Index review

# Check1 – Page density

## Description:

Low page density (also known as page fullness) can impact on query performance and resource consumption.

Each page in the database can contain a variable number of rows. If rows take all space on a page, page density is 100%. If a page is empty, page density is 0%. If a page with 100% density is split in two pages to accommodate a new row, the density of the two new pages is approximately 50%.

When page density is low, more pages are required to store the same amount of data. This means that more I/O is necessary to read and write this data, and more memory is necessary to cache this data. When memory is limited, fewer pages required by a query will be cached, causing even more disk I/O. Consequently, low page density negatively impacts performance. Also, for queries that read many pages using full or range index scans, low page density can degrade query performance because additional I/O may be required to read the data required by the query. Instead of a small number of large I/O requests, the query would require a larger number of small I/O requests to read the same amount of data.

Low page density may increase the number of intermediate B-tree levels. This moderately increases CPU and I/O cost of finding leaf level pages in index scans and seeks.

## Estimated Benefit:

Very High

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Review indexes and work to increase page density to fit more rows in a page.

### Detailed recommendation:

In many workloads, increasing page density results in a positive performance impact on query performance and resource consumption. To increase the page density, you should consider the following items:

* To avoid lowering page density unnecessarily, we do not recommend setting fill-factor to values other than 100 or 0, except in certain cases for indexes experiencing a high number of page splits.
* By default, SQL will store data from columns using large data types (varchar(max), nvarchar(max), varbinary(max), xml and etc) directly in the data row, up to a limit of 8000 bytes and as long as the value can fit in a page. This can cause a low page density as less rows would fit in a page. Consider to enable the option “large value types out of row” in a table to make SQL store the LOB data out of row, with a 16-byte pointer to the root page. This usually means more rows will fit per page, which can improve performance of queries that do not directly reference the LOB columns.
* Consider to use one of native data compression, such as, PAGE, ROW or Columnstore.
* Avoid fragmentation and low page density by reorganizing the indexes. Reorganizing compacts index pages to make page density equal to the fill-factor of the index.

# Check2 – Fragmentation

## Description:

In B-tree (rowstore) indexes, fragmentation exists when indexes have pages in which the logical ordering within the index, based on the key values of the index, does not match the physical ordering of index pages.

For queries that read many pages using full or range index scans, heavily fragmented indexes can degrade query performance because additional I/O may be required to read the data required by the query. Instead of a small number of large I/O requests, the query would require a larger number of small I/O requests to read the same amount of data.

## Estimated Benefit:

Very High

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Run index defragmentation.

### Detailed recommendation:

Reduce index fragmentation of biggest and most accessed objects by using one of the following methods:

* Reorganize an index
* Rebuild an index

For more details about the best index maintenance strategy that balances potential performance improvements against resource consumption required for maintenance check the following article: <https://learn.microsoft.com/en-us/sql/relational-databases/indexes/reorganize-and-rebuild-indexes?view=sql-server-ver16#index-maintenance-methods-reorganize-and-rebuild>

Note 1: Check the mentioned article for specific recommendations and details about an index maintenance on SQL Azure DB and SQL Managed Instances. An index maintenance may not be necessary in those environments as the rebuild or reorganize operation may degrade performance of other workloads due to resource contention.

Note 2: If available memory is enough to keep all the database pages in cache, fragmentation may be less important, but it is still important to use the available resources as best as possible and avoid extra storage space caused by the internal fragmentation.

Note 3:

Some important notes about why fragmentation still matters even on most modern storage hardware:

- Reading from memory is still a lot faster than reading from any storage (flash based or not) subsystem.

- Low page density (internal fragmentation) will require more pages to store the data, given the cost per gigabyte for high-end storage this could be quite significant.

- Index fragmentation affects the performance of scans and range scans through limiting the size of read-ahead I/Os. This could result in SQL Server not being able to take full advantage of the IOPS and I/O throughput capacity of the storage subsystem. Depending on the storage capability, SQL Server usually achieves a much higher I/O throughput as a direct consequence of requesting large I/Os, as an example, SQL Server can use read-ahead to do up to 8MB in a single I/O request on SQL EE and ColumnStore. It is definitely more efficient to issue 1 x 8-page read than 8 x 1-page reads.

- Index fragmentation can adversely impact execution plan choice: When the Query Optimizer compiles a query plan, it considers the cost of I/O needed to read the data required by the query. With low page density, there are more pages to read, therefore the cost of I/O is higher. This can impact query plan choice. For example, as page density decreases over time due to page splits, the optimizer may compile a different plan for the same query, with a different performance and resource consumption profile.

# Check12 – Index fill-factor

## Description:

The fill-factor option is provided for fine-tuning index data storage and performance. When an index is created or rebuilt, the fill-factor value determines the percentage of space on each leaf-level page to be filled with data, reserving the remainder on each page as free space for future growth. For example, specifying a fill-factor value of 80 means that 20 percent of each leaf-level page will be left empty, providing space for index expansion as data is added to the underlying table. The empty space is reserved between the index rows rather than at the end of the index.

