1. The rules governing default column nullability go like this:

• If you explicitly specify either NULL or NOT NULL, it will be used (if valid—see below).

• If a column is based on a user-dened data type, that data type's nullability specification is used.

• If a column has only one nullability option, that option is used. Timestamp columns always require values, and bit columns can require them as well, depending on the server compatibility setting (specified via the sp\_dbcmptlevel system stored procedure).

• If the session setting ANSI\_NULL\_DFLT\_ON is set to **true** (it defaults to the setting specified in the database), column nullability defaults to **true**. ANSI SQL species that columns are nullable by default. Connecting to SQL Server via ODBC or OLEDB (which is the normal way applications connect) sets ANSI\_ NULL\_DFLT\_ON to **true** by default, though this can be changed in ODBC data sources or by the calling application.

• If the database setting **ANSI null default** is set to **true** (it defaults to **false**), column nullability is set to **true**.

• If none of these conditions species an ANSI NULL setting, column nullability defaults to **false** so that columns don't allow NULL values.

1. TRUNCATE TABLE empties a table without logging row deletions in the transaction log. It can't be used with tables referenced by FOREIGN KEY constraints, and it invalidates the transaction log for the entire database.
2. Predefined constants or automatic variables can be used as parameters to stored procedures, but true functions cannot.
3. When the join criteria in an outer join aren't met, columns in the first table are returned normally, but columns from the second table are returned with no value—as NULLs. This is handy for finding missing values and broken links between tables.
4. A CROSS JOIN is an intentional Cartesian product. The size of a Cartesian product is the number of rows in one table multiplied by those in the other. So for two tables with three rows each, their CROSS JOIN or Cartesian product would consist of nine rows. By definition, CROSS JOINs don't need or support the use of the ON clause that other joins require.
5. SQL Server optimizer turns inner query into an inner join internally.
6. HAVING is less efficient than WHERE because it qualifies the result set after it's been organized into groups; WHERE does so beforehand so SQL Server translates HAVING into WHERE during query execution.
7. Datetime columns require eight bytes of storage and can store dates ranging from January 1, 1753, to December 31, 9999. Smalldatetime columns require four bytes and can handle dates from January 1, 1900, through June 6, 2079. Datetime columns store dates and times to the nearest three-hundredths of a second (3.33 milliseconds), while smalldatetime columns are limited to storing times to the nearest minute—they don't store seconds or milliseconds at all. Since accuracy is limited to 3.33 milliseconds, milliseconds are always rounded to the nearest threehundredths of a second. This means that the millisecond portion of a datetime column will always end in 0, 3, or 7. So, "19000101 12:00:00.564" is rounded to "19000101 12:00:00.563" and "19000101 12:00:00.565" is rounded to "19000101 12:00:00.567."
8. COMPUTE generates totals that appear as additional summary columns at the end of the result set. When used with BY, the COMPUTE clause generates control-breaks and subtotals in the result set. You can specify COMPUTE BY and COMPUTE in the same query (This feature will be removed in the next version of Microsoft SQL Server).

SELECT SalesPersonID, CustomerID, OrderDate, SubTotal, TotalDue FROM Sales.SalesOrderHeader ORDER BY SalesPersonID, OrderDate COMPUTE SUM(SubTotal), SUM(TotalDue) BY SalesPersonID;

1. The ROLLUP operator is useful in generating reports that contain subtotals and totals. For example, a simple table Inventory contains the following:

|  |  |  |
| --- | --- | --- |
| Item | Color | Quantity |
| Table | Blue | 124 |
| Table | Red | 223 |
| Chair | Blue | 101 |
| Chair | Red | 210 |

This query generates a subtotal report:

SELECT CASE WHEN (GROUPING(Item) = 1) THEN 'ALL' ELSE ISNULL(Item, 'UNKNOWN') END AS Item, CASE WHEN (GROUPING(Color) = 1) THEN 'ALL' ELSE ISNULL(Color, 'UNKNOWN') END AS Color, SUM(Quantity) AS QtySum FROM Inventory GROUP BY Item, Color WITH ROLLUP

|  |  |  |
| --- | --- | --- |
| Item | Color | QtySum |
| Chair | Blue | 101.00 |
| Chair | Red | 210.00 |
| Chair | ALL | 311.00 |
| Table | Blue | 124.00 |
| Table | Red | 223.00 |
| Table | ALL | 347.00 |
| ALL | ALL | 658.00 |

If the ROLLUP keyword in the query is changed to CUBE, the CUBE result set is the same, except these two additional rows are returned at the end:

ALL Blue 225.00

ALL Red 433.00

1. Following are the specific differences between CUBE and ROLLUP:

* CUBE generates a result set that shows aggregates for all combinations of values in the selected columns.
* ROLLUP generates a result set that shows aggregates for a hierarchy of values in the selected columns.

