Deep Science Publishing, 2025
https://doi.org/10.70593/978-93-7185-191-6_3

DeepScience
Open Access Books

Chapter 3: Performance tuning in SQL server environments: Execution plans, index strategies, and resource cost estimation

Mohanraju Muppala

1. Introduction to Performance Optimization

Performance Optimization in SQL Server relies on the Query Optimizer looking at an execution plan, the Parallelism Costs, and the memory grant that the query request says it needs [1-2]. The optimize for ad hoc workload setting looks to flag a query in the caching plan on whether this is a one time execution or will be re-used. If it is set then when a query is first executed only a lightweight plan is cached and does not have the full plan details of execution costs, memory grants, associated indexes and operations. It is a distinct cache plan from the full plan and greatly reduces the memory requirements for large queries that will not be re-run. As the Query Optimizer looks for the execution plan to minimize Resource Costs, Memory for the execution of the Query, overall execution time of the query, and potentially other factors that impact the cost of the plan, HVAC works on the time taken and resource requirements.

The overall execution time of the query includes factors such as the Cost of CPU Time taken to execute the query, the IO Cost of the queries use of disk and memory, the Cost of Network, Security grants and other factors that contribute to the query execution [3-5]. The HVAC algorithm adjusts the estimates for the execution plan based on the elapsed time so far and resource load being observed whilst the query is in execution. It then examines the degree of expected parallelism for the query and compares these estimates against the SQL Server System Resource Governor's settings for MaxDOP,

Memory Limits for MAX, and the setting configured for MAX Cost, abort threshold. The Compatibility Level is looked at to flag for SQL Server 2019 batch mode adaptive joins that are capable of reducing overall query execution time. The HVAC algorithm provides a mechanism for the overall execution plan for the query to be dynamically adapted based on the query's progress in the execution plan, and the system resources that it is consuming, such as Degree.of Parallelism and Memory Grants, where the expected time to complete exceeds the threshold set in the System Resource Governor's Delegation Threshold setting.

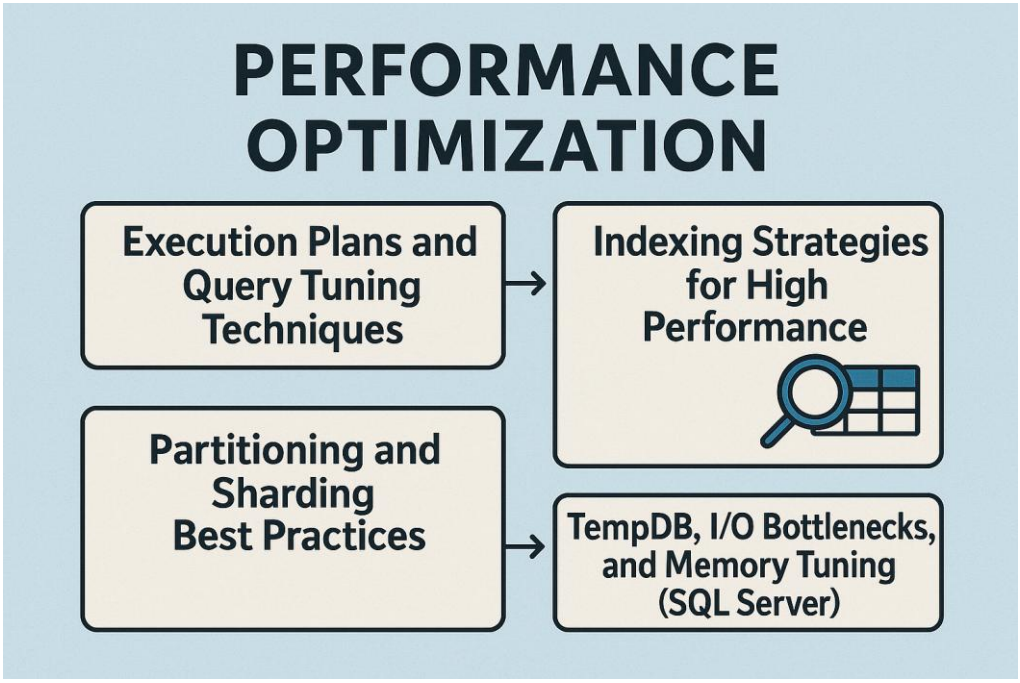

Fig 1. Performance optimization

2. Execution Plans and Query Tuning Techniques

Execution plans provide a wealth of information relating to SQL Server query execution. They reveal the actual path chosen by the optimizer, including the physical operators used; the estimated or actual number of rows flowing from one operator to another; the CPU cost for each operator; and the indexes used, alongside estimated or actual I/O costs associated with those indexes. Understanding execution plans is essential for analyzing slow-running queries and identifying performance hotspots. The Resource Monitor tool within SQL

Server Management Studio facilitates this analysis by displaying the query plan and highlighting costly operators along with their associated CPU and logical reads.

Having pinpointed the potential cause of suboptimal performance, the database administrator has several avenues for query optimization. One of the simplest methods is the selection of a different retrieval strategy. Other more involved techniques include rewriting the query to employ temporary tables as intermediate containers, using grouping and aggregation functions, leveraging CTEs instead of subqueries, and creating additional indexes, possibly with included columns. Remembering the adage "poor index, poor performance" underscores the significance of indexing strategies for query tuning and performance—topics that are more deeply explored in the following section.

