**How To and Not To Use Sub-Queries:**

* If you are going to use a sub-query, put your filters in the sub-queries if you can.
* Correlated sub-queries in the SELECT list are a really bad idea – generally speaking. Figure out how to rewrite that as a sub-query that you JOIN in the FROM clause.
  + Sometimes, the query optimizer can essentially turn it into a join, but I find it best not to count on that.
  + However, a correlated sub-query MIGHT make a better alternative to using a CURSOR in applicable situations.
* A VIEW is a sub-query… it is NOT a stored execution plan and will be incorporated and recalculated within the context of your query.
  + However, using VIEWs can cause table redundancy in your main query.
  + If the data from joined tables in the VIEW are not needed to either 1) get the VIEW to return proper results (i.e. as a filter) or 2) get values from other tables needed in your consuming/main query, then your VIEW will be costing you extra overhead.
* Don’t overcomplicate your sub-queries/views. This is often easy to do when you are using a view or sub-query that someone else wrote – they might include extra joins and fields that you don’t need.
* You may need to create a new version of the VIEW you are using that applies additional filter criteria… or just copy the guts of that VIEW into your SQL code to give you more control
  + The downside to this is that if the logic of the VIEW changes, you have to change your query or other version of the VIEW, too.

**Indexing Fundamentals (more advanced indexing topics to be covered later)**

* Types of indexes:
  + Clustered : This is the “best” type of index… for SELECT queries, anyhow
    - Clustered indexes actually dictate how the data is stored on disk. The data is actually stored on disk in the order specified by the index.
    - This makes reading data much faster because SQL doesn’t have to jump around to different pages to find data.
    - You can only have 1 clustered index on a table – for obvious reasons
    - Clustered indexes can be composite indexes – and often should be, especially if it takes more than one column to define a unique value – but keep it limited to as few columns as possible.
    - Don’t include columns that undergo frequent change, as SQL will have to resort the data on disk whenever a clustered index column value in a record is changed.
    - Clustered indexes don’t have to be unique, but it is usually a good idea to try to make them unique.
    - If not unique, try to engineer your clustered indexes based on columns that produce “mostly” unique values.
    - Columns that are frequently used in joins and filters – especially used in ranged filters, ORDER BY, and GROUP BY clauses – are usually good candidates to consider for your clustered index.
    - Almost every table should have a clustered index defined.
  + Non-Clustered : as many as you want – almost…
    - Non-clustered indexes include the key values from the clustered index on the table – which is one of the main reasons to keep your clustered index width to a minimum
    - If there is no clustered index on the table, then SQL will create a unique identifier column behind the scenes.
    - Multiple indexes can include the same column(s), but remember that all of these indexes must be maintained and updated during INSERT, UPDATE, and DELETE operations.
    - Create non-clustered indexes that make sense based on how often columns are being used throughout the database and/or front-end application in joins and filters. Creating an index to accommodate a single SELECT query that runs once per week on a table that is being updated and added to hundreds of times per day may not be the best choice.
  + UNIQUE vs. non-unique
    - Both clustered and non-clustered indexes can be defined as UNIQUE.
    - Use UNIQUE indexes if you can, and if the data is frequently filtered and/or joined on equality.
    - Non-unique indexes are useful for columns that are often filtered and/or joined based on non-equality and ranges – such as “start” and “end” date fields.
  + Generally, don’t index columns that will often contain NULL values.

**Analyzing Execution Plans**

* Analyzing Execution Plans
  + Estimated vs. Actual
    - Toolbar icon or right-click in query window and turn “Include Actual Execution Plan” option on.
  + Try to rewrite your query in the same window and view total Estimated Execution Plan to determine if one is better than the other.
  + Understanding the diagram
    - Read right-to-left, top-to-bottom
    - Thickness of arrowed lines between operations indicate how much data is being passed up the chain.
    - Hovering over an operation icon or data line displays detailed information. Useful properties include:
      * Cached plan size : useful to see how much memory plans consume
      * Est. # rows
      * Est. Subtree Cost : Only valid relative to other operations/subtrees in the batch, but provides cumulative cost (right-to-left) up to, and including the operation. Useful for comparing different branches to find most costly aspects of a complex query.
    - Table Scan vs. Index Scan vs. Index Seek
      * Sometimes, Table Scans are unavoidable, but try to minimize them by implementing indexes when possible.
      * A table scan has to go through every row/page of the table data
      * An Index Scan goes through every row/page of the index (which generally requires less memory and I/O than looking at the table data)
      * An Index Seek allows the engine to completely skip over most of the data in the index and look at a much smaller subset to identify matches. This is, by far, the most efficient of the 3 operations.
      * Scans generally require a lot more I/O and take longer to process.
    - RID Lookup
      * Row Identifier lookup
      * Used on a “heap” table – a table that has no Clustered index
      * A bit more resource intensive than a Key Lookup.
      * Could indicate potential for adding a clustered index.
    - Joins
      * NESTED LOOP join
        + For each record in the outer table, loops through the inner table to find matches.
        + Efficient if one table is small (the outer table) and the inner table is being joined on an indexed column
        + Also, always used if join is not based on equality
      * MERGE join
        + Optimizer first sorts both inputs on the join column
        + Then, it gets a row from each input and compares join column. If they match, then both rows are moved to the result set being built. If they don’t, SQL only adds the row with the “higher” value, then looks at the next row from that input for the next comparison.
        + This results in a single scan on each input (unlike a NESTED LOOP)
        + Particularly a good choice if both inputs are already sorted (i.e. via clustered indexes) by the join column.
        + Might also be good choice for the optimizer if adequate memory is available to sort the inputs first.
      * HASH join
        + Usually chosen if the “inner” table does not have a useful index.
        + Most useful when joining large tables with no useful index(es)
        + SQL has to build a hash table on the fly – essentially a temporary index.
        + Two phases: Build and Probe