1. The result set of a ROLLUP operation has functionality similar to that returned by a COMPUTE BY. However, ROLLUP has the following advantages:

* ROLLUP returns a single result set while COMPUTE BY returns multiple result sets that increase the complexity of application code.
* ROLLUP can be used in a server cursor while COMPUTE BY cannot.
* The query optimizer can sometimes generate more efficient execution plans for ROLLUP than it can for COMPUTE BY.

1. SQL Server string variables and fields, variable-length and fixed-length types are supported, with each limited to a maximum of 8000bytes. Use text when you need to store more than

8000bytes.

1. Whether you should choose to create character (char) or variable character (varchar) fields depends on your needs. If the

* Data you're storing is of a relatively fixed length and varies very little from row to row, fixed character fields make more sense. Each variable character field carries with it the overhead associated with storing a field's length in addition to its data. If the length of the data it stores doesn't vary much, a fixed-length character field will not only be more efficiently stored, it will also be faster to access.
* On the other hand, if the data length varies considerably from row to row, a variable-length field is more appropriate.fi Variable character fields can also be more efficient in terms of SQL syntax. For fixed character variable, concatenation doesn't work as we might expect. Unlike variable-length, these fields are right-padded with spaces to its maximum length.

1. When ANSI\_PADDING is OFF, field values are trimmed as they're **inserted**
2. PATINDEX() works very similarly to the LIKE predicate of the WHERE clause. PATINDEX() really comes in handy when you need to filter rows not only by the presence of a mask but also by its position.
3. The Transact-SQL EXEC() function and the sp\_executesql stored procedure allow you to execute a string variable as a SQL command. sp\_executesql is faster and more feature laden than EXEC(). When you need to execute a dynamically generated SQL string multiple times in succession (with only query parameters changing between executions), sp\_executesql should be your tool of choice. This is because it easily facilitates the reuse of the execution plan generated by the query optimizer the first time the query executes. It's more efficient than EXEC() because the query string is built only once, and each parameter is specified in its native data format, not first converted to a string, as EXEC() requires.
4. sp\_executesql allows you to embed parameters within its query string using standard variable names as placeholders, like so:

sp\_executesql N'SELECT \* FROM authors WHERE au\_lname LIKE @au\_lname', N'@au\_lname varchar(40)',@au\_lname='Green%'

Here, @au\_lname is a placeholder. Though the query may be executed several times in succession, the only thing that varies between executions is the value of @au\_lname. This makes it highly likely that the query optimizer will be able to avoid recreating the execution plan with each query run. Note the use of the "N" prefix to define the literal strings passed to the procedure as Unicode strings. sp\_executesql requires Unicode strings to be passed into it. That's why @execsql was defined using nvarchar.

1. Unicode expands the number of possible characters to 65,536, by using two bytes instead of one. This increased capacity facilitates the inclusion of the alphabets and symbols found in most of the world's languages. Transact-SQL's regular string types (char, varchar, and text) are constructed of characters from a particular single-byte character set. This character set is selected during installation and can't be changed afterward without recreating databases and reloading data. Since Unicode strings take twice as much storage space as regular strings, they can be only half as long (4000characters). SQL Server defines special Unicode-specific data types for storing Unicode strings: nchar, nvarchar, and ntext.
2. Prior to release 7.0 of SQL Server, dividing a numeric quantity by zero returned a NULL result. By default, that's no longer the case. Dividing a number by zero now results in a divide by zero exception.
3. There's an inconsistency between the monetary types—money and smallmoney—and the other numeric data types. All numerics except for money and smallmoney implicitly convert from character strings during INSERTs and UPDATEs.
4. In addition to using CAST() and CONVERT() to format numeric data types as strings, you can use the STR() function. STR() is better than the generic CAST() and CONVERT() because it provides for right justification and allows the number of decimal places to be specified.
5. SQL Server provides support for BLOB (binary large object) fields via its image and text (and ntext) data types. These data types permit the storage and retrieval of fields up to 2GB in size. BLOB fields are stored in a separate page chain from the row in which they reside. All that's stored in the BLOB column itself is a sixteen-byte pointer to the first page of the column's page chain.
6. BLOBs aren't stored like other data types, and you can't treat them as though they were.