A correctly chosen fill-factor value can reduce potential page splits by providing enough space for index expansion as data is added to the underlying table. When a new row is added to a full index page, the Database Engine moves approximately half the rows to a new page to make room for the new row. This reorganization is known as a page split. A page split makes room for new records, but can take time to perform and is a resource intensive operation.

Although a low, nonzero fill-factor value may reduce the requirement to split pages as the index grows, the index will require more storage space and can decrease read performance. Even for an application oriented for many insert and update operations, the number of database reads typically outnumber database writes by a factor of 5 to 10. Therefore, specifying a fill-factor other than the default can decrease database read performance by an amount inversely proportional to the fill-factor setting.

## Estimated Benefit:

Very High

## Estimated Effort:

Low

## Recommendation:

### Quick recommendation:

Review index fill-factor and work to increase page density to fit more rows in a page.

### Detailed recommendation:

It is particularly important to consider the costs and benefits of setting an index fill-factor other than 100, customers should perform it only when there is a demonstrated/documented need. For instance, setting a fill-factor to 80, will leave 20% of empty space on pages which will make your database 20% larger, table scans take 20% longer, maintenance jobs take 20% longer and your memory 20% smaller (as the empty space on pages are also in memory).

Set fill-factor to 100 to all indexes to avoid internal fragmentation, with exception of cases that there is a demonstrated/documented need.

Make sure that fill-factor of all indexes that first key is monotonically increasing are set to 100. If all the data is added to the end of the table, the empty space in the index pages will not be filled. For example, if the index key column is an IDENTITY column, the key for new rows is always increasing and the index rows are logically added to the end of the index. For those cases, a fill-factor of 100 is the recommended value.

To be able to do a fine tuning (correct balance and tradeoff between decrease read performance vs reduce page-splits and low page density) on the correct fill-factor value for an index, customers can create an session to capture the extended sqlserver.transaction\_log event and track mid-page splits in a database. For more details about this, check the following article: <https://www.sqlskills.com/blogs/jonathan/tracking-problematic-pages-splits-in-sql-server-2012-extended-events-no-really-this-time/>

Note: If available memory is enough to keep all the database pages in cache, fragmentation may be less important, but it is still important to use the available resources as best as possible and avoid extra storage space caused by the internal fragmentation.

# Check3 – Duplicated or overlapped indexes

## Description:

Duplicate indexes are indexes with a key identical or overlapped by another index. Like unused indexes, duplicates cost extra resources without providing a benefit.

## Estimated Benefit:

High

## Estimated Effort:

Very High

## Recommendation:

### Quick recommendation:

Remove or merge the indexes.

### Detailed recommendation:

Review the duplicate and overlapping indexes. Compare the keys, included columns and constraint to identify which indexes can be merged together and which can be safely dropped.

# Check9 – Hard coded indexes

## Description:

Some sql modules have fixed hard coded references to indexes. Because the SQL Server query optimizer typically selects the best execution plan for a query, we recommend that hints be used only as a last resort by experienced developers and database administrators. Also, if the referenced index is removed, the module will start to fail.

## Estimated Benefit:

Low

## Estimated Effort:

Medium

## Recommendation:

### Quick recommendation:

Avoid to use hard-coded indexes and review if they’re still best access option.

### Detailed recommendation:

Review the modules using the fixed indexes and if possible, remove its dependency.

Review the query execution plan using the forced index and make sure the plan is the best as compared to the query without the index hint.

# Check10 – Unused or rarely used indexes

## Description:

While proper indexes can greatly improve the performance, they can also negatively impact UPDATE, DELETE and INSERT operations. They also have a resource cost to maintain and can increase disk space consumption. Unused indexes have all these costs without providing any benefit.

Unused indexes can slow down database's performance. Time of write operations is increased because of index maintenance, but index is not used anywhere.

Note: We are considering indexes that are not part of a primary key or unique constraint and haven't been used since the last SQL restart. Index usage statistics are reset at each SQL restart, so it is best to collect these after the server has been running for some time.

## Estimated Benefit:

Medium

## Estimated Effort:

Very High

## Recommendation:

### Quick recommendation:

Remove unused indexes.

### Detailed recommendation:

Review the unused or rarely used indexes and drop them after confirm they’re unnecessary.

# Check11 – Key column size

## Description:

The maximum number of bytes in a clustered index key can't exceed 900. For a nonclustered index key, the maximum is 1,700 bytes.

You can define a key using variable-length columns whose maximum sizes add up to more than the limit. However, the combined sizes of the data in those columns can never exceed the limit.

## Estimated Benefit:

Low

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Review indexes with keys larger than the allowed size in bytes for the index.

### Detailed recommendation:

When you design an index that contains many key columns, or large-size columns, calculate the size of the index key to make sure that you do not exceed the maximum index key size. This excludes nonkey columns that are included in the definition of nonclustered indexes.

Review indexes key columns to avoid errors.

