Analyzing the Internal Behavior of the Costliest Query

Now you need to analyze the processing strategy for the query chosen by the optimizer to determine the internal factors affecting the query's performance. Analyzing the internal factors that can affect query performance involves these steps:

- Analyzing the query execution plan
- Identifying the costly steps in the execution plan
- Analyzing the effectiveness of the processing strategy

Analyzing the Query Execution Plan

To see the execution plan, click the Show Actual Execution Plan button to enable it and then run the stored procedure. Be sure you're doing these types of tests on a nonproduction system, while, at the same time, have it be as much like production as possible so that the behavior there mirrors what you see in production. We covered execution plans in Chapter 16. For more details on reading execution plans, check out my book *SQL Server Execution Plans* (Red Gate Publishing, 2018). Figure 27-7 shows the graphical execution plan of the worst-performing query.

Figure 27-7. *The actual execution plan of the worst-performing query*

The graphic of this plan is somewhat difficult to read. I'll break down a few of the interesting details in case you're not following along with code. You could observe the following from this execution plan:

- `SELECT` properties
    - `Optimization Level: Full`
    - `Reason for Early Termination: Good enough plan found`

    - Query Time Statistics: 30ms CpuTime and 244 ms ElapsedTime
    - WaitStats: WaitCount 4, WaitTimeMs 214, WaitType `ASYNC_NETWORK_IO`
- Data access
    - Index seek on nonclustered index, `Person.IX_Person_LastName_FirstName_MiddleName`
    - Clustered index scan on, `PurchaseOrderHeader.PK_PruchaseOrderHeader_PurchaseOrderID`
    - Clustered index seek on `PurchaseOrderDetail.PK_PurchaseOrderDetail_PurchaseOrderDetailID`
    - Clustered Index seek on `Product.PK_Product_ProductID`
    - Clustered Index seek on `Employee.PK_Employee_BusinessEntityID`
    - Join strategy
    - Nested loop join between the constant scan and `Person.Person` table with the `Person.Person` table as the outer table
    - Nested loop join between the output of the previous join and `Purchasing.PurchaseOrderHeader` with the `Purchasing.PurchaseOrderHeader` table as the outer table
    - Nested loop join between the output of the previous join and the `Purchasing.PurchaseOrderDetail` table that was also the outer table
    - Nested loop join between the output of the previous join and the `Production.Product` table with `Production.Product` as the outer table
    - Nested loop join between the previous join and the `HumanResources.Employee` table with the `HumanResource.Employee` table as the outer table

- Additional processing
  - Constant scan to provide a placeholder for the `@LastName` variable's `LIKE` operation
  - Compute scalar that defined the constructs of the `@LastName` variable's `LIKE` operation, showing the top and bottom of the range and the value to be checked
  - Compute scalar that combines the `FirstName` and `LastName` columns into a new column
  - Compute scalar that calculates the `LineTotal` column from the `Purchasing.PurchaseOrderDetail` table
  - Compute scalar that takes the calculated `LineTotal` and stores it as a permanent value in the result set for further processing

All this information is available by browsing the details of the operators exposed in the properties sheet from the graphical execution plan.

Identifying the Costly Steps in the Execution Plan

Once you understand the execution plan of the query, the next step is to identify the steps estimated as the most costly in the execution plan. Although these costs are estimated and don't reflect reality in any way, they are the only numbers you will receive that measure the function of the plan, so identifying, understanding, and possibly addressing the most costly operations can result in massive performance benefit. You can see that the following are the two costliest steps:

- *Costly step 1*: The clustered index scan on the `Purchasing.PurchaseOrderHeader` table is 36 percent.
- *Costly step 2*: The hash match join operation is 32 percent.

The next optimization step is to analyze the costliest steps so you can determine whether these steps can be optimized through techniques such as redesigning the query or indexes.

Analyzing the Processing Strategy

While the optimizer completed optimizing the plan, which you know because the reason for early termination of the optimization process was "Good Enough Plan Found" (or, because it showed `FULL` optimization without a reason for early termination), that doesn't mean there are not tuning opportunities in the query and structure. You can begin evaluating it by following the traditional steps.

Costly step 1 is a clustered index scan. Scans are not necessarily a problem. They're just an indication that a full scan of the object in question, in this case the entire table, was less costly than the alternatives to retrieve the information needed by the query.

