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 before hand 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 QtySumFROM InventoryGROUP 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 stringvariable as a SQL command. sp\_executesql is faster and more feature laden than EXEC(). When you need to execute a dynamicallygenerated 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 plangenerated by the query optimizer the first time the query executes. It's more efficient than EXEC() becausethe query string is built only once, and each parameter is specified in its native data format, not first convertedto a string, as EXEC() requires.
4. sp\_executesql allows you to embed parameters within its query string using standard variable names asplaceholders, likeso:

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

Here, @au\_lname is a placeholder. Though the query may be executed several times in succession, the onlything that varies between executions is the value of @au\_lname. This makes it highly likely that the queryoptimizer will be able to avoid recreating the execution plan with each queryrun.Note the use of the "N" prefix to define the literal strings passed to the procedure as Unicode strings. sp\_executesql requires Unicodestrings 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. Ittakes 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 orWRITETEXT command. UPDATETEXT can modify a portion of a BLOB field, while WRITETEXT rewrites itsentire contents. Generally speaking, UPDATETEXT is more flexible than WRITETEXT. Since both UPDATETEXT andWRITETEXT 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 asbytes, 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 possiblevalues would make a very poor index key.
5. A bitmap is acolumn 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 ormanipulate the bit switches in a bitmap. SQL Server makes extensive use of bitmaps and bit masks because they're an efficient way tostore 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 mnemonicpattern.
* The uniqueidentifier data type is four times as large as the four-byte int type that's typically used forrow identifiers. This makes accessing them slower in general, including building and accessingindexes overthem.
* The sequence in which a set of uniqueidentifier values were generated is not discernable from thevalues themselves—you can't tell which values came first and which ones came later by looking onlyat 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 localcursors (since you can't declare global variables) and can be used in place of direct cursor references incommands 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 SETcommand. Deallocating the original cursor doesn't prevent you from continuing toaccess it via the cursor variable.

1. A timestamp is a specialbinary(8) value that's guaranteed to be unique across a database. A timestamp column is updated each timethe data in a row changes. For example, if Juliet updates a row after Romeo reads itbut before he posts his own changes, Romeo's update attempt will fail because it will use the originaltimestamp 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().Ittakes 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 thesame as aggregating a column that contains both NULL and non-NULL values. In the former case, the endresult is always a NULL value. In the latter, the NULL values are ignored and the aggregation is performed.
2. TheCOUNT(\*) and COUNT(c1) return differentresults when NULLs enter the picture. Generally, it's preferable to use COUNT(\*) and let the optimizer choosethe 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 NULLvalues 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 stipulatesthat any expression involving comparison operators with NULL values returns NULL. Turning this setting off (it's on by default when you connect via ODBC orOLEDB) enables equality comparisons with NULL (including IN operation) to succeed if the column or variable in question contains aNULL value.
* SET CONCAT\_NULL\_YIELDS\_NULL determines whether string concatenation involving NULL values returnsa 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 triggerbecause it is enforced before the pending change is made. Triggers, by contrast, execute just after a changehas been recorded in the transaction log but before it's been written to the database. This notwithstanding, sometimes triggersare 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 isattempted: NO ACTION, SET NULL, SET DEFAULT, and CASCADE. Of these, only the first one, NOACTION, is supported directly by SQL Server.
3. The presence of foreign key constraints on a table precludes the use of TRUNCATE TABLE. This is trueregardless of whether deleting the rows would break a foreign key relationship. Rows deleted by TRUNCATETABLE aren't recorded in the transaction log, so no row-oriented operations (such as checking foreign keyconstraints) are possible. It's precisely because TRUNCATE TABLE deals with whole pages rather thanindividual rows that it's so much faster than DELETE.
4. Default constraints can be more than mere constant value. They canconsist of CASE expressions, functions, and other types of scalar expressions (but not sub-queries). Here's anexample:

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

1. To drop multiple tables, you can issue a single DROP TABLEfollowed by a comma-separated list of the tables to drop.This also applies to stored procedures, views, andother 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'svirtually useless as an aid in locating a row.
* SQL Serverdoesn't allow indexes on computed columns because computed columns do not actually exist in thedatabase—they don't store any real data. A computed column in a table is just like one in a view—they're bothrendered when queried, but they do not otherwise exist.

1. PAD\_INDEX, used in conjunction with FILLFACTOR in CREATE INDEX causes the intermediatepages 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 thatwill cause page splits and row relocation in an index's intermediate pages if sufficient space isn't set aside upfront for the newdata.

1. DROP\_EXISTING allows you to dropand recreate an index in one step. DROP\_EXISTING offers special performance enhancements for clusteredindexes in that it rebuilds dependent non-clustered indexes only once and only when the clustered key valueschange. If the data is already sorted in the correct order, DROP\_EXISTING doesn't resort the data but doescompact 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 columnsmay be safely omitted from any INSERT statement, even with target column lists.SET IDENTITY\_INSERT allows values to bespecified for identity columns.
* DEFAULT VALUES allows default values to be specified for all columns at once. If you usewith 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 columnsin 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 rejectedwithout 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 insertedwherever 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 thatusers attempting simultaneous INSERTs on the table will vie for the same resources).
3. Prior to SQLServer 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-bytesequence 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 thetransaction. Declarative constraints, by contrast, can be enforced via the inclusion of BULK INSERT'sCHECK\_CONSTRAINTS option. By default, except for UNIQUE constraints, the target table's declarativeconstraints are ignored. Notethat this can slow down the operation considerably.
5. BULK INSERT, by default, causes identity column values tobe regenerated as data is loaded. To override this behaviour, include BULK INSERT's KEEPIDENTITYkeyword.
6. The situation where an updated row moves within the list of rows being updated during the update, and istherefore changed erroneously multiple times, is known as the **HalloweenProblem**.
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 reuseregardless of what causes it to fire.
9. The execution plan for a DML statement branches to any triggers it firesjust 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'sconstraints.
10. To fire trigger and validate constraints while bulk insert, is to issue a fake update once the operation completes. This fake updatesimply sets each column's value to itself. If any ofthe rows contain bad data, the UPDATE will fail.
11. As column values referenced by an UPDATE statement always reflecttheir values before the operation, you don't need an intermediate variable in order to swap them, like this:

UPDATE #samplesSET samp1=samp2,samp2=samp1

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

* TRUNCATE TABLE command provides a way of deleting the rows in a tablewith a minimum of logging (page deallocationsare logged so that the command could be executed from within a transaction andits effects on the database could be reversed)
* TRUNCATE TABLE is not logging the process of deleting individual rows. That's because no rowdeletions 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 useTRUNCATE TABLE on a table that's been published for replication. This is because replication relies on thetransaction log to synchronize publishers and subscribers, and TRUNCATE TABLE does notgenerate row deletion log records.

1. DML runtime errors can be detected by inspecting the @@ERROR automatic variable. However, if aDML statement doesn't affect any rows, @@ERROR won't be set because that's technically not an errorcondition.
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 SETROWCOUNT command. SET ROWCOUNT is still available but SELECT TOP n ismostly used
* PERCENT keyword with TOP limits the rows returned to a percentage ofthe 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 TotalSalesFROM sales s JOIN titles t ON (s.title\_id=t.title\_id)GROUP BY t.titleORDER BY TotalSaletitle 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'sGROUP 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 inplace of a table or view. It can be queried and joined just like any other table or view.
2. The BETWEEN predicate indicateswhether 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 otherisn'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-SQLdoes 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 equivalentas 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 theequal operator (=).
9. SQL Server currently supports eight aggregate functions: COUNT(), SUM(), MIN(), MAX(), STDDEV()(standard deviation), STDDEVP() (population standard deviation), VAR() (variance), and VARP() (populationvariance). 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 functionshave a purpose beyond simulating SELECT DISTINCT queries.Sameexecution plan will be generated for these queries.
13. To reshape vertically oriented data into horizontally oriented tables suitable forreports 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 restrictsthose returned by SELECT. It is processed after the rows are collected from the underlying table(s) and istherefore less efficient for garden-variety row selection than WHERE. In fact, behind the scenes, SQL Serverimplicitly converts a HAVING that would be more efficiently stated as a WHERE automatically. This meansthat 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 thesame order as the other queries.
* The columns returned by each SELECT must be assignment compatible or be explicitly converted toa 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 thehigher type precedence of the two (e.g., combining a smallint and a float results in a float resultcolumn).
* 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. Removingduplicates may force the server to sort the data, an expensive proposition, especially with large tables.

1. When possible, the query optimizer will use anindex to service the sort request. When this is impossible or deemed suboptimal by the optimizer, a work tableis constructed to perform the sort. With large tables, this can take a while and can run tempdb out of space ifit's not sized sufficiently large. This is why you shouldn't order result sets unless you actually need a specificrow 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 SELECTstatement compiled ahead of time using SQL's CREATE VIEW command and referenced in the same manneras 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 orpermanent 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 VIEWjoins 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 ASSELECT TOP 50 \*FROM authorsORDER 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 twocolumns: 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 betweenidentifiers. Examples of sequences include time series, invoice numbers, account numbers, and so on.
* A run'skey column is also sequential, though there may or may not be gaps between identifiers. Examples of runsinclude those of regular sequences (with gaps, of course) as well as house numbers, version numbers, andthe like.
* A region is a subsequence whose members all meet the same criteria. The simplest example of aregion is a subsequence whose members all have the same value. An interval is the product of dividing asequence 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 anyof its individual SELECT statements. Conversely, GROUP BY and HAVING can be used by individualSELECT statements but not by the entire result set.
2. UNION syntax that allows a table to becreated en passant. To do this, you include an INTO table-name clause in the first SELECT statement of thoseincluded in the UNION operation, like so:

SELECT \* INTO #tempset FROM #set1UNION ALLSELECT \* FROM #set2

1. A left outer join returns columns from the rightmost table asNULL when the join condition fails.
2. CHAR(13) represents carriage return and CHAR(9) represents tab.
3. The types of cursors Transact-SQL supports and their attributes.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Scrollable**  (Not limited to accessing their rows sequentially) | **Membership/Order** | **Column Values** |
| FORWARD\_ONLY (default) | No | Dynamic (Newly added lines are also visible) | Dynamic (Changes made to the underlying data are visible) |
| DYNAMIC/SENSITIVE | Yes (FETCH PRIOR FROM) | Dynamic | Dynamic |
| STATIC / INSENSITIVE | Yes | Fixed | Fixed |
| KEYSET | Yes | Fixed | Dynamic |

1. Keyset Cursors, new row insertions are not reflected by the cursor. The set of unique key values for the cursor's rows are copied to a table in tempdb (hence the term keyset) when the cursor is opened. That's why membership in the cursor is fixed. If the underlying table doesn't have a primary or unique key, the entire set of candidate key columns is copied to the keyset table. Since changes to keyset columns aren't reflected by the cursor, failing to define a unique key of some type for the underlying data results in a keyset that doesn't reflect changes to any of its candidate key columns.
2. Use cursors only when you have to. Some examples of situations where cursor use is appropriate are dynamic queries, row-oriented operations and scrollable forms.
3. Transact-SQL cursors reside on the server and return only fetched rows, they can save a lots of time and resources when dealing with large result sets.
4. Following are the commands for Cursor:

|  |  |
| --- | --- |
| **Command or Function** | **Purpose** |
| DECLARE CURSOR | Defines a cursor |
| OPEN | Opens a cursor so that data may be retrieved from it |
| FETCH | Fetches a single row from the cursor |
| CLOSE | Closes the cursor, leaving intact the internal structures that service it. |
| DEALLOCATE | Frees the cursor's internal structures |
| @@CURSOR\_ROWS | Returns the number of rows exposed by the cursor |
| @@FETCH\_STATUS | Indicates the success or failure of the last FETCH |
| CURSOR\_STATUS() | Reports status info for cursors and cursor variables |