# Check14 – Non-unique clustered index

## Description:

Index uniqueness is highly desirable attribute of a clustering key, and goes hand-in-hand with index narrowness. SQL Server does not require a clustered index to be unique, but yet it must have some means of uniquely identifying every row. For non-unique clustered indexes, SQL Server adds to every duplicate instance of a clustering key value a 4-byte integer value called a uniqueifier. This uniqueifier is added everywhere the clustering key is stored. That means the uniqueifier is stored in both clustered and non-clustered indexes.

Non-unique clustered indexes will have an additional overhead at index creation as there's extra disk space and additional costs on INSERTs and UPDATEs.

## Estimated Benefit:

Low

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Review indexes cluster key and if possible, use a unique key.

### Detailed recommendation:

The choice of a “perfect clustered index key” depends of many factors (amount of memory available, DB size, number of non-clustered indexes, OLTP or OLAP workload, application orientation for writes vs reads, storage performance for random vs sequential I/Os, whether application queries are doing range or singleton reads using the key, just to list a few).

Experienced database administrators can design a good set of indexes, but this task is complex, time-consuming, and error-prone even for moderately complex databases and workloads. Understanding the characteristics of the database, queries, and data columns can help customers to design optimal indexes.

Since there are many factors should be considered, in general, we recommend customers to use the following general guidelines for a cluster key, the idea is that by following the general guidelines you should cover the best practices for most of the cases, and then, fine tune the exceptions.

General guidelines for a good clustered index key:

* Define the clustered index key with as few columns as possible.
* Keep the length of the index key short.
* Consider columns that have one or more of the following attributes:
  + Are unique or contain many distinct values.
  + Are accessed sequentially.
  + Used frequently to sort the data retrieved from a table.
  + Defined as IDENTITY.

Typically, a cluster index is good candidate when you have queries doing the following:

* Return a range of values by using operators such as BETWEEN, >, >=, <, and <=. After the row with the first value is found by using the clustered index, rows with subsequent indexed values are guaranteed to be physically adjacent. For example, if a query retrieves records between a range of sales order numbers, a clustered index on the column SalesOrderNumber can quickly locate the row that contains the starting sales order number, and then retrieve all successive rows in the table until the last sales order number is reached.
* Return large result sets.
* Reading many columns in a table.
* Use JOIN clauses; typically these are foreign key columns.
* Use ORDER BY or GROUP BY clauses.
* An index on the columns specified in the ORDER BY or GROUP BY clause may remove the need for the Database Engine to sort the data, because the rows are already sorted. This improves query performance.

# Check15 – GUID as index key

## Description:

A big reason for a clustered index is to retrieve rows for a range of values for a given column. Because the data is physically arranged in that order, the rows can be extracted very efficiently. Something like a GUID, while excellent for a primary key, could be positively detrimental to performance, as there will be additional cost for inserts and no perceptible benefit on selects.

## Estimated Benefit:

Medium

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Remove GUIDs in clustered indexes keys.

### Detailed recommendation:

The choice of a “perfect clustered index key” depends of many factors (amount of memory available, DB size, number of non-clustered indexes, OLTP or OLAP workload, application orientation for writes vs reads, storage performance for random vs sequential I/Os, whether application queries are doing range or singleton reads using the key, just to list a few).

Experienced database administrators can design a good set of indexes, but this task is complex, time-consuming, and error-prone even for moderately complex databases and workloads. Understanding the characteristics of the database, queries, and data columns can help customers to design optimal indexes.

Since there are many factors should be considered, in general, we recommend customers to use the following general guidelines for a cluster key, the idea is that by following the general guidelines you should cover the best practices for most of the cases, and then, fine tune the exceptions.

General guidelines for a good clustered index key:

* Define the clustered index key with as few columns as possible.
* Keep the length of the index key short.
* Consider columns that have one or more of the following attributes:
  + Are unique or contain many distinct values.
  + Are accessed sequentially.
  + Used frequently to sort the data retrieved from a table.
  + Defined as IDENTITY.

Typically, a cluster index is good candidate when you have queries doing the following:

* Return a range of values by using operators such as BETWEEN, >, >=, <, and <=. After the row with the first value is found by using the clustered index, rows with subsequent indexed values are guaranteed to be physically adjacent. For example, if a query retrieves records between a range of sales order numbers, a clustered index on the column SalesOrderNumber can quickly locate the row that contains the starting sales order number, and then retrieve all successive rows in the table until the last sales order number is reached.
* Return large result sets.
* Reading many columns in a table.
* Use JOIN clauses; typically, these are foreign key columns.
* Use ORDER BY or GROUP BY clauses.
* An index on the columns specified in the ORDER BY or GROUP BY clause may remove the need for the Database Engine to sort the data, because the rows are already sorted. This improves query performance.

Note: There are several cases where a GUID as a key column is acceptable, therefore, customers should evaluate each case to confirm.

# Check16 – Foreign keys without index

## Description:

Foreign key constraints should have corresponding indexes.

## Estimated Benefit:

Medium

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Create a corresponding index on each foreign key.

### Detailed recommendation:

A general guideline and a good rule of thumb is to have an associated **covered** index for any foreign key columns that are commonly used in join operations.

