**Tips for Boosting SQL Server Query Performance**

1. **Locate I/O Bottlenecks**

Use the DMF sys.dm\_io\_virtual\_file\_stats() to locate any areas in which you have excessive physical I/O or excessive stalls on that I/O. When you find that you have many physical I/O bottlenecks occurring, your first instinct should be to find the queries that are causing all the physical I/O, and then try to tune them before you add more hardware.

1. **Root Out Problem Queries**

sys.dm\_exec\_query\_stats

sys.dm\_exec\_cached\_plans

CROSS APPLY sys.dm\_exec\_sql\_text

1. **Monitor Index Usage**

The sys.dm\_db\_index\_operational\_stats() DMF is a widely underutilized source of information. It can provide you valuable information about your index usage.

1. **Separate Data and Log Files**

One of the most basic but often disregarded rules for good performance is to separate the data and the log files onto separate physical drive arrays whenever possible. This is especially true when you use DAS, but it also applies to a SAN.

1. **Use Separate Staging Databases**
2. **Pay Attention to Log Files**
3. **Minimize tempdb Contention**
4. **Change the MAX Memory Limit**
5. **Just Say No to Shrinking Data Files**
6. **Don’t share the SQL server hardware with other services**Other workloads are running on the same server where SQL Server is running, memory and other hardware resources will be shared among this workload. In this condition it will be more difficult to identify the cause of poor performances as they arise.
7. **Use Multiple Disk Controllers**  
   SQL Server can take advantage from scattering data across multiple disk drives.
8. **Use the Appropriate RAID Configuration**  
   When it comes to choosing a RAID (Redundant Array of Independent Disks) level, you may consider cost, performance, and availability requirements: RAID 5 is cheaper than RAID 0+1, and RAID 5 performs better for read operations than write operations. RAID 0+1 is more expensive and performs better for write-intensive operations.  
   If possible you should choose hardware-level RAID rather than software RAID. Software RAID is usually cheaper but uses CPU cycles while RAID controllers have onboard logic that will offset this workload from the CPU.
9. **Provide a separate disk for heavily used tables and indexes**If you have heavily accessed tables or indexes, you will boost performance by allocating those objects in their own file group on a separate physical disk.
10. **Separate OLAP and OLTP Workloads**OLAP (Online Analytical Processing) and OLTP (Online Transaction Processing) workloads on the same server have to be designed to not interfere with each other. OLAP and reporting workloads tend to be characterized by less frequent but long-running queries. OLTP workloads, on the other hand, tend to be characterized by lots of small transactions that return something to the user in less than a second. consider creating a reporting server that supports the OLAP and reporting workloads.
11. **Use fixed size databases**  
    If you allocate disk space for a database while creating it you can be confident enough that the allocated space will be contiguous and therefore you will get the best possible performances. Instead if you set the Autogrow option the disk space will be allocated only when needed and will be probably very fragmented.  A fragmented database will perform worse that a contiguous one. So, especially in production, it is better to allocate space when you first create a database.
12. **Put tempdb on a separate disk**  
    The tempdb database is a temporary storage area that is used when performing operations such as GROUP BY or ORDER BY. Keeping tempdb on a separate disk will ensure that such operation will not have a negative impact on the performance of other database operations.
13. **Separate data and logs on different physical disks**  
    Database and logs have different usage patterns: database is read and written in an almost random way, while logs are mostly written sequentially. Separating them on different physical disks allows the operation to be executed with the best possible performance.
14. **Use table partitioning**  
    Partitioning allows you to keep portions of the same table on different physical disks. Using a partition to separate current data from historical data, you can keep all data on the same table. But keep just current data on your faster disks and therefore improve your query performances.
15. **Create indexes**  
    Indexes allow searching for data inside database tables in the most optimized way, and it is very important that all necessary indexes are created for the queries that are going to be served by the database engine.  Consider creating indexes on columns frequently used in the WHERE, ORDER BY, and GROUP BY clauses. These columns are the best candidates for indexes.
16. **Create clustered indexes**  
    Create clustered indexes instead of non-clustered in order to increase the performance of the queries that return a range of values and for the queries that contain the GROUP BY or ORDER BY clauses that return the sort results.  Since a table can have only one clustered index, you should choose the columns for this index very carefully. Analyze all your queries, choose most frequently used queries and include into the clustered index only those columns which provide the most performance benefits from your creation.
17. **Create non-clustered indexes**  
    Create non-clustered indexes to increase performance of the queries that return fewer  rows and where the index has good selectivity. You should consider non-clustered index creation carefully because each index can take up disk space and has impact on data modification.
18. **Rebuild indexes periodically**  
    While you update, delete and create records in your tables your indexes becomes fragmented and performance may degrade over time. You should consider rebuilding indexes periodically in order to keep performance at the best level.
19. **Use covering indexes**  
    A covering index is an index that includes all the columns referenced in the query. Covering indexes can improve performance because all the data for the query is contained within the index itself and only the index pages–not the data pages–will be used to retrieve the data. Covering indexes can bring a lot of performance improvement  because it can save a huge amount of I/O operations.
20. **Drop indexes that are not used**  
    Limit the number of indexes if your application updates data very frequently. Because each index takes up disk space and slows the adding, deleting, and updating of rows, you should create new indexes only after analyzing data usage, the types and frequencies of queries performed and how your queries will use the new indexes. In many cases, the speed advantages of creating the new indexes outweigh the disadvantages of additional space used and slowly rows modification.  
    Use Index Wizard to identify indexes that are not used in your queries.
21. **Retrieve only the data you need**  
    Sometimes you may be tempted to use SELECT \* FROM … when writing your queries, this way you will retrieve all fields in a table when you only need some. In order to reduce the size of transferred data you should specify the list of just the columns you need.
22. **Use Locking and Isolation Level Hints to Minimize Locking**  
    Within transactions, use the “WITH NOLOCK” option when possible. You’ll avoid long wait times for concurrent instances of your application accessing the same rows.