Costly step 2 is the hash match join operation of the query. This again is not necessarily a problem. But, sometimes, a hash match is an indication of bad or missing indexes, or queries that can't make use of the existing indexes, so they are frequently an area that needs work. At least, that's frequently the case for OLTP systems. For large data warehouse systems, a hash match may be ideal for dealing with the types of queries you'll see there.

Tip At times you may find that no improvements can be made to the costliest step in a processing strategy. In that case, concentrate on the next costliest step to identify the problem. If none of the steps suggests indications for optimization, then you may need to consider changing the database design or the construction of the query.

Optimizing the Costliest Query

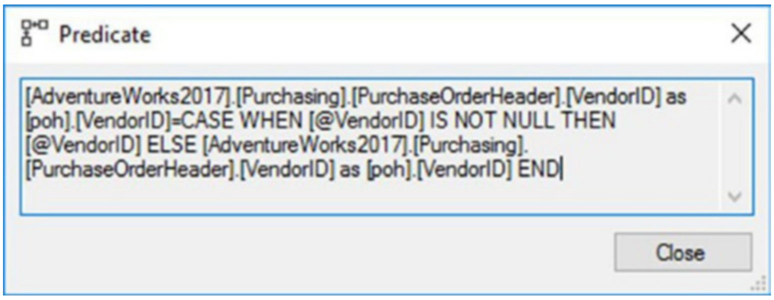
Once you've diagnosed the queries with costly steps, the next stage is to implement the necessary corrections to reduce the cost of these steps.

The corrective actions for a problematic step can have one or more alternative solutions. For example, should you create a new index or structure the query differently? In such cases, you should prioritize the solutions based on their expected effectiveness and the amount of work required. For example, if a narrow index can more or less do the job, then it is usually better to prioritize that over changes to code that might lead to business testing. Making changes to code may also be the less intrusive approach. You need to evaluate each situation within the business and application that you're dealing with.

Apply the solutions individually in the order of their expected benefit and measure their individual effect on the query performance. Finally, you can apply the solution (or solutions) that provides the greatest performance improvement to correct the problematic step. Sometimes, it may be evident that the best solution will hurt other queries in the workload. For example, a new index on a large number of columns can hurt the performance of action queries. However, since that's not always an issue, it's better to determine the effect of such optimization techniques on the complete workload through testing. If a particular solution hurts the overall performance of the workload, choose the next best solution while keeping an eye on the overall performance of the workload.

Modifying the Code

The costliest operation in the query is a clustered index scan of the `PurchaseOrderHeader` table. The first thing you need to do is understand if the clustered index scan is necessary for the query and data returned or may be there because of the code or even because another index or a different index structure could work better. To begin to understand why you're getting a clustered index scan, you should look at the properties of the scan operation. Since you're getting a scan, you also need to look to the code to ensure it's sargable. Specifically, you're interested in the `Predicate` property, as shown in Figure 27-8.



```
[AdventureWorks2017].[Purchasing].[PurchaseOrderHeader].[VendorID] as
[poh].[VendorID]=CASE WHEN [@VendorID] IS NOT NULL THEN
[@VendorID] ELSE [AdventureWorks2017].[Purchasing].
[PurchaseOrderHeader].[VendorID] as [poh].[VendorID] END
```

Figure 27-8. *The predicate of the clustered index scan*

This is a calculation. There is an existing index on the `VendorID` column of the `PurchaseOrderTable` that might be of use to this query, but because you're using a `COALESCE` statement to filter values, a scan of the entire table is necessary to retrieve the information. The `COALESCE` operator is basically a way to take into account that a given value might be NULL and, if it is NULL, to provide an alternate value, possibly several alternate values. However, it's a function, and a function against a column within a WHERE clause, the JOIN criteria, or a HAVING clause is likely to lead to scans, so you need to get rid of the function. Because of this function, you can't simply add or modify the index because you'd still end up with a scan. You could try rewriting the query with an OR clause like this:

```
...WHERE   per.LastName LIKE @LastName AND
        poh.VendorID = @VendorID
        OR poh.VendorID = poh.VendorID…
```