2.1. Understanding Execution Plans

Determining how a query is executed provides insight into why it runs slowly and enables proper tuning [6-8]. SQL Server generates an exploration path for the query, and examining this path—known as an Execution Plan—greatly assists in tuning. Execution plans are graphical or textual representations of the data retrieval methods chosen by the SQL Server Database Engine Optimaizer. An execution plan shows the methods SQL Server will use, or has used, for data retrieval, covering aspects such as which tables or indexes are accessed, the join order, and the size of the result set.

Execution plans can be displayed before executing the query to predict the execution method, or after executing to view what was actually used. Often, the plan chosen by the optimizer changes significantly after execution because the optimizer is sensitive to the statistics associated with the underlying table(s) and index(es). Obtaining an execution plan before executing the query is therefore usually not advised, as it provides only an estimated path lacking actual cardinality information. The optimizer will usually choose a better path after execution, informed by real-time statistics. The underlying indexing strategy—which is crucial to performance—is discussed separately in Section 3.

2.2. Analyzing Query Performance

The efficiency of a query is measurable only through an execution plan because the information provided by a query or stored procedure body is insufficient for judging performance. An execution plan is a graphical or textual visualization produced by an optimizer that describes the data access methods and join algorithms selected for a particular query or procedure. The execution plan used

by the SQL Server engine determines execution speed and efficiency of resource usage.

Analyzing query execution plans—particularly for poorly performing queries—guides the tuning process. Important factors for adjusting queries include the selection of indexes, join algorithms, join sequence, and row filters. If the actual execution plan differs significantly from the expected plan, a forced plan may be considered. For more information on indexing, consult section 2.3.4. Indexing for High Performance.

2.3. Common Query Tuning Techniques

Execution plans can be used to identify the execution path chosen by the SQL optimizer for a query. When a query runs slower than expected, analysing it often resolves performance issues. In extreme cases, it might be necessary to completely rewrite a query. The performance of a query is heavily influenced by the structure and load on the database. Query tuning is a complex discipline and not every aspect is discussed here.

T-SQL tuning seeks to improve the query execution plan and minimize logical I/O. Reducing logical I/O does not guarantee faster performance, especially if a query reads few blocks; in such cases, improving the physical location of the rows in the database file might have a greater effect. Standardised naming conventions, meaningful aliasing in the FROM clause, and indentation improve query readability, facilitating better understanding and maintenance.

3. Indexing Strategies for High Performance

Proper indexing allows the database engine to find the data quickly without having to perform dimension scans. It is significantly more time-consuming for the engine to scan through the table to select the correct data than to search through an index and then find the location of the data on the table. Tables can have primary indexes, unique indexes, and non-unique indexes.

SQL Server 7 has three types of indexes. Clustered indexes have their data in order and at the leaf of the index is the table data. Non-clustered indexes have more abstract mappings and use pointers back to the data in the table. Primary indexes cannot be dropped. Subscripts to the primary index on the table can be created or dropped. The first index created on a table—to which the primary key was applied or an index manually created as clustered—is the primary clustered index. Consider a copy of the base table that is indexed. SQL Server 6

clustered, SQL Server 7 primary indexes, and a clustered index all contain a physical copy of the data. Subscripts to the primary index are a logical copy of the data.

Indexes do add overhead to database modifications. Whenever the data changes, the indexes need to be maintained and changed. Creating more indexes has a direct impact on the response time of database modifications. Whenever data changes, the database needs to provide access to these changes in the index. This situation requires more processing power and disk activity than the database without indexes.

3.1. Types of Indexes

SQL Server indexes are the database's equivalent of table of contents to a book making data retrieval efficient and fast. They help create the data structure with database pages and database pointers [7,9-10]. These pages are known as index nodes or index levels. Index nodes are located at various level between last level and first level of index structure. These nodes consist of a row for every database page and a pointer to the page.

An index typically includes a key part. This consists of the key values from the index keys. The key value for each first index key is selected to provide a multilevel hierarchy. The key values for all other index keys are copied directly from the base data structures. There is also a non-key part. This holds the row locator data value that is used to locate the original data row. In a clustered index, the non-key columns of the index are included in the key part. Clustering determines the storage pattern of the remaining non-key columns for the index key in the table. In clustered-indexed tables, data rows are stored sorted by clustering. Therefore, clustered indexes will be called tables or clustered indexes.

3.2. Creating and Maintaining Indexes

Indexes are objects in the database that improve the efficiency of data retrieval operations. They do so by providing direct pointers to the data and reducing the number of data blocks that need to be read during a search operation. It is important to note that SQL statements that output large volumes of data, such as SELECT *, cannot effectively utilize any indexes created on the tables. Furthermore, the presence of indexes on a table necessitates extra processing during an update operation, as the index entries must also be updated. Consequently, it is recommended to create indexes primarily for tables that serve as the source for data retrieval operations and are not heavily involved in update operations [1,11-14].

SQL Server supports the creation of clustered and nonclustered indexes on tables. Although tables with no clustered index are referred to as heaps in the SQL books, the term heap is not used in the course's PowerPoint slides. For the CREATE INDEX statement, the Microsoft SQL Server Books Online specify that, if created as NONCLUSTERED, the index is created in ascending key order by default. In contrast, a clustered index is created in ascending key order by default, if created as CLUSTERED. Primary key and unique key constraints create a clustered index unless a clustered index already exists on the table, in which case the nonclustered index is created.