1. Transact-SQL's extended syntax follows this form:

DECLARE name CURSOR

[LOCAL | GLOBAL]

[FORWARD\_ONLY | SCROLL]

[STATIC | KEYSET | DYNAMIC | FAST\_FORWARD]

[READ\_ONLY | SCROLL\_LOCKS | OPTIMISTIC]

[TYPE\_WARNING]

FOR select

[FOR {READ ONLY | UPDATE [OF column [,…n]]}]

1. If you include the FOR UPDATE clause but specify a select that inherently prohibits updates (e.g., one that includes GROUP BY or DISTINCT), your cursor will be implicitly converted to a read-only (or static) cursor. The server converts cursors to static cursors that, by their very nature, cannot be updated. These types of automatic conversions are known as **implicit cursor conversions**.
2. If you specify the FOR UPDATE clause and include a column list, the column(s) you update must appear in that list. If you attempt to update a column not in the list using UPDATE's WHERE CURRENT OF clause, SQL Server will reject the change and generate an error message.
3. If select references a variable, the variable is resolved when the cursor is declared, not when it's opened.
4. A global cursor persists until it's explicitly de-allocated or until its host connection disconnects. Local cursors are implicitly de-allocated when they go out of scope. SQL Server creates global cursors by default. You can have global and local cursors with identical names.
5. If the cursor is an INSENSITIVE or STATIC cursor, OPEN copies the entirety of its result set to a temporary table. If there are both a local and a global cursor with the same name, use GLOBAL to indicate the global one. This is default to local cursor.
6. For dynamic cursors, @@CURSOR\_ROWS returns –1 since new row additions could change the number of rows returned by the cursor at any time. If the cursor is being populated asynchronously, @@CURSOR\_ROWS returns a negative number whose absolute value indicates the number of rows currently in the cursor.
7. @@FETCH\_STATUS is used as control variable. Scrollable cursors allow FETCH to retrieve rows other than the cursor's next row. As scrollable cursors allow FETCH to retrieve a cursor's previous row, its first row (FETCH FIRST FROM), its last row (FETCH LAST FROM), an absolute row number (FETCH ABSOLUTE 4 FROM), and a row relative to the current row (FETCH RELATIVE -1 FROM). FETCH RELATIVE 0 can be used to refresh the current record. This allows you to accommodate changes made to the current row while the cursor is being traversed.
8. The procedures sp\_configure and sp\_dboption and the SET command can be used to configure cursors. By default, SQL Server generates all keysets synchronously—that is, the call to OPEN doesn't return until the cursor's result set has been fully materialized. This may not be optimal for large data sets, and you can change it via the sp\_configure **'cursor threshold**' configuration option.
9. By default, a SQL Server cursor remains open until explicitly closed or until the connection that created it disconnects. To force SQL Server to close cursors when a transaction is committed, use the SET **CURSOR\_CLOSE\_ON\_COMMIT** command. ROLLBACK doesn't close cursors unless **CLOSE\_CURSOR\_ON\_COMMIT** has been enabled. Set the default to local cursor database option to true using sp\_dboption.
10. The **WHERE CURRENT OF** clause of the UPDATE and DELETE commands allows you to update and delete rows via a cursor.
11. We can't pass a cursor via an input parameter into a procedure—you can return cursors only via output parameters. You also cannot define table columns using the cursor data type—only variables are allowed—nor can you assign a cursor variable using the SELECT statement (as with scalar variables)—you must use SET.
12. DEALLOCATE differs in handling cursor used as name and used as variable. It doesn't actually de-allocate the cursor unless it's the last reference to it. It does, however, prevent future access using the specified cursor identifier. So if you have a cursor named foo and a cursor variable named foovar to which foo has been assigned, de-allocating foo will do nothing except prohibit access to the cursor via foo—foovar remains intact.
13. The following are the list of functions which work with cursors:

|  |  |
| --- | --- |
| **Procedure** | **Function** |
| sp\_cursor\_list | Returns a list of the cursors and their attributes that have been opened by a connection |
| sp\_describe\_cursor | Lists the attributes of an individual cursor. |
| sp\_describe\_cursor\_columns | Lists the columns (and their attributes) returned by a cursor |
| sp\_describe\_cursor\_tables | Returns a list of the tables referenced by a cursor |

1. Following are the tips for Cursors:

• Use the FAST\_FORWARD cursor option in lieu of FORWARD\_ONLY when working with unidirectional, read-only result sets. Using FAST\_FORWARD defines a FORWARD\_ONLY, READ\_ONLY cursor with a number of internal performance optimizations.

• Be careful with modifying large numbers of rows via a cursor loop that's contained within a transaction.

Depending on the transaction isolation level, those rows may remain locked until the transaction is committed or rolled back, possibly causing resource contention on the server.

