OPTIMIZING QUERIES IN SQL SERVER 2008

Professor Ph.D. Ion LUNGU[1], Nicolae MERCIOIU[2], Victor VLĂDUCU[3]

[1]Academy of Economic Studies, Bucharest, Romania,
[2]Prosecutor's Office attached to the High Court of Cassation and Justice
[3]Prosecutor's Office attached to Vâlcea Court

***Abstract**: Starting from the need to develop efficient IT systems, we intend to review the optimization methods and tools that can be used by SQL Server database administrators and developers of applications based on Microsoft technology, focusing on the latest version of the proprietary DBMS, SQL Server 2008. We'll reflect on the objectives to be considered in improving the performance of SQL Server instances, we will tackle the mostly used techniques for analyzing and optimizing queries and we will describe the "Optimize for ad hoc workloads", "Plan Freezing" and "Optimize for unknown" new options, accompanied by relevant code examples.*

***Keywords:** Query, SQL Server 2008, Optimization.*

JEL Classification: C88, D80

1. INTRODUCTION

Improving the performance of a DBMS includes identifying bottlenecks and their causes, using appropriate techniques and tools for solving them and evaluating the added performance obtained. It is generally accepted that aiming to achieve maximum theoretical performance is unrealistic and counterproductive, as the investment cost beyond reaching the "good enough" time increases exponentially with the performance gain.

Most times, efficiency and performance are the last criteria considered when designing and developing new applications using a database. These criteria become important only after the system goes into production. Sometimes it appears that the application does not display the information requested to the database in a reasonable time or completely fails to display it, the set timeout being exceeded. The reasons may be related to the application design, but in many cases the DBMS does not return the data quickly enough, due to the nonuse of indexes, deficient design of the queries and/or database schema, excessive fragmentation, use of inaccurate statistics, failure to reuse the execution plans or improper use of cursors.

Once the full set of running SQL queries captured, one should identify the queries exerting high pressure on system resources and those running the slowest. The component dealing with optimizing queries in SQL Server attempts to determine the most efficient way to execute a query, taking into account possible plans, and decides which one is the best. The optimization based on actual cost strategy involves the estimation of a runtime "cost" for every possible plan, allowing choosing the execution plan with the lowest "cost" in terms of disk I/O operations, CPU load, memory load etc.

2. GOALS FOR IMPROVING PERFORMANCE OF A SQL SERVER INSTANCE

The elements to be focused on for increasing performance have been found to be, in ascending order of importance: Windows operation system, SQL Server instance, hardware, database and application. According to Microsoft, the most important objectives to be considered in order to improve the performance of SQL Server are:

- Designing an efficient data schema.
- Optimizing queries.
- Optimizing indexes, stored procedures and transactions.
- Analyzing execution plans and avoiding recompiling them.
- Monitoring access to data.

Designing an efficient data schema requires initial normalization and subsequent de-normalization (e.g. persons/institutions address), if necessary. A reverse approach would involve additional activities meant to insure the data consistency. Another issue to be considered is the use of the declarative referential integrity. This approach is more efficient than using triggers employing system temporary tables. The use of primary, foreign and unique key constraints contributes to the creation of effective execution plans. It is also recommended to define data types as close as possible to the real ones, given that implicit and explicit conversions are intensive consumers of computational resources.

A very important aspect comes from the need to use indexed views, when the information is not frequently updated, as indexed views are stored physically as a table.

In optimizing indexes and stored procedures the fact that they allow a rapid response to selection operations but slow down insert, update and delete operations should be taken into consideration. Generally, the creation and use of indexes should be balanced among read and write operations i.e. indexes improve read operations but may positively or negatively alter write operations. Indexes have to be also created for all foreign keys on tables that are often queried and do not contain image or bit type fields. As regards transactions, they should be kept as short as possible. Transactions requiring user intervention are to be avoided and data validation is recommended before starting the transaction.

Stored procedures must include the SET NOCOUNT ON command. This command prevents sending the message regarding the number of affected records for each operation carried out within this procedure.

