```
AS
BEGIN;

    INSERT INTO @return_variable (SalesPersonID,
                                  ShippingCity,
                                  OrderDate,
                                  PurchaseOrderNumber,
                                  AccountNumber,
                                  OrderQty,
                                  UnitPrice)
    SELECT soh.SalesPersonID,
           a.City,
           soh.OrderDate,
           soh.PurchaseOrderNumber,
           soh.AccountNumber,
           sod.OrderQty,
           sod.UnitPrice
    FROM Sales.SalesOrderHeader AS soh
        JOIN Person.Address AS a
            ON a.AddressID = soh.ShipToAddressID
        JOIN Sales.SalesOrderDetail AS sod
            ON sod.SalesOrderID = soh.SalesOrderID
    WHERE soh.SalesPersonID = @SalesPersonID
          AND a.City = @ShippingCity;
    RETURN;
END;
GO
```

Instead of using a `WHERE` clause to execute the final query, we would execute it like this:

```
SELECT asi.OrderDate,
       asi.PurchaseOrderNumber,
       asi.AccountNumber,
       asi.OrderQty,
       asi.UnitPrice,
       sp.SalesQuota
```

```
FROM dbo.AllSalesInfo(277,'Odessa') AS asi
    JOIN Sales.SalesPerson AS sp
        ON asi.SalesPersonID = sp.BusinessEntityID;
```

By passing the parameters down to the function, we allow the interleaved execution to have values to measure itself against. Doing it this way returns exactly the same data, but the performance dropped to 65ms and only 1,135 reads. That's pretty amazing for a multistatement function. However, running this as a noninterleaved function also dropped the execution time to 69ms and 1,428 reads. While we are talking about improvements requiring no code or structure changes, those improvements are very minimal.

One additional problem can arise because of the interleaved execution, especially if you pass values as I did in the second query. It's going to create a plan based on the values it has in hand. This effectively acts as if it is parameter sniffing. It's using these hard-coded values to create execution plans directly in support of them, using these values against the statistics as the row count estimates. If your statistics vary wildly, you could be looking at performance problems similar to what we talked about in Chapter 15.

You can also use a query hint to disable the interleaved execution. Simply supply `DISABLE_INTERLEAVED_EXECUTION_TVF` to the query through the hint, and it will disable it only for the query being executed.

Summary

With the addition of the tuning recommendations in SQL Server 2017, along with the index automation in Azure SQL Database, you now have a lot more help within SQL Server when it comes to automation. You'll still need to use the information you've learned in the rest of the book to understand when those suggestions are helpful and when they're simply clues to making your own choices. However, things are even easier because SQL Server can make automatic adjustments without you having to do any work at all through the adaptive query processing. Just remember that all this is helpful, but none of it is a complete solution. It just means you have more tools in your toolbox to help deal with poorly performing queries.

The next chapter discusses methods you can use to automate the testing of your queries through the use of distributed replay.

CHAPTER 26

Database Performance Testing

Knowing how to identify performance issues and knowing how to fix them are great skills to have. The problem, though, is that you need to be able to demonstrate that the improvements you make are real improvements. While you can, and should, capture the performance metrics before and after you tune a query or add an index, the best way to be sure you're looking at measurable improvement is to put the changes you make to work. Testing means more than simply running a query a few times and then putting it into your production system with your fingers crossed. You need to have a systematic way to validate performance improvements using the full panoply of queries that are run against your system in a realistic manner. Introduced with the 2012 version, SQL Server provides such a mechanism through its Distributed Replay tool.

Distributed Replay works with information generated from the SQL Profiler and the trace events created by it. Trace events capture information in a somewhat similar fashion to the Extended Events tool, but trace events are an older (and less capable) mechanism for capturing events within the system. Prior to the release of SQL Server 2012, you could use SQL Server's Profiler tool to replay captured events using a server-side trace. This worked, but the process was extremely limited. For example, the tool could be run only on a single machine, and it dealt with the playback mechanism—a single-threaded process that ran in a serial fashion, rather than what happens in reality. Microsoft has added the capability to run from multiple machines in a parallel fashion to SQL Server. Until Microsoft makes a mechanism to use Distributed Replay through Extended Events output, you'll still be using the trace events for this one aspect of your performance testing.