• Be careful with updating dynamic cursors, especially those constructed over tables with nonunique clustered index keys, because they can cause the "Halloween Problem”

1. ACID is an acronym for atomic, consistent, isolated, and durable. A transaction is atomic if it's an all-or-nothing proposition. A transaction is consistent if it ensures that its underlying data never appears in an interim or illogical state—that is, if it never appears to be inconsistent. The trade-off with each isolation level is one of concurrency vs. consistency. The more airtight the isolation, the higher the degree of data consistency. A transaction is considered durable if it ensure that transactions committed but not yet stored in the database are written to the database following a system failure (rolled forward) and that transactions in progress are reversed (rolled back).
2. Any time a data modification occurs, SQL Server writes a record of the change to the transaction log. This occurs before the change itself is performed and is the reason SQL Server is described as having a "writeahead" log.
3. SQL Server supports four basic types of transactions: automatic, implicit, user-defined, and distributed. Implicit transactions are initiated automatically when any of numerous DDL or DML commands is executed. They continue until explicitly committed by the user. To toggle implicit transaction support, use the SET IMPLICIT\_TRANSACTIONS command. The BEGIN TRAN, COMMIT TRAN, and ROLLBACK TRAN commands are used to control user-defined transactions. Transactions that span multiple servers are known as distributed transactions.
4. The BULK INSERT, TRUNCATE TABLE, SELECT…INTO, and WRITETEXT/ UPDATETEXT commands minimize transaction logging by causing only page operations to be logged. They're non-logged in that they don't generate row-level log records.
5. Use the SET TRANSACTION ISOLATION LEVEL command to set a transaction's isolation level. Valid TILs include READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE.

• **READ UNCOMMITTED** permits dirty reads (reads of uncommitted changes by other transactions) and non-repeatable reads (data that changes between reads during a transaction).

• **READ COMMITTED** (default TIL) avoids dirty reads by initiating share locks on accessed data but permits changes to underlying data during the transaction, possibly resulting in non-repeatable reads and/or phantom data.

• **REPEATABLE READ** initiates locks to prevent other users from changing the data a transaction accesses but doesn't prevent new rows from being inserted, possibly resulting in phantom rows appearing between reads during the transaction.

• **SERIALIZABLE** prevents dirty reads and phantom rows by placing a range lock on the data it accesses. It is the most restrictive of SQL Server's four TILs. It's equivalent to using the HOLDLOCK hint with every table a transaction references.

1. You can also use the COMMIT WORK and ROLLBACK WORK commands in lieu of COMMIT TRANSACTION and ROLLBACK TRANSACTION, though you cannot use transaction names with them.
2. @@TRANCOUNT automatic variable can be queried to determine the level of nesting—0 indicates no nesting. In nested transactions, a COMMIT issued against any transaction except the outermost one doesn't commit any changes to disk. A ROLLBACK, on the other hand, works regardless of the level at which it is issued but rolls back all transactions, regardless of the nesting level.
3. ROLLBACK can reverse a named transaction only when it's the outermost transaction. Attempting to roll back our nested transaction yields the error message.
4. SAVE TRAN creates a save point to which you can roll back if you wish. Syntactically, you just pass the name of the save point to the ROLLBACK TRAN command.
5. Following Transact-SQL commands are illegal inside transactions: ALTER DATABASE, DROP DATABASE, RECONFIGURE, BACKUP LOG, DUMP TRANSACTION, RESTORE DATABASE, CREATE DATABASE, LOAD DATABASE, RESTORE LOG, DISK INIT, LOAD TRANSACTION, and UPDATE STATISTICS.
6. Following are two DBCC (database consistency checker) commands:

• **DBCC OPENTRAN** allows you to retrieve the oldest active transaction in a database.

• **DBCC LOG** lists the database transaction log. You can use it to look under the hood and see what operations are being carried out on your data.

1. Performance improvement sometimes comes at a cost as SELECT…INTO, for example, locks system tables until it completes.
2. Each time you successfully create a new procedure, its name and other vital information are recorded in sysobjects and its source code into the syscomments system table. The procedure isn't compiled until it's executed for the first time. Despite the name, syscomments stores far more than comments—it's the repository for the source code for stored procedures, views, triggers, rules, and defaults.
3. Stored Procedures can provide security mechanisms, allowing users controlled access to database objects they could not otherwise use. They can reduce network bandwidth use by greatly lessening the amount of Transact-SQL code that must traverse the network in order to accomplish tasks. Since their execution plans are retained by the server for reuse, they can improve application performance considerably.
4. There are four major steps involved with using stored procedures:
   * Creation—where you initially create the procedure with CREATEPROC
   * During User execution, the command processor constructs what's known as a sequence tree or query tree that will be passed to the query optimizer for compilation and optimization.
   * Compilation—where the server compiles and optimizes the procedure during an EXEC. This execution plan consists of the following:
     1. The steps required to carry out the work of the stored procedure.
     2. The steps necessary to enforce constraints.
     3. The steps needed to branch to any triggers red by the stored procedure.

Execution plans in SQL Server 7.0 and later are reentrant and read-only. This differs from previous releases, where each connection received its own copy of the execution plan for a given procedure.

* + Server execution—where the server runs its compiled execution plan during an EXEC. Execution plans are never stored on disk. The only portion of the stored procedure that's stored permanently is its source code (in syscomments). Since they're kept in memory, cycling the server disposes of all current execution plans.

1. Following are some important point related to SP:
   * Stored procedures can be altered with ALTER PROCEDURE. The advantage to using ALTER PROC rather than CREATE PROC to change a stored procedure is that it preserves access permissions, whereas CREATE PROC doesn't.
   * A procedure can contain any valid Transact-SQL command except these: CREATE DEFAULT, CREATE PROC, CREATE RULE, CREATE SCHEMA, CREATE TRIGGER, and CREATE VIEW.
   * To execute CREATE PROC you must be a member of the sysadmin role, the db\_owner role, or the db\_ddladmin\_role.
   * The maximum stored procedure size is the lesser of 65,536 \* the network packet size (which defaults to 4096 bytes) and 250 megabytes. The maximum number of parameters a procedure may receive is 1024.
2. A prefix of one pound sign (#) creates a local temporary procedure that's visible only to the current connection, while a prefix of two pound signs (##) creates a global temporary procedure that's visible to all connections.
3. System procedures are procedures that reside in the master database and are prefixed with sp\_. System procedures are executable from any database assumes the context of the database in which it's running.
4. Extended procedures are routines that reside in DLLs (Dynamic Link Libraries) that look and work like regular stored procedures. They receive parameters and return results via the Open Data Services framework and are normally written in C or C++. They reside in the master database (you cannot create them elsewhere). They aren't automatically located in master when referenced from other databases, and they don't assume the context of the current database when run.
5. Faux Procedures are system-supplied stored procedures that are neither true system procedures nor extended procedures. They're implemented internally by the server itself. Examples are sp\_executesql, sp\_prepare, most of the sp\_cursorXXXX routines, sp\_reset\_connection, etc. You can't list their source code because it's part of the server itself, and you can't trace into them with a T-SQL debugger because they're not written in Transact-SQL.
6. The INSERT command supports calling a stored procedure in order to supply rows for insertion into a table. Here's an example:

INSERT <table-name> (<comma separated fields name>) EXEC <stored procedure name>

1. The status of **QUOTED\_IDENTIFIER** and **ANSI\_NULLS,** is actually recorded in each procedure's status field in sysobjects. QUOTED\_IDENTIFIER controls whether strings within double quotes are interpreted as object identifiers and ANSI\_NULLS controls whether non-ANSI equality comparisons with NULLs are allowed. QUOTED\_IDENTIFIER facilitate references to objects whose names contain reserved words, spaces, or other normally disallowed characters.
2. Parameters can be passed to stored procedures by name or by position. Here's an example of each method:

EXEC sp\_msforeachtable @command1='sp\_help "?"', @replacechar = '?'

EXEC sp\_msforeachtable 'sp\_help "?"', '?'

1. Output parameters allow values to be returned from stored procedures. These parameters can be integers, character strings, dates, and even cursors. Here's an example:

CREATE PROC dbo.listsales @bestseller tid OUT, @topsales int OUT, @salescursor cursor varying OUT

EXEC listsales @bestseller OUT, @topsales OUT, @salescursor OUT

SELECT @bestseller, @topsales

1. A return code of 0 indicates success, values –1 through –14 indicate different types of failures, and values –15 through –99 are reserved for future use.

|  |  |
| --- | --- |
| **Code** | **Meaning** |
| –1 | Object missing |
| –2 | Data type error occurred |
| –3 | Process chosen as deadlock victim |
| –4 | Permission error |
| –5 | Syntax error |
| –6 | Miscellaneous user error |
| –7 | Resource error |
| –8 | Nonfatal internal error |
| –9 | System limit reached |
| –10 | Fatal internal inconsistency error |
| –11 | Fatal internal inconsistency error |
| –12 | Corrupt table or index |
| –13 | Corrupt database |
| –14 | Hardware error |

1. You can list a procedure's parameters (which include its return code—considered parameter0) using the undocumented procedure **sp\_procedure\_params\_rowset** and Stored procedures support up to 1024 parameters. The number of stored procedure local variables is limited only by available memory.
2. The following are the automatically available variables:

|  |  |
| --- | --- |
| **Variable Name** | **Returns** |
| @@NESTLEVEL | The current procedure nesting level (Stored procedures can be nested up to 32 levels deep) |
| @@OPTIONS | A bitmap of the currently specified user options |
| @@PROCID | The object ID of the current procedure |
| @@SPID | The process ID of the current process |
| @@TRANCOUNT | The current transaction nesting level |

1. Transact-SQL flow control language statements include IF…ELSE, WHILE, GOTO, RETURN, WAITFOR (WAITFOR DELAY '00:00:03' -- Delay 3 secs to view message), BREAK, CONTINUE, and BEGIN…END.
2. RAISERROR doesn't change the flow of the procedure, it merely displays an error message (optionally writing it to the SQL Server error log and the NT application event log) and sets the @@ERROR automatic variable. RAISERROR can reference predefined error messages that reside in the sysmessages table (you create these with sp\_addmessage), or you can supply it with a custom message string. If you supply a custom message during the call to RAISERROR, the error number is set to 50,000. RAISERROR allows both a severity and a state to be specified with each message. Severity values less than 16 produce informational messages in the system event log (when logged), a severity of 16 produces a warning message in the event log, and severity values greater than16 produce error messages in the event log. Severity values up through 18 can be specified by any user; severity values 19–25 are reserved for members of the sysadmin role and require the use of the WITH LOG option. Note that severity values over20 are considered fatal and cause the client connection to be terminated. State is an informational value that you can use to indicate state information to your front-end application. Raising an error with a state of 127 will cause the ISQL and OSQL utilities to set the operating system ERRORLEVEL value to the error number returned by RAISERROR.

RAISERROR('Setting the OS ERRORLEVEL variable',16,127) WITH NOWAIT

PRINT 'Prior to 7.0, execution would never make it here in ISQL'

1. Autostart procedures can be used to perform start-up processes and other administrative work or to load commonly used procedures into the procedure cache with each server boot. They must reside in the master database and must be owned by a member of the sysadmin role. You use the sp\_procoption stored procedure to flag a stored procedure as an autostart routine, like so:

EXEC sp\_procoption 'sp\_databases','startup',true

1. You can encrypt the source code that's stored in syscomments for a stored procedure, view, or trigger using the WITH ENCRYPTION option when you create the object. Encrypted objects have the third bit of the texttype column in syscomments set. Note that once you've encrypted an object, there's no supported way of decrypting it.
2. A trigger is a special type of stored procedure that executes when a specified DML operation (an INSERT, DELETE, or UPDATE or any combination of them) occurs.