23. **Use parameters in queries**  
    The SQL Server query optimizer keeps recently used query plans in memory. When you are not using parameters, the parameters themselves contribute to make queries different from each other, and therefore, the Query Optimizer will not reuse them. Using parameters, the number of query plans in memory will decrease and they will more likely be reused.
24. **Choose the smallest data type that works for each column**  
    Explicit and implicit conversions may be costly in terms of the time that it takes to perform the conversion itself. There is also a cost in terms of the table or index scans that may occur because the optimizer cannot use an index to evaluate the query.
25. **Use varchar instead of text**  
    Columns that use the text data type have extra overhead because they are stored separately on text/image pages rather than on data pages. Use the varchar type instead of text for superior performance for columns that contain less than 8,000 characters.
26. **Use unicode only when necessary**  
    Unicode data types like nchar and nvarchar take twice as much storage space compared to ASCII data types like char and varchar.
27. **Limit the use of cursors**  
    Cursors can result in some performance degradation compared to select statements. Try to use correlated subquerìes or derived tables if you need to perform row-by-row operations.
28. **Avoid long actions in triggers**Trigger code is often overlooked when developers evaluate systems for performance and scalability problems. Because triggers are always part of INSERT, UPDATE, or DELETE calling transactions, a long-running action in a trigger can cause locks to be held longer than intended, resulting in the blocking of other queries. Keep your trigger code as small and as efficient as possible. If you need to perform a long-running or resource-intensive task, consider using message queuing to accomplish the task asynchronously.
29. **Avoid expensive operators such as “NOT LIKE”**  
    Some operators in joins or predicates tend to produce resource-intensive operations. The LIKE operator with a value enclosed in wildcards (“%a value%”) almost always causes a table scan. This type of table scan is a very expensive operation because of the preceding wildcard. “LIKE” operators with only the closing wildcard can use an index because the index is part of a B+ tree, and the index is traversed by matching the string value from left to right.  Negative operations, such as <> or NOT LIKE, are also very difficult to resolve efficiently. Try to rewrite them in another way if you can. If you are only checking for existence, use the “IF EXISTS” or the “IF NOT EXISTS” construct instead. You can use an index. If you use a scan, you can stop the scan at the first occurrence.
30. **Evaluate the query execution plan**  
    In SQL Query Analyzer, enable the Display Execution Plan option, and run your query against a meaningful data load to see the plan that is created by the optimizer.  
    Evaluate this plan and then identify any good indexes that the optimizer could use. Also, identify the part of your query that takes the longest time to run and that might be better optimized. Understanding the actual plan that runs is the first step toward optimizing a query. As with indexing, it takes time and knowledge of your system to be able to identify the best plan.
31. **Use Sp\_executesql for dynamic code**  
    If you must use dynamic code in your application, try to wrap it in the sp\_executesql system stored procedure. This allows you to write parametrized queries in T-SQL and you save the execution plan for the code. If the dynamic code has little chance of being called again, there is no value in saving the execution plan because the execution plan will eventually be removed from the cache when the execution plan expires. Evaluate whether an execution plan should be saved or not. Note that wrapping code in the sp\_executesql system stored procedure without using parameters does not provide compile time performance savings.
32. **Keep Statistics Up to Date**  
    Statistics are used by SQL Server Query Optimizer to select the best index to use when extracting data from your table. If statistics are not up to date you may end up keeping an index that is never used.
    * 1. **Don’t use the \* in your queries. A SELECT \* does an overload on the table, I/O and network bandwidth.**
      2. Verify if a critical query gains performance by turning it in a **stored procedure**.
      3. **Avoid too much JOINs** on your query: use only what is necessary!
      4. **Always restrict the number of rows and columns of your result.** That way, you save disk, memory and network of the database server. Always verify your WHERE clause and use TOP if necessary.
      5. **Verify if your server isn’t suffering from not-enough-disk-space illness.** Some times you lose time searching for all kind of problems only to find out that the server’s disk are almost full a few hours later. Always reserve at least 30% of available space on your disc.
      6. The decreasing performance order of operators is: = (faster)>, >=, <, <=, LIKE, <> (slower)
      7. If a query is slow and your index is not being used by it (remember to check your execution plan), you can force it using **WITH(INDEX=index\_name)**, right after the table declaration on the FROM clause.
      8. **Use EXISTS or NOT EXISTS instead of IN or NOT IN.** IN operators creates a overload on database.
      9. Try to use **BETWEEN instead of IN,** too.
      10. When using **LIKE** operator, try to leave the wildcards on the right side of the VARCHAR.
      11. **Always avoid to use functions on your queries.** SUBSTRING is your enemy. Try to use LIKE instead.
      12. **Queries with all operations on the WHERE clause connected by ANDs are processed from the left to right**. So, if a operation returns false, all other operations in the right side of it are ignored, because they can’t change the AND result anyway. It is better then to start your WHERE clause with the operations that returns false most of the time.
      13. Sometimes is better to make various queries with **UNION ALL** than a unique query with too much OR operations on WHERE clause. Test it.
      14. When there is a HAVING clause, it is better to **filter most results on the WHERE clause and use HAVING only for what it is necessary**.
      15. If there is a need of returning some data fast, even if it is not the whole result, use the **FAST** option.
      16. Use, if possible, UNION ALL instead of UNION. The second eliminates all redundant rows and requires more server’s resources.
      17. Use less **subqueries**. If you must use it, try to nest all of them on a unique block.
      18. Avoid to do much operations on your WHERE clause. If you are searching for **a + 2 > 7, use a > 5 instead**.
      19. Use **more variable tables and less temporary tables**.
      20. Use **functions to reuse code. But don’t exaggerate** on using them!
      21. To delete **all rows** on a table, use **TRUNCATE TABLE statement instead of DELETE.**
      22. If you have a **IDENTITY** primary key and do dozens of simultaneous insertions on in, make it a non-clusterized primary key index to avoid bottlenecks.
      23. Create Highly-Selective Indexes
      24. Avoid Indexing Small Tables
      25. Understand Response Time vs. Total Time
      26. Rewrite Subqueries to Use JOIN

