**Advanced Indexing**

* Filtered Indexes (SQL Server 2008+)
  + Essentially, just create a nonclustered index on the table and include a WHERE clause with your CREATE NONCLUSTERED INDEX statement.
    - Through SSMS, define the filter on the Filter page of the index creation window.
  + Optimized, non-clustered index for covering queries that select from a well-defined subset of data.
  + Uses a filter predicate to index only a portion of rows in the table – instead of indexing the entire table.
  + Can improve performance
    - smaller than a full-table index and has filtered statistics.
    - Reduced index maintenance cost: filtered indexes are updated/maintained only when data in the index is changed.
    - Filtered statistics are more accurate because they only cover the subset of rows that meets the predefined filter.
    - Query optimizer will consider a Filtered Index in more situations than it will an indexed view.
  + You may be able to replace a single full-table non-clustered index with multiple filtered indexes without significantly increasing the storage requirements.
    - Use when table has heterogeneous (scattered/diverse) data rows to create indexes on various “categories” of data
  + If you need to index frequently changing columns, but only for a certain subset of rows, a filtered index would be a happy medium between having no index and doing a full-table index.
    - Use when indexed column(s) often have NULL values, but you usually query for rows with non-NULL
  + Filtered indexes can be non-unique – as opposed to indexed views which must be unique.
  + Limitations/Considerations
    - Only support simple comparison operators.
    - Comparison expressions used for the filtered index must not result in an explicit or implicit data conversion or an error will occur.
    - A column in the filtered index expression should be a key or included column if that column is going to be used in your query result set(s)
* Indexed Views
  + Indexed Views may be utilized by the query optimizer even in situations where the view itself is not specified in the query. If the tables used by the Indexed View are used and joined in a similar way to your query, the optimizer recognizes this and may choose to use that view index.
  + Once you have a view that meets all of the below requirements, create and index (a UNIQUE CLUSTERED INDEX) on the view just as you would on a table.
    - You can create multiple indexes on a view, but the first index you create must be a unique clustered index
  + Requirements:
    - Must create the view using WITH SCHEMABINDING option
    - View definition must be deterministic
      * All expressions in the SELECT, WHERE, and GROUP BY clauses must yield the same result any time they are evaluated against a given set of parameters/inputs.
      * Float data-type columns are non-deterministic because they are not precise. While float columns can be used in the SELECT list and WHERE clause of the view, they cannot be used as part of the index key.
      * Expressions that depend on GETDATE() function are generally non-deterministic.
    - Tables must be referenced using two-part names: *schema.tablename*
    - The view cannot reference other views
    - There are 7 SET options that require specific values to use indexed views:
      * ANSI\_NULLS : ON
      * ANSI\_PADDING : ON
      * ANSI\_WARNINGS : ON
      * ARITHABORT : ON
      * CONCAT\_NULL\_YIELDS\_NULL : ON
      * NUMERIC\_ROUNDABORT : OFF
      * QUOTED\_IDENTIFIER : ON
    - Additional requirements for creating an indexed view can be found here: <http://msdn.microsoft.com/en-us/library/ms191432.aspx>
* Choosing a filegroup for your index(es)
  + Clustered index is always going to reside on the same filegroup as the base table (filegroups, if the base table is partitioned across more than one filegroup)
  + You can create nonclustered indexes on a filegroup other than that of the base table
  + This MIGHT be useful for performance if you put the nonclustered index on a filegroup that is on a different physical drive that is run off of a separate controller, so that the index and data can both be read in parallel.
  + To put you nonclustered index on a different filegroup, simply include … ON *filegroup\_name*  in your CREATE NONCLUSTERED INDEX statement.
* Partitioning indexes
  + Generally, partition the table over filegroups instead of specifically partitioning indexes separately. When the table is partitioned, indexes that are created will automatically follow the same partitioning structure. Thus the index will be *aligned* with the table.
  + There are situations where partitioning the index separately from the base table can be useful:
    - The base table has not been partitioned
    - The base table IS partitioned, but the index key is unique and does not contain the partitioning column of the table.
    - You need the base table to participate in collocated joins with more tables using different join columns – i.e. some of the other tables that you frequently join with use similar/same partitioning as the base table and you want your new index to include those join columns using different filegroups (and thus, different physical disks) to improve parallel processing of the joins.
  + The index key (for a unique, nonclustered index) or clustering key (for a clustered index) must contain the partitioning column. For nonunique, non-clustered indexes, SQL automatically adds the partition column as a nonkey.
  + 3 steps to create a partitioned table or index:
    - Create a Partition Function to specify how a table or index that uses that function can be partitioned
    - Create a Partition Scheme to specify the placement of the partitions of a partition function on filegroups.
    - Create a table or index using the Partition Scheme
* Columnstore indexes (for datawarehousing applications)
  + Only available in SQL 2012 and later
  + Offer high performance gains for queries that use full table scans, but not well-suited for queries that seek into the data for particular values.
  + Instead of storing data from rows sequentially on disk, the COLUMN is read like a row and stored sequentially. This generally allows for much greater compression which reduces memory requirements, and yields higher performance.

