IDERA® | Whitepaper

THE TOP 4 PERFORMANCE MAKERS AND BREAKERS FOR SQL SERVER

TABLE OF CONTENTS

INTRODUCTION

CONSIDER THIS: SUCCESSFUL SOLUTIONS ARE SUCCESSFUL SOLELY BECAUSE SYSTEMS, USERS, AND ORGANIZATIONS HAVE COME TO DEPEND UPON THEM. NOT NECESSARILY BECAUSE THEY WERE WELL CRAFTED, INTELLIGENTLY DESIGNED, OR ADHERE TO ANY OTHER THEORETICAL STANDARDS OR BEST PRACTICES. THE MOST NECESSARY AND ADOPTED APPLICATION MAY BE (1) A WELL-OILED MACHINE, (2) SOMEHOW SUCCEEDING "DESPITE ITSELF", OR (3) SOMETHING IN BETWEEN.

One of the beautiful things about SQL Server is how accessible it is. It's about as difficult to install as say, Microsoft Office. This is great if you want a basic sandbox server to play around with, or a database to support just a handful of users. For SQL Server to perform as an Enterprise Level DBMS, the "out of the box" install rarely cuts it. In addition, there are design considerations, configuration and maintenance that must be considered and implemented to enable the true capability of the platform.

A common scenario is that what started out as a prototype solution or something designed for just a few users becomes widely adopted by a company, and mission-critical. More and more users jump on board, and performance degrades. When a solution's success starts to overwhelm its bearing capacity, there is a problem. Some applications reach this point almost immediately, either because they were more successful than anticipated or because they were haphazardly designed from inception.

SQL Server can be an extremely scalable and fast solution – but you must work with it and not against it!

INDEXES

Indexes, Indexes, Indexes. They can be the lifesaver that changes the performance of a query like some kind of magic fairy dust. They can also be the reason a server is overworked and sluggish. Indexes is a broad subject, but there are some general areas on which you can focus if you're looking at performance and scalability: Clustered Index Design, Unused Indexes, Desired Indexes, and Fragmentation.

Clustered Index Design

Clustered indexes are one of those things that can on one hand be wonderful and enabling, but on the other hand (when not well architected) be the culprit of a lot of unnecessary work for a SQL Server instance. The ideal clustered index should have the following qualities (in no particular order):

Sequential in Nature

When designing a clustered index, choosing a column (or combination of columns) that are sequential in nature (or ever-increasing) allows SQL Server to keep writing on the last page of the table until full, at which point it allocates a new page and begins writing there. This accomplishes two things at the same time:

1. Reduced Fragmentation – by always inserting records at the end of the table and not in random places throughout the table, there is always room on the current page for the data. This avoids page splits to accommodate an insert that will not logically fit on a page.

2. More Condensed Data – If a clustered index is sequential in nature (or any index for that matter), then taking the fill factor up to 100 or close to it becomes a design choice that may make sense. More fill factor = less data pages = smaller table size = faster scans. It's all good stuff here if the shoe fits.

By the way, when it comes to Fill Factor, 0 and 100 are the same thing – which can be very misleading. On the other hand, 1 and 99 couldn't be more different!

Small Data Type(s)

Choosing columns with the smallest reasonable data type when designing a clustered index has multiple performance-related benefits. The columns in a Unique Clustered Index are used in all non-clustered indexes as the lookup key when tying back to a clustered index to retrieve additional fields needed while executing a query. In other words, the columns in a Unique Clustered Index become like an invisible suffix on every non-clustered index. The larger the datatype of the clustered index key, the more space is used with each row of each non-clustered index to accommodate the key values. This can get especially burdensome when a combination clustered key is chosen (more than one column).

In the example below, two tables are created. A data generator is used to create 1 million rows of data in the first table, which is then propagated to the 2nd table (data in both tables is identical). In Table A, a unique, clustered index is applied to a single INT data type column. In Table B, a unique, clustered index is created by using a combination of two columns, the first being a varchar(50), followed by a char(12). Two identical non-clustered indexes are then applied to each table. The script to create this same scenario is below (sans data insert).

UNIQUE CLUSTERED INDEX DESIGN EXAMPLE

TABLE A **INT – SINGLE COLUMN**	**TABLE B** **VARCHAR(50), CHAR(12) – TWO COLUMNS**
CREATE TABLE CUSTOMER_EXAMPLE_INT (CUSTOMER_ID INT NOT NULL IDENTITY(1,1), CUSTOMER_KEY CHAR(12) NOT NULL, CUSTOMER_NAME VARCHAR(50) NOT NULL, CUSTOMER_ADDRESS VARCHAR(200), CUSTOMER_CITY VARCHAR(100), CUSTOMER_PROVINCE CHAR(2), CUSTOMER_POSTAL_CODE VARCHAR(10)) GO CREATE UNIQUE CLUSTERED INDEX IX_CUSTOMER_ EXAMPLE_INT ON CUSTOMER_EXAMPLE_INT (CUSTOMER_ID) GO CREATE INDEX IX_CUSTOMER_EXAMPLE_INT_POSTALCODE ON CUSTOMER_EXAMPLE_INT (CUSTOMER_POSTAL_CODE) GO CREATE INDEX IX_CUSTOMER_EXAMPLE_INT_CUSTOMER_ NAME ON CUSTOMER_EXAMPLE_INT (CUSTOMER_NAME) GO	CREATE TABLE CUSTOMER_EXAMPLE_NAME_KEY (CUSTOMER_ID INT NOT NULL, CUSTOMER_KEY CHAR(12) NOT NULL, CUSTOMER_NAME VARCHAR(50) NOT NULL, CUSTOMER_ADDRESS VARCHAR(200), CUSTOMER_CITY VARCHAR(100), CUSTOMER_PROVINCE CHAR(2), CUSTOMER_POSTAL_CODE VARCHAR(10)) GO CREATE UNIQUE CLUSTERED INDEX IX_CUSTOMER_ EXAMPLE_NAME_KEY ON CUSTOMER_EXAMPLE_NAME_KEY (CUSTOMER_NAME, CUSTOMER_KEY) GO CREATE INDEX IX_CUSTOMER_EXAMPLE_NAME_KEY_ POSTALCODE ON CUSTOMER_EXAMPLE_NAME_KEY (CUSTOMER_POSTAL_CODE) GO CREATE INDEX IX_CUSTOMER_EXAMPLE_NAME_KEY_ CUSTOMER_NAME ON CUSTOMER_EXAMPLE_NAME_KEY (CUSTOMER_NAME) GO

The above script can be downloaded here

One can investigate the Index Depth and Page Usage of both Table A and Table B by using the following DMO and replacing [TABLENAME] with each table name. This represents the B-Tree structure of the index. One way to visualize this is to imagine forks in the road on the way to your destination, each with a sign telling you to go left or right, directing you closer to the information you are trying to acquire. Table A would have 2 intersections at which to decide “left or right”, whereas Table B would have 3 (more navigation and a longer road to your destination).

```
SELECT
INDEX_DEPTH,
INDEX_LEVEL,
RECORD_COUNT,
AVG_PAGE_SPACE_USED_IN_PERCENT,
AVG_RECORD_SIZE_IN_BYTES
FROM
SYS.DM_DB_INDEX_PHYSICAL_STATS(DB_ID(), OBJECT_ID([TABLENAME]), 1, NULL, 'DETAILED')
ORDER BY INDEX_LEVEL DESC
GO
```

TABLE A
INT – SINGLE COLUMN

Index Depth	Level	Rec Qty	Avg % Page Used	Avg Rec Size in Bytes
3	2	185	29%	11
3	1	50000	43%	11
3	0	100000	98%	395

TABLE B
VARCHAR(50), CHAR(12) – TWO COLUMNS

Index Depth	Level	Rec Qty	Avg % Page Used	Avg Rec Size in Bytes
4	3	8	7%	73
4	2	471	54%	73
4	1	50004	98%	73
4	0	1000000	98%	395

The most obvious difference is that the table with the combination unique clustered index (two columns) has an additional non-leaf level, giving 4 levels to its tree, as opposed to only 3 levels for the int key. The simple reason for this is that the two columns take up significantly more space than the int data type, and so when we create a clustered key, fewer rows can be packed into each index page, and the clustered key requires an additional non-leaf level to store the keys.