But, the best guidance is “it depends” and is better to base it on the workload.

In a perfect world, you would have to analyze the workload and understand how the data is used, the types of queries and the frequency they run. You would also have to review every single statement and analyze the query execution plan to make sure all required indexes are available and the performance comply with the application user experience expectations, requirements and the reads vs writes tradeoff performance.

In a real world, is very unlikely that you will have time to analyze case by case. You’ll have to decide whether the impact caused by the “auto-indexed foreign keys” is higher than the benefits, you'll need to do some experiments to find your own sweet spot. In my experience, the cost of having extra indexes is often lower than the benefits it provides.

Some benefits of having indexed foreign keys:

* Improve joins and table access:
  + Foreign key columns are frequently used in join criteria when the data from related tables is combined in queries by matching the column or columns in the foreign key constraint of one table with the primary or unique key column or columns in the other table. An index enables the database engine to quickly find related data in the foreign key table. However, creating an index is not required, but, the presence of a foreign key relationship between two tables indicates that the two tables have been optimized to be combined in a query that uses the keys as its criteria.
* Improve delete (modifications) statements:
  + When you delete a key row, the database engine must check to see if there are any rows which reference the row being deleted. Let’s suppose a classic scenario containing an users table with column user\_id being replicated in several tables across the database. So, customers, products, orders and other 200 tables will have an user\_id column with a FK pointing back to users table. Now, suppose someone is trying to remove an user and trying to run “delete from users where user\_id = 10”, if you’re lucky, you’ll receive an exception with a “The query processor ran out of internal resources and could not produce a query plan.”, but, if you’re not lucky, you’ll have a huge query plan with an access to each table that depends on the user\_id. If the foreign keys are not indexed, you’ll have to pay for a scan (that can be very expensive) to check if the user\_id you’re trying to remove exists on the related tables, that will include acquire and hold all required locks related to the operation while the transaction is running. So, an index can not also help on performance of select statements, but it can also be very helpful to improve modifications.
* Trusted foreign keys can help query optimizer to create better plans.

Other considerations:

* Make sure you’re creating the correct index structure, although an index can help to speed up a query, a seek in a covered index is faster than a seek + lookup, so, make sure that when possible (the overhead is not too high), you are including the correct columns in the index to avoid the expensive and unordered lookup operations. If the optimizer can retrieve all the data it needs from a nonclustered index without having to reference/lookup the underlying table, it will do so and have better performance.
* This debate is very similar to the “what is the maximum number of indexes I should have in a single table?”, well, it depends, but, it is often a lot better to have indexes then don’t have it. Remember that any data you want to modify has to be found first, so, the extra cost you pay for a modification in a indexed table is often lower than the benefit of find the row you want to modify quickly.
* Before create an index, check if it is possible to adjust and consolidate any existing index to achieve the same goal.
* It is always a good practice to analyze the index usage to identify non-used of rarely used indexes. If you decide to create indexes on all foreign keys, after a few days of index usage monitoring, you could identify whether they’re being useful or not and then remove the non-used indexes.
* If possible, I highly recommend you to create the indexes in a dev/test environment, replay the production workload and measure the performance impact/differences.
* If your query is joining tables by using columns that does not have a foreign key, it may be helpful to create indexes on those columns for the same reason already mentioned.

Interesting note: MySQL requires foreign keys to be indexed, as [per their documentation](https://dev.mysql.com/doc/refman/8.0/en/constraint-foreign-key.html): “MySQL requires that foreign key columns be indexed; if you create a table with a foreign key constraint but no index on a given column, an index is created.”. I’m not saying this is good, but, just saying.

# Check38 – Heaps and tables with no index

## Description:

Poorly designed indexes and a lack of indexes are primary sources of database application bottlenecks. Designing efficient indexes is paramount to achieving good database and application performance.

When a table has a clustered index, the table is called a clustered table. If a table has no clustered index, its data rows are stored in an unordered structure called a heap. If a table is a heap and does not have any nonclustered indexes, then the entire table must be read (a table scan) to find any row.

## Estimated Benefit:

High

## Estimated Effort:

Low

## Recommendation:

### Quick recommendation:

Create appropriate indexes to improve performance of queries.

### Detailed recommendation:

There are sometimes good reasons to leave a table as a heap instead of creating a clustered index, but using heaps effectively is an advanced skill. Most tables should have a carefully chosen clustered index unless a good reason exists for leaving the table as a heap.

Review all tables with no indexes and make sure they’re really used.

Create appropriate indexes to improve performance of queries.

Usually, some acceptable usages for heaps are:

* Heaps can be used as staging tables for large, unordered insert operations.
* Sometimes data professionals also use heaps when data is always accessed through nonclustered indexes, and the RID is smaller than a clustered index key.
* Very small tables.

# Check18 – Heap with a non-clustered index

## Description:

In some cases, it may be possible to have a non-clustered created index created in a heap table, as the access through nonclustered indexes are fast than scanning the heap, and the RID (row identifier consisting of the file number, data page number, and slot on the page) is smaller than a clustered index key and an efficient structure.