But logically, that's not the same as the COALESCE operation. Instead, it's substituting one part of the WHERE clause for another, not just using the OR construct. So, you could rewrite the entire stored procedure definition like this:

```
CREATE OR ALTER PROCEDURE dbo.PurchaseOrderBySalesPersonName
    @LastName NVARCHAR(50),
    @VendorID INT = NULL
AS
IF @VendorID IS NULL
BEGIN
    SELECT poh.PurchaseOrderID,
           poh.OrderDate,
           pod.LineTotal,
           p.Name AS ProductName,
           e.JobTitle,
           per.LastName + ', ' + per.FirstName AS SalesPerson,
           poh.VendorID
    FROM Purchasing.PurchaseOrderHeader AS poh
        JOIN Purchasing.PurchaseOrderDetail AS pod
            ON poh.PurchaseOrderID = pod.PurchaseOrderID
        JOIN Production.Product AS p
            ON pod.ProductID = p.ProductID
        JOIN HumanResources.Employee AS e
            ON poh.EmployeeID = e.BusinessEntityID
        JOIN Person.Person AS per
            ON e.BusinessEntityID = per.BusinessEntityID
    WHERE per.LastName LIKE @LastName
    ORDER BY per.LastName,
             per.FirstName;
END
ELSE
BEGIN
    SELECT poh.PurchaseOrderID,
           poh.OrderDate,
           pod.LineTotal,
           p.Name AS ProductName,
           e.JobTitle,
           per.LastName + ', ' + per.FirstName AS SalesPerson,
           poh.VendorID
    FROM Purchasing.PurchaseOrderHeader AS poh
        JOIN Purchasing.PurchaseOrderDetail AS pod
            ON poh.PurchaseOrderID = pod.PurchaseOrderID
        JOIN Production.Product AS p
            ON pod.ProductID = p.ProductID
        JOIN HumanResources.Employee AS e
            ON poh.EmployeeID = e.BusinessEntityID
        JOIN Person.Person AS per
            ON e.BusinessEntityID = per.BusinessEntityID
    WHERE per.LastName LIKE @LastName
          AND poh.VendorID = @VendorID
    ORDER BY per.LastName,
             per.FirstName;
END
GO
```

Using the IF construct breaks the query in two. Running it with the same set of parameters resulted in a change in execution time from 434ms to 128ms (as measured in Extended Events), which is a fairly strong improvement. The reads went up from 8,671 to 9,243. While the execution time went down quite a lot, we had a small increase in reads. The execution plan is certainly different, as shown in Figure 27-9.

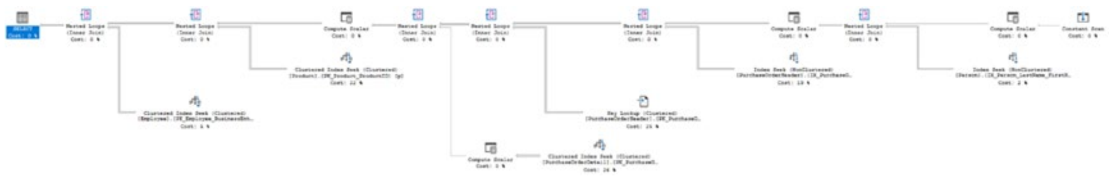


Figure 27-9. *New execution plan after breaking apart the query*

The two costliest operators are now different. There are no more scan operations, and all the join operations are now loop joins. But, a new data access operation has been added. You're now seeing a `Key Lookup` operation, as described in Chapter 12, so you have more tuning opportunities.

Fixing the Key Lookup Operation

Now that you know you have a key lookup, you need to determine whether any of the methods for addressing it suggested in Chapter 12 can be applied. First, you need to know what columns are being retrieved in the operation. This means accessing the properties of the `Key Lookup` operator. The properties show the `VendorID` and `OrderDate` columns. This means you only need to add those columns to the leaf pages of the index through the `INCLUDE` part of the nonclustered index. You can modify that index as follows:

```
CREATE NONCLUSTERED INDEX IX_PurchaseOrderHeader_EmployeeID
ON Purchasing.PurchaseOrderHeader
(
    EmployeeID ASC
)
INCLUDE
(
    VendorID,
    OrderDate
)
WITH DROP_EXISTING;
```

Applying this index results in a change in the execution plan and a modification in the performance. The previous structure and code resulted in 128ms. With this new index in place, the query execution time dropped to 110ms, and the reads have dropped to 7748. The execution plan is now completely different, as shown in Figure 27-10.