3.3. Index Usage and Performance Impact

Indexes are an important part in optimizing SQL Server performance. They help to speed up reads and search operations. Indexes can be single column or composite; they can also be unique indexes, which do not allow duplicates. There are two types of indexes: the traditional nonclustered ones, where the data itself is stored in the data pages and index key columns are stored in B-tree structure only, and clustered indexes, which store both the clustered key column and the data in the B-tree structure. Every table can have only one clustered index, as the data pages are sorted physically according to the clustered index key.

When a table is regularly being updated, updated, or deleted, indexes can have a huge negative impact on its performance. Maintenance of indexes during every modification query can delay the operation. It can also have a negative impact on memory utilization, as more read/write is done on the main memory for index structures. Hence choosing the right indexes, limiting the number of nonclustered indexes, and creating the right query can improve performance significantly.

4. Partitioning and Sharding Best Practices

Partitioning and sharding in SQL Server constitute complementary techniques that improve application responsiveness and optimize DBMS throughput under heavy workloads. Both approaches allow the DBMS to scan a smaller subset of the data, reducing response times and boosting concurrency.

Range partitioning is a natural modeling choice for temporal data. Large tables representing the recent past partitions are queried frequently and updated sporadically, while older partitions are accessed by occasional large-scale

analytics. PARTITION functions declare the partitioning strategy, while PARTITION SCHEME references the FILEGROUPs housing the underlying data files. Historical data can be archived to cheaper storage tiers or decommissioned, depending on maintenance policies. Query optimizer performance is directly affected by, for example, the absence of partition-aligned indexes—a suboptimal indexing strategy where the granularity of the indexes on a partitioned table is coarser than the partitions.

4.1. Overview of Partitioning

Partitioning is a technique for splitting individual database "objects" horizontally into smaller "pieces" enabling certain queries or data-management operations to access or affect only some of those pieces [13,15-17]. The two types of objects partitioned are tables and indexes, which have the same structure but somewhat different functions. Tables store the base data, and indexes help with searches and sorts. Partitioning tables or indexes can help improve query performance, backup and restore performance, index maintenance performance, and data-management operations such as switching.

Partitioned tables and indexes has been around since SQL Server 2005. Partitioning requires a partition function, a partition scheme, a partitioning column in the table or in the index, and a partition-aligned clustered index to take advantage of all the performance features that partitioning provides. The partition function specifies how the data will be mapped to a number of partitions by defining the partitioning column and the number of partitions. A set of range values allocates the rows in the partitioning column to the partitions. The partition scheme allocates these partitions to a number of filegroups, so that certain partitions can be placed in specific filegroups.

4.2. Implementing Partitioning in SQL Server

Table and Index Partitioning in SQL Server Table and index partitioning facilitates the management, maintenance, and access to large tables and indexes. For example, a partitioned table can be split between multiple filegroups and windows of data pruned during a query, thereby limiting the amount of data accessed through partition elimination [18-20]. When working with very large tables, consider using the following tasks to improve your overall performance in querying and maintaining the data: Historical data, which is rarely updated but is growing, can be partitioned into its own filegroups [19,21-22]. These filegroups can then be moved to slower, cheaper storage. Current data, where daily processing occurs, can also be partitioned and maintained independently from historical data. The entire table can be queryable so that areas of the table

can be zipped (compressed) and restored independently. Partitioning also plays a great role in the maintenance window. For example, when index or statistics maintenance occurs on a large table, or even when a data warehouse table is being bulk loaded, the maintenance window can be reduced.

4.3. Sharding Strategies for Scalability

Any application that needs to scale horizontally will implement some sharding strategy. The default Microsoft Azure SQL database shard map provides the basis for a sharding strategy, but customers of all sizes prefer a sharding strategy that matches their application's scalability requirements and natural distribution of their data. Understanding common sharding strategies for Azure SQL Database helps in making effective decisions about how the data should be distributed.

Choice of shard keys is critical for the performance and scalability of any application that uses a sharding strategy. For example, if TenantId is selected as the shard key, then all data of every tenant can be located in a single shard. This configuration allows the application to read and write the tenant data by issuing operations to only one shard. The selection of TenantId impacts how well the overall sharding strategy supports large large tenant scenarios. If the sharding strategy's tenant selection is such that the system cannot accommodate large tenants, then the tenant shard key does not optimally support the data distribution.

5. Managing TempDB for Optimal Performance

TempDB is the workspace all SQL Server instances must share. All use TempDB, but when a few sessions or users are hogging its capacity, all others get hammered. Some users or sessions need it more than others, so the best solution to TempDB contention might be to enroll in a club or use a different feature. One of TempDB’s greater ironies is that it’s a singleton shell.

TempDB needs to be shrinkable so that it can be used by other organizations. TempDB needs to be easy to administer and use, which is why many DBAs prefer to disable the service feature altogether. TempDB needs to be easy to code and manage, which are two reasons many developers turn to the collection class instead.