## Estimated Benefit:

Medium

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Review heap tables and consider to create a clustered index.

### Detailed recommendation:

Review queries used to access the tables, and if possible, create a clustered index on the table.

Review queries used to access the tables, and if necessary, adjust the non-clustered index to add any missing columns that may be accessed via RID lookup operation. In other words, create a covered index.

# Check20 – Table index not aligned with PS

## Description:

A non-aligned index is partitioned differently from its corresponding table. That is, the index has a different partition scheme that places it on a separate filegroup or set of filegroups from the base table.

## Estimated Benefit:

High

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Re-create indexes to have it aligned with current schema.

### Detailed recommendation:

Review indexes and make sure they’re aligned the same way as your clustered index to avoid valuable data management features available on partitioned tables.

Check21 – Missing index DMV

Description:

Potentially missing indexes were found based on missing index SQL Server DMVs. It is important to revise them.

## Estimated Benefit:

High

## Estimated Effort:

High

Recommendation:

Quick recommendation:

Review recommended missing indexes and if possible, create them.

Detailed recommendation:

Review missing index suggestions to effectively tune indexes and improve query performance. Review the base table structure, carefully combine indexes, consider key column order, and review included column suggestions, examine missing indexes and existing indexes for overlap and avoid creating duplicate indexes.

It's a best practice to review all the missing index requests for a table and the existing indexes on a table before adding an index based on a query execution plan.

Missing index suggestions are best treated as one of several sources of information when performing index analysis, design, tuning, and testing. Missing index suggestions are not prescriptions to create indexes exactly as suggested.

Review the missing index recommendations for a table as a group, along with the definitions of existing indexes on the table. Remember that when defining indexes, generally equality columns should be put before the inequality columns, and together they should form the key of the index. To determine an effective order for the equality columns, order them based on their selectivity: list the most selective columns first (leftmost in the column list). Unique columns are most selective, while columns with many repeating values are less selective.

It's important to confirm if your index changes have been successful, “is the query optimizer using your indexes?”. Keep in mind that while indexes can dramatically improve query performance but indexes also have overhead and management costs.

Check22 – Missing index plan cache

Description:

Potentially missing indexes were found based on SQL Server query plan cache. It is important to revise them.

## Estimated Benefit:

Very High

## Estimated Effort:

High

Recommendation:

Quick recommendation:

Review recommended missing indexes and if possible, create them.

Detailed recommendation:

Review missing index suggestions to effectively tune indexes and improve query performance. Review the base table structure, carefully combine indexes, consider key column order, and review included column suggestions, examine missing indexes and existing indexes for overlap and avoid creating duplicate indexes.

It's a best practice to review all the missing index requests for a table and the existing indexes on a table before adding an index based on a query execution plan.

Missing index suggestions are best treated as one of several sources of information when performing index analysis, design, tuning, and testing. Missing index suggestions are not prescriptions to create indexes exactly as suggested.

Review the missing index recommendations for a table as a group, along with the definitions of existing indexes on the table. Remember that when defining indexes, generally equality columns should be put before the inequality columns, and together they should form the key of the index. To determine an effective order for the equality columns, order them based on their selectivity: list the most selective columns first (leftmost in the column list). Unique columns are most selective, while columns with many repeating values are less selective.

It's important to confirm if your index changes have been successful, “is the query optimizer using your indexes?”. Keep in mind that while indexes can dramatically improve query performance but indexes also have overhead and management costs.

# Check23 – Index naming convention

## Description:

There are different schools of thought on how to name indexes and there is not a single best practice that works for everyone, but it is important to whatever you do, be consistent and avoid to use “test”, “\_DTA\_”, “missing index” on index name to avoid confusion. The purpose of having a good naming convention is to increase code readability.

## Estimated Benefit:

Low

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Review index name and if possible, rename it to follow a defined naming convention.

### Detailed recommendation:

Before applying indexes from the DTA or a [<Name of Missing Index, sysname,>], it is recommended that the names of indexes be changed to match your organization’s index naming standards.

# Check26 – Lock escalation

## Description:

Lock escalation is the process of converting many fine-grained locks (such as row or page locks) to table locks. SQL Server dynamically determines when to do lock escalation. When it makes this decision, SQL Server considers the number of locks that are held on a particular scan, the number of locks that are held by the whole transaction, and the memory that's used for locks in the system as a whole.

## Estimated Benefit:

Medium

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Check if lock escalation is causing unexpected blocking and work to resolve it.

### Detailed recommendation:

Review TOP N indexes by number of lock escalations and make sure this is expected and it is not blocking other users trying to access the index.

Some application or query designs might trigger lock escalation at a time when this action not desirable, and the escalated table lock might block other users. To determine whether lock escalation is occurring at or near the time when you experience blocking issues, start an Extended Events session that includes the lock\_escalation event.

The simplest and safest method to prevent lock escalation is to keep transactions short and reduce the lock footprint of expensive queries so that the lock escalation thresholds are not exceeded.