Conversely, using a narrow int column for the key allows SQL Server to store more data per page, meaning that it must traverse fewer levels to retrieve a data page, which minimizes the IO required to read the data. The potential benefit of this is large, especially for queries that utilize a range scan. In general, the more data you can fit onto a page, the better your table can perform. A purposefully efficient data type is an essential component of good database design.

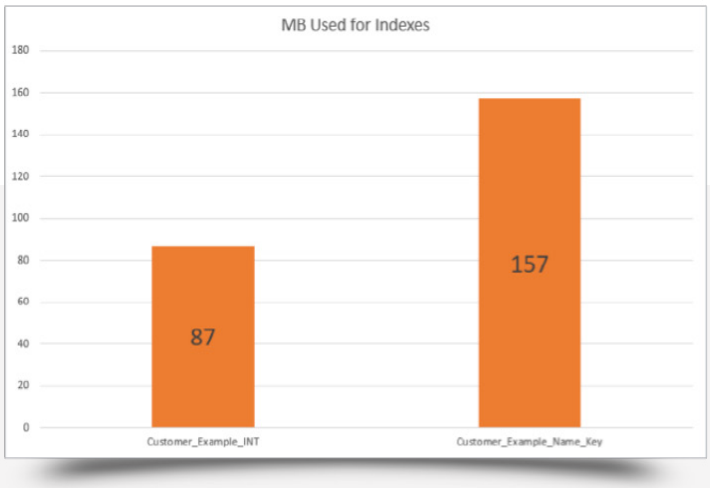
The choice of the clustering key can affect the performance of not only the clustered index, but also any non-clustered indexes that rely on the clustered index. A non-clustered index contains the clustered index key(s) in every level of its b-tree structure, as a pointer back into the clustered index (see fields in RED in the table below). This happens whether the clustering key was explicitly included in the non-clustered index structure or not.

CUSTOMER_EXAMPLE_INT (TABLE A)		
INDEX_ID	**INDEXED COLUMN(S)**	**ACTUAL COLUMNS IN INDEX**
1	CUSTOMER_ID	CUSTOMER_ID
2	POSTAL_CODE	POSTAL_CODE, CUSTOMER_ID
3	CUSTOMER_NAME	CUSTOMER_NAME, CUSTOMER_ID
CUSTOMER_EXAMPLE_NAME_KEY (TABLE B)		
1	CUSTOMER_NAME, CUSTOMER_ID	CUSTOMER_NAME, CUSTOMER_KEY
2	POSTAL_CODE	POSTAL_CODE, CUSTOMER_NAME, CUSTOMER_ID
3	CUSTOMER_NAME	CUSTOMER_NAME, CUSTOMER_NAME, CUSTOMER_ID

The two example tables above each have identical non-clustered indexes applied. These indexes were applied after all data was inserted, so there should be little to no fragmentation complicating our analysis. Using the system stored procedure sp_spaceused to get an estimate of the total size of each table and the indexes used by each is quite illuminating. With only 1 million rows inserted and 2 non-clustered indexes the table with a combination unique clustered index is 14% larger overall. The per index size is even more significantly affected, with the MB required for indexes being 180% larger in the combination key example than in the single field clustered index. This is with only two non-clustered indexes added. The more rows and non-clustered indexes that are added to a table, the resulting size differential between the two architectural decisions will be.

EXEC SP_SPACEUSED CUSTOMER_EXAMPLE_INT
EXEC SP_SPACEUSED CUSTOMER_EXAMPLE_NAME_KEY

Table	Rows	Reserved	Data	Index
Customer_Example_INT	1000000	480 MB	390 MB	87 MB
Customer_Example_Name_Key	1000000	548 MB	390 MB	157 MB



Few columns, preferably only 1

The example above demonstrated both the benefit of choosing a unique clustered index key with the smallest reasonable data type, and well the additional burden added when more than one column is used to define a unique clustered index.

- More columns = more B-Tree levels to traverse and less densely packed pages
- More columns = more invisible, suffix “key values” on every non-clustered index.

Keep your unique clustered index key as small as is reasonably possible – that means small data types and preferably a single, unique column!

Non-Changing Values (immutable, permanent)

It is best to choose a column for your unique, clustered index that will never change. Think about an Order Number or a Check Number (for those of us who remember checkbooks). You’d rarely, if ever change either of these values once issued. There are a couple of reasons that permanency is desirable in a unique, clustered index, both are tied to what happens if you change a clustered key value:

- The entire row (all columns) must be physically deleted and re-inserted, this is managed by SQL Server under the covers. The I/O and log space required to pull this off is dependent upon the full size of the row contents. As a result, the relocation of the row may cause page splits and fragmentation.
- A change of the clustered key requires all the non-clustered indexes to be updated to reflect the new clustered key value(s).

Useful to queries. Meaningful to users

As the clustered index designates the actual sort order of the table data as it is stored on disk, the clustered index field(s) chosen plays a large role in performance. Take for example a table that contains order information. A table like his might typically contain the following fields:

ORDER_ID
ORDER_DATE
CUSTOMER_ID
STORE_ID
TOTAL_CHARGE

If ORDER_ID is the Unique, Clustered Index that would many that the table is stored on disk sequentially by ORDER_ID. Most daily-use queries against the table are going to naturally contain ORDER_ID in the where clause (accessing a particular order's information) or use ORDER_ID to join to another table (say to get ORDER_DETAILS information).

Choosing TOTAL_CHARGE as the clustered index would not be very useful in what you would expect to be common queries. The orders would be stored on disk in ascending order, sorted by the amount spent. Very rarely would that field be included in a where clause, and it would be even rarer to have it found in a join.

Unique

As previously covered, the clustered index column(s) is appended to every non-clustered index to match the index rows back to the clustered index. These matches, called key lookups, are used to look up any columns requested by a query that isn't part of a non-clustered index that was searched.

Because of this need to connect each non-clustered index row to exactly one matching clustered index row it is imperative that the clustered index be unique. The previous sections encouraged creating explicitly unique clustered indexes for this reason.

But what if the clustered index isn't unique? SQL Server will allow a clustered index to be created without a unique constraint and will further allow non-unique values on such an index. When non-unique values are entered into a clustered index SQL Server makes the rows unique by adding and populating an invisible 4-byte column commonly referred to as a "uniquifier". This "uniquifier" is appended to the table and all non-clustered indexes to allow key lookups to be completed accurately.

This is much less efficient than just creating the clustered index with a unique constraint. This invisible column wastes space which slows all key lookup operations. As a rule, clustered indexes should always be created with a unique constraint to avoid having any implicit uniquifiers in the database.

Unused Indexes

Indexes that are not useful to queries (have never been used or are so rarely used that their cost outweighs the benefit) are best removed from a SQL Server database. For those that are rarely used, it is extremely important to make sure there are not hard coded index hints that are forcing the use of the index, or you will break code by removing the index. For those that you can confirm over a long period of time (1 month to 1 year, your comfort level determines this) that the index has never, not 1 time EVER been used to help a query along, get rid of that stuff! Indexes like this require inserts, updates and deletes to take place as part of their upkeep and existence. They also require defragmentation, statistic updates, corruption checks, and backing up. All for no benefit. Who wants to pay a bunch of money for something that you've never once looked at, and has never been useful to you? Throw that stuff out. Kick it to the curb.