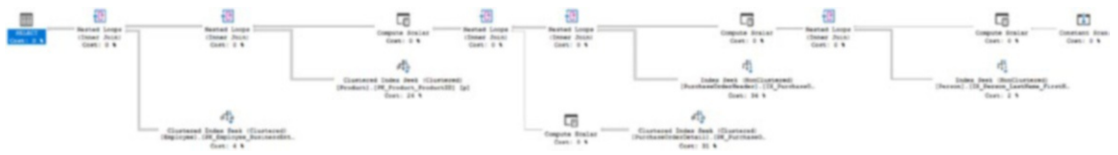


Figure 27-10. *New execution plan after modifying the index*

At this point there are nothing but nested loop joins and index seeks. There's not even a sort operation anymore despite the `ORDER BY` statement in the query. This is because the output of the index seek against the `Person` table is `Ordered` and the rest of the operations maintain that order. In short, you're largely in good shape as far as this query goes, but there were two queries in the procedure now.

Tuning the Second Query

Eliminating `COALESCE` allowed you to use existing indexes, but in doing this you effectively created two paths through your query. Because you've explored the first path only because you have used only the single parameter, you've been ignoring the second query. Let's modify the test script to see how the second path through the query will work.

```
EXEC dbo.PurchaseOrderBySalesPersonName @LastName = 'Hill%',
    @VendorID = 1496;
```

Running this query results in a different execution plan entirely. You can see the most interesting part of this in Figure 27-11.

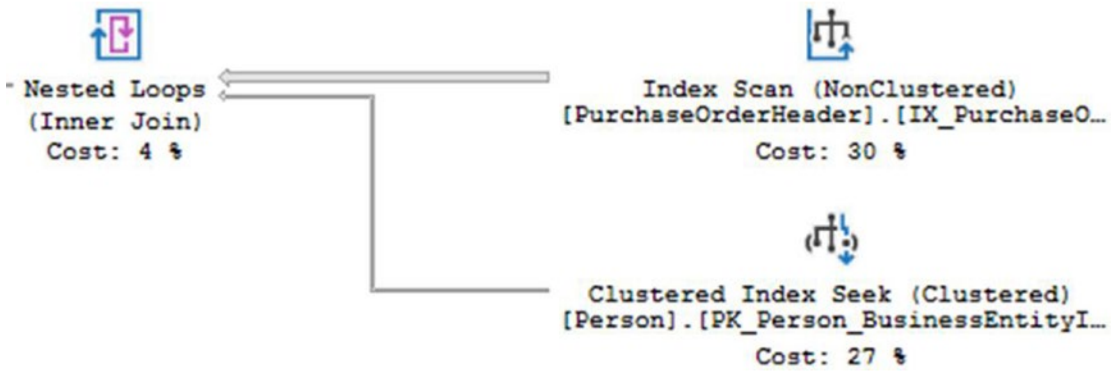

Nested Loops
(Inner Join)
Cost: 4 %
Index Scan (NonClustered)
[PurchaseOrderHeader].[IX_PurchaseO...
Cost: 30 %
Clustered Index Seek (Clustered)
[Person].[PK_Person_BusinessEntityI...
Cost: 27 %

Figure 27-11. *Execution plan for the other query in the procedure*

This new query has different behaviors because of the differences in the query. The main issue here is a clustered index scan against the `PurchaseOrderHeader` table. You're seeing a scan despite that there is an index on `VendorID`. Again, you can look to see what the output of the operator includes. This time, it's more than just two columns: `OrderDate`, `EmployeeID`, `PurchaseOrderID`. These are not very large columns, but they will add to the size of the index. You'll need to evaluate whether this increase in index size is worth the performance benefits of the elimination of the scan of the index. I'm going to go ahead and try it by modifying the index as follows:

```
CREATE NONCLUSTERED INDEX IX_PurchaseOrderHeader_VendorID
ON Purchasing.PurchaseOrderHeader
(
    VendorID ASC
)
INCLUDE
(
    OrderDate,
    EmployeeID,
    PurchaseOrderID
)
WITH DROP_EXISTING;
GO
```

Prior to applying the index, the execution time was around 4.3ms with 273 reads. After applying the index, the execution time dropped to 2.3ms and 263 reads. The execution plan now looks like Figure 27-12.