Distributed Replay is not a widely adopted tool. Most people just skip the idea of implementing repeatable tests entirely. Others may go with some third-party tools that provide a little more functionality. I strongly recommend you do some form of testing to

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_26

ensure your tuning efforts are resulting in positive impact on your systems that you can accurately measure.

This chapter covers the following topics:

- The concepts of database testing
- How to create a server-side trace
- Using Distributed Replay for database testing

Database Performance Testing

The general approach to database performance and load testing is pretty simple. You need to capture the calls against a production system under normal load and then be able to play that load over and over again against a test system. This enables you to directly measure the changes in performance caused by changes to your code or structures. Unfortunately, accomplishing this in the real world is not as simple as it sounds.

To start with, you can't simply capture the recording of queries. Instead, you must first ensure that you can restore your production database to a moment in time on a test system. Specifically, you need to be able to restore to exactly the point at which you start recording the transactions on the system because if you restore to any other point, you might have different data or even different structures. This will cause the playback mechanism to generate errors instead of useful information. This means, to start with, you must have a database that is in Full Recovery mode so that you can take regular full backups as well as log backups in order to restore to a specific point in time when your testing will start.

Once you establish the ability to restore to the appropriate time, you will need to configure your query capture mechanism—a server-side trace definition generated by Profiler, in this case. The playback mechanism will define exactly which events you'll need to capture. You'll want to set up your capture process so that it impacts your system as little as possible.

Next, you'll have to deal with the large amounts of data captured by the trace. Depending on how big your system is, you may have a large number of transactions over a short period of time. All that data has to be stored and managed, and there will be many files.

You can set up this process on a single machine; however, to really see the benefits, you'll want to set up multiple machines to support the playback capabilities of the Distributed Replay tool. This means you'll need to have these machines available to you as part of your testing process. Unfortunately, with all editions except Enterprise, you can have only a single client, so take that into account as you set up your test environment.

Also, you can't ignore the fact that the best data, database, and code to work with is your production system. However, depending on your need for compliance for local and international law, you may have to choose a completely different mechanism for recording your server-side trace. You don't want to compromise the privacy and protection of the data under management within the organization. If this is the case, you may have to capture your load from a QA server or a preproduction server that is used for other types of automated testing. These can be difficult problems to overcome.

When you have all these various parts in place, you can begin testing. Of course, this leads to a new question: what exactly are you doing with your database testing?

A Repeatable Process

As explained in Chapter 1, performance tuning your system is an iterative process that you may have to go through on multiple occasions to get your performance to where you need it to be and keep it there. Since businesses change over time, so will your data distribution, your applications, your data structures, and all the code supporting it. Because of all this, one of the most important things you can do for testing is to create a process that you can run over and over again.

The primary reason you need to create a repeatable testing process is because you can't always be sure that the methods outlined in the preceding chapters of this book will work well in every situation. This no doubt means you need to be able to validate that the changes you have made have resulted in a positive improvement in performance. If not, you need to be able to remove any changes you've made, make a new set of changes, and then repeat the tests, repeating this process iteratively. You may find that you'll need to repeat the entire tuning cycle until you've met your goals for this round.

Because of the iterative nature of this process, you absolutely need to concentrate on automating it as much as possible. This is where the Distributed Replay tool comes into the picture.

Distributed Replay

The Distributed Replay tool consists of three pieces of architecture.

- *Distributed Replay Controller*: This service manages the processes of the Distributed Replay system.
- *Distributed Replay Administrator*: This is an interface to allow you to control the Distributed Replay Controller and the Distributed Replace process.
- *Distributed Replay Client*: This is an interface that runs on one or more machines (up to 16) to make all the calls to your database server.

You can install all three components onto one machine; however, the ideal approach is to have the controller on one machine and then have one or more client machines that are completely separated from the controller so that each of these machines is handling only some of the transactions you'll be making against the test machine. Only for the purposes of illustration, I have all the components running on a single instance.

Begin by installing the Distributed Replay Controller service onto a machine. There is no interface for the Distributed Replay utility. Instead, you'll use XML configuration files to take control of the different parts of the Distributed Replay architecture. You can use the distributed playback for various tasks, such as basic query playback, server-side cursors, or prepared server statements. Since I'm primarily covering query tuning, I'm focus on the queries and prepared server statements (also known as *parameterized queries*). This defines a particular set of events that must be captured. I'll cover how to do that in the next section.