5.1. Understanding TempDB Usage

In SQL Server, the tempDB database provides temporary storage by creating free pages on data files or allocating request pages as needed [11,23-25]. Tempobjects are data objects created on the SQL Server TempDB database and are respectively divided into local and global. Temporary tables and indexes used in the execution of the request work this way. Internally created work tables on tempDB or on-disk work tables with low transaction isolation levels are also part of tempobjects. Allocation structures such as of allocation maps (GAM, SGAM, and IAM) are updated when new worktables are created on tempDB. Furthermore, SQL Server manages the LOB allocation maps for updateable LOB data types.

poptable occurs when tempDB allocation structures are repeatedly updated and then accessed by concurrent sessions, creating a bottleneck. The update pages used for tracking allocations are also placed on tempDB, and contention might occur on these pages during concurrent allocation requests. This specific form of internal contention within tempDB is mainly created at the allocation level of the allocation pages (GAM, SGAM, IAM) or the allocation pages of the LOB data types. High multiples of such allocation calls within a short amount of time might therefore lead to contention and the following result:

5.2. Configuring TempDB for Performance

Because of the use of temporary database objects in the TempDB database, configuring TempDB becomes very important for efficient record selection from logical pages. You can achieve impressive performance improvements when these configurations are applied.

Although the version of SQL Server is not indicated explicitly in the SQL Server error log, you can use the following feature to identify it: CREATE DATABASE tempdb ON

— Primary Data File (NAME = tempdev, FILENAME = 'E:\MSSQL12.MSSQLSERVER\MSSQL\DATA\tempdb.mdf', SIZE = 128MB, MAXSIZE = 2048GB, FILEGROWTH = 256MB), — Additional TempDB Files (NAME = tempdb2, FILENAME = 'E:\MSSQL12.MSSQLSERVER\MSSQL\DATA\tempdb2.ndf', SIZE = 64MB, MAXSIZE = 2048GB, FILEGROWTH = 64MB), (NAME = tempdb3, FILENAME = 'E:\MSSQL12.MSSQLSERVER\MSSQL\DATA\tempdb3.ndf', SIZE = 64MB, MAXSIZE = 2048GB, FILEGROWTH = 64MB), — Log File LOG ON (NAME = templog, FILENAME =

'E:\MSSQL12.MSSQLSERVER\MSSQL\DATA\templog.ldf', SIZE = 128MB, MAXSIZE = 2048GB, FILEGROWTH = 256MB);

5.3. Best Practices for TempDB Management

SQL Server uses TempDB as a system database for temporary tables, stored procedures, triggers, cursors, table variables, and for sorting large data sets, among other sources. During these operations, SQL Server creates and writes to file structures in TempDB. Proper TempDB performance is essential for optimum performance of SQL Server only queries. Many TempDB activities are shared by all SQL Server processes, including OLMs: the database snapshot bitmap, row versioning data for online operations and online index rebuilds, the harmful IRL used by all active transactions and used by every query with a serializable or Repeatable Read Isolation level, the active portion of the Temp Table Garbage Collection cycle, and so on. Under-provisioning TempDB can severely impact the overall server performance.

Allocation contention in TempDB can occur when the system cannot find pages for certain types of allocation, so threads have to wait for allocation bitmaps to be freed. SQL Server generates TempDB alerts when contention on allocation structures causes a thread wait more than 5 seconds.

6. Identifying and Resolving I/O Bottlenecks

An examination of SQL Server's performance metrics reveals that, whenever I/O stalls approach 10 ms or more, the bottleneck resides at the SQL Server level [29-32]. For instance, a read stall of 100 ms indicates that SQL Server had to wait an unusually long time for an I/O operation to complete. Conversely, a 1-ms read stall is generally considered acceptable in the context of SQL Server's workload.

When SQL Server I/O stalls exceed 10 ms, it is worthwhile to explore options for storage subsystem enhancement. The deductions presented here derive from analysis of aggregate read stalls, but the same reasoning applies to read, write, or other types of I/O at the system, file, or even database level.

6.1. Understanding I/O Performance

Understanding I/O performance is crucial when tuning SQL Server. SQL Server reads and writes ARM64 pages, with each page having a size of 8 KB. When it encounters a page needed for query execution that is not in the buffer cache, it must be read from disk [26-28]. Additionally, when a new row is

added to a page in the buffer, the page needs to be written back to disk for persistence. However, the question arises: are these reading and writing operations random or sequential? Does the pattern depend on the load pattern or the query? Can the random or sequential nature of these operations change depending on the type of RAID implemented on the disk? This section attempts to answer all those questions.

For cache misses, SQL Server issues read commands to disk to bring the pages into the buffer cache. Depending on whether the read operation is sequential or random, one or more pages will be read. Some of the internals of SQL Server are explained here to differentiate between sequential and random-read operations. Besides the buffer cache, SQL Server also has a separate cache called the procedure cache, not to be confused with the plan cache and execution cache that are part of the procedure cache. The plan cache is used to cache ad-hoc queries, whereas execution objects are cached when a stored procedure plan is compiled. For instance, upon execution, a stored procedure compiles into a plan that is added to the cache. Execution objects can be related to a procedure, function, trigger, and so on. Execution objects usually consume the bulk of the memory.

6.2. Tools for Monitoring I/O Bottlenecks

Several tools in SQL Server assist during performance tuning to calculate the I/O cost for a query. During query optimization, the cost of various plans for executing the query is calculated and the one with the lowest cost is chosen. However, the cost sometimes doesn’t correlate with actual query execution time. When this happens, the suggested tools can help find bottlenecks that may indicate an I/O problem.

The query execution plan shown in the previous example is quite useful as it displays the type of joins, the sequence in which the tables are joined, the physical operators used, and the join predicates used. The plan also shows the estimated I/O cost, CPU cost, and the combined cost for each operator and for the entire query. To spell out the details of the procedure, query trace options is another option. The query trace output contains the same details as the query execution plan along with the statistics on the number of pages of data read from the data file, number of pages for which lock was acquired, memory allocated for the query, and the number of buffers read from the buffer cache.