CREATE TRIGGER SalesQty\_INSERT\_UPDATE ON sales FOR INSERT, UPDATE AS

Some important points related to triggers are:

* + When Trigger’s host table is dropped, so it is.
  + Triggers re once per statement, not per row.
  + Triggers associated with the same DML statement will re in succession in no particular order. DRI (declarative referential integrity) constraints have precedence over triggers.
  + Inside triggers, UPDATE() function returns true or false based on whether the value of a specified column is being set (regardless of whether it's actually changing). COLUMNS\_UPDATED() returns a bitmap representing which columns are being set.
  + When a user transaction is not active, a trigger and the DML operation that red it, are considered a single transaction.
  + SQL Server exposes special logical tables for use by triggers: the inserted and deleted tables. For INSERT operations, the inserted table lists the row(s) about to be appended to the table. For DELETE operations, the deleted table lists the row(s) about to be removed from the table. For UPDATE operations, the deleted table lists the old version of the row(s) about to be updated, and the inserted table lists the new version.
  + Non-logged operations do not re triggers.
  + You can disable a trigger via the ALTER TABLE…DISABLE TRIGGER command. Disabled triggers can be re-enabled using ALTER TABLE…ENABLE TRIGGER.
  + Triggers re just after the work has been completed by the DML statement but before it has been committed to the database.

1. Database Design Performance Tips are as follows:
   * Keeping clustered index keys as narrow as possible will help reduce the size of nonclustered indexes since they now reference the clustered index (if one exists) rather than referencing the table directly.
   * Use Declarative Referential Integrity constraints to ensure relational integrity when possible because they're generally faster than triggers and stored procedures. DRI constraints cause highly optimized native machine code internal to SQL Server to run.
   * Use fixed-length character data types when the length of a column's data doesn't vary significantly throughout a table. Processing variable-length columns requires more processing resources than handling fixed-length columns.
   * Disallow NULLs when possible—handling NULLs adds extra overhead to storage and query processing. It's not unheard of for developers to avoid NULLs altogether, using placeholders to signify missing values as necessary.
   * If the primary key for a given table is sequential (e.g., an identity column), consider making it a non-clustered primary key. A clustered index on a monotonically increasing key is less than optimal since you probably won't ever query the table for a range of key values or use the primary key column(s) with ORDER BY. A clustered sequential primary key can cause users to contend for the same area of the database as they add rows to the table, creating what's known as a "hotspot." Avoid this if you can by using clustered keys that sort the data more evenly across the table.
   * If a table frequently experiences severe contention, especially when multiple users are attempting to insert new rows, page locks may be at fault. Consider using the sp\_indexoptions system stored procedure to disable page locks on the suspect table. Disabling page locks forces the server to use row locks and table locks. This will prevent the automatic escalation of row locks to page locks from reducing concurrency.
2. Index Performance Tips are as follows:
   * Clustered indexes are best for range selections and ordered queries. Clustered indexes are also appropriate for keys with a high density (those with many duplicate values). Since rows are physically sorted, queries that search using these non-unique values will find them with a minimum number of I/O operations. Nonclustered indexes are better for singleton selects and individual row lookups.
   * Make nonclustered indexes as highly selective (i.e., with as low densities) as possible. Index selectivity can be calculated using the formula Selectivity = Number of Unique Keys / Number of Rows. Nonclustered indexes with selectivity less than 0.1 are not efficient, and the optimizer will refuse to use them. Nonclustered indexes are best used to find single rows. Obviously, duplicate keys force the server to work harder to locate a particular row.
   * Along the lines of making indexes highly selective, order the key columns in a multicolumn index by selectivity, placing more selective columns first. As the server traverses the index tree to find a given key column value, the use of highly selective key columns means that it will have to perform fewer I/Os to reach the leaf level of the index, resulting in a faster query.
   * Consider creating indexes on foreign key references. Foreign keys require a unique key index on the referenced table but make no index stipulations on the table making the reference. Creating an index on the dependent table can speed up foreign key integrity checks that result from modifications to the referenced table and can improve join performance between the two tables.
   * If the optimizer can retrieve all the data it needs from a non-clustered index without having to reference the underlying table, it will do so. This is called **index covering**, and indexes that facilitate it are known as covered indexes. If adding a small column or columns to an existing nonclustered index would give it all the data a popular query needs, you may find that it speeds up the query significantly.
   * You can use DBCC DBREINDEX() to rebuild the indexes on a table. This is one way of removing dead space from a table. Here's an example:

**DBCC DBREINDEX('Customers','PK\_Customers') or DBCC DBREINDEX('Customers','',100)**

In the first example, we pass the name of the clustered index into DBREINDEX. Rebuilding its clustered index rebuilds a table's nonclustered indexes as well. In the second example, we pass an empty string for the index name. This also causes all indexes on the table to be rebuilt. The nice thing about DBREINDEX is that it's atomic—either the specified index or indexes are all dropped and recreated or none of them are. In fact, DBREINDEX is the only way to rebuild primary and unique key indexes.

* + Thanks to the query optimizer's use of multiple indexes on a single table, multiple single-key indexes can yield better overall performance than a compound-key index. This is because the optimizer can query the indexes separately and then merge them to return a result set. This is more flexible than using a compound-key index because the single-column index keys can be specified in any combination. That's not true with a compound key—you must use compound-key columns in a left-to right order.