**Isolation Levels and Locking**

* SET TRANSACTION ISOLATION LEVEL
  + Use to set locking option for all queries in a batch
  + <http://msdn.microsoft.com/en-us/library/ms173763.aspx>
  + READ UNCOMMITTED : just like the NOLOCK option on an individual query – only applicable to SELECT statements in the batch.
  + READ COMMITTED : The default, prevents other SQL statements from modifying or reading pages not yet committed. However, data can be changed by other transactions BETWEEN STATEMENTS within the current transaction – which could result in phantom data.
  + REPEATABLE READ : Hold locks until end of entire transaction (instead of end of each statement). Results in lower concurrency (i.e. ability for SQL server to handle multiple transactions against same tables/data) and, thus, should be used with caution if ANY tables referenced within queries for the transaction are frequently used by other processes.
  + SNAPSHOT :
    - ALLOW\_SNAPSHOT\_ISOLATION database option must be set to ON to use this.
    - Limits data being read to committed data only.
    - Does not issue any locks.
    - Will not prevent other transactions from modifying or reading data
    - Statements within the transaction CAN see modifications made within the same transaction.
  + SERIALIZABLE :
    - Most restrictive locking level.
    - Locks entire ranges of data based on key values that match all search conditions in all statements used in the transaction.
    - Blocks other transactions from updating or inserting rows that would fall within key value ranges used by the current transaction
    - Locks are active for duration of the entire transaction
* Pages, locking, and deadlocks
  + SQL Server stores data physically in chunks (pages) of 8KB
  + SQL Server reads and writes entire 8KB chunks at a time.
  + Depending on the isolation level of the query, locks are placed on pages (and other objects) when transactions are acting on the data. Other transactions that want to use that same data have to wait until any existing lock is released.
  + Therefore, if a particular record in a table only requires, say, 900 bytes of storage, modifying that one record could essentially lock up to 8 records in that table.
  + A “deadlock” can happen when two transactions are each creating locks on multiple resources – at least one of which the other transaction requires access to.
    - Example: <http://etutorials.org/SQL/microsoft+sql+server+2000/Part+V+SQL+Server+Internals+and+Performance+Tuning/Chapter+38.+Locking+and+Performance/SQL+Server+Lock+Types/>
      * Transaction A puts a shared lock on a record in Table 1.
      * Transaction B puts a shared lock on a record in Table 2.
      * Transaction A now needs to access the record in Table 2, but Transaction B has it locked.
      * Transaction B needs to access the record in Table 1 before it can complete, but it is locked by Transaction A.
      * Neither transaction can complete because they are each waiting for the other to release its lock.
* Some DBA tools to be familiar with:
  + Use sp\_lock to view locks
  + Use sp\_who and sp\_who2 to view sessions, blocks, high utilization, and identify spid’s
    - For even better troubleshooting info, use this sp\_who3:
      * <http://sqlserverplanet.com/dba/a-better-sp_who2-using-dmvs-sp_who3>
    - Particularly, look for values in the BlkBy column (in sp\_who2). These indicate queries/operations that are waiting on another process/transaction to release a lock. These can be chained.
    - For more information:
      * <http://msdn.microsoft.com/en-us/library/ms174313.aspx>
      * Sp\_who2 is undocumented and specs could change.
  + SELECT OBJECT\_NAME(*object\_id*); to identify database object referenced.
  + To get the *object\_id* for a particular table (or other object):
    - SELECT OBJECT\_ID(‘*[database.][schema.]object\_name*’);
* Types of locks:
  + Resource Types:
    - RID : Row identifier used to lock a single row within a table
      * Used whenever they are more efficient to improve concurrent access to the page-level
    - KEY : Row lock within an index. Used to protect key ranges in serializable transactions.
    - PAG : Data or index page.
    - EXT : Extent - Contiguous group of eight data pages or index pages
    - TAB : Entire table, including all data and indexes
    - DB : Database
      * Always Shared locks – can generally be ignored, but do indicate database usage.
  + Modes:
    - Sch-S : Schema stability. Ensures that a schema element is not dropped while any session holds this lock on the element.
    - Sch-M : Schema modification. Ensures that no other sessions are referencing the indicated object while it is being modified.
    - S : Shared. Session is granted shared access to the resource.
      * Very common – used on all read operations.
      * Does not block other processes from also reading and acquiring shared locks
      * Also allows for updates to the resource.
      * DOES block Exclusive locks from being acquired.
      * Only held for the duration of the actual read at whatever resource level (be it row, page, or table)
    - U : Update. Held by a session that may eventually perform an update to data.
      * Compatible with Shared locks and can be acquired on the same resource at the same time.
      * Only ONE Update lock can exist on a resource at a time.
      * Partially exclusive to avoid common deadlock scenarios.
      * Once all Shared locks are released, the Update lock is escalated to an Exclusive lock.
    - X : Exclusive. Session has exclusive access to the resource.
      * No other process can obtain any type of lock on this resource
      * Exclusive lock being held for a long time may indicate a problem or deadlock.
    - I{S|U|X} : Intent locks
      * Intent locks are not so much locks, as they are indicators of locks at deeper levels.
      * Intent locks improve performance by allowing SQL Server to look at the higher level object first – like at the table level – to determine types of locks that are present rather than having to look at all of the individual element locks.
    - SIU : Shared Intent Update.
    - SIX : Shared Intent Exclusive.
    - UIX : Update Intent Exclusive.
    - RANGES\_[] and RANGE\_[] : ranged locks (similar to those listed above, but for entire key ranges)
* NOLOCK option
  + Add to FROM clause after table specification: WITH (NOLOCK)
  + Same as READUNCOMMITTED
  + Causes SELECT to perform a “dirty read” – read/display data that has not yet been committed to the database.
  + Can improve performance if “dirty reads” are not an issue – especially if query accesses a table that is frequently being updated/added to.
  + Only valid for SELECT
  + SELECT can still be blocked if table structure is being modified and hasn’t been committed.

**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>