Once the information is captured in a trace file, you will have to run that file through the preprocess event using the Distributed Replay Controller. This modifies the basic trace data into a different format that can be used to distribute to the various Distributed Replay Client machines. You can then fire off a replay process. The reformatted data is sent to the clients, which in turn will create queries to run against the target server. You can capture another trace output from the client machines to see exactly which calls they made, as well as the I/O and CPU of those calls. Presumably you'll also set up standard monitoring on the target server to see how the load you are generating impacts that server.

When you go to run the system against your server, you can choose one of two types of playback: Synchronization mode or Stress mode. In Synchronization mode, you will get an exact copy of the original playback, although you can affect the amount of idle

time on the system. This is good for precise performance tuning because it helps you understand how the system is working, especially if you're making changes to structures, indexes, or T-SQL code. Stress mode doesn't run in any particular order, except within a single connection, where queries will be streamed in the correct order. In this case, the calls are made as fast as the client machines can make them—in any order—as fast as the server can receive them. In short, it performs a stress test. This is useful for testing database designs or hardware installations.

One important note, as a general rule, is that you're safest when using the latest version of SQL Server for your replay only with the latest version of trace data. However, you can replay SQL Server 2005 data on SQL Server 2017. Also, Azure SQL Database is not supported by Distributed Replay or trace events, so you won't be able to use any of this with your Azure database.

Capturing Data with the Server-Side Trace

Using trace events to capture data is similar to capturing query executions with Extended Events. To support the Distributed Replay process, you'll need to capture some specific events and specific columns for those events. If you want to build your own trace events, you need to go after the events listed in Table 26-1.

Table 26-1. *Events to Capture*

Events	Columns
Prepare SQL	Event Class
Exec Prepared SQL	EventSequence
SQL:BatchStarting	TextData
SQL:BatchCompleted	Application Name
RPC:Starting	LoginName
RPC:Completed	DatabaseName
RPC Output Parameter	Database ID
Audit Login	HostName
Audit Logout	Binary Data
Existing Connection	SPID
Server-side Cursor	Start Time
Server-side prepared SQL	EndTime
	IsSystem

You have two options for setting up these events. First, you can use T-SQL to set up a server-side trace. Second, you can use an external tool called Profiler. While Profiler can connect directly to your SQL Server instance, I strongly recommend against using this tool to capture data. Profiler is best used as a way to supply a template for performing the capture. You should use T-SQL to generate the actual server-side trace.

On a test or development machine, open Profiler and select TSQL_Replay from the Template list, as shown in Figure 26-1.