6.3. Strategies for I/O Optimization

Since SQL Server performs table and index scans during query execution, it is wise to put the most commonly used tables and clustered index s on their own

physical disk or disk array so that the operating system does not need to read reads from multiple tables and indexes on the same physical disk or disk array.

SQL Server is able to use all of the columns in a covering index to answer query side and index-side joins [31,33-35]. Because of this, it may be useful to create covering indexes with the most frequently retrieved columns—even if it means including only nonclustered columns—and to use INCLUDE for columns that are never part of the search conditions. Put the nonclustered indexes on their own disks or disk arrays as well. The data files and log can be put together on a fast disk or disk array.

7. Memory Tuning Techniques

SQL Server uses its Memory Manager to allocate and deallocate memory based on its needs during query execution. It allocates and deallocates buffers for cache or execution work areas, manages conditions when memory request can't be met, and controls the amount of caching for other resources such as connections and locks.

There are many configuration options to control memory usage, but these should be set only on systems where multiple applications are competing for available memory resources. Memory configuration should be left at their defaults on other systems since SQL Server can determine the amount of memory it needs more effectively than the DBA. One exception is when SQL Server is the sole application on the computer and the amount of RAM exceeds approximately 1GB. Without setting the maximum memory option, SQL Server will grow its memory usage until it fills all available RAM on the computer.

7.1. Understanding SQL Server Memory Architecture

SQL Server Memory Architecture

Examining the architecture of SQL Server memory provides insight into the factors that affect buffer pool usage, CPU utilization, and overall memory consumption. This understanding guides the allocation of the correct amount of RAM to the server and its database instances. SQL Server memory comprises the Procedure Cache, Buffer Pool, and Other Memory allocations.

The Procedure Cache allocates memory pages for optimizing query execution by SQL Server. The Buffer Pool contains Data Cache and Log Cache. The Data Cache holds frequently accessed data pages, enabling faster retrieval, while the

Log Cache stores information about transactions and system transactions that utilize the logging mechanism. Other Memory consists of allocations for external links, extended procedures, linked servers, Replication, Service Broker, and various other features—for example, memory space for the Service Broker is allocated here, and some of it can be freed when Smart Scan is enabled.

7.2. Configuring Memory Settings

In Chapter 7, Memory, look beyond the typical causes of poor server performance and learn how to control and monitor SQL Server's use of memory. The maximum amount of memory that SQL Server can use should be set in most cases. The default of 2147483647 MByte is basically unlimited—SQL Server can take as much RAM as the operating system allows. This means that SQL Server can consume all the available RAM and leave none for the operating system.

The consequences of this will only become evident once the system is under excessive memory pressure and start to swap excessively. Validator even recommends setting both the minimum and maximum memory to the same value. This is what the configuration script provided does, but setting the minimum memory can be controversial. For more information, review the discussion related to the MDASQLMemory script.

Caution The maximum memory value should never be set by an application or part of an automated SQL Server install process because it varies per server. It's bad practice for an application or installation procedure to set it.

7.3. Monitoring Memory Usage

Why Monitor Memory Usage? SQL Server is a data-intensive product and, as such, it loves to use memory. Caching data in memory allows for quicker data retrieval and hence better overall performance. However, be aware that the operating system also requires memory in order to perform its functions, so SQL Server must give some memory back to the operating system. If SQL Server occupies all available memory, the operating system becomes slower. If SQL Server does not store enough data in memory, performance degrades due to excessive trips to disk.

When running SQL Server on a dedicated server, it is recommended to set the MAX SERVER MEMORY to the largest value possible while still leaving enough memory for the operating system and other applications. Memory tuning is the management of a balance between having enough memory for

SQL Server in execution and having enough free memory in the operating system to operate without swapping memory to disk. There are two key indicators of memory optimization, Cache Hit Ratio and Page Life Expectancy.

Cache Hit Ratio The primary goal in memory tuning is to reduce disk I/O. Cache Hits indicate queries retrieving data from memory, while cache misses indicate queries going to disk to retrieve data. This counter will help determine if more memory needs to be supplied to SQL Server. Cache Hits are generally expressed as a Cache Hit Ratio. Too many Cache Misses indicate that the buffer cache size is too small. However, it is possible that the data page was simply not in the cache because it had not been accessed before, rather than as a result of buffer cache size. In this case, an excessively high Cache Hit Ratio does not necessarily indicate a large cache size.

8. Best Practices for Performance Monitoring

Anyone who uses SQL Server on a daily basis, whether it be a system administrator, developer, or business analyst, has, at some point, encountered slow-running queries or waited on a dataset to be gathered. These experiences allow typical complaints to be heard throughout corporate America, such as “Why is this report delayed?” or “Where is that sales analysis?” To curtail the negative effects of such queries, developers and DBAs have a variety of tools at their disposal to find the appropriate combinations of execution plans, update statistics, and index tuning. When used properly, however, the tools for tuning SQL Server can prevent slow-performing queries from ever reaching production.

There are many ways to monitor the health and status of a SQL Server instance, but the Holy Grail of performance tuning is the timely detection of a slow SQL query. A SQL Server server-side tracing tool, such as Profiler, will allow you to monitor what is taking longer than expected. Profiler can also identify long-running queries and the users executing these queries. The best-case scenario is running Profiler during a slow time period identified by the WARN function

8.1. Key Performance Indicators (KPIs)

One of the essential skills that a DBA needs is the ability to monitor, tune, and improve the performance of a database regardless of the platform in use. Database performance involves many components and considerations; it's important to start by monitoring key performance indicators (KPIs) to ensure

the system meets its service-level agreement (SLA). KPIs not only help to keep track of the current system state but can also pinpoint potential future or imminent problems.