* You can't, for example, declare text or image local variables
* You also can't refer to BLOB columns in the WHERE clause using the equal sign—the LIKE predicate, PATINDEX(), or DATALENGTH() is required instead.

1. READTEXT command can be used to access blob objects in pieces. READTEXT works with image as well as text columns. It takes four parameters: the column to read, a valid pointer to its underlying text, offset at which to begin reading, and the size of the chunk to read. Use the TEXTPTR() function to retrieve a pointer to a BLOB column's underlying data.
2. READTEXT doesn't allow reading past the end of the BLOB. That is, if the BLOB is 100characters long, you can't specify a starting point of 90 and a chunk size of 30 and expect to get the last 10characters of the BLOB—READTEXT will return an error instead. So, the query is forced to do READTEXT's work for it—it computes the exact size of the remainder of the BLOB and is careful not to exceed it.
3. Supplying BLOB columns with text or image data that's less than or equal to 8000bytes in size is as straightforward as updating any other type of column. Writing values larger than 8000 bytes via Transact-SQL requires the use of the UPDATETEXT or WRITETEXT command. UPDATETEXT can modify a portion of a BLOB field, while WRITETEXT rewrites its entire contents. Generally speaking, UPDATETEXT is more flexible than WRITETEXT. Since both UPDATETEXT and WRITETEXT require a valid text pointer, you can't use them to write data to a BLOB field that's NULL.
4. Bit columns and variables can have one of three values: 0, 1, or NULL. Bits are stored in groups of eight as bytes, so if there are fewer than eight of them, they require just one byte of storage. Bits are not allowed to serve as index keys, and for good reason. A column that's limited to three possible values would make a very poor index key.
5. A bitmap is a column or variable of a type integer or image—that stores an array of bit switches-a map of them. A bit mask is a collection of bits—usually in the form of an integer—that's used to extract or manipulate the bit switches in a bitmap. SQL Server makes extensive use of bitmaps and bit masks because they're an efficient way to store and track status flags.
6. The uniqueidentifier data type stores GUIDs (global unique identifiers). A GUID is a 16-byte binary number. Uniqueidentifier have a number of disadvantages:

* Their values are unwieldy and cryptic. They're random and don't fit or match any sort of mnemonic pattern.
* The uniqueidentifier data type is four times as large as the four-byte int type that's typically used for row identifiers. This makes accessing them slower in general, including building and accessing indexes over them.
* The sequence in which a set of uniqueidentifier values were generated is not discernable from the values themselves—you can't tell which values came first and which ones came later by looking only at the data. Among other things, this means that they make poor ORDER BY columns.

1. A cursor variable stores a reference to a cursor definition. Cursors defined via variables are by definition local cursors (since you can't declare global variables) and can be used in place of direct cursor references in commands such as OPEN, FETCH, CLOSE, and DEALLOCATE. Cursor variables and the cursor data type can’t be used as:

* You can't define a table column of type cursor.
* You can't define stored procedure input parameters as cursors (but you can define cursor output parameters.
* You can't assign a cursor variable with a SELECT statement. (They must be assigned using the SET command. Deallocating the original cursor doesn't prevent you from continuing to access it via the cursor variable.

1. A timestamp is a special binary(8) value that's guaranteed to be unique across a database. A timestamp column is updated each time the data in a row changes. For example, if Juliet updates a row after Romeo reads it but before he posts his own changes, Romeo's update attempt will fail because it will use the original timestamp value to try to locate the row. The TSEQUAL() function can be used to compare timestamp values. If the timestamps aren't equal, TSEQUAL() raises an error and aborts the current command batch.
2. There are a no of methods related to NULL values:

* ISNULL()'s parameters aren't limited to constants. Both arguments can consist expressions (including the one returned by the function) or even handle SELECT statements.
* The NULLIF() function is a rough inverse of ISNULL(). It takes two parameters and returns NULL if they're equal; otherwise it returns the first parameter.
* COALESCE() returns the first non-NULL value from a horizontal list.