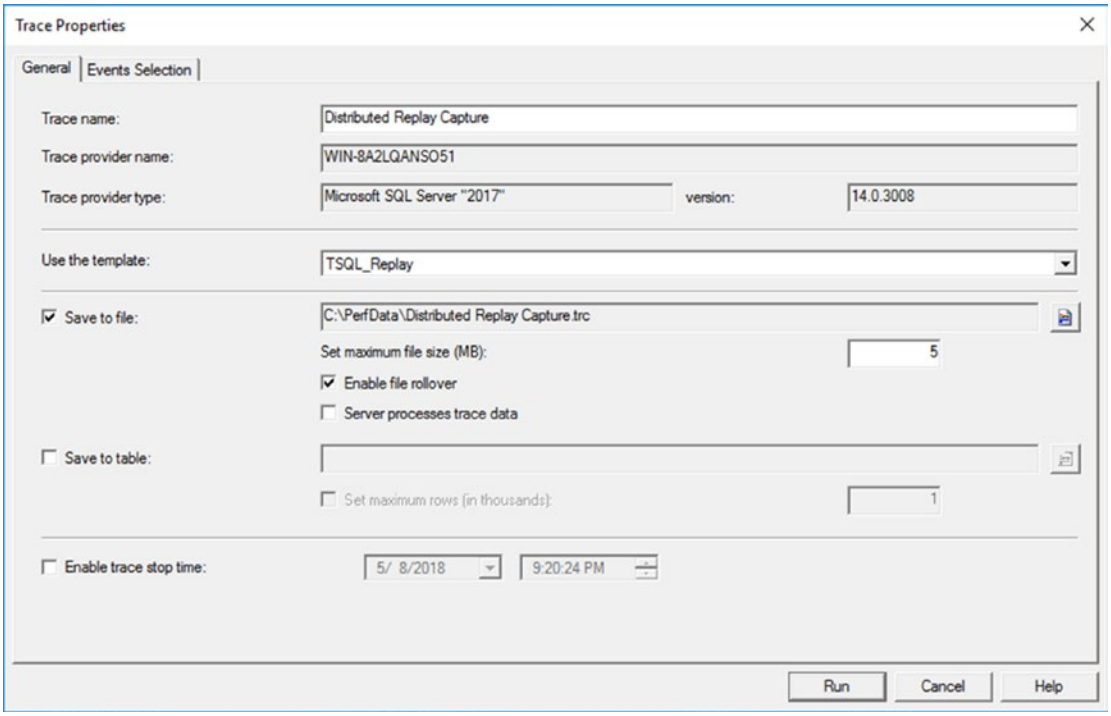

Figure 26-1. *The Distributed Replay trace template*

Since you need a file for Distributed Replay, you'll want to save the output of the trace to file. It's the best way to set up a server-side trace anyway, so this works out. You'll want to output to a location that has sufficient space. Depending on the number of transactions you have to support with your system, trace files can be extremely large. Also, it's a good idea to put a limit on the size of the files and allow them to roll over, creating new files as needed. You'll have more files to deal with, but the operating system can actually deal with a larger number of smaller files for writes better than it can deal with a single large file. I've found this to be true because of two things. First, with a smaller file size, you get

a quicker rollover, which means the previous file is available for processing if you need to load it into a table or copy it to another server. Second, in my experience, it generally takes longer for writes to occur with simple log files because the size of such files gets very large. I also suggest defining a stop time for the trace process; again, this helps ensure you don't fill the drive you've designated for storing the trace data.

Since this is a template, the events and columns have already been selected for you. You can validate the events and columns to ensure you are getting exactly what you need by clicking the Events Selection tab. Figure 26-2 shows some of the events and columns, all of which are predefined for you.

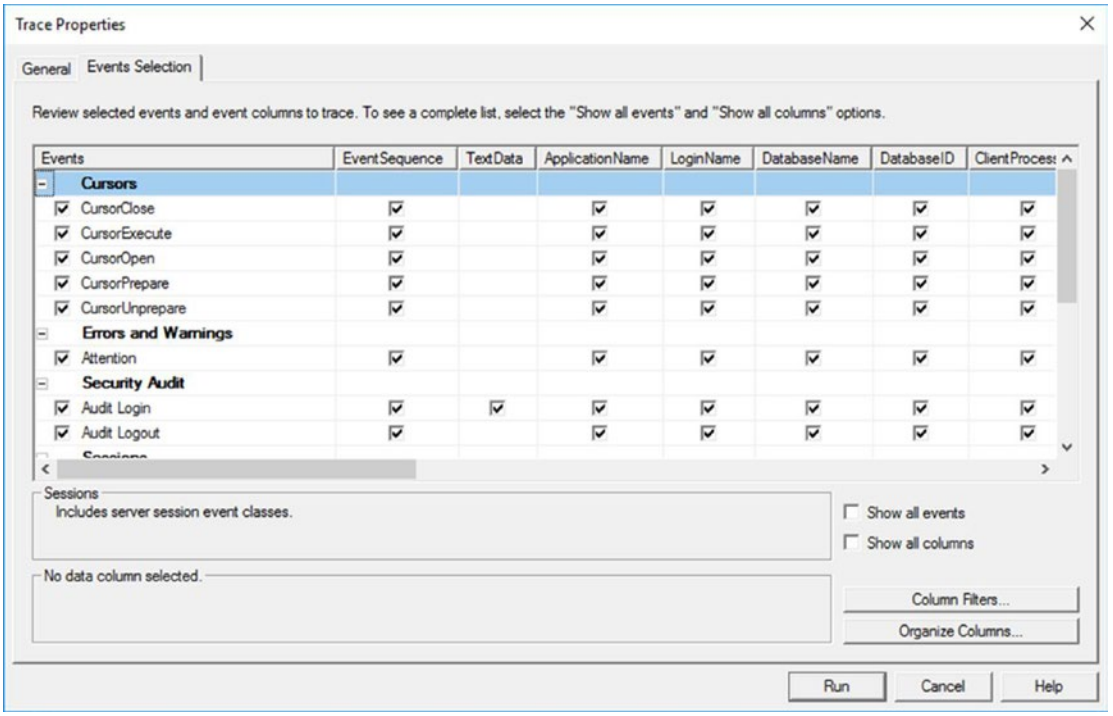

Figure 26-2. *The TSQL_Replay template events and columns*

This template is generic, so it includes the full list of events, including all the cursor events. You can edit it by clicking boxes to deselect events; however, I do not recommend removing anything other than the cursor events, if you're going to remove any.