It is important to analyze and run execution plans on representative data so that the best plan suggested by the optimizer may be chosen. In this regard, plans involving scanning tables and indexes should be avoided. Scanning is worthy only for tables containing up to hundreds of records. Also, major CPU and memory resources consumers are the records sorting and filtering operations. In general, recompiling execution plans should be avoided due to the loss in performance. This can be avoided by using parameterized queries and stored procedures and avoiding cursors over temporary tables. However, recompiling plans may bring benefits when the optimizer is able create a more efficient execution plan.

It is very important to monitor through statistics (if they are kept up to date) and use the profiler for queries running a long time, as well as for scanning and monitoring the use of resources. The queries optimization will be tackled in detail below.

3. QUERY ANALYSIS AND OPTIMIZATION TECHNIQUES

Analysis and optimization techniques require individual approach but also of the whole set of queries, on the premise that although individual queries can be optimized enough, the whole set performance may be poor.

First and most important optimization technique is to limit the amount of returned data by limiting the number of records (the WHERE clause) and fields specified in the SELECT list. This will lead to an efficient use of the indexes. In principle, a WHERE clause should be selective as it is the one establishing the use of indexes on columns.

For an efficient use of indexes, according to the utilization requirements for the system, the SQL Server provides clustered and non-clustered type indexes. Within the Online Transaction Processing schemes — whose tables are frequently updated — the clustered indexes are recommended, but on as few columns as possible. A large number of indexes in these systems

will affect the performance of the INSERT, UPDATE, DELETE and MERGE commands, as all indexes must be accordingly adjusted when data in the tables are modified. Clustered type indexes are effective when operators like BETWEEN, >, >=, < and <= are used, because after the record containing the first value is found, subsequent records with indexed values are physically adjacent. Also, if the query contains clauses like JOIN, ORDER BY or GROUP BY, the clustered type indexes are most appropriate. Non-clustered indexes use is recommended only for databases where updates are infrequent and gives the optimal solution for the "exact match" type queries. Queries can effectively use indexes only if within the WHERE clause functions and arithmetic operations are as much as possible avoided. For example, using the LIKE clause instead of the SUBSTRING function will cause the optimizer to use the index on the Name column:

```
SELECT Name
FROM AdventureWorks.Production.Location
WHERE SUBSTRING(Name,1,1) ='P'
```

The recommended option is:

```
SELECT Name
FROM AdventureWorks.Production.Location
WHERE Name LIKE 'P%'
```

Likewise, the exclusion conditions <>, ! =, !>, !<, NOT EXISTS, NOT IN, NOT LIKE IN, OR or the LIKE example '% <literal>' will determine the DBMS's optimizer not to use the indexes on columns in the WHERE clause. Instead, the inclusion conditions, BETWEEN and the LIKE example '<literal>%', allow the optimizer to increase query performance because the SQL Server will find the record in the index and will return the adjacent records too, as long as the condition in the WHERE clause remains true.

Another optimization method is using, where possible, the BETWEEN clause instead the IN or OR conditions. The SQL Server 2008 will resolve the IN condition by accessing the index for a number of times equal to the number of values by which to search. Using the BETWEEN clause, the optimizer will turn it into a pair of conditions >= <=, the index being accessed once.

```
SELECT *
FROM AdventureWorks.Purchasing.PurchaseOrderDetail
WHERE PurchaseOrderID IN (651,652,653)
```

The recommended option is:

```
SELECT *
FROM AdventureWorks.Purchasing.PurchaseOrderDetail
WHERE PurchaseOrderID BETWEEN 651 AND 653
```

While the two previous examples have the same execution plan and use the clustered type index, in the first case the SQL Server version will resolve the IN clause using three values in three OR conditions and the index will be accessed three times. In the second option, the index will be accessed once, from the first to the last record satisfying the condition in the WHERE clause. In general, using the BETWEEN clause instead of the IN/OR conditions will reduce the number of logical reads. Using arithmetic operators on a column in the WHERE clause will make the optimizer not to use the column index.

```
SELECT PurchaseOrderID
FROM AdventureWorks.Purchasing.PurchaseOrderDetail
WHERE PurchaseOrderID * 3 =1953
```

The recommended option is:

```
SELECT PurchaseOrderID
FROM AdventureWorks.Purchasing.PurchaseOrderDetail
WHERE PurchaseOrderID =1953/3
```

The difference in actual cost for the previous queries, run on a SQL Server 2008 Express, is

about 70 percent. The SQL Server 2008 optimizer dynamically determines a query processing strategy based on the current structure of the table and indexes, as well as on the existing data. However, this process can be overwritten if the optimizer suggestions are used, its behavior becoming static as the query processing strategy will not be permanently updated and self-parametrization will be omitted. In general, the cost-effective strategy based on the data distribution, indexes and other factors is efficient and it is not recommended to force the optimizer to execute specific strategies, due to potential loss of performance.