1. Adding a NULL value to a number is not the same as aggregating a column that contains both NULL and non-NULL values. In the former case, the end result is always a NULL value. In the latter, the NULL values are ignored and the aggregation is performed.
2. The COUNT(\*) and COUNT(c1) return different results when NULLs enter the picture. Generally, it's preferable to use COUNT(\*) and let the optimizer choose the best method of returning a row count rather than forcing it to count a specific column.
3. Following are the parameters which determine the behaviour of NULL:

* SET ANSI\_NULL\_DFLT\_ON/\_OFF determines whether columns in newly created tables can contain NULL values by default. You can query this setting via the GETANSINULL() system function.
* SET ANSI\_NULLS controls how equality comparisons with NULL work. The ANSI SQL standard stipulates that any expression involving comparison operators with NULL values returns NULL. Turning this setting off (it's on by default when you connect via ODBC or OLEDB) enables equality comparisons with NULL (including IN operation) to succeed if the column or variable in question contains a NULL value.
* SET CONCAT\_NULL\_YIELDS\_NULL determines whether string concatenation involving NULL values returns a NULL value.

1. The standard allows you to assign column values using "= NULL" in update queries but does not allow you to search for them using the same syntax.
2. One might think that this query:

SELECT \* FROM #values WHERE c1=1

followed by this one:

SELECT \* FROM #values WHERE c1<>1

would return all the rows in the #values table, but that's not the case. Remember that SQL is based on three values logic. To return all rows, we have to allow for NULL values, so something like this is necessary:

SELECT \* FROM #values WHERE c1=1 OR c1 IS NULL

1. RI (referential integrity) is faster than a comparable trigger because it is enforced before the pending change is made. Triggers, by contrast, execute just after a change has been recorded in the transaction log but before it's been written to the database. This notwithstanding, sometimes triggers are a better choice due to their increased power and flexibility.
2. The ANSI SQL-92 specification defines four possible actions that can occur when a data modification for referenced keys is attempted: NO ACTION, SET NULL, SET DEFAULT, and CASCADE. Of these, only the first one, NO ACTION, is supported directly by SQL Server.
3. The presence of foreign key constraints on a table precludes the use of TRUNCATE TABLE. This is true regardless of whether deleting the rows would break a foreign key relationship. Rows deleted by TRUNCATE TABLE aren't recorded in the transaction log, so no row-oriented operations (such as checking foreign key constraints) are possible. It's precisely because TRUNCATE TABLE deals with whole pages rather than individual rows that it's so much faster than DELETE.
4. Default constraints can be more than mere constant value. They can consist of CASE expressions, functions, and other types of scalar expressions (but not sub-queries). Here's an example:

CREATE TABLE #testdc (c1 int DEFAULT CASE WHEN SUSER\_SNAME()='JOE' THEN 1 ELSE 0 END)

1. To drop multiple tables, you can issue a single DROP TABLE followed by a comma-separated list of the tables to drop. This also applies to stored procedures, views, and other types of objects.
2. You can't create indexes on bit, text, ntext, or image and Computed columns because:

* The purpose of an index is to locate a row within a table. SQL Server builds balanced trees (B-trees) using the distinct values in the index's underlying data. If a column has only two distinct values, it's virtually useless as an aid in locating a row.
* SQL Server doesn't allow indexes on computed columns because computed columns do not actually exist in the database—they don't store any real data. A computed column in a table is just like one in a view—they're both rendered when queried, but they do not otherwise exist.

1. PAD\_INDEX, used in conjunction with FILLFACTOR in CREATE INDEX causes the intermediate pages in an index to assume the same fill percentage as that specified by FILLFACTOR for the leaf nodes. Here's an example:

IF INDEXPROPERTY(OBJECT\_ID('titles'),'typeind','IsClustered') IS NOT NULL

DROP INDEX titles.typeind

GO

CREATE INDEX typeind ON titles (type) WITH PAD\_INDEX, FILLFACTOR = 10

PAD\_INDEX is useful when you know in advance that you're about to load a sizable portion of new data that will cause page splits and row relocation in an index's intermediate pages if sufficient space isn't set aside up front for the new data.