I started this template connected to a test server instead of a production machine because once you've set it up appropriately, you have to start the trace by clicking Run. I wouldn't do that on a production system. On a test system, however, you can watch the

screen to ensure you're getting the events you think you should. It will display the events, as well as capture them to a file. When you're satisfied that it's correct, you can pause the trace. Next, click the File menu and then select Export ➤ Script Trace Definition. Finally, select For SQL Server 2005 – 2014 (see Figure 26-3).

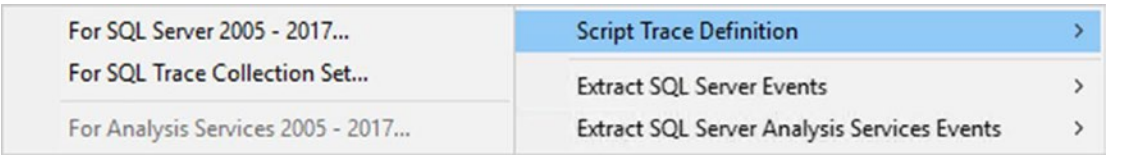

Figure 26-3. *The menu selection to output the trace definition*

This template will allow you to save the trace you just created as a T-SQL file. Once you have the T-SQL, you can configure it to run on any server that you like. The file path will have to be replaced, and you can reset the stop time through parameters within the script. The following script shows the beginning of the T-SQL process used to set up the server-side trace events:

```
/****************************************************/
/* Created by: SQL Server 2017 Profiler          */
/* Date: 05/08/2018  08:27:40 PM         */
/****************************************************/

-- Create a Queue
declare @rc int
declare @TraceID int
declare @maxfilesize bigint
set @maxfilesize = 5

-- Please replace the text InsertFileNameHere, with an appropriate
-- filename prefixed by a path, e.g., c:\MyFolder\MyTrace. The .trc extension
-- will be appended to the filename automatically. If you are writing from
-- remote server to local drive, please use UNC path and make sure server has
-- write access to your network share

exec @rc = sp_trace_create @TraceID output, 0, N'InsertFileNameHere',
@maxfilesize, NULL
if (@rc != 0) goto error
```

You can edit the path where it says `InsertFileNameHere` and provide different values for `@DateTime`. At this point, your script can be run on any SQL Server 2017 server. You can probably run the same script all the way back to SQL Server 2008R2; there have been so few changes made to trace events since then that this is a fixed standard now. However, always test to be on the safe side.

The amount of information you collect really depends on what kind of test you want to run. For a standard performance test, it's probably a good idea to collect at least one hour's worth of information; however, you wouldn't want to capture more than two to three hours of data in most circumstances. Plus, it can't be emphasized enough that trace events are not as lightweight as extended events, so the longer you capture data, the more you're negatively impacting your production server. Capturing more than that would entail managing a lot more data, and it would mean you were planning on running your tests for a long time. It all depends on the business and application behaviors you intend to deal with in your testing.

Before you capture the data, you do need to think about where you're going to run your tests. Let's assume you're not worried about disk space and that you don't need to protect legally audited data (if you have those issues, you'll need to address them separately). If your database is not in Full Recovery mode, then you can't use the log backups to restore it to a point in time. If this is the case, I strongly recommend running a database backup as part of starting the trace data collection. The reason for this is that you need the database to be in the same condition it's in when you start recording transactions. If it's not, you may get a larger number of errors, which could seriously change the way your performance tests run. For example, attempting to select or modify data that doesn't exist will impact the I/O and CPU measured in your tests. If your database remains in the same state that it was at or near the beginning of your trace, then you should few, if any, errors.

With a copy of the database ready to go and a set of trace data, you're ready to run the Distributed Replay tool.

Distributed Replay for Database Testing

Assuming you used the Distributed Replay template to capture your trace information, you should be ready to start processing the files. As noted earlier, the first step is to convert the trace file into a different format, one that can be split up among multiple client machines for playback. But there is more to it than simply running the executable

against your file. You also need to make some decisions about how you want the Distributed Replay to run; you make those decisions when you preprocess the trace file.