Index IDs 0 and 1

A couple of things to be careful of here – Index ID 0 isn't something you'd want to remove. That is SQL's way of telling you that the table doesn't have a clustered index at all. Index ID 0 IS the table. Same goes for Index ID 1, except that numeral designates that the table has a clustered index. You either have a clustered index or you don't, so you'll never have a table that has both Index IDs 0 and 1. It will be either/or (either an Index ID 0 or an index ID 1). Both of those Index IDs represent the table itself. Removing either of these indexes is not recommended when going about "cleaning up" unused indexes. Your investigation should be looking at Indexes with IDs higher than 1.

Index Hints

Before removing an index, be 100% sure there are no referenced index hints. It isn't too difficult to find a script or tool that will let you search for the presence of hints in functions, views or stored procedures. Much more difficult is finding the little buggers in dynamically written code coming from an application. If your server has been up and running for a good amount of time (1 month + generally), you shouldn't have to worry too much about hidden index hints. An index hint forces an index to be used, and you can see this usage in the DMV report (see below). If an index hint specifies that index should cbe explicitly used, and the index is no longer present, an error is thrown and the T-SQL will not run. In other words – if you remove a hinted index you break things.

Index Usage Info

Thankfully, you can ask SQL Server what indexes have been used, not used, and how much effort has been expended keeping those indexes updated. This information is "in-memory", so restarting the SQL Server resets these values. Think of them like an index usage "odometer" that gets reset anytime a SQL Server instance is restarted. If your server has been up for some time and heavily used, any non-clustered index with high Updates, but absolutely ZERO Seeks, Scans, or Lookups is a great candidate for a swift kick to the curb.

Unused Indexes Analysis

Desired Indexes

Just like Amazon's shopping cart, SQL Server has a "wish-list" of indexes. The thing is, SQL doesn't always realize exactly what "toys" it already owns, and that if you just added a game to that X-Box console it'd be happy. In other words, SQL Server will tell you about indexes it really wishes that it had, but without an understanding that perhaps if you simply added 1 field to an existing index it'd have exactly what it wants. It will also sometimes ask for two or more nearly identical indexes. Perhaps a review of this DMO will show a request for 2 separate indexes on the same table and same search columns, but one asks for an included column A and the other for included column B. Don't make both indexes. Just make one that includes A and B.

Most "Desired Indexes" scripts on the internet simply pull from the DMOs without considering the fact that the columns included in the results are ordered by the table's numeric column ID. Yes, you read that correctly. The script in the below takes this into consideration and re-orders the columns into their truly desired order. This list is well worth occasional perusal; items at the top of the list may well be the panacea fix you've been hoping to find, you never know.

For these reasons, it still requires a DBA to review the wish-list, see at what's already in place and make an informed judgment about whether to add an index, alter an existing one, or merge the DMO requests.

Desired Indexes with Adjusted Column Order - Part A

Desired Indexes with Adjusted Column Order - Part B

Fragmentation

(See #4 Maintenance / Fragmentation section below)

ISOLATION LEVELS

Isolation is a key component of any relational database. Isolation guarantees that any 2 transactions which run concurrently will not interfere with each other and will perform the same as if they had been instead run consecutively.

This is completed by having a process lock objects being modified and using that lock to block others from trying to work on the same rows until the process is complete at which point the locks are released. If two processes did try to concurrently update the same rows, one would be forced to wait until the other was done and its locks released – consecutive execution! If the two processes were not working in the same part of the database then they would not run into each other's locks and could finish concurrently.

That is the good part about locking and blocking. The downside is that queries can be slowed down because they are blocked -- waiting for someone else's locks to be released before they can proceed.

Read Uncommitted Isolation

A common way to reduce this delay is to use a query hint called "NOLOCK" which tells read-only queries to ignore locks and continue as if there was no lock (what a well-named query hint). This same behavior achieved by adding "SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED" to a query. This is not a desirable approach.

Blowing past those locks and getting faster query performance sounds great, but those locks – and the process of getting blocked – are features that are there for a reason. When queries are run using an isolation-busting query hint the query risks getting a dirty read. A dirty read is where a query returns a value to a user that was technically never in the database.

What? How can someone read a value that was never in the database? The timeline for a dirty read looks like this.

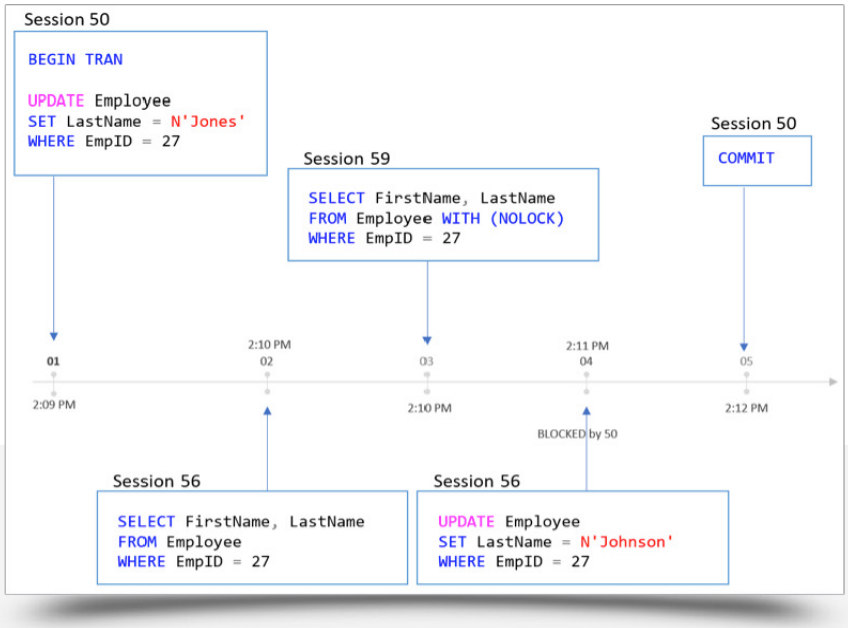
1. Process 100 takes out some locks and starts modifying data.
2. Process 200, operating without regard to locks, reads data that was modified by process 100.
3. Process 100 issues a rollback command.
4. Process 200 returns to its user a value that was NEVER committed to the database. A dirty read.

This is where the name of the isolation level, READ UNCOMMITTED, comes from. There are other technicalities that can lead to a dirty read even without a rollback being issued, but above is the most common scenario.

Read Committed Snapshot Isolation

The Read Committed Snapshot Isolation (RCSI) level can often improve performance for applications that have frequent blocking or deadlocks. When this isolation level is implemented blocking is reduced. This is because the Database Engine uses row versioning and snapshot isolation instead of locks to protect the data. Only SCH-S table locks are acquired.

Let's take a look at how a transaction behaves when using RCSI.
Consider you have the following 3 sessions: 50, 56, and 59.
Each runs a query between 2:09 PM and 2:12 PM.



First session 50 updates the Employee table without a commit. Session 56 and 59 both query the same row which has been updated. Session 56 is running under RCSI and 59 specifies NOLOCK, which is equal to READ UNCOMMITTED. Next session 56 updates the row and is blocked by 50. Finally, 50 issues a commit and 56 is unblocked and completes.

In this scenario the initial SELECT for session 56 would return 'Brown', which was the original data and session 59 would return 'Jones', since the query ran using the read uncommitted isolation with NOLOCK. Finally, the record would be updated to 'Jones' and then immediately to 'Johnson' after the commit from session 50. This shows us that writer's block writers and writers do not block readers when RCSI is enabled.

