**Temp tables, Table variables and Common Table Expressions (CTEs)**

* Temp tables can carry a lot of overhead, but can also be more efficient than alternatives:
  + Created just like creating regular tables, but with the pound sign “#” as a prefix to the table name.
  + Temp tables utilize the hard drive – requiring I/O operations.
  + To use a temp table, you have to Create, Insert data into, Select from, and then finally Drop.
  + Supports non-clustered indexing – which can provide a significant performance improvement on SELECTs, and UPDATEs over table variables – especially on larger data sets.
  + Avoid using them if you can - use sub-queries instead if you don’t need to manipulate the data through multiple operations.
* Table Variables:
  + Only choice inside a user-defined function if you need to perform DML (INSERT, UPDATE, and DELETE) operations on the temporary object
  + Has less impact on tempdb transaction log because table variable log activity is truncated immediately upon deletion of the object.
  + Do not utilize the main transaction log (because transactions to the table variable cannot be rolled back).
  + Queries using table variables, however, do NOT support parallelism.
  + Cannot have non-clustered indexes, constraints, or default values and cannot have statistics created against them.
    - You can (and should) specify PRIMARY KEY when setting up your table variable – by default, it will be clustered; but you can specify for it to be non-clustered and use CLUSTERED with a different UNIQUE constraint instead.
    - Exception: You can create UNIQUE CONSTRAINTs when defining your table variable – which act as indexes
    - More on limitations of table variables can be found here: <http://databases.aspfaq.com/database/should-i-use-a-temp-table-or-a-table-variable.html>
  + Useful for storing and then using return value of a Table-valued function
  + Use for small data sets and when indexing is not a factor
  + Query plan is compiled before table variable has any data, which can cause the cardinality estimate (i.e. # of rows) to be inaccurate – in particular, it’s always 1.
  + SQL Server 2012 SP2 added feature to be able to recompile query after execution started when certain thresholds are met so that row estimation will be more accurate and affect execution plan accordingly.
    - Requires using trace flag 2453 by executing statement: dbcc traceon(2453, -1)
* CTEs (Commen Table Expressions)
  + Essentially a temporary view. Useful for making queries more readable than using in-line sub-queries.
  + Can be used in Stored Procedures, User-defined Functions, Triggers, and Views… but not Indexed Views.
  + Because a CTE is “memory only”, you need to be careful about defining CTEs for large data sets that could suck up a lot of server memory resources.
    - As with any View or sub-query, CTEs should filter and limit results as much as possible to maximize performance.
    - Use temp tables for large result sets.
  + CTEs do NOT store the result set and reuse it (like with a temp table).
  + The CTE query is rerun every time it is referenced – this can have negative performance implications. SQL query optimizer ***ought*** to recognize a CTE referenced multiple times and calculate/run it just once, but it does not do this – for some specific reasons that become apparent in situations like recursive queries.
  + You can define multiple CTEs to be used with a query by simply separating their definitions with a comma.
    - Useful for breaking your logic down into bite-sized chunks instead of creating separate Views.
    - Only specify WITH keyword one time.
    - After a comma, just define the next CTE the way you normally would without the WITH keyword
  + CTE can reference itself within its own definition – key to making recursive queries.
    - For development and ad-hoc query execution, you can use the OPTION (MAXRECURSION n) on your main query to limit the number of recursive references. This works, but also generates an error, so it is not a good solution for a consumable query.
    - To get around this, you can add a “Level” calculated field to your CTE to indicate the recursion level of a record and then filter based on that.
    - Good example in AdventureWorks is the Stored Procedure: [uspGetBillOfMaterials].
    - Recursive CTE requires 3 things:
      * First part of the CTE is a select statement that sets the root/seed record(s). This is the ***anchor member*** of the recursive query.
      * Then, a UNION ALL is performed with another SELECT that self-references the CTE and joins it with the core table(s) also used in the first query. This is the ***recursive member*** of the recursive query.
      * Finally, the CTE is used (and possibly joined) in the outer query.
    - Have to be careful not to create a circular reference…
    - <http://technet.microsoft.com/en-us/library/ms186243(v=sql.105).aspx>