The decisions are fairly straightforward. First, you need to decide whether you're going to replay system processes along with the user processes. Unless you're dealing with the potential of specific system issues, I suggest setting this value to `No`. This is also the default value. Second, you need to decide how you want to deal with idle time. You can use the actual values for how often calls were made to the database; or, you can put in a value, measured in seconds, to limit the wait time to no more than that value. It really depends on what type of playback you're going to run. Assuming you use Synchronization mode playback, the mode best suited for straight performance measurement, it's a good idea to eliminate idle time by setting the value to something low, such as three to five seconds.

If you choose to use the default values, you don't need to modify the configuration file. But if you've chosen to include the system calls or to change the idle time, then you'll need to change the configuration file, `DReplay.Exe.Preprocess.config`. It's a simple XML configuration file. The one I'm using looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Options>
<PreprocessModifiers>
<IncSystemSession>No</IncSystemSession>
<MaxIdleTime>2</MaxIdleTime>
</PreprocessModifiers>
</Options>
```

I've made only one change, adjusting `MaxdIdleTime` to limit any down period during the playback.

Before you run the preprocessing, make sure have installed the `DRController` and that the `DReplay` service is running on your system. If so, you'll just need to call `DReplay.exe` to execute the preprocessing.

```
dreplay preprocess -i c:\perfdata\dr.trc -d c:\DRProcess
```

In the preceding code, you can see that `DReplay` runs the preprocess event. The input file was supplied by the `-i` parameter, and a folder to hold the output was supplied through the `-d` parameter. The trace files will be processed, and the output will go to the folder specified. The output will look something like Figure 26-4.

```
C:\Program Files (x86)\Microsoft SQL Server\140\Tools\DReplayClient>dreplay preprocess -i c:\perfdata\dr.trc
cess
2018-05-09 08:12:05:182 Info DReplay    Preprocessing pass 1 of 2 in progress.
2018-05-09 08:12:11:995 Info DReplay    Preprocessing pass 1 of 2 completed.
2018-05-09 08:12:12:011 Info DReplay    Preprocessing pass 2 of 2 in progress.
2018-05-09 08:12:12:011 Info DReplay    Preprocessing pass 2 of 2 completed.
2018-05-09 08:12:12:011 Info DReplay    4407 replayable events written to intermediate file in c:\DRProcess.
2018-05-09 08:12:12:011 Info DReplay    Elapsed time: 0 day(s), 0 hour(s), 0 minute(s), 7 second(s).
```

Figure 26-4. *Output from the preprocessing steps of Distributed Replay*

With the preprocessing complete, you're ready to move ahead with running the Distributed Replay process. Before you do so, however, you need to make sure you have one or more client systems ready to go.

Configuring the Client

The client machines will have to be configured to work with the Distributed Replay controller. Begin by installing your clients to the different machines. For illustration purposes only, I'm running everything on a single machine; however, the setup is no different if you use multiple machines. You need to configure the client to work with a controller, and a client can work with only one controller at a time. You also need to have space on the system for two items. First, you need a location for working files that are overwritten at each replay. Second, you need room for trace file output from the client if you want to collect execution data from that client. You also get to decide on the logging level of the client process. All of this is set in another XML configuration file, `DReplayClient.config`. Here is my configuration:

```
<Options>
<Controller>PerfTune</Controller>
<WorkingDirectory>C:\DRClientWork\</WorkingDirectory>
<ResultDirectory>C:\DRClientOutput\</ResultDirectory>
<LoggingLevel>CRITICAL</LoggingLevel>
</Options>
```

The directories and logging level are clear. I also had to point the client to the server where the Distributed Replay service is running. No other settings are required for multiple clients to work; you just need to be sure they're going to the right controller system.

Running the Distributed Tests

So far you have configured everything and captured the data. Next, you need to go back to the command line to run things from the `Dreplay.exe` executable. Most of the control is accomplished through the configuration files, so there is little input required in the executable. You invoke the tests using the following command:

```
Dreplay replay –d c:\data –w DOJO
```

You need to feed in the location of the output from the preprocessing, which means you need to list the client machines that are taking part in a comma-delimited list. The output from the execution would look something like Figure 26-5.