For more information review the following Microsoft documentation:
https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms189050%28v%3dsql.105%29

To enable Read Committed Snapshot Isolation you must set ALLOW_SNAPSHOT_ISOLATION and READ_COMMITTED_SNAPSHOT for each of the databases you want to adjust.

```
ALTER DATABASE [Database]
SET ALLOW_SNAPSHOT_ISOLATION ON
ALTER DATABASE [Database]
SET READ_COMMITTED_SNAPSHOT ON
```

Be aware that when row versioning is enabled the tempdb database can grow significantly and may require additional IOPS or throughput to keep up with the additional workload. In other words, the disk where your tempdb database lives must be able to handle the additional burden for this configuration to work well. The amount of additional load can greatly vary depending on your application. It's always best to test any database configuration changes before applying them to production.

In addition to the increased workload, another concern is potential race conditions for existing applications that have report queries running against the database with RCSI. This is because reads under RCSI only read committed data, which may be older than the data which is not yet committed.

One final concern is triggers. You'll want to fully test your database and application to be sure that it is compatible with this isolation level. In some cases, trigger code may insert outdated data if the application has not taken RCSI into account.

You can monitor the version store with the built-in Dynamic Management Views (DMVs). These DMVs allow you to observe the version store and troubleshoot performance problems that may be caused by RCSI.
https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms175492%28v%3dsql.105%29

Note the DMVs used to query version store data may contain millions of rows and should be queried carefully.

CODE AND PERFORMANCE ANTI-PATTERNS

Too Much Data

Queries that bring back more columns than necessary or more rows than necessary can be inhibitors of scalability and performance. While it may seem like a good idea for "code reuse" to create a view that contains every column that anyone could ever need, then use that view in lieu of writing similar and selective SQL over and over. This "one size fits all" type of code reuse can lead to problems.

First of all, SQL written to be used for many things will contain more columns than needed (just in case, let's get it all). This prevents the sought after "covering index" phenomenon, which is where you're going to get the best possible performance. If a non-clustered index can be used in a seek to satisfy a query 100% without having to go back to the Clustered Index or Heap for any additional column retrieval, that's the best situation anyone could ever hope for when it comes to performance. Designing a view or SQL that brings back excessive columns "just in case" pretty much guarantees you're going to have to use the lookup mechanism, and any chance of a covering index happening are out the window.

Bringing back too many rows can also be a problem. If your data is being displayed in a grid, and 20-50 rows are showing on a "page", returning 10,000 rows to the application is wasteful and does unnecessary I/O. Learn to use page functions in SQL (Offset and Fetch) in conjunction with application page numbers to only return the rows needed for current viewing by a user.

Excessive row and column retrieval can be a performance killer in the following non-obvious ways:

- Row locks, and possibly escalation to more invasive lock types depending upon row quantity being retrieved and memory constraints on the server
- Data transfer across the network and time for the client to digest / display (google ASYNC_NETWORK_IO)
- Memory allocation within the application and on the client

Parameterization and Plan Re-Use

Every time a query is run against a SQL Server the optimizer must decide how best to find the rows no matter whether they are to be selected, updated, or deleted. Is there an index that suits this query? Is the row count so high that a table scan would be better? These types of questions are answered via a process called compiling. The product of this compilation is a query execution plan.

Compiling plans is not a free operation. While most compilation takes only a fraction of a second it is not entirely uncommon for compilation to take multiple seconds. These costs can be measured by running a query in SSMS after first running SET STATISTICS TIME ON.

This is real clock time that appears to the user and displays the real cost of CPU cycles on the SQL Server to build the plan.

This sample query took 32 milliseconds to compile and all of the time was spent on CPU.

```
SET STATISTICS TIME ON
GO

SELECT C.CustomerName
FROM
```

```
   Sales.Invoices a
   INNER JOIN
Sales.CustomerTransactions b ON a.InvoiceID = b.InvoiceID
   INNER JOIN
   Sales.Customers c ON a.CustomerID = C.CustomerID
WHERE
   c.CustomerName LIKE 'C%'
      OR
    c.DeliveryCityID < 1000
      OR
    a.InvoiceDate > '1/1/2002'
```

Sample output from messages tab:

```
SQL Server parse and compile time:
  CPU time = 32 ms, elapsed time = 32 ms.
```

In an effort to limit the time and server resources spent compiling query execution plans, SQL Server holds a cache of these plans in memory that they might be reused by similar future queries without having to be compiled again. This is simply called the query plan cache. It exists only in memory and is not persisted to disk. The query plan cache is not to be confused with Query Store which does persist plans to disk, but for another purpose.

Especially in OLTP environments, plans can be expected to be reused hundreds or even thousands of times. Imagine how often the most popular screens listing customer details or order history are used. If each calls the same query there is no reason for the optimizer to have to compile a new plan every time.

In order to achieve this plan reuse, proper coding techniques must be utilized. Query plans are only reused when the query being executed is 100% identical to a prior execution that remains in cache. Take these 3 queries that pull up customer details as an example.

```
SELECT CustomerName, PhoneNumber FROM Sales.Customers WHERE CustomerID = 444;
SELECT CustomerName, PhoneNumber FROM Sales.Customers WHERE CustomerID = 68;
SELECT CustomerName, PhoneNumber FROM Sales.Customers WHERE CustomerID = 6;
```

These 3 queries are extremely similar. Clearly, they would all benefit from sharing the same query plan as compiled and saved upon the execution of the first query. But there is a problem - the queries ARE NOT identical! Each has a different value for which to be parsed. This difference in query text is enough to make the compiler not recognize them as identical, and to instead compile and save a new plan for each - wasting CPU cycles for the compilation and memory in the query plan cache.

To combat this problem, always parameterize queries. In the example below the same 3 queries will execute as in the example above, but this time the actual SQL statements are truly identical. In this example the plan used for the first query can be reused by the subsequent queries saving server resources and improving the user experience.

```
DECLARE @Cust INT;
SET @Cust = 444;
SELECT CustomerName, PhoneNumber FROM Sales.Customers WHERE CustomerID = @Cust;

SET @Cust = 68;
SELECT CustomerName, PhoneNumber FROM Sales.Customers WHERE CustomerID = @Cust;

SET @Cust = 6;
SELECT CustomerName, PhoneNumber FROM Sales.Customers WHERE CustomerID = @Cust;
```

Sometimes application queries are called using dynamic SQL. This is especially popular with ORM (Object-Relational Mapping) Software. These programs ease the programming burden for front-end developers by automating the SQL code development. While more difficult, parameterizing dynamic SQL is still possible. Consider the same 3 example queries as above, except dynamic.

```
EXEC sp_ExecuteSQL N'SELECT CustomerName, PhoneNumber FROM Sales.Customers WHERE CustomerID = 444';
EXEC sp_ExecuteSQL N'SELECT CustomerName, PhoneNumber FROM Sales.Customers WHERE CustomerID = 68';
EXEC sp_ExecuteSQL N'SELECT CustomerName, PhoneNumber FROM Sales.Customers WHERE CustomerID = 6';
```

Queries like these are incredibly common coming from ORM software. This example will exhibit the same faults as the non-parameterized queries above. Since the text is not identical they will not share a plan.

To parametrize these queries, call sp_executeSQL with 2 more optional parameters.

1. The first is an NVARCHAR list of variables with data types. Treat the text inside this parameter as a declare statement without the word declare or a semicolon at the end.

2. The third parameter starts a list of assignments. Assign each defined parameter from the prior string to the desired value. Multiple assignments are allowed and will be separated by a comma.

```
DECLARE @Cust INT;
SET @Cust = 68;
EXEC sp_ExecuteSQL N'SELECT CustomerName, PhoneNumber FROM Sales.Customers WHERE CustomerID = @CustInside'
, N'@CustInside INT'--Treat me like a DECLARE statement without the word DECLARE
, @CustInside = @Cust;--Assign as many variables as defined above. Separate with commas.

SET @Cust = 6;
EXEC sp_ExecuteSQL N'SELECT CustomerName, PhoneNumber FROM Sales.Customers WHERE CustomerID = @CustInside'
, N'@CustInside INT'
, @CustInside = @Cust;

SET @Cust = 444;
EXEC sp_ExecuteSQL N'SELECT CustomerName, PhoneNumber FROM Sales.Customers WHERE CustomerID = @CustInside'
, N'@CustInside INT'
, @CustInside = @Cust;
```

Since the actual query text is, again, identical, these queries will be able to share a query plan saving both time and resources. Bonus: Parameterizing sp_ExecuteSQL calls is one step to reducing the effectiveness of SQL injection attacks.