Reduce the query's lock footprint by making the query as efficient as possible. Large scans or many bookmark lookups can increase the chance of lock escalation. Additionally, these increase the chance of deadlocks, and adversely affect concurrency and performance. Review the execution plan and potentially create new nonclustered indexes to improve query performance.

Although it's possible to disable lock escalation in SQL Server, we recommend it to be used only as a last resort by experienced developers and database administrators.

# Check24 – Indexes waiting on physical disk I/O operations

## Description:

SQL Server tracks how much time the Database Engine spend waiting to complete a physical disk I/O request. This information is stored at the index level access and can be used to identify what are the objects that are spending most of the time on disk I/O operations. Long I/O waits may indicate problems with the disk subsystem.

## Estimated Benefit:

High

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Review TOP N indexes and work to minimize the physical disk I/O operations.

### Detailed recommendation:

Long waits for disk I/O operations usually indicate the I/O subsystem is overloaded, but it is also very common that the problem is with SQL Server queries other than with the I/O subsystem. The question you need to ask is, “why SQL Server is doing so many reads?”. It is not unusual to see that, a missing index or an unexpected table scan are doing a very high number of I/O requests.

Apply index compression to increase chances of having the data in memory and avoid the physical reads.

Reduce index fragmentation and increase page density. Queries that read many pages can degrade query performance because number of I/O requests required to read the data. Instead of a high number of large I/O requests, a query using an optimized index would require a small number of I/O requests to read the same amount of data.

# Check25 – Indexes by memory usage

## Description:

Measures the amount of memory used in the buffer cache by the largest object (based on the number of pages). It checks the sys.dm\_os\_buffer\_descriptors to identify the object, and returns the relative percentage used. This information is important if you want to monitor what is in the buffer area, or if you are having performance-related disk read problems.

Memory is one of the most important resources for SQL Server, so it’s important to make sure SQL Server is using it efficiently. For example, if 90% of the buffer pool (memory area) is being used to store data from one table, it is important to try to optimize the size of this table to save space for other tables in memory. It is very common for one or two objects to be responsible for using a large amount of the buffer cache. To increase the efficiency of the buffer cache area, these objects may benefit from a schema revision (datatype changes or sparse columns), and are great candidates for compression.

## Estimated Benefit:

High

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Work to improve memory usage.

### Detailed recommendation:

Review the queries using the TOP N indexes and make sure they’re not doing table scan, if necessary, create the needed indexes to avoid it.

Review the buffer pool free space and work to increase page density by defragmenting the index and adjusting the fill-factor.

Apply index compression to increase amount of data we’ll be able to cache using the available memory.

# Check27 – Index scans

## Description:

Reports the TOP indexes and number of scans by user queries that did not use 'seek' predicate. Normally the index seek is faster than index scan since a scan reads all the rows in an index – B-tree in the index order whereas index seek traverses a B-tree and walks through leaf nodes seeking only the matching or qualifying rows based on the filter criteria.

## Estimated Benefit:

Medium

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Review queries using the indexes and optimize it.

### Detailed recommendation:

Identify the queries doing the scans and work in a performance tuning review to improve it and create the necessary indexes to solve the problem.

The plan cache and/or query store can be used to find the plans doing the scan.

# Check28 – Indexes per key column count

## Description:

Reports the TOP indexes and number columns in the index key.

## Estimated Benefit:

Medium

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Review the number of key columns on TOP N indexes.

### Detailed recommendation:

Review the number of key columns on TOP N indexes and make sure they’re following the best practices do define an index. It is usually a best practice to define the index key with as few columns as possible and keep the length of the index key short.

Redesign nonclustered indexes with a large index key size so that only columns used for searching and lookups are key columns. Make all other columns that cover the query included nonkey columns. In this way, you will have all columns needed to cover the query, but the index key itself is small and efficient.

Avoid adding unnecessary columns. Adding too many index columns, key or nonkey, can have the performance implications such as: Fewer index rows will fit on a page (which could create I/O increases and reduced cache efficiency); more disk space will be required to store the index;

# Check29 – Tables with more indexes

## Description:

Reports the TOP tables and number of indexes. Large numbers of indexes on a table affect the performance of INSERT, UPDATE, DELETE, and MERGE statements because all indexes must be adjusted appropriately as data in the table changes.

## Estimated Benefit:

Medium

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Review TOP N tables and make sure all indexes are used and required.

### Detailed recommendation:

Review the TOP tables and make sure there are no duplicated, overlapped or non-used indexes. Compare the keys, included columns and constraint to identify which indexes can be merged together and which can be safely dropped.

Databases or tables with low update requirements, but large volumes of data can benefit from many nonclustered indexes to improve query performance. So, in some cases, it may be ok to have a table with a large number of indexes.

The selection of the right indexes for a database and its workload is a complex balancing act between query speed and update cost. You will have to determine whether the gains in query performance outweigh the effect to performance during data modification and in additional disk space requirements.

# Check31 – Most accessed indexes and number of singleton lookups and range scans

## Description:

Reports the TOP most accessed indexes and number of singleton lookups and range scans. If most of reads are singleton lookups, it may be a good idea to re-create the index as a nonclustered and use cluster in a column that requires range scans.