```
C:\Program Files (x86)\Microsoft SQL Server\140\Tools\DReplayClient>dreplay replay -d c:\drprocess -w WIN-8
 WIN-8A2LQANSO51
2018-05-09 08:33:27:886 Info DReplay    Dispatching in progress.
2018-05-09 08:33:27:902 Info DReplay    0 events have been dispatched.
2018-05-09 08:33:32:662 Info DReplay    Dispatching has completed.
2018-05-09 08:33:32:662 Info DReplay    62 events dispatched in total.
2018-05-09 08:33:32:662 Info DReplay    Elapsed time: 0 day(s), 0 hour(s), 0 minute(s), 4 second(s).
2018-05-09 08:33:32:662 Info DReplay    Event replay in progress.
2018-05-09 08:33:57:921 Info DReplay    Event replay has completed.
2018-05-09 08:33:57:921 Info DReplay    62 events (100 %) have been replayed in total. Pass rate 100.00 %.
2018-05-09 08:33:57:937 Info DReplay    Elapsed time: 0 day(s), 0 hour(s), 0 minute(s), 26 second(s).
```

Figure 26-5. *The output from running DReplay.exe*

As you can see, 62 events were captured, and all 62 events were successfully replayed. If, on the other hand, you had errors or events that failed, you might need to establish what information might exist about why some of the events failed. This information is available in the logs. Then, simply reconfigure the tests and run them again. The whole idea behind having a repeatable testing process is that you can run it over and over. The preceding example represents a light load run against my local copy of AdventureWorks2017, captured over about five minutes. However, I configured the limits on idle time, so the replay completes in only 26 seconds.

From here, you can reconfigure the tests, reset the database, and run the tests over and over again, as needed. Note that changing the configuration files will require you to restart the associated services to ensure that the changes are implemented with the next set of tests. One of the best ways to deal with testing here is to have the Query Store enabled. You can capture one set of results, reset the test, make whatever changes to the system you're going to make, and then capture another set of results from a second test. Then, you can easily look at reports for regressed queries, queries that used the most resources, and so on.

Conclusion

With the inclusion of the Distributed Replay utilities, SQL Server now gives you the ability to perform load and function testing against your databases. You accomplish this by capturing your code in a simple manner with a server-side trace. If you plan to take advantage of this feature, however, be sure to validate that the changes you make to queries based on the principles put forward in this book actually work and will help improve the performance of your system. You should also make sure you reset the database to avoid errors as much as possible.

CHAPTER 27

Database Workload Optimization

So far, you have learned about a number of aspects that can affect query performance, the tools that you can use to analyze query performance, and the optimization techniques you can use to improve query performance. Next, you will learn how to apply this information to analyze, troubleshoot, and optimize the performance of a database workload. I'll walk you through a tuning process, including possibly going down a bad path or two, so bear with me as we navigate the process.

In this chapter, I cover the following topics:

- The characteristics of a database workload
- The steps involved in database workload optimization
- How to identify costly queries in the workload
- How to measure the baseline resource use and performance of costly queries
- How to analyze factors that affect the performance of costly queries
- How to apply techniques to optimize costly queries
- How to analyze the effects of query optimization on the overall workload

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_27

Workload Optimization Fundamentals

Optimizing a database workload often fits the 80/20 rule: 80 percent of the workload consumes about 20 percent of server resources. Trying to optimize the performance of the majority of the workload is usually not very productive. So, the first step in workload optimization is to find the 20 percent of the workload that consumes 80 percent of the server resources.

Optimizing the workload requires a set of tools to measure the resource consumption and response time of the different parts of the workload. As you saw in Chapters 4 and 5, SQL Server provides a set of tools and utilities to analyze the performance of a database workload and individual queries.

In addition to using these tools, it is important to know how you can use different techniques to optimize a workload. The most important aspect of workload optimization to remember is that not every optimization technique is guaranteed to work on every performance problem. Many optimization techniques are specific to certain database application designs and database environments. Therefore, for each optimization technique, you need to measure the performance of each part of the workload (that is, each individual query) before and after you apply an optimization technique. You can use the techniques discussed in Chapter 26 to make this happen.