Limit Using Outer JOINs

### Create Highly-Selective Indexes

Indexing on columns used in the WHERE clause of your critical queries frequently improves performance. However, this depends on how selective the index is. Selectivity is the ratio of qualifying rows to total rows. If the ratio is low, the index is highly selective. It can get rid of most of the rows and greatly reduce the size of the result set. It is therefore a useful index to create. By contrast, an index that is not selective is not as useful.

A unique index has the greatest selectivity. Only one row can match, which makes it most helpful for queries that intend to return exactly one row. For example, an index on a unique ID column will help you find a particular row quickly.

You can evaluate the selectivity of an index by running the sp\_show\_statistics stored procedures on SQL Server Compact tables.

Create Multiple-Column Indexes

Avoid Indexing Small Tables

Choose What to Index

We recommend that you always create indexes on primary keys. It is frequently useful to also create indexes on foreign keys. This is because primary keys and foreign keys are frequently used to join tables. Indexes on these keys lets the optimizer consider more efficient index join algorithms. If your query joins tables by using other columns, it is frequently helpful to create indexes on those columns for the same reason.

## Understand Response Time vs. Total Time

Response time is the time it takes for a query to return the first record. Total time is the time it takes for the query to return all records. For an interactive application, response time is important because it is the perceived time for the user to receive visual affirmation that a query is being processed. For a batch application, total time reflects the overall throughput. You have to determine what the performance criteria are for your application and queries, and then design accordingly.

For example, suppose the query returns 100 records and is used to populate a list with the first five records. In this case, you are not concerned with how long it takes to return all 100 records. Instead, you want the query to return the first few records quickly, so that you can populate the list.

Many query operations can be performed without having to store intermediate results. These operations are said to be pipelined. Examples of pipelined operations are projections, selections, and joins. Queries implemented with these operations can return results immediately. Other operations, such as SORT and GROUP-BY, require using all their input before returning results to their parent operations. These operations are said to require materialization. Queries implemented with these operations typically have an initial delay because of materialization. After this initial delay, they typically return records very quickly.