Performance-related KPIs fall under the broad categories of availability and scalability. If the system isn't operational, it can't provide service, and the KPIs should trigger alerts of this state. Checkpoints to keep track of system operational status are known as health checks and should include disk space, memory allocation, processor usage, database size, and backup jobs. The next area that impacts scalability is response-time. Checkpoints to monitor response-time include read and write performance; lookups into system metadata; and lookups into user databases. The final area that affects scalability is throughput. Like response-time, throughput is measured in terms of read and write activity and the number of users connected to the system.

8.2. Tools for Performance Monitoring

The chapter examines the tools available for monitoring SQL Server performance. After acquiring essential knowledge in SQL Server performance tuning, it is natural to desire real-time performance analysis. Currently, three tools facilitate this process: SQL Trace, Performance Monitor, and System Monitor.

Performance Monitor, also known as Sysmon, differs from SQL Trace by allowing performance analysis without the need to define specific traces before execution. Although monitoring can occur in real time, results must be saved in a file or database for thorough examination. While using a file is acceptable, storing results in a SQL Server database demands considerable space for later analysis. Performance Monitor thus serves as a native Windows tool for OS-level performance monitoring.

SQL Server 7.0 offers convenient collection sets within Performance Monitor, accompanied by detailed explanations that simplify their use.

8.3. Regular Maintenance and Performance Reviews

Sensible, regular maintenance and periodic and thorough performance reviews are vital. Properly written database maintenance code can drastically reduce the number of interruptions while improving reliability and scalability. Carefully performed, because incorrect intervals or improper scheduling can actually hurt performance, these activities maintain a healthy, well performing database that is ready to operate in an LOB environment.

It is best to review and ensure that each of the database tables is indexed accordingly and that those indexes are current and used frequently. Index fragmentation can make even the best indexes almost useless. Ensure that statistics are updated regularly and thoughtfully. Locking can cause a system to hang, so it is important to ensure that statistics can still be updated without negatively impacting reporting reasons [36-38]. A regularly scheduled statistics maintenance job will be able to perform the updates quickly during working hours without any problems or systems hung in a pending state awaiting an update. Updating statistics can truly be a high-impact operation. Properly updating statistics can ensure the system remains performing at a high level, but allowing statistics to be updated haphazardly can also negatively impact business. Statistics should be updated at least every time index rebuilds occur.

Fragmentation can be one of the biggest contributors to lower system performance. It causes the system to work harder to fulfill the same request, thus making the system respond slower and slower and, in some cases, apparently hang, when in reality, it just lags behind. Although performance enhancement is a reasonable reason to rebuild indexes, the primary reason to rebuild indexes is a regular maintenance plan that uses the degree of fragmentation as a ratio for deciding what to rebuild and what to reorganize. Most maintenance plans use reorganization for indexes with fragmentation levels between 10 and 40 percent and rebuilds for indexes with fragmentation levels over 40 percent. Segment reorganization is a light, almost instant, activity; index rebuilds are a much heavier operation and should be scheduled accordingly.

9. Case Studies and Real-World Applications

Performance optimization of SQL Server functioning in the financial management branch of a commercial bank in the northwest region of China is studied. The internal structure of SQL Server is analyzed, and optimization methods for the structure of SQL Server components, dynamic link library files, and configuration parameters of the database are proposed. After the optimized method is applied on SQL Server, the internal structure, DLLs, and parameters of the SQL Server database are reconfigured. The optimized database structure lowers the memory usage of operating SQL Server, the optimized DLLs reduce the dependence on hardware, and the optimized parameters of SQL Server improve the execution efficiency. Concluding, the performance of SQL Server in financial management is notably improved.

Operational databases in a variety of industries continuously sustain their export and usage over time. Accordingly, the coupling of the operational database with other systems also increases. This trend results in high operational database utilization, which leads to performance declines in export or sharing of operational data. Job control requires a steady stream of data export to maintain a consistent state, whereas data sharing generates operational database query requests. As more requests arrive, the operational database experiences performance issues with updates, inserts, and other operations. Therefore, the optimal utilization of operational databases plays an important role in business or external export processes. In these pursuit, the operational database is typically offloaded to a secondary database.

9.1. Case Study 1: Query Optimization

One of the biggest Customer Service Issues is being able to execute stored procedures quickly. There are also many other common reasons to keep execution time to a minimum. No one wants to have to wait for a query to run, especially when there are other customers waiting in line for service at the same time. Ideally, you would like to spend as little time waiting for the query to run as possible. In addition, you can improve response time to all users in the system just by reducing the execution time of each command. Processing time on the database server often tends to be only part of the story. In many cases the network between the client application and the database server can also determine how long it takes to get data to the user. No matter which one is the bottleneck, the best way to reduce wait time is to write or tune your queries so that SQL Server spends less time processing the query and your computer network processes less data.

Included with this chapter are solutions to a few queries that could use a little fine-tuning. As you sift through the examples, keep in mind that this process of query optimization contains a lot of possibilities. The goal of these particular examples is to show you how to make queries run faster by putting a little intelligence into query writing and index design. Optimizing query performance also involves a thorough understanding of the business purpose of queries. Call Center Managers, other managers, and system support personnel all need to let Database Administrators know when a query that “hogs” the CPU or network is an accepted fact of life—at least until someone can schedule the time needed to create a more efficient solution.