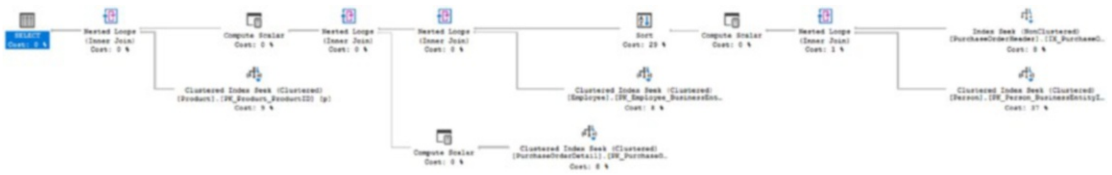

SELECT
Nested Loops (Inner Join)
Compute Scalar
Clustered Index Seek (Clustered) [Product].[PK_Product_ProductID] [p]
Nested Loops (Inner Join)
Nested Loops (Inner Join)
Compute Scalar
Clustered Index Seek (Clustered) [PurchaseOrderDetail].[PK_PurchaseO...
Sort
Clustered Index Seek (Clustered) [Employee].[PK_Employee_BusinessEn...
Compute Scalar
Nested Loops (Inner Join)
Index Seek (NonClustered) [PurchaseOrderHeader].[IX_PurchaseO...
Clustered Index Seek (Clustered) [Person].[PK_Person_BusinessEntityI...

Figure 27-12. *The second execution plan after modifying the index*

The new execution plan consists of index seeks and nested loops joins. There is a sort operator, the second costliest in the plan, ordering the data by `LastName` and `FirstName`. Getting this to be taken care of by the retrieval process might help to improve performance, but I've had a fairly successful tuning to this point, so I'll leave it as is for now.

One additional consideration should be made for the split query. When the optimizer processes a query like this, both statements will be optimized for the parameter values passed in. Because of this, you may see bad execution plans, especially for the second query that uses the `VendorID` for filtering, because of parameter sniffing gone bad. To avoid that situation, one additional tuning effort should be made.

Creating a Wrapper Procedure

Because you've created two paths within the procedure to accommodate the different mechanisms of querying the data, you have the potential for getting bad parameter sniffing because both paths will be compiled, regardless of the parameters passed. One mechanism around this is to run the procedure you have into a wrapper procedure. But first, you have to create two new procedures, one for each query like this:

```
CREATE OR ALTER PROCEDURE dbo.PurchaseOrderByLastName @LastName
NVARCHAR(50)
AS
SELECT poh.PurchaseOrderID,
       poh.OrderDate,
       pod.LineTotal,
       p.Name AS ProductName,
       e.JobTitle,
       per.LastName + ', ' + per.FirstName AS SalesPerson,
       poh.VendorID
FROM Purchasing.PurchaseOrderHeader AS poh
    JOIN Purchasing.PurchaseOrderDetail AS pod
        ON poh.PurchaseOrderID = pod.PurchaseOrderID
    JOIN Production.Product AS p
        ON pod.ProductID = p.ProductID
    JOIN HumanResources.Employee AS e
        ON poh.EmployeeID = e.BusinessEntityID
    JOIN Person.Person AS per
        ON e.BusinessEntityID = per.BusinessEntityID
WHERE per.LastName LIKE @LastName
ORDER BY per.LastName,
         per.FirstName;
GO

CREATE OR ALTER PROCEDURE dbo.PurchaseOrderByLastNameVendor
    @LastName NVARCHAR(50),
    @VendorID INT
AS
SELECT poh.PurchaseOrderID,
       poh.OrderDate,
       pod.LineTotal,
       p.Name AS ProductName,
       e.JobTitle,
       per.LastName + ', ' + per.FirstName AS SalesPerson,
       poh.VendorID
FROM Purchasing.PurchaseOrderHeader AS poh
    JOIN Purchasing.PurchaseOrderDetail AS pod
        ON poh.PurchaseOrderID = pod.PurchaseOrderID
    JOIN Production.Product AS p
        ON pod.ProductID = p.ProductID
    JOIN HumanResources.Employee AS e
        ON poh.EmployeeID = e.BusinessEntityID
    JOIN Person.Person AS per
```

```
        ON e.BusinessEntityID = per.BusinessEntityID
WHERE per.LastName LIKE @LastName
      AND poh.VendorID = @VendorID
ORDER BY per.LastName,
         per.FirstName;
GO
```