The query plan cache is a very powerful tool when used properly. Consider the requirements to reuse a query plan when writing code to make sure you get the most out of it.

Implicit Conversions

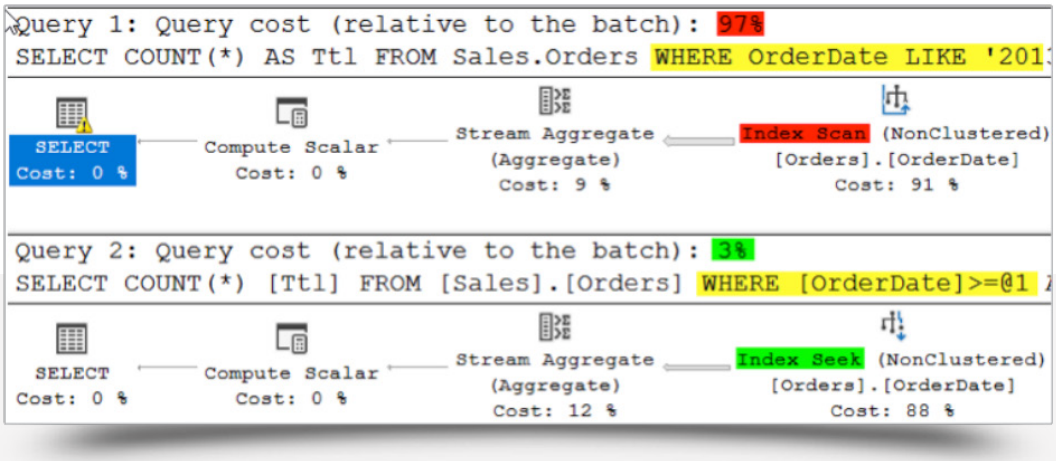
SQL Server tends to be pretty forgiving when comparing values of two different data types in a single argument. It achieves this by implicitly converting the data type of one side to match the other. Sometimes it is too forgiving.

Consider these 2 WideWorldImporters queries designed to get a count of orders from February 2013. Both compile, execute, and return the exact same result.

```
SELECT COUNT(*) Ttl
FROM Sales.Orders
WHERE OrderDate LIKE '2013-02-__'

SELECT COUNT(*) Ttl
FROM Sales.Orders
WHERE OrderDate >= '2/1/2013'
    AND OrderDate <= '2/28/2013'
```

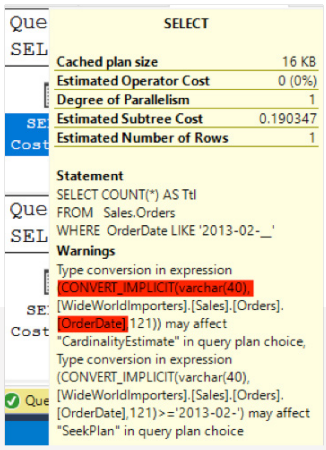
The first version treats the OrderDate column, a DATE data type, as a VARCHAR column. It looks for a pattern match on year and month and allows for any days within the month. This is allowed because the DATE data type can be implicitly converted to a VARCHAR.

The second query makes a direct comparison that does not require the OrderDate column to be converted.
This data set is relatively small so both return their answers pretty quickly. Including the execution plan, however, tells a very different story.



Wow! The first version took 32x more effort than the second. Since the first had an implicit conversion on the column it had to scan the index where the second version was able to seek on that same index. Imagine how much different the execution time would be if the table had more than a few thousand rows.

Notice the yellow exclamation point warning sign on the top left of the screenshot? This warning was explicitly highlighting the fact that an implicit conversion had happened. The following screenshot came from floating a mouse pointer over the warning sign.



Take care to avoid implicit conversions in T-SQL code. Do so by taking the time to choose the correct data types for columns and variables based on what data they'll store and how and where they'll be compared. Use argument operators that are appropriate for data types. For instance, doing a pattern match on dates or numeric columns will always result in a performance-killing implicit conversion.

Set-Based Approach, Avoid Looping

Most people familiar with SQL think of Cursors when they hear the word "looping". And many have been told that "cursors are bad". It's not that the declaration and fetching of rows within a cursor are in itself "bad" for performance – it's the looping part, the avoidance of set-based operations that is the real problem. With this in mind – there are a couple of ways to loop that aren't declaratively called cursors. Make no mistake, these are looped approaches and circumvent set-based operations.

- While
- For Each

Within SQL Code

It is very easy to think procedurally. It's something most programmers have been taught their whole life. Procedural thinking says "I'll look at each row in the table and if it meets my requirements I'll update it. If not, I'll skip it."

But that doesn't work well in SQL Server. SQL Server wants programmers to think set-based. Set-based programming wants the programmer to not think of the row first, but rather the column. For instance, I'll update column X if column Y and Z meet my requirements.

Consider this WideWorldImporters example. In it there are 2 methods to update all orders that have more than 3 dry items and have Amy, person 7, listed as the salesperson. The first considers Amy's rows, decides if they meet the final criterion, then updates if true, skips if not.

The second method is set-based. It simply says to update the orders that have Amy listed as a salesperson and also have more than 3 dry items. It does not consider the rows singularly. It only considers the columns.

```
T NOCOUNT ON;
DECLARE @TimeStart DATETIME;
SET @TimeStart = GETDATE();

--Method 1
DECLARE @TotalDryItems INT;
DECLARE A_Bad_Cursor CURSOR LOCAL FORWARD_ONLY FOR
SELECT TotalDryItems
FROM Sales.Invoices
WHERE SalespersonPersonID = 7;

OPEN A_Bad_Cursor;
FETCH NEXT FROM A_Bad_Cursor INTO @TotalDryItems;
WHILE @@FETCH_STATUS = 0
BEGIN
```

```
  IF @TotalDryItems > 3
  BEGIN
    UPDATE Sales.Invoices
    SET InternalComments = 'Big order for Amy!'
    WHERE CURRENT OF A_Bad_Cursor
END;
  FETCH NEXT FROM A_Bad_Cursor INTO @TotalDryItems;
END;

CLOSE A_Bad_Cursor;
DEALLOCATE A_Bad_Cursor;
PRINT DATEDIFF(ms, @TimeStart, GETDATE())

--Method 2
UPDATE Sales.Invoices
SET InternalComments = 'Big order for Amy!'
WHERE SalespersonPersonID = 7
  AND TotalDryItems > 3;
PRINT DATEDIFF(ms, @TimeStart, GETDATE())
```

Download the "Big Order for Amy" Code Here

In a test environment procedural method 1 took 674ms and set-based method 2 took 30 milliseconds – a drop of 95%! This example is quite forced but is illustrative of the type of performance savings one can get by changing a single operation from procedural to set-based.

Any place in the code where a programmer has written or is considering a cursor or other type of loop, very serious scrutiny should be applied to see if a set-based approach can be used. The performance benefit will still be there even if a series of joins, a CTE, and/or case statements are needed to reach a working set-based conclusion.

