning Advisor may not recognize possible tuning opportunities. It has a sophisticated testing engine, but in some scenarios, its capabilities are limited.

For a production server, you should ensure that the SQL trace includes a complete representation of the database workload. For most database applications, capturing a trace for a complete day usually includes most of the queries executed on the database, although there are exceptions to this such as weekly, monthly, or year-end processing. Be sure you understand your load and what’s needed to capture it appropriately. A few of the other considerations/limitations with the Database Engine Tuning Advisor are as follows:

- Trace input using the SQL: BatchCompleted event:* As mentioned earlier, the SQL trace input to the Database Engine Tuning Advisor must include the SQL:BatchCompleted event; otherwise, the wizard won’t be able to identify the queries in the workload.
- Query distribution in the workload:* In a workload, a query may be executed multiple times with the same parameter value. Even a small performance improvement to the most common query can make a bigger contribution to the performance of the overall workload, compared to a large improvement in the performance of a query that is executed only once.

- *Index hints*: Index hints in a SQL query can prevent the Database Engine Tuning Advisor from choosing a better execution plan. The wizard includes all index hints used in a SQL query as part of its recommendations. Because these indexes may not be optimal for the table, remove all index hints from queries before submitting the workload to the wizard, bearing in mind that you need to add them back in to see whether they do actually improve performance.

Remember that the Tuning Advisor's recommendations are just that, recommendations. The suggestions it offers may not work as suggested by the advisor, and you may already have indexes in place that would serve just as well as the suggested indexes. Test and validate all suggestions prior to implementation.

Summary

As you learned in this chapter, the Database Engine Tuning Advisor can be a useful tool for analyzing the effectiveness of existing indexes and recommending new indexes for a SQL workload. As the SQL workload changes over time, you can use this tool to determine which existing indexes are no longer in use and which new indexes are required to improve performance. It can be a good idea to run the wizard occasionally just to check that your existing indexes really are the best fit for your current workload. This assumes you're not capturing metrics and evaluating them yourself. The Database Engine Tuning Advisor also provides many useful reports for analyzing the SQL workload and the effectiveness of its own recommendations. Just remember that the limitations of the tool prevent it from spotting all tuning opportunities. Also remember that the suggestions provided by the DTA are only as good as the input you provide to it. If your database is in bad shape, this tool can give you a quick leg up. If you're already monitoring and tuning your queries regularly, you may see no benefit from the recommendations of the Database Engine Tuning Advisor.

Capturing query metrics and execution plans used to be a lot of work to automate and maintain. However, capturing that information is vital in your query tuning efforts. Starting with SQL Server 2016, the Query Store provides a wonderful mechanism for capturing query metrics and so much more. The next chapter will give you a thorough understanding of all the functionality that the Query Store offers.

CHAPTER 11

Query Store

The Query Store was introduced originally in Azure SQL Database in 2015 and was first introduced to SQL Server in version 2016. The Query Store provides three pieces of functionality that you're going to want to take advantage of. First, you get query metrics and execution plans, stored permanently in the database in structures that are easy to access so that you have good, flexible information about the performance of the queries on your system. Second, the Query Store creates a mechanism for directly controlling execution plan behavior in a way we've never had before. Finally, the Query Store acts as a safety and reporting mechanism for database upgrades that will enable you to protect your systems in new ways.

In this chapter, I cover the following topics:

- How the Query Store works and the information it collects
- Reports and mechanisms exposed through Management Studio for Query Store behavior
- Plan forcing, a method for controlling which execution plans are used by SQL Server and Azure SQL Database
- An upgrade method that helps you protect your system behavior

While Extended Events sessions are your go-to measure for precision, for most systems, the Query Store should be the principal means of monitoring your query performance.

Query Store Function and Design

The Query Store is probably the most lightweight mechanism in terms of impact on the system. It provides the core of what you need to properly understand the query performance of your system. Plus, because all the work around the Query Store is done with system views, you get to use T-SQL to work with it, so using it becomes

incredibly easy. Using the DMOs, trace events, and even, to a degree, Extended Events as mechanisms for monitoring query metrics really can be considered old school with the introduction of the Query Store.

Query Store Behavior

The Query Store collects two pieces of information. First, it collects an aggregate of each query’s behavior on your system. Second, the Query Store captures, by default, every execution plan created on your system, up to the maximum number of plans per query (200 by default). You can turn the Query Store on and off on a database-by-database basis. When it’s on, the Query Store functions as shown in Figure 11-1.

