**Resources:**

* GitHub repository for this class:
  + <https://github.com/CodeFlo-JHenry/SQLOptimize>
* SQL Server Express 2014 w/ Tools download
  + <http://www.microsoft.com/en-us/server-cloud/products/sql-server-editions/sql-server-express.aspx>
* Adventure Works 2014 full database download
  + <https://msftdbprodsamples.codeplex.com/releases>
* Adventure Works Cycles company/database overview:
  + <http://technet.microsoft.com/en-us/library/ms124825(v=sql.100).aspx>
* My e-mail:
  + JeremyH416@gmail.com

**General Tips**

* If you are not already use to using semi-colons at the end of your SQL statements, get into the habit.
  + Microsoft has announced that they will (in some future release of SQL Server yet TBD) eliminate the support for NOT ending your Transact-SQL statements with a semi-colon.
  + Other DB systems (like MySQL) have long subscribed to this SQL standard, so it will confuse you less if you ever work with other DBMSs to just end your statements with a semi-colon.
* Use a consistent, and visually appealing layout for your query text – capitalization, indentation, etc. The easier your query is to read, the easier it is to debug and troubleshoot performance.
  + Every SQL developer ultimately develops their own style, but I have some suggestions…
    - Use ALL CAPS for keywords and functions – makes it easier to read and identify parts of your code when intellisense coloring isn’t on.
    - Indent the items in your SELECT list… I like to put my commas at the beginning of each line – makes it easier to add and remove additional columns later
    - Indent each level of JOIN in the FROM clause
    - Start each JOIN on a new line and put the ON clause on a new line at the same indentation level as its associated JOIN.
    - Put each filter/criteria expression in the WHERE clause on its own line
      * I like to start my ANDs and ORs lined up with the WHERE, and then tab so my filter/criteria expressions are all lined up.
* Although a lot of the tips and rules we are going to go over in this class can often reduce queries that take minutes to run down to a few seconds, remember that **milliseconds count!** In a production environment with dozens (or hundreds or thousands) of queries being executed constantly and simultaneously, every little bit adds up!
* To view total execution time (down to the millisecond), open the Properties window by pressing F4 or View -> Properties Window
* To view execution plan for all batches in current window or selected SQL in the window, [Ctrl]+L

**Top 12+ (18) Rules to Ensure Best SQL Performance**

1. Don’t use \* in your SELECT – always name the columns you want returned.
   1. Blindly pulling all columns often pulls extra data you don’t need and eats up resources
   2. In addition, the query engine has to do a lookup to get the list of columns at execution time.
2. Avoid using correlated sub-queries in the SELECT list at all costs! Almost ALL sub-queries in a SELECT can be rewritten into a JOIN in the FROM clause.
   1. Sub-queries in the SELECT list have to be run once for every record being returned.
   2. Spending some time rethinking your design strategy for such queries can have big performance payoffs – even if you have to create several nested queries, the performance will usually be better.
3. Make sure all fields used in filters (WHERE clause) and JOINs are indexed/included in an index (if possible).
   1. Note: Be careful… if rows are frequently INSERTED, UPDATED, or DELETED, too much indexing can actually cause performance problems with those operations.
   2. Make use of composite indexes (i.e. indexes defined as a combination of fields). For example, when you have a table that you know you are going to frequently JOIN on the ID (PK) field, but also have an “Active” bit flag field that you will also be frequently concerned with, create a composite index so that Active records can be identified more quickly.
4. Avoid applying functions in comparison (i.e. join) and filter expressions if possible.
   1. SUBSTRING() is not your friend – at least not in a WHERE clause
   2. Functions applied to the items in the SELECT list, while they certainly require a little overhead, are not really a problem
   3. But when used as filter or join criteria, those functions have to be run against every record of the source table. Avoid this if you can!
5. Filter tables/sub-queries used in JOINs first in the FROM clause instead of the main WHERE clause.
   1. The query optimizer will usually apply filters first to the table(s) being joined to limit the records returned prior to the join… but not always.
   2. To help ensure that data sets from tables and sub-queries used in JOINs are minimized, try moving your filter criteria into the sub-queries or ON clauses of the JOINs to help make sure the query optimizer performs the filters first.
6. When using a calculation expression in a search argument, try to apply the expression to the non-column side of the comparison operator:
   1. Instead of : [Column] + 2 > @value
   2. Use: [Column] > @value – 2
   3. The “bad” way causes the engine to have to do a calculation for every record. By moving the calculation to the other side of the operator, the calculation only has to happen once.
   4. Likewise don’t use LEFT() and RIGHT() string functions against the column to compare to a particular string, instead use the LIKE operator and use the wildcard (%) character with the part of the string you are seeking.
7. Avoid using CURSORS unless there is a REALLY good reason to (which there definitely are… on occasion)
   1. Cursors are notoriously slow (relatively speaking). It’s not even so much that the cursor itself (and the FETCH) is slow, it’s more because of what people are typically DOING with the items returned that creates all the overhead – e.g. calling an INSERT or UPDATE statement with each iteration…
   2. About the only valid reason to use them is when you have to call another stored procedure or series of updates based on values returned on a per-record basis.
   3. Get into “set-think” / “batch-think”. A LOT of things that you might initially want to use a CURSOR for can actually be accomplished with JOINs
8. Sequence your filter criteria in your WHERE clause
   1. Generally, the query optimizer will select the best order to apply the filters… but not always.
   2. It is extremely RARE that the sequence should matter, but when you have a very complex query with 10+ joins/tables/sub-queries, it can make a difference. I have quite literally seen the sequencing of criteria take a query from 2+ minutes down to a few seconds.
   3. Put the most selective criteria first.
9. Don’t use HAVING clause to apply basic filters.
   1. HAVING was specifically added to SQL because the WHERE clause cannot analyze aggregate functions. Don’t use HAVING for any other purpose.
10. Use UNION ALL instead of UNION if possible (which is most of the time)
    1. Regular UNION eliminates duplicates – this means extra overhead.
    2. Usually, you should want all of the records from both side of the UNION, so use UNION ALL instead whenever you can… i.e. when you don’t expect duplicates, or duplicates from different sources are not an issue.
11. NOT can truly be a negative influence. Be careful when using the NOT operator.
    1. NOT can produce unexpected results when comparing Nullable fields.
    2. Using NOT or other negative operators (<>, !=, etc.) can sometimes prevent the query optimizer from using a useful index.
12. Don’t use sub-queries in the WHERE clause – and don’t do with 2 sub-queries what you could do with one:
    1. Unless it is a necessary EXISTS sub-query, try to include the sub-query as a JOIN instead.
    2. If you have multiple sub-queries, try to combine them into a single JOINed sub-query if possible.
13. If you are just trying to see if any records exist given certain criteria, use EXISTS() instead of COUNT() to determine if any records exist.
    1. COUNT() requires the query to get all rows. EXISTS() stops as soon as it finds ONE record that meets the criteria.
14. Remove the need for OUTER JOINs by creating a “dummy” record in the joined table and use its key as the default for the joining table. (not always possible, but a good hack when you can implement it)
    1. This ensures that there is never NULL data from the join.
    2. This is particularly useful when joining to simple “list” / “lookup” tables that don’t get updated often but are used often in queries for reports, etc.
15. RAID – it does more than just kill bugs dead… Optimize disk read performance – generally make sure data files are on a RAID 5 or RAID 10 array
    1. RAID 1 (mirroring) is typically used for SQL Log files. It provides good read and write performance.
    2. RAID 5 (Stiping with parity) is usually used for data files as it provides better read performance (but slightly worse write performance because of calculating the parity bits)
    3. RAID 10 (1+0 – Striped with mirror) is provides the best performance (no parity to slow down writing, but the read benefits of striping)… but this is the most expensive because it requires a minimum of 4 drives to implement with only the capacity of 2 of those drives.
16. For ADVANCED USE ONLY: Specify the Index to use with WITH (Index(idxName)) after the table in the FROM clause (very rare, but sometimes necessary)
    1. Especially if you analyze the execution plan and the query optimizer does not seem to be using the index you think it should… And it DOES happen – particularly in more complex queries.
17. Foreign Keys – use them, but be aware of their dark side.
    1. Foreign keys add a bit of overhead to write (INSERT, UPDATE, and DELETE) operations
    2. But, foreign keys can provide clues to the query optimizer so that it can eliminate unnecessary tables, scans, etc…. especially if the FK is indexed.
18. Consider dropping indexes before loading data through multiple INSERTs or UPDATEs
    1. Every time an INSERT or UPDATE occurs, SQL Server has to add to and/or rebuild the indexes on the table(s) affected.
    2. Add the index(es) back after the loading operations are complete. Overall, this will generally take less time than having the index(es) modified with every data insert/update.