Then you have to modify the existing procedure so that it looks like this:

```
CREATE OR ALTER PROCEDURE dbo.PurchaseOrderBySalesPersonName
    @LastName NVARCHAR(50),
    @VendorID INT = NULL
AS
IF @VendorID IS NULL
BEGIN
    EXEC dbo.PurchaseOrderByLastName @LastName;
END
ELSE
BEGIN
    EXEC dbo.PurchaseOrderByLastNameVendor @LastName, @VendorID;
END
GO
```

With that in place, regardless of the code path chosen, the first time these queries are called, each procedure will get its own unique execution plan, avoiding bad parameter sniffing. And, this won't negatively impact performance time. If I run both the queries now, the results are approximately the same. This pattern works very well for a small number of paths. If you have some large number of paths, certainly more than 10 or so, this pattern breaks down, and you may need to look to dynamic execution methods.

Taking the performance from 434ms to 110ms or 2.3ms, depending on our new queries, is a pretty good reduction in execution time, and we also had equally big wins on reads. If this query were called hundreds of times in a minute, that level of reduction would be quite serious indeed. But, you should always go back and assess the impact on the overall database workload.

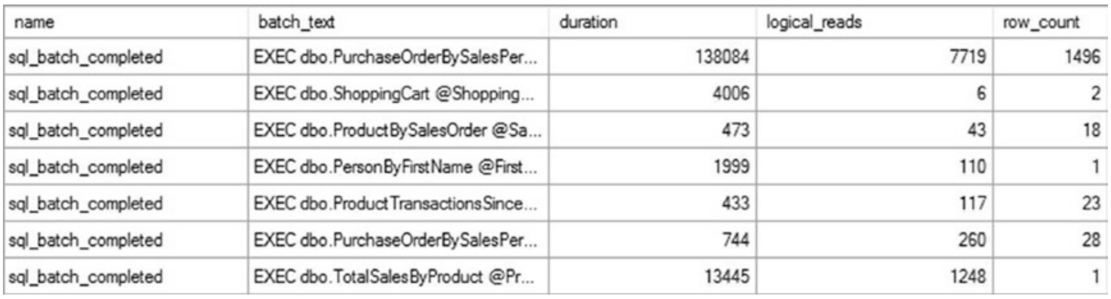
Analyzing the Effect on Database Workload

Once you've optimized the worst-performing query, you must ensure that it doesn't hurt the performance of the other queries; otherwise, your work will have been in vain.

To analyze the resultant performance of the overall workload, you need to use the techniques outlined in Chapter 15. For the purposes of this small test, reexecute the complete workload and capture extended events to record the overall performance.

Tip For proper comparison with the original extended events, please ensure that the graphical execution plan is off.

Figure 27-13 shows the corresponding Extended Events output captured.



name	batch_text	duration	logical_reads	row_count
sql_batch_completed	EXEC dbo.PurchaseOrderBySalesPer...	138084	7719	1496
sql_batch_completed	EXEC dbo.ShoppingCart @Shopping...	4006	6	2
sql_batch_completed	EXEC dbo.ProductBySalesOrder @Sa...	473	43	18
sql_batch_completed	EXEC dbo.PersonByFirstName @First...	1999	110	1
sql_batch_completed	EXEC dbo.ProductTransactionsSince...	433	117	23
sql_batch_completed	EXEC dbo.PurchaseOrderBySalesPer...	744	260	28
sql_batch_completed	EXEC dbo.TotalSalesByProduct @Pr...	13445	1248	1

Figure 27-13. *The Extended Events output showing the effect of optimizing the costliest query on the complete workload*

It's possible that the optimization of the worst-performing query may hurt the performance of some other query in the workload. However, as long as the overall performance of the workload is improved, you can retain the optimizations performed on the query. In this case, the other queries were not impacted. But now, there is a query that takes longer than the others. It too might need optimization, and the whole process starts again. This is also a place where having the Query Store in place so that you can look for regression or changes in behavior easily becomes a great resource.

Iterating Through Optimization Phases

An important point to remember is that you need to iterate through the optimization steps multiple times. In each iteration, you can identify one or more poorly performing queries and optimize the query or queries to improve the performance of the overall workload. You must continue iterating through the optimization steps until you achieve adequate performance or meet your service level agreement (SLA).