ORM Generated Code

ORMs (Entity Framework, LINQ and others) make it faster and easier for developers to code by removing the necessity to understand and write yet another language (SQL). By creating objects in the development environment that represent entities in a database, and defining how they relate to each other, a developer can drag and drop objects and have the SQL automatically generated. It's not uncommon to capture trace info from an application that is dynamically generating SQL via this method and seeing a distinct lack of joins and many, many, MANY statements being executed to accomplish something that could have been done in just a single or few statements. For example, imagine a query that needs to retrieve a list of 100 people's first and last names along with the total amount of $ ordered. A typical set-based SQL statement would look something like:

```
SELECT C.FIRSTNAME, C.LASTNAME, SUM(O.GRANDTOTAL) AS TOTALSPENT
FROM CUSTOMER C INNER JOIN ORDERS O ON C.CUSTOMERID = O.CUSTOMERID
GROUP BY C.FIRSTNAME, C.LASTNAME
```

Watch out for SQL code generated by an ORM that resembles something along these lines:

```
SELECT C.CUSOTMERID,  C.FISTNAME, C.LASTNAME FROM CUSTOMER
SELECT SUM(GRANDTOTAL) FROM ORDERS WHERE CUSTOMERID = @P1
SELECT SUM(GRANDTOTAL) FROM ORDERS WHERE CUSTOMERID = @P2
SELECT SUM(GRANDTOTAL) FROM ORDERS WHERE CUSTOMERID = @P3
SELECT SUM(GRANDTOTAL) FROM ORDERS WHERE CUSTOMERID = @P4
SELECT SUM(GRANDTOTAL) FROM ORDERS WHERE CUSTOMERID = @P5…TO 100.
```

1 statement vs 101 statements. 1 round trip to the database vs 101 round trips. Now let's put 10,000 users on the system at once and make nearly all of the generated queries behave this way.

My suggestion if you're up against this (because it's a hard one to fight if you're already down this road), pick your battles. Do some tracing and identify areas that users would notice a difference. Choose something to tackle that you stand a chance at pulling off and that will prove a point. Then choose another item to go after. This one requires methodical baby steps. I have a presentation called "Three Methods to End the Madness – Application Slowness Diagnosis" that shows how to figure out if these issues are affecting your application's performance and to summarize your findings. You can find the scripts for the presentation here.

Scalar Functions

Scalar functions are very useful when trying to maximize code reuse in SQL Server. Their benefit largely ends there. Don't get me started on their detriment to performance. Actually, I'm already started, how about an example?

Consider this simple scalar function designed to return a formatted name from a PersonID in WideWorldImporters. This code could be reused many, many times. It could remove a whole bunch of joins to the Person table when the only column needed is this formatted name.

```
CREATE FUNCTION GetFriendlyName (@PersonID INT) RETURNS NVARCHAR(203)
AS
BEGIN
DECLARE @ReturnValue NVARCHAR(203);
SELECT @ReturnValue = FullName + ' (' + PreferredName + ')'
FROM Application.People
WHERE PersonID = @PersonID;
RETURN @ReturnValue;
END
```

Now compare these 2 queries that return the exact same result set. Method 1 utilizes the scalar function creating a simpler query while maximizing code reuse. Method 2 writes the query using traditional T-SQL joins.

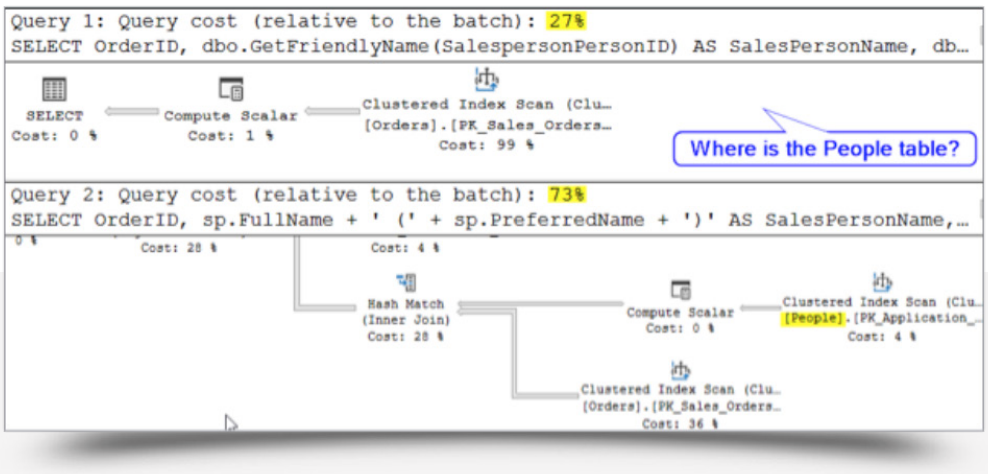
```
DECLARE @TimeStart DATETIME;
SET @TimeStart = GETDATE();

--Method 1
SELECT
  OrderID
, dbo.GetFriendlyName(SalespersonPersonID) SalesPersonName
, dbo.GetFriendlyName(PickedByPersonID) PickedByPersonName
FROM Sales.Orders;
PRINT DATEDIFF(ms, @TimeStart, GETDATE());

--Method 2
SELECT
  OrderID
, sp.FullName + ' (' + sp.PreferredName + ')' SalesPersonName
, pb.FullName + ' (' + pb.PreferredName + ')' PickedByPersonName
FROM Sales.Orders O
  INNER JOIN Application.People sp ON o.SalespersonPersonID = sp.PersonID
  LEFT OUTER JOIN Application.People pb ON o.PickedByPersonID = pb.PersonID;

PRINT DATEDIFF(ms, @TimeStart, GETDATE());
```

Scalar Function Demonstration SQL

In a test environment method 1 consistently ran about 3x longer than method 2. Ouch. That is a steep penalty to pay for code reuse.

If that wasn't reason enough to avoid scalar functions, there is more. Up until SQL Server 2017 CU3 the work done by scalar functions didn't appear in execution plans or statistics io output.

This is the query plan screen for these 2 queries when running in SQL Server 2016. Even though query 1 ran 3x longer than query 2 it shows as being 3x smaller! There is also no mention of the people table in query 1's plan even though it clearly queried it.



This is the statistics IO output for the same query batch. Again, the people table is completely skipped in the output. These gaps can make it very hard to debug poorly performing code that uses scalar functions as the root of the problem is hidden from view.

Query 1:

- Table 'Orders'. Scan count 1, logical reads 692, physical reads 0

Query 2:

- Table 'Workfile'. Scan count 0, logical reads 0, physical reads 0
- Table 'Worktable'. Scan count 0, logical reads 0, physical reads 0
- Table 'Orders'. Scan count 1, logical reads 692, physical reads 0
- Table 'People'. Scan count 2, logical reads 160, physical reads 0

Lastly –yes, there's more—any query that has a scalar function is not eligible for a parallel query execution plan. This means any large query that would benefit from a parallel execution plan, but also has a scalar function, will be forced to run single-threaded causing it to run even longer.

All of this adds up to the rule that scalar functions should be used sparingly. Developers commonly ask, "If I can't use scalar functions for code reuse, how do I achieve good code reuse in T-SQL?" The answer is that, in short, generally you don't. Not if the code needs to perform. While there are times when a scalar function may be useful, more often than not it is just not a good idea. It's easier to talk about when Scalar Functions may be OK (which are only in a couple of cases) rather than when not:

- When the query results are few and the scalar function is only in the select columns of the query
- When the scalar function is not referencing any tables
- When the scalar function is not involved in the where clause, join, group by or order clauses of a SQL statement

Massive Queries / Views / Procs

Due to the way that SQL Server caches plans for reuse, performance can often suffer from massive queries for a few reasons:

1. The compiled plan and parameters used for compilation doesn't fit the use-case for subsequent parameter values. If the procedure has 20 parameters, and upon compilation only parameter A and D are given actual meaningful values, while the rest of the parameters are either given NULL values or plugged with some value that represents "no value provided", the plan that is created and stored by the SQL Server will fit that scenario as well as it can. A subsequent call the stored procedure that provides meaningful data to parameters B and G isn't going to work as well as it may have (if at all – which will result in recompilation). Think about what you're asking SQL Server to do. It's not magic, it's logic.