## Estimated Benefit:

Medium

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Review the cluster index access and check if it is worthy to re-create it as a nonclustered.

### Detailed recommendation:

One of most important aspects of a clustered index, is that all columns of a table are stored on it. This means that it is very good for supporting range queries and reading several columns.

Although a clustered is often (because it is usually defined as the primary key) used to return a single row (a singleton lookup) from the index, it could be better used on range queries and queries reading several columns.

If most of access to the clustered index is returning a single row, consider to re-create it as a non-clustered and define the clustered in another column (or an existing non-clustered index) that is used on range filters and/or return several columns. The non-clustered indexes are considered better (and good candidates to become clustered indexes) compared to the existing clustered indexes if the number of user seeks on those indexes is greater than the number of lookups on the related to the table clustered index.

# Check32 – Index with size greater than table

## Description:

Reports all tables that space to store indexes are greater than the base table size.

## Estimated Benefit:

Medium

## Estimated Effort:

High

## Recommendation:

### Quick recommendation:

Review indexes to confirm they are really necessary.

### Detailed recommendation:

Review the reported indexes to confirm they are really necessary and are being used.

Check if there are indexes with a high number of include columns and confirm queries are really using all the indexed columns. Avoid adding unnecessary columns in an index, adding too many index columns, key or nonkey, can have performance implications.

# Check33 – Row/Page lock disabled

## Description:

Reports indexes that have options ALLOW\_ROW\_LOCKS or ALLOW\_PAGE\_LOCKS set to OFF (default is ON).

Setting those options to OFF can cause unexpected blocking issues and degrade query performance.

## Estimated Benefit:

Very High

## Estimated Effort:

Medium

## Recommendation:

### Quick recommendation:

Set ALLOW\_ROW\_LOCKS or ALLOW\_PAGE\_LOCKS to ON.

### Detailed recommendation:

Review the reported indexes to confirm that disable ALLOW\_ROW\_LOCKS or ALLOW\_PAGE\_LOCKS is really necessary and if not, set it back to ON.

# Check34 – Tables greater than 10mi and not partitioned

## Description:

Reports all indexes greater than 10 million rows and are not using partitioning.

There are several performance and manageability benefits of configure partitioning on large tables or indexes.

## Estimated Benefit:

Medium

## Estimated Effort:

Very High

## Recommendation:

### Quick recommendation:

Consider to implement a purge/archive strategy or implement partitioning on large tables.

### Detailed recommendation:

Review reported tables and implement a purge/archive strategy.

Review reported tables and consider to implement SQL Server native partitioning or partitioned views.

# Check35 – Indexes that are not compressed

## Description:

SQL Server support row and page compression for rowstore tables and indexes, and support columnstore and columnstore archival compression for columnstore tables and indexes.

In addition to saving space, data compression can help improve performance of I/O intensive workloads because the data is stored in fewer pages and queries need to read fewer pages from disk.

## Estimated Benefit:

Very High

## Estimated Effort:

Medium

## Recommendation:

### Quick recommendation:

Consider to enable SQL Server native page, row or columnstore compression on reported indexes.

### Detailed recommendation:

Consider to enable SQL Server native page, row or columnstore compression on reported indexes. Compression will help to reduce I/O reads and writes operations and increase buffer pool data cache efficiency.

Off-row data is not compressed when enabling data compression. For example, an XML record that's larger than 8060 bytes will use out-of-row pages, which are not compressed. As an alternative to compress data for those columns, you can consider the following options:

* Use SQL native COMPRESS function that will use the GZIP algorithm format.
* Create a CLR function to apply the compression.
* Create a clustered columnstore index.

Note: Extra CPU resources are required on the database server to compress and decompress the data, while data is exchanged with the application. It is recommended to evaluate the trade-off of reduce storage space vs the extra CPU cost.

Check36 – Indexes in a key column set to ascending

Description:

Review reported indexes and make sure the frequency you update the index statistics is enough to provide accurate information. Statistics on ascending or descending key columns, such as IDENTITY or real-time timestamp columns, might require more frequent statistics updates than the Query Optimizer performs.

Insert operations append new values to ascending or descending columns. The number of rows added might be too small to trigger a statistics update. If statistics are not up-to-date and queries select from the most recently added rows, the current statistics will not have cardinality estimates for these new values. This can result in inaccurate cardinality estimates and slow query performance.

For example, a query that selects from the most recent sales order dates will have inaccurate cardinality estimates if the statistics are not updated to include cardinality estimates for the most recent sales order dates.

## Estimated Benefit:

Medium

## Estimated Effort:

High

Recommendation:

Quick recommendation:

Review queries using reported indexes and make sure you’re update statistics on those indexes.

Detailed recommendation:

Review queries using reported indexes and check if queries using those indexes are trying to read latest inserted rows. Check if queries are using predicates beyond the RANGE\_HI\_KEY value of the existing statistics, if so, make sure you've a script to update the statistic more often to guarantee those queries will have Information about newest records.