Besides analyzing the workload for resource-intensive queries, you must also analyze the workload for error conditions. For example, if you try to insert duplicate rows into a table with a column protected by the unique constraint, SQL Server will reject the new rows and report an error condition to the application. Although the data was not entered into the table and no useful work was performed, valuable resources were used to determine that the data was invalid and must be rejected.

To identify the error conditions caused by database requests, you will need to include the following in your Extended Events session (alternatively, you can create a new session that looks for these events in the `errors` or `warnings` category):

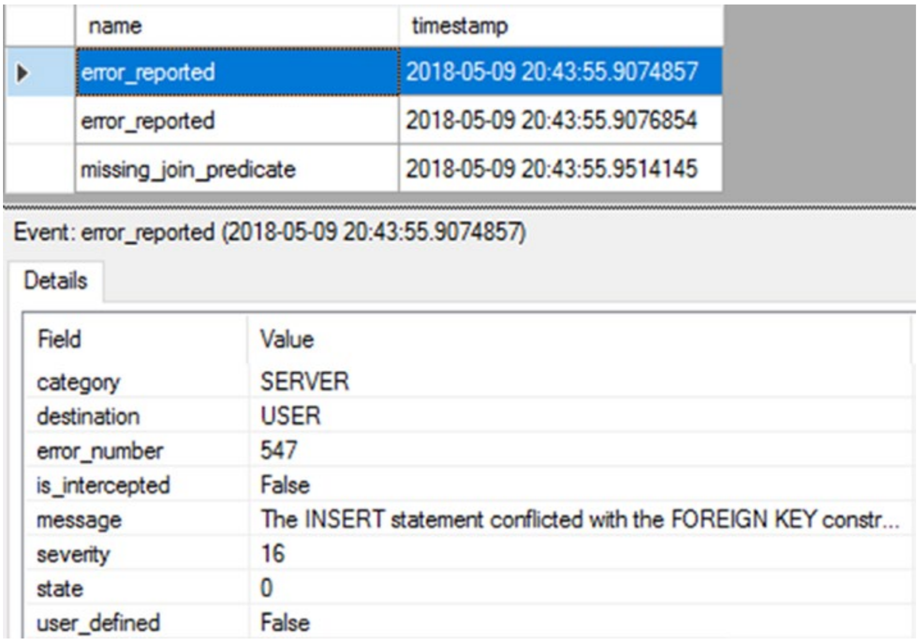
- `error_reported`
- `execution_warning`
- `hash_warning`
- `missing_column_statistics`
- `missing_join_predicate`
- `sort_warning`
- `hash_spill_details`

For example, consider the following SQL queries:

```
INSERT  INTO Purchasing.PurchaseOrderDetail
        (PurchaseOrderID,
         DueDate,
         OrderQty,
         ProductID,
         UnitPrice,
         ReceivedQty,
         RejectedQty,
         ModifiedDate
        )
VALUES  (1066,
         '1/1/2009',
         1,
         42,
         98.6,
         5,
         4,
         '1/1/2009'
        ) ;
GO

SELECT  p.[Name],
        psc.[Name]
FROM    Production.Product AS p,
        Production.ProductSubCategory AS psc ;
GO
```

Figure 27-14 shows the corresponding session output.



name	timestamp
error_reported	2018-05-09 20:43:55.9074857
error_reported	2018-05-09 20:43:55.9076854
missing_join_predicate	2018-05-09 20:43:55.9514145

Event: error_reported (2018-05-09 20:43:55.9074857)

Details

Field	Value
category	SERVER
destination	USER
error_number	547
is_intercepted	False
message	The INSERT statement conflicted with the FOREIGN KEY constr...
severity	16
state	0
user_defined	False

Figure 27-14. *Extended Events output showing errors raised by a SQL workload*

From the Extended Events output in Figure 27-14, you can see that the two errors I intentionally generated occurred.

- `error_reported`
- `missing_join_predicate`

The `error_reported` error was caused by the `INSERT` statement, which tried to insert data that did not pass the referential integrity check; namely, it attempted to insert `Productld = 42` when there is no such value in the `Production.Product` table. From the `error_number` column, you can see that the error number is 547. The `message` column shows the full description for the error. It's worth noting, though, that `error_reported` can be quite chatty with lots of data returned and not all of it useful.