1. DROP\_EXISTING allows you to drop and recreate an index in one step. DROP\_EXISTING offers special performance enhancements for clustered indexes in that it rebuilds dependent non-clustered indexes only once and only when the clustered key values change. If the data is already sorted in the correct order, DROP\_EXISTING doesn't resort the data but does compact it using the current FILLFACTOR value. Its atomic—either all the indexes will be created or none of them will be.
2. DDL stands for Data Definition Language while DML stands for Data Manipulation Language.
3. Some interesting facts for INSERT statements:

* To insert a default value for columns, use the DEFAULT keyword in place of an actual value.
* Identity columns may be safely omitted from any INSERT statement, even with target column lists. SET IDENTITY\_INSERT allows values to be specified for identity columns.
* DEFAULT VALUES allows default values to be specified for all columns at once. If you use with columns that do not have defaults of some type defined, your INSERT will fail and a target column list is illegal with DEFAULT VALUES. If you supply one (even if it includes all the columns in the table), your INSERT command will fail.
* If an INSERT that fails due to a constraint or invalid duplicate value will not cause the command batch to fail.
* INSERT command can remove duplicate rows by way of a unique index with the IGNORE\_DUP\_KEY option set. The rows that violate the index's unique constraint will be rejected without causing the other inserts to fail.

1. Extended stored procedures allow you to create your own external routines in a programming language such as C.
2. A table without a clustered index is known as a heap table. Rows inserted into a heap table are inserted wherever there's room in the table. If there's no room on any of the table's existing pages, a new page is created. This can create a hotspot at the end of the table (meaning that users attempting simultaneous INSERTs on the table will vie for the same resources).
3. Prior to SQL Server 7.0, they caused the creation of overflow pages as new rows with duplicate keys were inserted, slowing the operation and fragmenting the table. Beginning with version 7.0, a "uniqueifier" (a four-byte sequence number) is appended to each duplicate clustered index key in order to force it to be unique.
4. BULK INSERT uses the BCP (Bulk Copy Program) facility. When you insert rows via BULK INSERT, INSERT triggers do not fire as it avoids logging inserted rows in the transaction. Declarative constraints, by contrast, can be enforced via the inclusion of BULK INSERT's CHECK\_CONSTRAINTS option. By default, except for UNIQUE constraints, the target table's declarative constraints are ignored. Note that this can slow down the operation considerably.
5. BULK INSERT, by default, causes identity column values to be regenerated as data is loaded. To override this behaviour, include BULK INSERT's KEEPIDENTITY keyword.
6. The situation where an updated row moves within the list of rows being updated during the update, and is therefore changed erroneously multiple times, is known as the **Halloween** **Problem**.
7. SQL Server triggers fire once per statement, not per row.
8. A trigger's code is not compiled into the execution plan for the INSERT, UPDATE, or DELETE that fires it. Rather, it's compiled and cached separately so that it's available for reuse regardless of what causes it to fire.
9. The execution plan for a DML statement branches to any triggers it fires just before it terminates, after its work is otherwise complete. This isn't true of constraints. Steps are added directly to the DML execution plan for each of a table's constraints.
10. To fire trigger and validate constraints while bulk insert, is to issue a fake update once the operation completes. This fake update simply sets each column's value to itself. If any of the rows contain bad data, the UPDATE will fail.
11. As column values referenced by an UPDATE statement always reflect their values before the operation, you don't need an intermediate variable in order to swap them, like this:

UPDATE #samples SET samp1=samp2, samp2=samp1

1. You can use the UPDATE and DELETE command to modify rows returned by updatable cursors. This is facilitated via WHERE CURRENT OF clause.
2. Some interesting fact regarding TRUNCATE are:

* TRUNCATE TABLE command provides a way of deleting the rows in a table with a minimum of logging (page deallocations are logged so that the command could be executed from within a transaction and its effects on the database could be reversed)
* TRUNCATE TABLE is not logging the process of deleting individual rows. That's because no row deletions actually occur— all that really happens is the deallocation of the pages that make up the table.
* TRUNCATE TABLE is many times faster than an unqualified DELETE statement.
* You also can't use TRUNCATE TABLE on a table that's been published for replication. This is because replication relies on the transaction log to synchronize publishers and subscribers, and TRUNCATE TABLE does not generate row deletion log records.