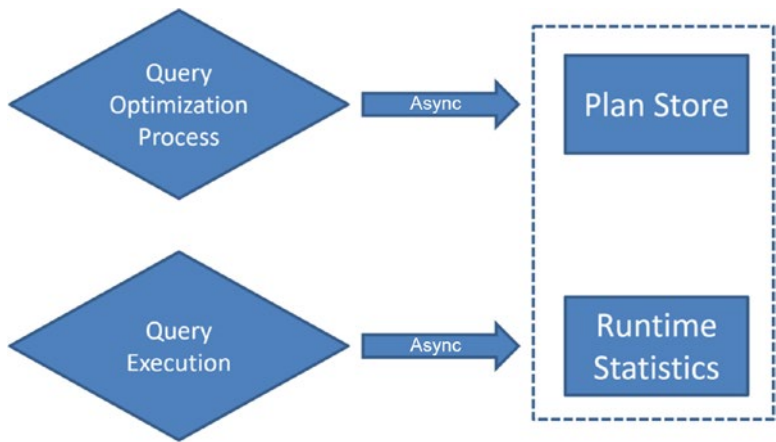


Figure 11-1. Behavior of the Query Store in collecting data

The query optimization process occurs as normal. When a query is submitted to the system, an execution plan is created (covered in detail in Chapter 15) and stored in the plan cache (which will be explained in Chapter 16). After these processes are complete, an asynchronous process runs for the Query Store to capture execution plans from the plan cache. Initially it writes these plans to a separate piece of memory for temporary storage. Another asynchronous process will then write these execution plans to the Query Store in the database. All these are asynchronous processes to ensure that there is minimal, although not zero, impact on other processes within the system. The only exception to the flow of this process is plan forcing, which we’ll cover later in the chapter.

The query execution then occurs just as with any other query. Once the query execution is complete, query runtime metrics, such as the number of reads, the number of writes, the duration of the query, and wait statistics, are written to a separate memory space, again, asynchronously. At a later point, another asynchronous process will write that information to disk. The information that is gathered and written to disk is aggregated. The default aggregation time is 60-minute intervals.

All the information stored within the Query Store system tables is written permanently to the database on which the Query Store is enabled. The query metrics and the execution plans for the queries are kept with the database. They get backed up with the database, and they get restored with the database. In the event of your system going offline or failing over, it is possible to lose some of the Query Store information that was still in memory and not yet written to disk. The default interval for writing to the disk is 15 minutes. Considering this is aggregate data, that's not a bad interval for the possibility of some Query Store data loss for what should not be considered production-level data.

When you query the information from the Query Store, it combines both the in-memory data and the data written to disk. You don't have to do anything extra to access that information.

Before continuing with the rest of the chapter, if you want to follow along with some of the code and processing, you'll need to enable the Query Store on one of the databases. This command will make it happen:

```
ALTER DATABASE AdventureWorks2017 SET QUERY_STORE = ON;
```

To ensure you have queries in the Query Store as you follow along, let's use this stored procedure:

```
CREATE OR ALTER PROC dbo.ProductTransactionHistoryByReference (
    @ReferenceOrderID int
)
AS
BEGIN
    SELECT p.Name,
           p.ProductNumber,
           th.ReferenceOrderID
```

```

FROM    Production.Product AS p
JOIN    Production.TransactionHistory AS th
        ON th.ProductID = p.ProductID
WHERE   th.ReferenceOrderID = @ReferenceOrderID;
END

```

If you execute the stored procedure with these three values, removing it from cache each time, you'll actually get three different execution plans.

```

DECLARE @Planhandle VARBINARY(64);

EXEC dbo.ProductTransactionHistoryByReference @ReferenceOrderID = 0;

SELECT @Planhandle = deps.plan_handle
FROM sys.dm_exec_procedure_stats AS deps
WHERE deps.object_id = OBJECT_ID('dbo.ProductTransactionHistoryByReference');

IF @Planhandle IS NOT NULL
BEGIN
    DBCC FREEPROCCACHE(@Planhandle);
END

EXEC dbo.ProductTransactionHistoryByReference @ReferenceOrderID = 53465;

SELECT @Planhandle = deps.plan_handle
FROM sys.dm_exec_procedure_stats AS deps
WHERE deps.object_id = OBJECT_ID('dbo.ProductTransactionHistoryByReference');

IF @Planhandle IS NOT NULL
BEGIN
    DBCC FREEPROCCACHE(@Planhandle);
END

EXEC dbo.ProductTransactionHistoryByReference @ReferenceOrderID = 3849;

```

With this, you can be sure that you'll have information in the Query Store.