Build runs the join key value through a hash algorithm – usually on the smaller of the two tables. This often results in a shortened version of the value – which is more efficient for comparison. These values with a pointer to the original record are essentially stored in a temp table. Furthermore, these hashed values are “grouped” into *hash buckets* in an even distribution.

Probe then runs the join key value for each row of the other table through the same algorithm, determines the relevant *hash bucket* to look in, and then tries to find a match in the hash table (within that *hash bucket*).

* + - * + Because of the *hash bucket* mechanism, HASH joins are useless for inequality joins.
        + HASH joins can’t flow output rows to the next operation until the hash operation is complete – unlike MERGE and NESTED LOOP joins. In addition, HASH joins require a memory grant which imposes additional limitations. Therefore, HASH joins remove an opportunity for some parallelism when processing the query.
        + Why choose/allow a HASH join? Can be efficient when you don’t want to incur the overhead of a permanent index because you only have a few (or infrequent) queries based on that join.
    - Hash Match (aggregation)
      * This operation will show up in the plan if you are aggregating data, and these can be expensive operations.
      * Make sure you are limiting rows to be aggregated using a WHERE clause (filter) as much as possible.
      * HAVING filters happen AFTER aggregation, and generally won’t help you here.
    - Sort operation
      * Obviously indicates a sorting operation.
      * If you are utilizing clustered indexes in your tables and/or joins, you might not need to add the ORDER BY clause to your query. Rows will already be sorted as they come out of the join operations.
    - Filter operation
      * Again, pretty self-explanatory. Excludes rows passed to it so far that don’t match criteria.
      * Sometimes, you might be able to eliminate a Filter operation by leveraging an existing join operation.
  + "Deep trees" vs "Bushy trees"
    - # of Deep trees is calculated as *n!* (where n = # of tables)
    - # of Bushy trees is calculated as (2n – 2)! / (n – 1)!
    - Possible bushy-tree combinations quickly far outgrow deep-tree combinations once you get to 3+ tables.
      * Query optimizers usually lean toward deep-trees for this reason. Analyzing bushy-tree combinations would take too long.
    - Bushy trees offer a greater potential for parallelism in processing the query.
    - Query optimizer cannot possibly analyze all the possible combinations, so it depends on heuristics and rules – such as “avoid bushy trees”.
    - Finding the optimal join order is the most difficult job of the query optimizer and has been the subject of extensive research, theory, and study for decades.
    - Because of this, however, as the designer of the query, you MIGHT already know and be able to tell the SQL engine the best order to join your tables.
  + Problem indicators
    - Table Scans
    - Thick lines (further to the right) could indicate an opportunity for a filter
    - RID Lookups

**Providing Hints to SQL with and without using query Hints**

* <http://msdn.microsoft.com/en-us/library/ms181714.aspx>
* How can the query optimizer get it wrong? This can happen sometimes due to lack of statistics or uneven data distributions.
* { HASH | ORDER } GROUP
* HASH | LOOP | MERGE | REMOTE join hints
* OPTIMIZE FOR (@variable\_name = literal\_contant [, …n])
* FAST number\_rows
  + Specifies that the query is optimized for fast retrieval of the first number\_rows. After the first number\_rows are returned, the query continues execution and produces its full result set.
* FORCE ORDER
  + Specifies that the join order indicated by the query syntax is preserved during query optimization. Using FORCE ORDER does not affect possible role reversal behavior of the query optimizer (e.g. choosing the “outer” vs. the “inner” table for Merge joins, etc.).
* RECOMPILE
  + Forces query optimizer to discard execution plan (normally cached) after execution
  + Why would you want to do this? This can be useful in stored procedures/queries that use dynamic WHERE clauses. Sometimes the execution plan will be cached based on the first run of the SP since SQL Server was most recently restarted, but that execution plan may not be the best given various parameter values and you would actually prefer that SQL recompile and re-compute the execution plan with each run.
* Other
  + EXPAND VIEWS, KEEP PLAN, KEEPFIXED PLAN, INGOR\_NONCLUSTERED\_COLUMNSTORE\_INDEX, MAXDOP, MAXRECURSION, …