1. DML runtime errors can be detected by inspecting the @@ERROR automatic variable. However, if a DML statement doesn't affect any rows, @@ERROR won't be set because that's technically not an error condition.
2. Some interesting facts about TOP is:

* Prior to SQL Server 7.0, restricting the number of rows returned by a query required the use of the SET ROWCOUNT command. SET ROWCOUNT is still available but SELECT TOP n is mostly used
* PERCENT keyword with TOP limits the rows returned to a percentage of the total number of rows.
* Add WITH TIES clause if you want to include ties (duplicate values) in the result set.

SELECT TOP 4 WITH TIES t.title, SUM(s.qty) AS TotalSales FROM sales s JOIN titles t ON (s.title\_id=t.title\_id) GROUP BY t.title ORDER BY TotalSale title TotalSales

* If a tie is present at last position, this query will return 5 values rather 4
* TOP n can't return grouped top segments in conjunction with a query's GROUP BY clause. Though the syntax is supported, it doesn't do what we might like

1. A derived table is a sub-query that's used in place of a table or view. It can be queried and joined just like any other table or view.
2. The BETWEEN predicate indicates whether a given value falls between two other values, inclusively. In addition to simple constant arguments, BETWEEN accepts sub-query, variable, and expression arguments.
3. ANSI SQL specifies two pattern wildcard characters: the % character and the \_. % matches any number of characters, while \_ matches exactly one. Transact-SQL also supports regular expression wildcards. [ai] is a regular expression wildcard that matches any string with either a or I while to exclude strings, prefix its characters with a caret ^.
4. It's possible for a string to survive an equality test but fail a LIKE test. LIKE are less restrictive than a plain equality test. The reason this is that ANSI SQL padding rules require that two strings compared for equality be padded to the same length prior to the comparison. That's not true for LIKE. If one term is padded with blanks and the other isn't, the comparison will probably fail.
5. As a rule, you should use SELECT \* in the sub-queries you pass EXISTS. This allows the optimizer to select the column to use and should generally perform better.
6. Conversion between EXISTS and IN can be tricky when NULLs are involved because the IN predicate compares a scalar value with a series of values and as per ANSI/ISO SQL guidelines, an expression that compares a value for equality to NULL always returns NULL.
7. Though ANSI SQL-92 allows row values to be used with IN, Transact-SQL does not. You can specify scalar values only.
8. IN and =ANY are functionally equivalent, you might tend to think that NOT IN and <>ANY are equivalent as well, but that's not the case. Instead, <>ALL is the equivalent of NOT IN. This brings up the interesting point that ALL is more often used with the not equal operator (<>) than with the equal operator (=).
9. SQL Server currently supports eight aggregate functions: COUNT(), SUM(), MIN(), MAX(), STDDEV() (standard deviation), STDDEVP() (population standard deviation), VAR() (variance), and VARP() (population variance). All of these except COUNT() automatically ignore NULL values.
10. GROUP BY ALL generates all possible groups—even those that do not meet the query's search criteria. Aggregate values in groups that fail the search criteria are returned as NULL.
11. With the exception of bit, text, ntext, and image columns, any column can participate in the GROUP BY clause.
12. GROUP BY clauses without aggregate functions have a purpose beyond simulating SELECT DISTINCT queries. Same execution plan will be generated for these queries.
13. To reshape vertically oriented data into horizontally oriented tables suitable for reports and user interfaces. These tables are known as pivot tables or cross-tabulations (cross-tabs).
14. HAVING restricts the rows returned by GROUP BY similarly to the way that WHERE restricts those returned by SELECT. It is processed after the rows are collected from the underlying table(s) and is therefore less efficient for garden-variety row selection than WHERE. In fact, behind the scenes, SQL Server implicitly converts a HAVING that would be more efficiently stated as a WHERE automatically. This means that the execution plans generated for the following queries are identical.
15. There are a few simple rules you should keep in mind when using UNIONs:

* Each query listed as a UNION term must have the same number of columns and must list them in the same order as the other queries.
* The columns returned by each SELECT must be assignment compatible or be explicitly converted to a data type that's assignment compatible with their corresponding columns in the other SELECTs.
* Combining columns that are assignment compatible but of different types produces a column with the higher type precedence of the two (e.g., combining a smallint and a float results in a float result column).
* The column names returned by the UNION are derived from those of the first SELECT.
* UNION ALL is faster than UNION because it doesn't remove duplicates before returning. Removing duplicates may force the server to sort the data, an expensive proposition, especially with large tables.