1. SELECT Performance Tips are as follows:
   * Match query search columns with those leftmost in the index when possible. An index on stor\_id, ord\_num will not be of any help to a query that filters results on the ord\_num column.
   * Don't use DISTINCT or ORDER BY "just in case.". Unless the optimizer can locate an index to service them, they can force the creation of an intermediate work table, which can be expensive in terms of performance.
   * If a query includes an IN predicate that contains a list of constant values (rather than a subquery), order the values based on frequency of occurrence in the outer query, if you know the bias of your data well enough. Since the predicate returns true as soon as any of its values match, moving those that appear more often to the first of the list should speed up the query, especially if the column being searched is not indexed.
   * If you are benchmarking one query against another to determine the most efficient way to access data, be sure to clear the cache. It can be done using undocumented DBCC command verbs to clear out the relevant caches. DBCC FREEPROCCACHE frees the procedure cache; DBCC DROPCLEANBUFFERS clears all caches.
2. INSERT Performance Tips are as follows:
   * Because individual row inserts aren't logged, SELECT...INTO is often many times faster than a regular logged INSERT. It locks system tables, so use it with care. If you use SELECT...INTO to create a large table, other users may be unable to create objects in your database until the SELECT...INTO completes.
   * BULK INSERT is faster than INSERT for loading external data, even when fully logged, because it operates at a lower level within the server.
3. Bulk Copy Performance Tips are as follows:
   * Use the new BULK INSERT command rather than the bcp utility to perform bulk load operations. Though, at the lowest level, they use the same facility that's been in SQL Server since its inception, data loaded via BULK INSERT doesn't navigate the Tabular Data Stream protocol, go through Open Data Services, or traverse the network. It's sent directly to SQL Server as an OLE-DB rowset. The upside of this is that it's much faster—sometimes twice as fast—as the bcp utility. The downside is that the data file being loaded must be accessible over the network by the machine on which SQL Server is running. This can present problems over a WAN (wide area network) where different segments of the network may be isolated from one another but where you can still access SQL Server via a routable protocol such as TCP/IP.
   * If possible, lock tables during bulk load operations (e.g., BULK INSERT). This can significantly increase load speed by reducing lock contention on the table. The best way to do this is to enable the table lock on bulk load option via the sp\_tableoption system procedure, though you can also force table locks for specific bulk load operations via the TABLOCK hint.
   * Consider directing bulk inserts to a staging area when possible, preferably to a separate database. Since the minimally logged version of BULK INSERT prohibits indexes on the target table (including those created as a result of a PRIMARY KEY constraint), it's sensible to set up staging tables whose whole purpose is to receive data as quickly as possible from BULK INSERT. By placing these tables in a separate database, you avoid invalidating the transaction log in your other databases during bulk load operations. In fact, you might not have to enable select into/bulkcopy in any database except the staging area. Once the data is loaded into the staging area, you can then use stored procedures to move it in batches from one database to another.
4. DELETE and UPDATE Performance Tips is:
   * Because individual row deletions aren't logged, TRUNCATE TABLE is usually many times faster than a logged DELETE. Like all minimally logged operations, it invalidates the transaction log, so use it with care.
5. Stored Procedure Performance Tips is:
   * Minimize the number of network round-trips between clients and the server. One very effective way to do this is to disable DONE\_IN\_PROC messages. You can disable them at the procedure level via SET NOCOUNT or at the server level with the trace flag 3640.
6. A SARG, or search argument, can be referenced as Column *op* Constant/Variable (the terms can be reversed) where op is one of the following inclusive operators:=,>=,<=,>,<, BETWEEN, and LIKE (LIKE expressions that can be restated in terms of " x is greater than value y and less than value z " are useful to the optimizer as SARGs—otherwise they aren't). The rule of thumb for identifying SARGs is that a clause can be a useful search argument if the optimizer can detect that it's a comparison between an index key value and a constant or variable. A clause that compares two columns or one that compares two expressions is not a SARG clause.
7. Query optimizer convert ‘$1 LIKE 'Gr%'’ to $1 > 'GQ\_' AND $1 < 'GS' which allows specific key values in the index to be referenced. The '\_'character has the ASCII value of 254, so 'GQ\_' is two values before 'Gr' followed by any character. The same is not valid for ‘%Gr%’, so in later case none of the index will be used. Similarly, $1 !=0 is changed to $1 < 0 OR $1 > 0.
8. When we joined the two SARG clauses via OR, they're processed in parallel using the appropriate index and then combined using a "hash match" operation just before being returned as a result set.
9. If the amount of data being returned is large, even proper SARGs query scans the clustered index. As optimizer has estimated that it's less expensive to scan the entire table and filter results sequentially than to use the nonclustered index because each row located via the index must then be looked up in the clustered index (or underlying table if no clustered index exists) in order to retrieve the other columns the query requests. The step in the execution plan where this occurs is called the "**Bookmark lookup**".
10. Widening nonclustered index key columns results in slower updates, because they must be kept up to date, and slower query processing, because they're physically larger—they require more I/O and more memory to process.
11. The following are the point related to data types:
    * Avoid use of float, real, or datetime data types for primary keys.
    * For best performance, columns should ideally be set to NOT NULL. For example, use of the IS NULL keywords in the WHERE clause makes that portion of the query non-SARGable.
    * If you are using fixed length columns (CHAR, NCHAR) in your table, consider avoiding storing NULLs in them.
12. A number of techniques that you can use to de-normalize a database and hopefully improve performance exist:
    * **Creating contrived or virtual columns**: is one that's composed of the values from other columns. Setting up a computed column saves you from having to include its underlying expression each time you query the table. You define computed columns with the CREATE TABLE or ALTER TABLE command.

ALTER TABLE <table> ADD <column> AS <Conditional values like CASE>

* + **Maintaining redundant copies of data**: is to maintain multiple copies of the same data. For example, you may find that it's worthwhile to look up and store joined values in advance. Of course, you'll want to be careful with this because it adds additional overhead to maintaining data integrity. The more copies of data you have, the more work required to keep it up to date and the more likely a mishap can compromise database integrity.
  + **Keeping summary tables:** summarize detail data from other tables. Building a summary table typically consists of running a popular query ahead of time and storing its results in a summary table. When applications need access to the data, they access this static table. During off-peak, the summary query can be rerun and the table updated with the latest info. One problem with this approach is in administration. If you had ten stored procedures on the original table, you are likely to need twenty now.
  + **Partitioning data horizontally or vertically**. **Vertical Partitioning**: Since SQL Server uses a fixed database page size of 8KB and a single row cannot span pages, the number of rows that will fit on a page is determined by row width. The wider a row, the fewer rows that fit on each page. Physically splitting a table into multiple tables (create new table using some of the column) allows more rows to fit on a page, potentially increasing query performance. **Horizontal Partitioning**: As rows are added to a table, the infrastructure required to support it grows in size. Eventually, it gets so large that index navigation alone is an expensive and time-consuming proposition. Traversing an index B-tree that contains millions of keys can require more time than accessing the data itself. One answer to this is to partition the table horizontally—to break it into multiple tables based on the value of some column or columns. Then, the number of rows any one query will have to navigate is far less.