If you can't spend time looking at all queries using those tables, you can create a job to update those statistics more often. Make sure your script is smart enough to only run update if number of modified rows changed. Probably an update with sample is enough, as long as you do a bigger sample in the regular maintenance window update.

Note: On KB3189645 (SQL2014 SP1 CU9(12.00.4474) and SP2 CU2(12.00.5532)) filtered indexes are exempted from quickstats queries because it had a bug with filtered indexes and columnstore, but, that ended up fixing another problem that when the quickstats query was issued for filtered index stats it has no filter, which was making a full scan (unless a nonfiltered index with the same first column happens to be around to help).

Check37 – Indexes and lock wait time

Description:

Queries using indexes with high wait on lock operations can experience degraded concurrency and performance.

## Estimated Benefit:

Medium

## Estimated Effort:

High

Recommendation:

Quick recommendation:

Review reported indexes and check if queries using those indexes are experiencing unexpected blocking and work to resolve it.

Detailed recommendation:

Identify queries holding locks on the reported indexes and causing blocks and work to reduce the query's lock footprint by making the query as efficient as possible. Large scans or many bookmark lookups can increase the chance of blocking problems. Additionally, these increase the chance of deadlocks, and adversely affect concurrency and performance. Review the execution plan and potentially create new nonclustered indexes to improve query performance.

Avoid to use lock table hints and review transaction isolation level to confirm this is using the correct option.

Consider to implement snapshot isolation level.

Check39 – Identity columns

Description:

Tests that identity columns values are not getting close to the maximum value for the column data type.

## Estimated Benefit:

Medium

## Estimated Effort:

High

Recommendation:

Quick recommendation:

Review identity columns with low percent of available values.

Detailed recommendation:

If an identity column is getting close to the limit of the datatype, you need to know so that you can avoid logical problems in your application and SQL Server errors. For example, if you created an IDENTITY column of smallint datatype, if you try to insert more than 32767 rows in the table, you will get the following error:

Server: Msg 8115, Level 16, State 1, Line 1

Arithmetic overflow error converting IDENTITY to data type smallint. Arithmetic overflow occurred.

This is a limitation of the datatype, not the identity.

Compression savings (check 45)

Description:

Estimate compression savings

## Estimated Benefit:

Very High

## Estimated Effort:

High

Recommendation:

Quick recommendation:

Review indexes and apply recommended compression algorithm.

Detailed recommendation:

ROW compression adds minimal CPU utilization while providing substantial storage and memory savings. For most workloads, ROW compression should be enabled by default for new tables and indexes.

PAGE compression adds higher CPU utilization, but also provides higher storage and memory savings.

Additional analysis and testing may be required to select data compression types optimally.

Any data compression requires additional CPU processing; thus, testing is particularly important when low query latency must be maintained in transactional workloads.

Notes:

- Writes\_Ratio shows the ratio of update/modifications operations on a specific index relative to total operations on that object. The lower the ratio (that is, the table, index, or partition is infrequently updated), the better candidate it is for page compression.

- Don't ignore compression for small objects, consider future growth when deciding what to compress. It is easier to compress a table while it is small than do it once it has become large.

Recommendation based on check analysis:

Use the following script to apply compression in the most appropriate compression algorithm and order based in our analysis that is considering a few items like:

* Index buffer pool usage
* Index size
* The compression ratio
* If index is already compressed and NONE compression ratio is >= -3 and <= 3%, then set preferable compression algorithm to NONE, as it is probably better to leave it as NONE to avoid extra overhead for a little gain.
* If index is already compressed with PAGE and ROW compression ratio is >= -3 and <= 3%, then set preferable compression algorithm to ROW, as it is probably better to leave it as ROW to avoid extra overhead for a little gain compared to PAGE.
* If row compression ratio is >= 10% and ratio difference from ROW to PAGE is less than 10%, then set preferable compression algorithm to ROW.
* If page compression ratio is >= 25% then set preferable compression algorithm to PAGE.
* An index >= 5% of total buffer pool data cache size (since "big" can vary on each environment, I'm using BP as a measure to help answer this.) is considered a big table, will gain an extra weight on final formula. The idea is to favor compression on big tables.
* Indexes with a high percent (default of 40%) of reads compared to writes will get an extra weight on final formula. The idea is to favor compression on tables with a high number of reads compared to writes.
* Indexes with a high percent (default of 40%) of writes compared to reads will get a -1 weight on final formula. The idea is to disfavor compression on tables with a high number of writes compared to reads.
* Indexes with high waits (default of TOP 500 indexes by page IO latch) on page IO latch will get an extra weight on final formula. The idea is to favor compression on indexes waiting to be read from disk.
* Indexes with a high (default of TOP 500 indexes by buffer pool memory usage) buffer pool memory usage will get an extra weight on final formula. The idea is to favor compression on indexes using a lot of memory.
* If PLE is low (default of 500), double the @TopNBufferPool (default of 500) value to get more indexes compressed.
* Most used (default of TOP 200 indexes by avg of access per minute) indexes will get an extra weight on final formula. The idea is to favor compression on most used indexes.