1. When possible, the query optimizer will use an index to service the sort request. When this is impossible or deemed suboptimal by the optimizer, a work table is constructed to perform the sort. With large tables, this can take a while and can run tempdb out of space if it's not sized sufficiently large. This is why you shouldn't order result sets unless you actually need a specific row order—doing so wastes server resources.
2. A few things to keep in mind regarding ORDER BY:

* You can't use ORDER BY in views, derived tables, or subqueries without also using the TOP n extension (see the section on TOP n earlier in this chapter for more information). A technique for working around this is to include a TOP n clause that specifies more rows than exist in the underlying table(s).
* You can't sort on text, ntext, or image columns.
* If your query is a SELECT DISTINCT or combines result sets via UNION, the columns listed in the ORDER BY clause must appear in the SELECTlist.
* If the SELECT includes the UNION operator, the column names and aliases you can use are limited to those of the first table in the UNION.

1. VIEWs are static queries that you can use as though they were tables. A VIEW consists of a SELECT statement compiled ahead of time using SQL's CREATE VIEW command and referenced in the same manner as a table.
2. Some interesting facts about VIEWs are:

* Transact-SQL doesn't support temporary VIEWs, though you can create static VIEWs in tempdb and achieve a similar effect.
* VIEWs aren't allowed to reference temporary tables—only references to other VIEWs or permanent base tables are permitted.
* An UPDATE, INSERT or DELETE to a VIEW is not allowed to affect more than one underlying base table at a time. If the VIEW joins two or more tables together, these operations may alter only one of them
* ORDER BY is not allowed in VIEWs, however, you can use Transact-SQL's TOP n extension to allow ORDER BY in VIEWs, like this:

CREATE VIEW myauthors AS SELECT TOP 50 \* FROM authors ORDER BY au\_lname

1. There are a number of factors affecting whether a VIEW is updatable. For a VIEW to allow updates, the following criteria must be met:

* Aggregate functions, the TOP, GROUP BY, UNION, or DISTINCT clauses or keywords are not allowed.
* Derived columns (columns constructed from complex expressions) are not updatable.
* SELECT lists consisting entirely of nontabular expressions are not allowed.

1. If an insert or update you make through a VIEW that has WITH CHECK OPTION enabled, would cause the row to fail the VIEW's WHERE criteria, the insert/update will be rejected.
2. A dynamic VIEW is simply one whose selection criteria can change based on the evaluation of the expressions in its WHERE or HAVING clauses.
3. Runs, regions, sequences, and series are related data constructs that usually include a minimum of two columns: a key column that is more or less sequential and a value column.

* The key column of a sequence (or series) is sequential, with no gaps between identifiers. Examples of sequences include time series, invoice numbers, account numbers, and so on.
* A run's key column is also sequential, though there may or may not be gaps between identifiers. Examples of runs include those of regular sequences (with gaps, of course) as well as house numbers, version numbers, and the like.
* A region is a subsequence whose members all meet the same criteria. The simplest example of a region is a subsequence whose members all have the same value. An interval is the product of dividing a sequence or run into multiple, evenly sized subsequences or subsets.

1. You can use COMPUTE and ORDER BY with the result set returned by the UNION operation but not with any of its individual SELECT statements. Conversely, GROUP BY and HAVING can be used by individual SELECT statements but not by the entire result set.
2. UNION syntax that allows a table to be created en passant. To do this, you include an INTO table-name clause in the first SELECT statement of those included in the UNION operation, like so:

SELECT \* INTO #tempset FROM #set1 UNION ALL SELECT \* FROM #set2

1. A left outer join returns columns from the rightmost table as NULL when the join condition fails.