9.2. Case Study 2: Indexing Strategies

In addition to the expense data presented above, actual elapsed time will be used to demonstrate the validity of these results. Since the expense data already reflects the elapsed time required to perform the queries shown, the actual elapsed query times were gathered and compared with the estimated query costs as well. The conclusions and resulting recommendations will therefore be based on a compilation of both sets of statistics.

Figure 9-3 displays the query costs versus the actual elapsed time for two of the more costly queries, both with and without the missing index. After comparing the two, the possible correlation between the estimated query cost and the actual query time becomes apparent. Such a link is particularly useful when the actual query times are not available or there are insufficient queries being captured in the execution plan cache.

9.3. Case Study 3: I/O Bottleneck Resolution

Here is a case study on the resolution of an I/O bottleneck in a production system. Data was acquired with the topsql utility (previously discussed in Chapter 2). The most demanding query identified returns numerous rows, with each row requiring T-SQL code to be executed. Consequently, the query funds 90% of the batch text time, and specifically, the T-SQL code execution accounts for 99% of the query time. OFF CPU time corresponds to waiting for the accumulator cursor from SQL Server to return a few result sets. The statement is repeatedly executed against distinct parameters. The following chart illustrates the TOP SQL Batch Texts.

The query responsible exhibits the following characteristics:

- Requesting large amounts of data (over 2 million rows) - Taking approximately 13 minutes to transfer data from the database engine to the client application

Given that the client application (formerly known as "Data Browser" and currently referred to as the legacy application) was recently redesigned, it was decided to optimize the query internally and load only the very limited number of rows required in the legacy application. The redesigned application, known here as "DataStudio," is prepared to handle large amounts of data.

10. Conclusion

Everything related to SQL Server performance tuning and optimization is covered in this book. As previously discussed, SQL Server performance tuning requires a substantial amount of time and commitment. The methods used to find the root cause problem are also very difficult. However, the information available here can make it easier to identify where to look for the problem and how to solve it. Implementation of performance-tuning and optimization can significantly improve performance and save time and money. It is important to note that the solution to tune and optimize SQL Server may change over time. Therefore, it is essential to update the knowledge gained from this book and use it on the live server during performance and optimization.

Performance tuning is the practice of enhancing the response time or throughput of the Server. Keeping performance optimal is a continuous task. Make sure to use the correct data types, employ user-defined functions appropriately, and create indexes where necessary. Configuring CPU affinity and understanding how CPU and memory function are also beneficial. It is also advisable to find an optimal fill factor and defragment indexes. Many settings can be configured in SQL Server; however, decisions regarding enabling or disabling these settings should be based on throughput or performance. In SQL Server, there are multiple server configuration options. Use them with discipline and monitor the system after enabling or disabling them; otherwise, the server will exhibit very slow response times.

References:

[1] Nguyen XB, Phan XH, Piccardi M. Fine-tuning text-to-SQL models with reinforcement-learning training objectives. Natural Language Processing Journal. 2025 Mar 1;10:100135.

[2] Li J, Ye J, Mao Y, Gao Y, Chen L. LOFTune: A Low-Overhead and Flexible Approach for Spark SQL Configuration Tuning. IEEE Transactions on Knowledge and Data Engineering. 2025 Mar 18.

[3] Chopra A, Azam R. Enhancing natural language query to SQL query generation through Classification-Based table selection. InInternational Conference on Engineering Applications of Neural Networks 2024 Jun 22 (pp. 152-165). Cham: Springer Nature Switzerland.

[4] Ling X, Liu J, Liu J, Wu J, Liu J. Finetuning LLMs for Text-to-SQL with Two-Stage Progressive Learning. InCCF International Conference on Natural Language Processing and Chinese Computing 2024 Nov 1 (pp. 449-461). Singapore: Springer Nature Singapore.