In order to avoid the execution of queries involving intensive resource consumption, it is recommended to verify the existence of data using the EXISTS () function instead of the COUNT (*) function. In the first version, the SQL Server 2008 will scan and stop at the first record meeting the criterion between the EXISTS brackets, while using COUNT (*), the DBMS will scan all table records. Implicit data type conversion is to be avoided, meaning that variables declared in the query must be of the same type with the columns to be compared with. In the SQL Server, implicit data conversion is done following the rules for data types precedence. Normally, the low precedence data type is converted into a higher precedence data type. Implicit conversion worsens the query performance duet o the inefficient execution plan, materialized additional CPU load. An alternative is using the CAST and CONVERT functions.

It is recommended to avoid local variables in scripts containing multiple queries, especially when the variable values are transmitted from one query to another. The SQL Server optimizer will generate an inefficient execution plan if the WHERE clause contains local variables. Between queries in the following script the performance difference is about 50%.

```
USE AdventureWorks
DECLARE @id INT = 1
SELECT *
FROM    Sales.SalesOrderDetail
JOIN Sales.SalesOrderHeader
ON Sales.SalesOrderHeader.SalesOrderID = Sales.SalesOrderDetail.SalesOrderID
WHERE   Sales.SalesOrderHeader.SalesOrderID >= @id
SELECT *
FROM Sales.SalesOrderDetail
JOIN Sales.SalesOrderHeader
ON Sales.SalesOrderHeader.SalesOrderID = Sales.SalesOrderDetail.SalesOrderID
WHERE Sales.SalesOrderHeader.SalesOrderID >=1
```

For each query in an executed stored procedure or script, the SQL Server will return the number of affected records (2323333 row(s) affected).To save resources, it is preferable to use the SET NOCOUNT ON <Queries> SET NOCOUNT OFF sequence.

4. NEW OPTIMIZING OPTIONS IN SQL SERVER 2008

Some applications run scripts once in a session (*ad hoc workloads*). The fact that the SQL Server stores each execution plan for a possible reuse results in an excessive increase of the used memory, leading to a lower efficiency of the instance. In order to solve this problem, Microsoft introduced the option of optimizing this kind of queries — *optimize for ad hoc workload* — for all its variants of SQL Server 2008. Once this option activated, when the script is compiled for the first time, the DBMS will save only a small part of the ad hoc query execution plan. This part will help in a later phase to determine whether that script has been compiled. If the script is re-executed, the SQL Server 2008 removes that small part of the originally compiled plan and will recompile the script in order to get the complete execution plan. On first running of the following script, the second query will return a single result, showing that the first query execution plan was memorized.

```
DBCC FREEPROCCACHE
DBCC DROPCLEANBUFFERS
GO
USE AdventureWorks
GO
```

```
SELECT * FROM Person.Contact
GO
SELECT usecounts, cacheobjtype, objtype, text
FROM sys.dm_exec_cached_plans
CROSS APPLY sys.dm_exec_sql_text(plan_handle)
WHERE usecounts > 0 AND
text like '%SELECT * FROM Person.Contact%'
ORDER BY usecounts DESC;
GO
```

Before re-executing the script, we activate the optimization option and change slightly the LIKE clause in the second query (`text like 'SELECT * FROM Person.Contact%'`):

```
SP_CONFIGURE 'show advanced options',1
RECONFIGURE
GO
SP_CONFIGURE 'optimize for ad hoc workloads',1
RECONFIGURE
GO
```

In this case, the second query will not return any results, reflecting the fact that no execution plan has been memorized for the first query. If we went on re-executing the script, we would notice that the SQL Server 2008 is saving the query execution plan, regardless of the state of the discussed option. Thus we can conclude that after activating the *optimize for ad hoc workload* option only new execution plans are affected and those already memorized are not affected.