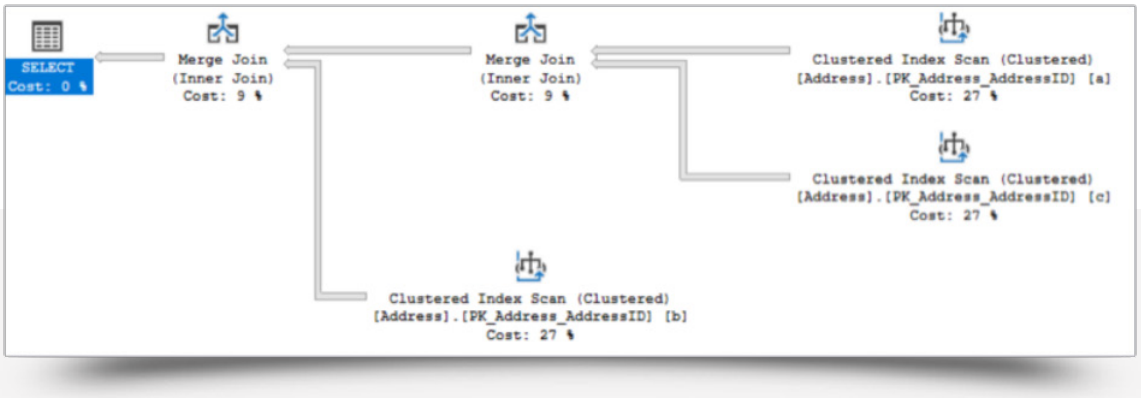
2. Gigantic strings of SQL may not be something that SQL can find a great execution plan for in the time allotted. In this case, the optimizer will time out and go with the best plan it was able to come up with in a reasonable amount of time.

3. Long and verbose SQL will take up more space in the plan cache, not only will compilation take a long time, identification and retrieval of the plan is also more work for SQL as the plans are stored with a hash and found by matching the hash. All that hashing to match on long SQL statements is more work than on short ones.

IFs and ORs

Try to avoid using too many IF statements and statements that use too many ORs. Queries and procedures should avoid excessive nesting and repeated branches.

Note the 3 repeated branches of the clustered index scan on the Address table. This behavior can often be avoided.



Endless Nesting

Avoid creating views that are based on other views. It negatively affects the optimizer's ability to get good row estimates. While a view based upon a single other view is usually ok, once a 3rd level is introduced then the estimates fall way off. These bad estimates lead the optimizer to generate bad query execution plans. Compilation time is also negatively affected as the optimizer engine tries to unwind the views back to the underlying tables. Having code that takes an extra-long time to come up with a subpar execution plan is a recipe for bad performance.

Chronic Recompiles

Recompiles can be a performance killer because the time it takes to compile a plan can often be greater than the time to run the query. Chronic recompilation (meaning that every execution results in a recompile) essentially reduces a piece of code to a "one person at a time" experience. "Everyone get in line to execute my stored procedure – single file, one at a time folks, no pushing!" Ouch.

In my experience, the most common reasons for unintended and surprise chronic recompilation are:

- Set Options Changed in Batch
- Statistics Updates on Temporary Objects

To identify queries that recompile often, use the plan_generation_num column from the sys.dm_exec_query_stats DMV. This column shows the number of plan instances after a recompile.

The "warning" threshold for recompiles is RECOMPILES > 10% of COMPILES. You can use the following query to identify when recompiles exceed this threshold.

```
SET NOCOUNT ON

DECLARE @comps bigint, @recomps bigint
SELECT @comps = cntr_value
FROM sys.dm_os_performance_counters
WHERE counter_name = 'SQL Compilations/sec'

SELECT @recomps = cntr_value
FROM sys.dm_os_performance_counters
WHERE counter_name = 'SQL Re-Compilations/sec'

IF (@recomps > (@comps *.1))
BEGIN
        PRINT CONCAT('Recompiles are high: ', @recomps, ' recompiles of ', @comps, ' compiles')
END
SET NOCOUNT OFF
```

In some cases you may force a recompile when a poor plan is often chosen due to parameter changes; however, in this case it is often better to use the OPTIMIZE FOR UNKNOWN hint or to configure a plan guide to help the optimizer choose the right plan.

A very thorough article about recompilation, monitoring, diagnosis and strategies to avoid can be found here:
http://sqlchitchat.com/sqldev/tsql/tempst/

Triggers

A trigger is a type of T-SQL object similar to a stored procedure except that it is not run on demand, but rather attached to a table and executed when rows are inserted, updated, and/or deleted from that table. This can be an enticing proposition as it can simplify business logic and maximize code reuse. But triggers are like potato chips, it's hard for folks to eat just one. Once a problem has been solved by implementing a trigger, it seems to become a common go-to in many development shops. I've seen what should have been a single insert result in literally 20 triggers being executed and thousands of lines of code. What could have been a millisecond operation is made into a sluggish 4-5 second user experience as SQL bounces around inside the server like a ricocheting bullet inside a metal dome.

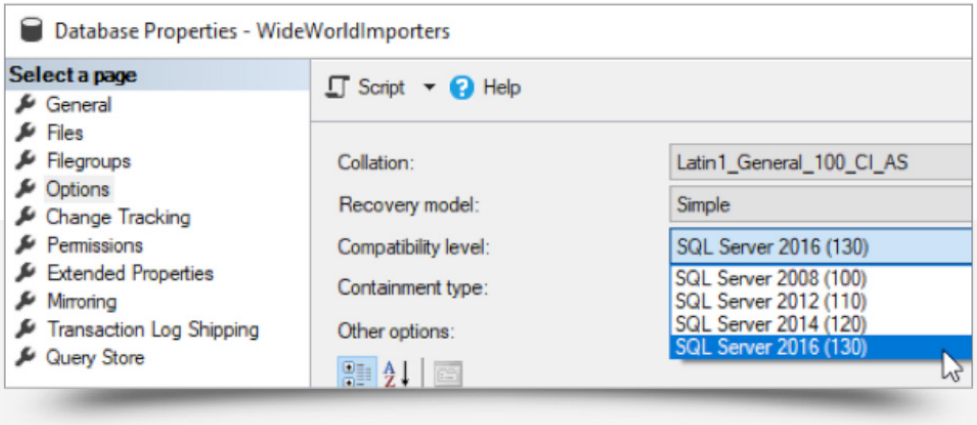
If there is no avoiding a trigger, it should be kept short and simple (fast compilation, solid plan) and not perform actions that will result in the calling of yet another trigger. Also, logic inside triggers is best kept to set-based operations that avoid looping. Triggers that unintentionally chronically recompile can be particularly nasty.

SNAPPY NEW FEATURES

Compatibility Mode

When upgrading to a new version of SQL Server any databases that are brought over from an old server are restored with a compatibility level to match their old server. Sometimes this can ease the transition as the older compatibility level will unlock features that have otherwise been removed in the current version. On the downside, the old compatibility level will lock out new features such as a new cost estimator designed to improve query performance. Check out your databases to make sure they are all running the latest compatibility level by going to the database properties, options pane, and viewing the level in the drop-down menu. Any databases that are not running the latest, consider upgrading them by selecting a new value from that same menu.

There might be a concern that this change will break something in the programs that use the database. There are several methods



to mitigate this risk. One such method would be to make the change in a test environment and see what happens. The change isn't permanent so if an error comes up after making this change the compatibility level can be dropped back down until a fix is put into place.