[5] Shivadekar S, Kataria DB, Hundekar S, Wanjale K, Balpande VP, Suryawanshi R. Deep learning based image classification of lungs radiography for detecting covid-19 using a deep cnn and resnet 50. International Journal of Intelligent Systems and Applications in Engineering. 2023;11:241-50.
[6] Ashlam AA, Badii A, Stahl F. Multi-phase algorithmic framework to prevent SQL injection attacks using improved machine learning and deep learning to enhance database security in real-time. In2022 15th International Conference on Security of Information and Networks (SIN) 2022 Nov 11 (pp. 01-04). IEEE.
[7] Tanimura C. SQL for Data Analysis: Advanced Techniques for Transforming Data Into Insights. " O'Reilly Media, Inc."; 2021 Sep 9.
[8] Brunner U, Stockinger K. Valuenet: A natural language-to-sql system that learns from database information. In2021 IEEE 37th International Conference on Data Engineering (ICDE) 2021 Apr 19 (pp. 2177-2182). IEEE.
[9] Panda SP. Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems. Deep Science Publishing; 2025 Jun 22.
[10] Lawson JG, Street DA. Detecting dirty data using SQL: Rigorous house insurance case. Journal of Accounting Education. 2021 Jun 1;55:100714.
[11] Zhang B, Ren R, Liu J, Jiang M, Ren J, Li J. SQLPsdem: A proxy-based mechanism towards detecting, locating and preventing second-order SQL injections. IEEE Transactions on Software Engineering. 2024 May 14;50(7):1807-26.
[12] Panda SP. Artificial Intelligence Across Borders: Transforming Industries Through Intelligent Innovation. Deep Science Publishing; 2025 Jun 6.
[13] Gandhi N, Patel J, Sisodiya R, Doshi N, Mishra S. A CNN-BiLSTM based approach for detection of SQL injection attacks. In2021 International conference on computational intelligence and knowledge economy (ICCIKE) 2021 Mar 17 (pp. 378-383). IEEE.
[14] Dhanaraj RK, Ramakrishnan V, Poongodi M, Krishnasamy L, Hamdi M, Kotecha K, Vijayakumar V. Random forest bagging and x-means clustered antipattern detection from SQL query log for accessing secure mobile data. Wireless communications and mobile computing. 2021;2021(1):2730246.
[15] Panda SP, Muppala M, Koneti SB. The Contribution of AI in Climate Modeling and Sustainable Decision-Making. Available at SSRN 5283619. 2025 Jun 1.
[16] Shivadekar S. Artificial Intelligence for Cognitive Systems: Deep Learning, Neuro-symbolic Integration, and Human-Centric Intelligence. Deep Science Publishing; 2025 Jun 30.
[17] Davidson L. Pro SQL Server Relational Database Design and Implementation: Best Practices for Scalability and Performance. Apress; 2021.
[18] Zhang W, Li Y, Li X, Shao M, Mi Y, Zhang H, Zhi G. Deep Neural Network-Based SQL Injection Detection Method. Security and Communication Networks. 2022;2022(1):4836289.
[19] Roy P, Kumar R, Rani P. SQL injection attack detection by machine learning classifier. In2022 International conference on applied artificial intelligence and computing (ICAAIC) 2022 May 9 (pp. 394-400). IEEE.

[20] Katsogiannis-Meimarakis G, Koutrika G. A survey on deep learning approaches for text-to-SQL. The VLDB Journal. 2023 Jul;32(4):905-36.
[21] Khan W, Kumar T, Zhang C, Raj K, Roy AM, Luo B. SQL and NoSQL database software architecture performance analysis and assessments—a systematic literature review. Big Data and Cognitive Computing. 2023 May 12;7(2):97.
[22] Hong Z, Yuan Z, Zhang Q, Chen H, Dong J, Huang F, Huang X. Next-generation database interfaces: A survey of llm-based text-to-sql. arXiv preprint arXiv:2406.08426. 2024 Jun 12.
[23] Islam S. Future trends in SQL databases and big data analytics: Impact of machine learning and artificial intelligence. Available at SSRN 5064781. 2024 Aug 6.
[24] de Oliveira VF, Pessoa MA, Junqueira F, Miyagi PE. SQL and NoSQL Databases in the Context of Industry 4.0. Machines. 2021 Dec 27;10(1):20.
[25] Rockoff L. The language of SQL. Addison-Wesley Professional; 2021 Nov 4.
[26] Shivadekar S, Halem M, Yeah Y, Vibhute S. Edge AI cosmos blockchain distributed network for precise ablh detection. Multimedia tools and applications. 2024 Aug;83(27):69083-109.
[27] Fotache M, Munteanu A, Strîmbei C, Hrubaru I. Framework for the assessment of data masking performance penalties in SQL database servers. Case Study: Oracle. IEEE Access. 2023 Feb 22;11:18520-41.
[28] Panda SP. Augmented and Virtual Reality in Intelligent Systems. Available at SSRN. 2021 Apr 16.
[29] Karwin B. SQL Antipatterns, Volume 1: Avoiding the Pitfalls of Database Programming. The Pragmatic Programmers LLC; 2022 Oct 24.
[30] Nasereddin M, ALKhamaiseh A, Qasaimeh M, Al-Qassas R. A systematic review of detection and prevention techniques of SQL injection attacks. Information Security Journal: A Global Perspective. 2023 Jul 4;32(4):252-65.
[31] Chakraborty S, Aithal PS. CRUD Operation on WordPress Database Using C# SQL Client. International Journal of Case Studies in Business, IT, and Education (IJCSBE). 2023 Nov 28;7(4):138-49.
[32] Panda SP. The Evolution and Defense Against Social Engineering and Phishing Attacks. International Journal of Science and Research (IJSR). 2025 Jan 1.
[33] Choi H, Lee S, Jeong D. Forensic recovery of SQL server database: Practical approach. IEEE Access. 2021 Jan 18;9:14564-75.
[34] Thalji N, Raza A, Islam MS, Samee NA, Jamjoom MM. Ae-net: Novel autoencoder-based deep features for sql injection attack detection. IEEE access. 2023 Nov 28;11:135507-16.
[35] Chakraborty S, Paul S, Hasan KA. Performance comparison for data retrieval from nosql and sql databases: a case study for covid-19 genome sequence dataset. In2021 2nd International Conference on Robotics, electrical and signal processing techniques (ICREST) 2021 Jan 5 (pp. 324-328). IEEE.
[36] Crespo-Martínez IS, Campazas-Vega A, Guerrero-Higueras ÁM, Riego-DelCastillo V, Álvarez-Aparicio C, Fernández-Llamas C. SQL injection attack detection in network flow data. Computers & Security. 2023 Apr 1;127:103093.
[37] Antas J, Rocha Silva R, Bernardino J. Assessment of SQL and NoSQL systems to store and mine COVID-19 data. Computers. 2022 Feb 21;11(2):29.