Another feature introduced in the SQL Server 2008 version is the one allowing a direct creation of the guide (*Plan Freezing*) for any execution plan for a query existing in the SQL Server memory. This feature adds to the support extension for the execution plans guides for all DML commands. Note that in the 2005 version of the SQL Server, this feature was available only for the SELECT command and involved the possibility to specify suggestions (guides) for the queries that can not be changed directly from the application.

In the following example we will run a query first in order to get the execution plan that will later be "frozen".

```
USE AdventureWorks
GO
DBCC FREEPROCCACHE
GO
SET STATISTICS XML ON
EXEC sp_executesql
N'SELECT  *
FROM    Sales.SalesOrderDetail
        JOIN Sales.SalesOrderHeader
        ON Sales.SalesOrderHeader.SalesOrderID = Sales.SalesOrderDetail.SalesOrderID'
SET STATISTICS XML OFF
GO
```

Next we will "freeze" the previously created execution plan.

```
DECLARE @plan_handle varbinary(1000)
SELECT @plan_handle = plan_handle
FROM sys.dm_exec_query_stats qs
cross apply sys.dm_exec_sql_text(qs.sql_handle) sqt
WHERE text like '%SalesOrderDetail%'
SELECT @plan_handle
EXEC sp_create_plan_guide_from_handle 'TEST_Plan_Guide_1', @plan_handle=@plan_handle
```

The `sp_create_plan_guide_from_handle` procedure allows us to ensure that the optimizer will always use the same plan for a specific query. We'll use the "frozen" execution plan and run the query again:

```
SET STATISTICS XML ON
EXEC sp_executesql
N'SELECT  *
FROM    Sales.SalesOrderDetail
        JOIN Sales.SalesOrderHeader
```

```
        ON Sales.SalesOrderHeader.SalesOrderID = Sales.SalesOrderDetail.SalesOrderID'
SET STATISTICS XML OFF
GO
```

We can easily see that the plan used is the one previously created.

The *Optimize for unknown* option causes the optimizer to use a standard algorithm to be permanently employed to generate a query plan. Instead of using the actual values of the parameters submitted by the application, the optimizer will consult all the statistical data to determine which values could be used to generate an effective plan.

The approach known from the earlier versions of SQL Server was to use parameterized queries which allowed saving and reusing the execution plans, avoiding recompilation. The problem arose when the parameters values sent in recalling queries were not comparable to those originally transmitted. On the first execution, the SQL Server would compile and save an effective plan for those values. Subsequently, the same plan would cause a suboptimal query execution.

```
DECLARE @C1 INT
DECLARE @C2 INT
SET @C1 = 5000000
SET @C2 = 400
SELECT * FROM TEST WHERE COL1 > @C1 or COL2 > @C2 ORDER BY COL1;
```

When the plan created by the previous query is deleted and the non-typical values 1 and 90 are given to the two variables C1 and C2, the optimizer creates, upon the re-execution of the query, a plan that may be different from the first. In practice, as the query is the same, the optimizer will keep and use the plan originally created, with the corresponding loss of performance, for the subsequent values of the C1 and C2 variables. In order to avoid this situation and benefit of the query parameterization, the previous query may be rewritten as:

```
SELECT * FROM TEST WHERE COL1 > @C1 or COL2 > @C2 ORDER BY COL1
OPTION (OPTIMIZE FOR (@C1 UNKNOWN, @C2 UNKNOWN))
```

5. CONCLUSIONS

The optimization process is iterative and includes steps like identifying bottlenecks, solving them, measuring the impact of changes and reassessing the system from the first step as to determine if satisfactory performance is achieved. This process allows a gradual improvement of the performance but we should always keep in mind that performance depends on the amount of data and the distribution of users' activities within the application. These elements are dynamic in time so regular performance review is needed.

There are many aspects that should be addressed in order to achieve optimal performance of queries and SQL instance.

In principle, superior performance can be obtained by writing an efficient code at the application level and properly using the design and database development techniques. Nevertheless, changing the SQL Server configuration will not result in a significant improvement of the performance.

REFERENCES

[1] G. Fritchey, S. Dam, "*SQL Server 2008 Query Performance Tuning Distilled*" Apress, 2009.
[2] MSDN, "*Data Type Conversion*".
[3] MSDN, "*Microsoft Patterns & Practices*".
[4] Pinal Dave, "*Journey to SQL Authority* ".