SQL Server has a method of informing the DBA of any deprecated features currently in use. The query below will show any deprecated features in use in the instance right now. Keep in mind that just because a feature is deprecated doesn't necessarily mean it has been removed in the newest compatibility level. Any results from this query should be validated to see if they would result in a failure in the event of a compatibility level upgrade.

```
SELECT instance_name, cntr_value
FROM sys.dm_os_performance_counters
WHERE object_name = 'SQLServer:Deprecated Features'
AND cntr_value > 0
```

Columnstore Indexes

Columnstore indexes allow for ultra-fast aggregate queries over large datasets. They have been around in various forms since SQL Server 2012, but in SQL Server 2016, for the first time, they can be combined with a rowstore index in a single object while remaining read/write. This means you can add a row-store index to a clustered columnstore index to enforce primary or foreign keys. Or add a columnstore index to a heap or cluster for report query performance.

By combining these indexes, you can get the best of both worlds without having to make a second copy of the data and introducing a delay between the OLTP and reporting data sets.

Compression

Index compression has been a feature of SQL Server since SQL Server 2008, but was limited to Enterprise Edition until SQL Server 2016 SP1 and SQL Server 2017 when it was enabled in Standard Edition.

There are 2 levels of compression available, PAGE and ROW. In simple terms, row compression stores fixed length columns as if they were variable length columns and allows numeric columns to be stored using smaller data types when not using the entire width available (i.e. a bigint column where the largest number stored is 1,000,000). Page compression first performs row compression then adds prefix and dictionary compression which are standard compression techniques.

Index compression only compresses data stored on-page and does not consider off-page data types such as the MAX data types. Compressed indexes stay compressed in the buffer pool meaning that compressing an index will reduce the disk AND memory footprints of a database! Your mileage may vary, but compression commonly results in a 40-60% reduction in index size. There is a CPU penalty associated with compressing indexes that is generally about 10%.

As recently as SSMS 17.8.1 the GUI does not support compressing indexes during index creation. It must be done using T-SQL DML or as a separate activity after the index is built. Below are examples of rebuilding an existing index using PAGE and ROW compression taken from WideWorldImporters . If unsure of which compression method to use a built-in stored procedure called sp_estimate_data_compression_savings (Books Online Link) can help make that decision.

```
---PAGE Compression
CREATE INDEX FK_Purchasing_PurchaseOrderLines_PackageTypeID ON
Purchasing.PurchaseOrderLines(PackageTypeID)
WITH (DROP_EXISTING = ON, DATA_COMPRESSION=PAGE) ON USERDATA

--Row Compression
CREATE INDEX FK_Purchasing_PurchaseOrderLines_PackageTypeID ON
Purchasing.PurchaseOrderLines(PackageTypeID)
WITH (DROP_EXISTING = ON, DATA_COMPRESSION=ROW) ON USERDATA
```

Improved DBCC Check DB

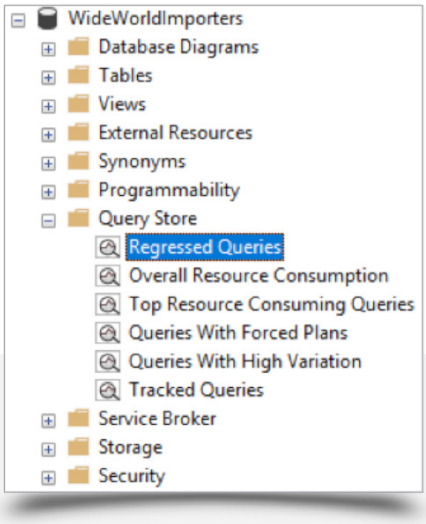
Starting with SQL Server 2016 changes were made to the DBCC CheckDB command to make it run faster -- especially on machines with 8 or more CPUs. Internal testing at Microsoft showed up to a 7x improvement in command performance with SQL Server 2016 when checking the exact same database on the exact same hardware when compared to SQL Server 2014. For more technical details, read this article from MSDN.

https://blogs.msdn.microsoft.com/psssql/2016/02/25/sql-2016-it-just-runs-faster-dbcc-scales-7x-better/

Query Store and Adaptive Query Plans

SQL Server 2016 introduced a new feature called the Query Store where the database engine could record query plan and execution statistic history with more detail than ever before. When Query Store is enabled this information is also persisted to disk so that it can live beyond a service restart.

The information from the Query Store can be used by a data professional to track performance issues by viewing the performance history of an object or by viewing queries that consume large amounts of resources. View this information using a series of built-in reports that can be found in the Query Store directory within the database in SSMS.



Query Store is especially helpful in resolving issues of plan regression where a newly compiled plan performs significantly worse than an older plan. When such a case if found the Data Professional can take actions such as forcing the optimizer to choose a more optimal plan from the past or refusing to allow it to select a poorly performing plan in favor of a more favorable option. Read more about this feature at this link, https://docs.microsoft.com/en-us/sql/relational-databases/performance/best-practice-with-the-query-store?view=sql-server-2017.

SQL Server 2017 took this a step further and added a feature where the database engine automatically reviews the information from Query Store and makes recommendations where forcing plans might be beneficial. This information can be found in a DMO, sys.dm_db_tuning_recommendations. Read more about that feature at this link, https://docs.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-db-tuning-recommendations-transact-sql?view=sql-server-2017.

It can get even easier than that! Another added feature for SQL Server 2017 is the ability to automatically accept the recommendations made by the query engine. Simply tell the database to automatically tune (That just sounds so cool!) using the command below and it will automatically remove bad query plans from use in favor of prior versions that had better performance metrics.

```
ALTER DATABASE [MyDatabase]
SET AUTOMATIC_TUNING (FORCE_LAST_GOOD_PLAN = ON);
```

Read more about this feature at this link, https://docs.microsoft.com/en-us/sql/relational-databases/automatic-tuning/automatic-tuning.

About the Authors

Mindy Curnutt

Mindy Curnutt is a business owner and 5X Microsoft Data Platform MVP and has been involved in the SQL Community for 20+ years. She has been an SME on multiple MS SQL Server Cert Exams and is the co-author of 4 books, most recently MS Press SQL Server 2017 Administration Inside-Out. She is the President of the North Texas SQL Server User's Group, a national public speaker and active Girls and Stem volunteer and mentor. You can often find Mindy in her hometown of Dallas, TX at the Sons of Hermann Hall Acoustic Picker's Jam on Thursday nights.

Eric Blinn

Eric Blinn is the Senior Database Architect at Squire Patton Boggs, an international law firm with 47 offices in 20 countries, based out of Cleveland, Ohio. He is a content creator and facilitator for the data track at DriveIT, an Akron, Ohio based IT training firm. Eric is an active member of the SQL Server community and currently serves as the Vice President of the Ohio North PASS Local Group. He has spoken on SQL Server topics at a number of technology conferences and training events. When not working on SQL Server Eric enjoys smoked meats and going fishing with his family.

Daniel Janik

Daniel Janik is an independent consultant from Austin, TX and a 2X Microsoft Data Platform MVP. He has been supporting SQL Server solutions in many roles from DBA to developer over his 20-year career. Daniel is also former Microsoft and was a field engineer (PFE) for 6 years. He speaks regularly at SQL Saturday events and user groups. You can find him on his blog SQLTechBlog.com.

With **SQL Diagnostic Manager**, unscheduled downtime on SQL Server has been cut by a third.

Neil Leslie **IT Architect** for General Electric Company

ACHIEVE 24/7 SQL MONITORING

WITH SQL DIAGNOSTIC MANAGER

- Monitor performance for physical, virtual, and cloud environments.
- Monitor queries and query plans to see the causes of blocks and deadlocks.
- Monitor application transactions with SQL Workload Analysis add-on.
- View expert recommendations from SQL Doctor to optimize performance.
- Alert predictively with settings to avoid false alerts.
- View summary of top issues and alerts with the web console add-on.

Start for FREE