Queries with response time requirements should avoid materialization. For example, using an index to implement ORDER-BY yields better response time than does using sorting. The following section describes this in more detail.

### Index the ORDER-BY / GROUP-BY / DISTINCT Columns for Better Response Time

The ORDER-BY, GROUP-BY, and DISTINCT operations are all types of sorting. The SQL Server Compact query processor implements sorting in two ways. If records are already sorted by an index, the processor needs to use only the index. Otherwise, the processor has to use a temporary work table to sort the records first. Such preliminary sorting can cause significant initial delays on devices with lower power CPUs and limited memory, and should be avoided if response time is important.

In the context of multiple-column indexes, for ORDER-BY or GROUP-BY to consider a particular index, the ORDER-BY or GROUP-BY columns must match the prefix set of index columns with the exact order. For example, the index CREATE INDEX Emp\_Name ON Employees ("Last Name" ASC, "First Name" ASC) can help optimize the following queries:

Update plans can be complicated, as they need to update existing indexes alongside data and, because of objects like check constraints, referential integrity constraints and triggers, those plans may also have to access multiple tables and enforce existing constraints. Updates may also require the updating of multiple tables when cascading referential integrity constraints or triggers are defined. Some of these operations, such as updating indexes, can have a big impact on the performance of the entire update

operation, and we'll take a deeper look at that now.

Update operations are performed in two steps, which can be summarized as a read

section followed by the update section. The first step provides the details of the changes

to apply and which records will be updated. For INSERT operations, this includes the

values to be inserted and, for DELETE operations, it includes obtaining the keys of the

records to be deleted, which could be the clustering keys for clustered indexes or the RIDs

for heaps. Just to keep you on your toes, for update operations, a combination of both the

keys of the records to be updated and the data to be inserted is needed. In this first step,

SQL Server may read the table to be updated just like in any other SELECT statement

In the second step, the update operations are performed, including updating indexes,

validating constraints and executing triggers. The update operation will fail and roll back

if it violates a constraint

Per-row and per-index plans

An important operation performed by updates is the modifying and updating of existing

non-clustered indexes, which is done by using per-row or per-index maintenance plans

(also called narrow and wide plans, respectively). In a per-row maintenance plan, the

updates to the base table and the existing indexes are performed by a single operator, one

row at a time. On the other hand, in a per-index maintenance plan, the base table and

each non-clustered index are updated in separated operations.

Except for a few cases where per-index plans are mandatory, the Query Optimizer can

choose between a per-row and per-index plan based on performance reasons, and on an

index-by-index basis. Although factors like the structure and size of the table, as well as

the other operations performed by the UPDATE statement, are all considered, choosing

between per-index and per-row plans will mostly depend on the number of records

being updated. The Query Optimizer is more likely to choose a per-row plan when a

small number of records are being updated, and a per-index plan when the number of

records to be updated increases, as this choice scales better. A drawback with the per-row

approach is that the storage engine updates the non-clustered index rows using the

clustered index key order, which is not efficient when a large number of records need to

be updated..

In summary, keep in mind that, except for a few cases where per-index plans are

mandatory, the Query Optimizer can choose between a per-row and per-index plan

on an index-by-index basis, so it is even possible to have both maintenance choices in

the same execution plan.

Halloween protection

update operations have a read section followed by an update section, and that is a crucial distinction to bear in mind at this stage.

To avoid the Halloween problem, the read and update sections must be completely

separated; the read section must be completed in its entirety before the write section is

run.

The System R team was testing a query optimizer when they ran a query

to update the salary column on an Employee table. The query was supposed to give a

10% raise to every employee with a salary of less than $25,000 but, to their surprise, no

employee had a salary under $25,000 after the update query was completed. They noticed

that the query optimizer had selected the salary index and had updated some records

multiple times, until they reached the $25,000 salary. Since the salary index was used to

scan the records, when the salary column was updated, some records were moved within

the index and were then scanned again later, updating those records more than once. The

problem was called Halloween problem simply because it was discovered on Halloween

around 1976 or 1977.

Table Spool operator, which is a blocking operator, separating the read section from the write section. A blocking operator has to read *all* of the relevant rows before producing any output rows to the next operator. the table spool separates the Clustered Index Scan from the Clustered Index Update The spool operator scans the original data and saves a copy of it in a hidden spool

table in tempdb before it is updated. A Table Spool operator is usually used to avoid the Halloween problem as it is a cheap operator. However, if the plan already includes another operator that can be used, such as a Sort, then the Table Spool operator is not

needed, and the Sort can perform the same blocking job instead