It is not unusual to find that an optimization technique has little effect—or even a negative effect—on the other parts of the workload, thereby hurting the overall performance of the workload. For instance, a nonclustered index added to optimize a `SELECT` statement can hurt the performance of `UPDATE` statements that modify the value of the indexed column. The `UPDATE` statements have to update index rows in addition to the data rows. However, as demonstrated in Chapter 6, sometimes indexes can improve the performance of action queries, too. Therefore, improving the performance of a particular query could benefit or hurt the performance of the overall workload. As usual, your best course of action is to validate any assumptions through testing.

Workload Optimization Steps

The process of optimizing a database workload follows a specific series of steps. As part of this process, you will use the set of optimization techniques presented in previous chapters. Since every performance problem is a new challenge, you can use a different set of optimization techniques for troubleshooting different performance problems. Just

remember that the first step is always to ensure that the server is well configured and operating within acceptable limits, as defined in Chapters 2 and 3.

To understand the query optimization process, you will simulate a sample workload using a set of queries.

The core of query tuning comes down to just a few steps.

1. Identify the query to tune.
2. Look at the execution plan to understand resource usage and behavior.
3. Modify the query or modify the structure to improve performance.

Most of the time, the answer is, modify the query. In a nutshell, that's all that's necessary to do query tuning. However, this assumes a lot of knowledge of the system, and you've looked at things like statistics in the past. When you're approaching query tuning for the first time or you're on a new system, the process is quite a bit more detailed. For a thorough and complete definition of the steps necessary to tune a query, here's what you're going to do. These are the optimization steps you will follow as you optimize the sample workload:

1. Capture the workload.
2. Analyze the workload.
3. Identify the costliest/most frequently called/longest-running query.
4. Quantify the baseline resource use of the costliest query.
5. Determine the overall resource use.
6. Compile detailed information on resource use.
7. Analyze and optimize external factors.
8. Analyze the use of indexes.
9. Analyze the batch-level options used by the application.
10. Analyze the effectiveness of statistics.
11. Assess the need for defragmentation.
12. Analyze the internal behavior of the costliest query.

13. Analyze the query execution plan.
14. Identify the costly operators in the execution plan.
15. Analyze the effectiveness of the processing strategy.
16. Optimize the costliest query.
17. Analyze the effects of the changes on database workload.
18. Iterate through multiple optimization phases.

As explained in Chapter 1, performance tuning is an iterative process. Therefore, you should iterate through the performance optimization steps multiple times until you achieve the desired application performance targets. After a certain period of time, you will need to repeat the process to address the impact on the workload caused by data and database changes. Further, as you find yourself working on a server over time, you may be skipping lots of the previous steps since you've already validated transaction methods or statistics maintenance or other steps. You don't have to follow this slavishly. It's meant to be a guide. I'll refer you to Chapter 1 for the graphical representation of the steps needed to tune a query.

Sample Workload

To troubleshoot SQL Server performance, you need to know the SQL workload that is executed on the server. You can then analyze the workload to identify causes of poor performance and applicable optimization steps. Ideally, you should capture the workload on the SQL Server facing the performance problems. In this chapter, you will use a set of queries to simulate a sample workload so that you can follow the optimization steps listed in the previous section. The sample workload you'll use consists of a combination of good and bad queries.

Note I recommend you restore a clean copy of the AdventureWorks2017 database so that any artifacts left over from previous chapters are completely removed.

The simple test workload is simulated by the following set of sample stored procedures; you execute them using the second script on the AdventureWorks2017 database:

```
USE AdventureWorks2017;
GO

CREATE OR ALTER PROCEDURE dbo.ShoppingCart @ShoppingCartId VARCHAR(50)
AS
--provides the output from the shopping cart including the line total
SELECT sci.Quantity,
       p.ListPrice,
       p.ListPrice * sci.Quantity AS LineTotal,
       p.Name
FROM Sales.ShoppingCartItem AS sci
    JOIN Production.Product AS p
        ON sci.ProductID = p.ProductID
WHERE sci.ShoppingCartID = @ShoppingCartId;
GO

CREATE OR ALTER PROCEDURE dbo.ProductBySalesOrder @SalesOrderID INT
AS
/*provides a list of products from a particular sales order,
and provides line ordering by modified date but ordered
by product name*/

SELECT ROW_NUMBER() OVER (ORDER BY sod.ModifiedDate) AS LineNumber,
       p.Name,
       sod.LineTotal
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
    JOIN Production.Product AS p
        ON sod.ProductID = p.ProductID
WHERE soh.SalesOrderID = @SalesOrderID
ORDER BY p.Name ASC;
GO
```