1. DBMSs take a variety of approaches to optimizing queries.
   * Some optimize based on heuristics—internally reordering and reorganizing queries based on a predefined set of algebraic rules. Query trees are dissected and associative and commutative rules are applied in a predetermined order until a plan for satisfying the entire query emerges.
   * Some optimize queries based on syntactic elements. This places the real burden of optimization on the user because WHERE clause predicates and join criteria are not reordered. They generate the same execution plan with each run.
   * Some use cost-based optimization which weighs several different methods of satisfying a query against one another and selects the one that will execute in the shortest time. A cost-based optimizer bases this determination on estimates of I/O, CPU utilization, and other factors that affect query performance.
2. Join Optimizations flexibility allows the optimizer to find the best way of linking one table with another using all the information at its disposal:
   * **Nested Loops** joins consist of a loop within a loop. They designate one table in the join as the outer loop and the other as the inner loop. For each iteration of the outer loop, the entire inner loop is traversed. This works fine for small to medium-sized tables, but as the loops grow larger, this strategy becomes increasingly inefficient.
   * **Merge joins** perform much more efficiently with large data sets than nested loop joins. A row from each table in the join is retrieved and compared. Both tables must be sorted on the merge column for the join to work. The optimizer usually opts for a merge join when working with a large data set and when the comparison columns in both tables are already sorted.
   * **Hash joins** are also more efficient with large data sets than nested loop joins. Additionally, they work well with tables that are not sorted on the merge column(s). The server performs hash joins by hashing the rows from the smaller of the two tables (designated the "build" table), inserting them into a hash table, processing the larger table (the "probe" table) a row at a time, and scanning the hash table for matches. Because the smaller of the two tables supplies the values in the hash table, the table size is kept to a minimum, and because hashed values rather than real values are used, comparisons can be made between the tables very quickly.
3. Index Optimizations made possible by the optimizer's ability to make use of multiple indexes on the same table.
   * **Index Joins:** SQL Server can join multiple nonclustered indexes to create covered indexes on the fly. This is often faster than using the indexes separately and certainly quicker than sequentially scanning the table itself. Now that the optimizer can join indexes, it may be more sensible to split these keys into multiple indexes and allow the optimizer to join them as necessary. This allows the optimizer to use them individually as well, which wouldn't be the case with a compound key.
   * **Index Merging and Intersection** is the optimizer's ability to merge and intersect indexes. This allows it to merge the matching keys in multiple indexes into a set of key values that it may then look up in the clustered index or underlying table in order to retrieve columns not found in the indexes.
4. **Grouping Optimizations**: The normal order of events when GROUP BY is present in a query containing joins is to perform the joins before grouping the data. Sometimes, however, it's faster to group the data first, especially when working with a huge number of rows that will be coalesced into a relatively small number of groups. Now, the optimizer can potentially recognize situations where grouping first would be beneficial and act accordingly.
5. **Predicate Clause Optimizations:** The optimizer can detect when predicate clauses are associative and eliminate unnecessary join steps. Suppose if we are joining all three of them on the same column. Normally, the flow of execution would be to join #tmp1 and #tmp2, then join #tmp2 and #tmp3, and then join the results of the first to joins. If all three tables are joined on the same column(s), the query optimizer can eliminate one of these three steps by joining #tmp1 and #tmp3 and then joining the result with #tmp2.
6. The Database Engine often has to acquire locks at multiple levels of granularity to fully protect a resource. This group of locks at multiple levels of granularity is called a lock hierarchy.
7. The following table shows the resource lock modes that the Database Engine uses.

|  |  |
| --- | --- |
| **Lock mode** | **Description** |
| Shared (S) | Used for read operations that do not change or update data, such as a SELECT statement. |
| Update (U) | Used on resources that can be updated. Prevents a common form of deadlock that occurs when multiple sessions are reading, locking, and potentially updating resources later. |
| Exclusive (X) | Used for data-modification operations, such as INSERT, UPDATE, or DELETE. Ensures that multiple updates cannot be made to the same resource at the same time. |
| Intent | Used to establish a lock hierarchy. The types of intent locks are: intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX). |
| Schema | Used when an operation dependent on the schema of a table is executing. The types of schema locks are: schema modification (Sch-M) and schema stability (Sch-S). |
| Bulk Update (BU) | Used when bulk copying data into a table and the **TABLOCK** hint is specified. |
| Key-range | Protects the range of rows read by a query when using the serializable transaction isolation level. Ensures that other transactions cannot insert rows that would qualify for the queries of the serializable transaction if the queries were run again. |

1. In a repeatable read or serializable transaction, the transaction reads data, acquiring a shared (S) lock on the resource (page or row), and then modifies the data, which requires lock conversion to an exclusive (X) lock. If two transactions acquire shared-mode locks on a resource and then attempt to update data concurrently, one transaction attempts the lock conversion to an exclusive (X) lock. The shared-mode-to-exclusive lock conversion must wait because the exclusive lock for one transaction is not compatible with the shared-mode lock of the other transaction; a lock wait occurs. The second transaction attempts to acquire an exclusive (X) lock for its update. Because both transactions are converting to exclusive (X) locks, and they are each waiting for the other transaction to release its shared-mode lock, a deadlock occurs. To avoid this potential deadlock problem, update (U) locks are used. Only one transaction can obtain an update (U) lock to a resource at a time. If a transaction modifies a resource, the update (U) lock is converted to an exclusive (X) lock.
2. The cost is the time needed to execute a statement/query/batch. Logical processing stages decide the order in which the different clauses are logically processed.