The second type of error, `missing_join_predicate`, is caused by the `SELECT` statement.

```
SELECT p.Name,
       c.Name
FROM Production.Product AS p,
     Production.ProductSubcategory AS c;
```

If you take a closer look at the `SELECT` statement, you will see that the query does not specify a `JOIN` clause between the two tables. A missing join predicate between the tables usually leads to an inaccurate result set and a costly query plan. This is what is known as a *Cartesian join*, which leads to a *Cartesian product*, where every row from one table is combined with every row from the other table. You must identify the queries causing such events in the `Errors and Warnings` section and implement the necessary fixes. For instance, in the preceding `SELECT` statement, you should not join every row from the `Production.ProductCategory` table to every row in the `Production.Product` table—you must join only the rows with matching `ProductCategorylD`, as follows:

```
SELECT p.Name,
       c.Name
FROM Production.Product AS p
    JOIN Production.ProductSubcategory AS c
        ON p.ProductSubcategoryID = c.ProductSubcategoryID;
```

Even after you thoroughly analyze and optimize a workload, you must remember that workload optimization is not a one-off process. The workload or data distribution on a database can change over time, so you should periodically check whether your queries are optimized for the current situation. It's also possible that you may identify shortcomings in the design of the database itself. Too many joins from overnormalization or too many columns from improper denormalization can both lead to queries that perform badly, with no real optimization opportunities. In this case, you will need to consider redesigning the database to get a more optimized structure.

Summary

As you learned in this chapter, optimizing a database workload requires a range of tools, utilities, and commands to analyze different aspects of the queries involved in the workload. You can use Extended Events to analyze the big picture of the workload and identify the costly queries. Once you've identified the costly queries, you can use the execution plan and various SQL commands to troubleshoot the problems associated with the costly queries. Based on the problems detected with the costly queries, you can apply one or more sets of optimization techniques to improve the query performance. The optimization of the costly queries should improve the overall performance of the workload; if this does not happen, you should roll back the change or changes.

In the next chapter, I summarize the performance-related best practices in a nutshell. You'll be able to use this information as a quick and easy-to-read reference.

CHAPTER 28

SQL Server Optimization Checklist

If you have read through the previous 27 chapters of this book, then you understand the major aspects of performance optimization. You also understand that it is a challenging and ongoing activity.

What I hope to do in this chapter is to provide a performance-monitoring checklist that can serve as a quick reference for database developers and DBAs when in the field. The idea is similar to the notion of tear-off cards of *best practices*. This chapter does not cover everything, but it does summarize, in one place, some of the major tuning activities that can have a quick and demonstrable impact on the performance of your SQL Server systems. I have categorized these checklist items into the following sections:

- Database design
- Configuration settings
- Database administration
- Database backup
- Query design

Each section contains a number of optimization recommendations and techniques. Where appropriate, each section also cross-references specific chapters in this book that provide more detailed information.

© Grant Fritchey 2018
G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_28

Database Design

Database design is a broad topic, and it can't be given due justice in a small section in this query tuning book; nevertheless, I advise you to keep an eye on the following design aspects to ensure that you pay attention to database performance from an early stage:

- Use entity-integrity constraints.
- Maintain domain and referential integrity constraints.
- Adopt index-design best practices.
- Avoid the use of the `sp_` prefix for stored procedure names.
- Minimize the use of triggers.
- Put tables into in-memory storage.
- Use columnstore indexes.

Use Entity-Integrity Constraints

Data integrity is essential to ensuring the quality of data in the database. An essential component of data integrity is *entity integrity,* which defines a row as a unique entity for a particular table; that is, every row in a table must be uniquely identifiable. The column or columns serving as the unique row identifier for a table must be represented as the primary key of the table.

Sometimes, a table may contain an additional column (or columns) that also can be used to uniquely identify a row in the table. For example, an `Employee` table may have the `EmployeeID` and `SocialSecurityNumber` columns. The `EmployeeID` column serves as the unique row identifier, and it can be defined as the *primary key*. Similarly, the `SocialSecurityNumber` column can be defined as the *alternate key*. In SQL Server, alternate keys can be defined using unique constraints, which are essentially the younger siblings to primary keys. In fact, both the unique constraint and the primary key constraint use unique indexes behind the scenes.

It's worth noting that there is honest disagreement regarding the use of a natural key (for example, the `SocialSecurityNumber` column in the previous example) or an artificial key (for example, the `EmployeeID` column). I've seen both designs succeed, but each approach has strengths and weaknesses. Rather than suggest one over the other,