**Proper JOINs:**

* Be careful with your LEFT JOINs and filter criteria in your WHERE clause. Filtering on the right-side table in the WHERE clause might yield unintended results.
* If you don’t need to do a LEFT (or other OUTER) join, then DON’T!
* Also, when performing a LEFT join, make sure your left table is filtered prior to the JOIN taking place… The query optimizer is USUALLY pretty good at making this happen, but sometimes it needs a little help.
* Remember: a RIGHT join is just a LEFT join with dyslexia. A RIGHT join rarely ever makes the query easier to understand.
* Performing a LEFT join and a check for NULL to find records that do NOT have a match in another table may not be as efficient as using NOT IN.
* Putting basic filter expressions up in the ON clause of the related table join instead of the WHERE clause can make a significant performance difference in some rare cases. Usually, the optimizer will apply such filters to the table before the join, but sometimes the optimizer gets confused.
* SELF joins are often useful – particularly on self-referencing tables where records can have a parent-child relationship (like the common employee-supervisor paradigm), as well as for getting particular combinations of records from within the same table.
* FULL OUTER joins can be useful, but use them wisely.
  + When the rows from the two tables match, you get one combined record. For all other records you get just the data from either the left or the right table.
  + Can be useful for identifying orphaned or unused data/records
    - Typically, you add a filter in the WHERE clause to identify the records that have missing data from one or the other sides.
  + Can be useful in parent/multiple-children situations
* Legitimate cases for CROSS joins are somewhat rare. CROSS joins are dangerous (from a performance perspective)… but also occasionally extremely useful
  + A CROSS join gives you every combination of record between two tables
    - The records can multiply into a very large dataset very quickly, so be careful before you tie up your server with one of these bad boys…
  + You can explicitly specify a CROSS join (which I recommend), use a shortened syntax, or set join condition that is always true.
  + I’ve seen developers mistakenly create a CROSS JOIN and filter out unneeded records using DISTINCT… Two big performance mistakes.
  + The CROSS join requires no ON clause
  + You can turn a CROSS join into an INNER JOIN by matching columns from the two tables in the WHERE clause… but why would you?
  + The CROSS join is useful when you want to get a count of how many times some combination of data occurred, but you also want to see the combinations with 0 counts.
  + The CROSS join can be useful for generating numbers (a trick we will see later)
  + It can also be used to analyze different possible outcomes, such as in “path analysis” and floating time window analysis.
  + CROSS joins can be very useful for populating data when you need to cover every possible combination – say, like of Size and Color… or to create dummy data like First Name and Last Name