No code may be placed between END TRY and BEGIN CATCH
TRY and CATCH blocks may be nested

CATCH block defined by BEGIN CATCH...END CATCH

Execution moves to the CATCH block when catchable errors occur within the TRY block

```
BEGIN TRY
    -- Generate a divide-by-zero error.
 SELECT 1/0;
END TRY
BEGIN CATCH
SELECT
         ERROR_NUMBER() AS ErrorNumber
        ,ERROR_SEVERITY() AS ErrorSeverity
        ,ERROR_STATE() AS ErrorState
        ,ERROR_PROCEDURE() AS ErrorProcedure
        ,ERROR_LINE() AS ErrorLine
        ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO
```

ErrorNumber	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLine	ErrorMessage
8134	16	1	NULL	3	Divide by zero error encountered.

Not all errors can be caught by TRY / CATCH:

Syntax, compilation errors, and name resolution errors

```
BEGIN TRY
    -- Table does not exist; object name resolution
    -- error not caught.
SELECT * FROM IDontExist;
END TRY
BEGIN CATCH
    SELECT
         ERROR_NUMBER() AS ErrorNumber
        ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH
```

```
Invalid object name 'IDontExist'.
```

<u>THROW statement</u>:

SQL Server 2012 provides the new THROW statement

Successor to the RAISERROR statement

Does not require defining errors in the sys.messages table

THROW allows choices when handling errors:

Handle specific errors in the local CATCH block

Pass errors to another process

Use THROW:

With parameters to pass a user-defined error

Without parameters to re-raise the original error (must be within a CATCH block)

```
BEGIN TRY
       SELECT 100/0 AS 'Problem';
END TRY
BEGIN CATCH
       PRINT 'Code inside CATCH is beginning'
       PRINT 'MyError: ' + CAST(ERROR_NUMBER()
              AS VARCHAR(255));
       THROW;
END CATCH
```

```
(0 row(s) affected)
Code inside CATCH is beginning
MyError: 8134
Msg 8134, Level 16, State 1, Line 6
Divide by zero error encountered.
```

<u>Transactions</u>:

unstructured error handling
- @@ERROR → gets error no. caused by immediately previous statement.
-RAISERROR vs THROW

A transaction is a group of tasks defined as a unit of work that must succeed or fail together – no partial completion is permitted

```
--Two tasks that make up a unit of work INSERT INTO Sales.SalesOrderHeader...
INSERT INTO Sales.SalesOrderHeader...
INSERT INTO Sales.SalesOrderDetail...
```

Individual data modification statements are automatically treated as standalone transactions
User transactions can be managed with T-SQL commands:
BEGIN/ COMMIT/ROLLBACK TRANSACTION
SQL Server uses locking mechanisms and the transaction log to support transactions

The need for transactions: issues with batches:
Some runtime errors during a batch may result in unacceptable partial success:
Part of the batch succeeds and part fails, leaving behind the results from the part of the batch that succeeded
Simple error handling within a batch cannot repair partial success

```
--Batch without transaction management
BEGIN TRY
 INSERT INTO Sales.SalesOrderHeader... --Insert succeeds
 INSERT INTO Sales.SalesOrderDetail... --Insert fails
END TRY
BEGIN CATCH
 --First row still in Sales.SalesOrderHeader Table
 SELECT ERROR_NUMBER()
 ...
END CATCH;
```

select @@TRANCOUNT; to see if u are in a transaction.

Transactions extend batches:
Transaction commands identify blocks of code that must succeed or fail together and provide points where database engine can roll back, or undo, operations:

```
BEGIN TRY
 BEGIN TRANSACTION
  INSERT INTO Sales.SalesOrderHeader... --Succeeds
  INSERT INTO Sales.SalesOrderDetail... --Fails
 COMMIT TRANSACTION -- If no errors, transaction completes
END TRY
BEGIN CATCH
 --Inserted rows still exist in Sales.SalesOrderHeader SELECT ERROR_NUMBER()
 ROLLBACK TRANSACTION --Any transaction work undone
END CATCH;
```

'ACID' properties of transactions.

BEGIN TRANSACTION marks the starting point of an explicit, user-defined transaction
Transactions last until a COMMIT statement is issued, a ROLLBACK is manually issued, or the connection is broken and the system issues a ROLLBACK
Transactions are local to a connection and cannot span connections
In your T-SQL code: Mark the start of the transaction's work

```
BEGIN TRY
 BEGIN TRANSACTION -- marks beginning of transaction
 INSERT INTO Sales.SalesOrderHeader... --Completed
 INSERT INTO Sales.SalesOrderDetail... --Completed
...
```

→ In a nested tran, a ROLLBACK issued at any nested level will rollback the entire TRAN

COMMIT ensures all of the transaction's modifications are made a permanent part of the database
COMMIT frees resources, such as locks, used by the transaction
In your T-SQL code: If a transaction is successful, commit it

```
BEGIN TRY
 BEGIN TRAN -- marks beginning of transaction
 INSERT INTO Sales.SalesOrderHeader...
 INSERT INTO Sales.SalesOrderDetail...
 COMMIT TRAN -- mark the transaction as complete
END TRY
```

→ In a nested TRAN, u have to COMMIT at each level although only the last COMMIT actually commits the changes of entire TRAN to db.

A ROLLBACK statement undoes all modifications made in the transaction by reverting the data to the state it was in at the beginning of the transaction
ROLLBACK frees resources, such as locks, held by the transaction
Before rolling back, you can test the state of the transaction with the XACT_STATE function

In your T-SQL code: If an error occurs, ROLLBACK to the point of the BEGIN TRANSACTION statement

```
BEGIN CATCH
 SELECT ERROR_NUMBER() --sample error handling
 ROLLBACK TRAN
END CATCH;
```

SQL Server does not automatically roll back transactions when errors occur
To roll back, either use ROLLBACK statements in error-handling logic or enable XACT_ABORT
XACT_ABORT specifies whether SQL Server automatically rolls back the current transaction when a runtime error occurs
When SET XACT_ABORT is ON, the entire transaction is terminated and rolled back on error, unless occurring in TRY block
SET XACT_ABORT OFF is the default setting
Change XACT_ABORT value with the SET command:

```
SET XACT_ABORT ON;
```

Explict TRAN, IMPLICIT TRAN & AUTOCOMMIT
↓ single DML statement
↓ u do not do a 'BEGIN TRAN' but do ROLLBACK & COMMIT

System catalog views:
Built-in views that provide information about the system catalog
Use standard query methods to return metadata
    Column lists, JOIN, WHERE, ORDER BY
Some views are filtered to display only user objects, some views include system objects

```
--Pre-filtered to exclude system objects
SELECT  name, object_id, schema_id, type, type_desc
FROM sys.tables;

--Includes system and user objects
SELECT name, object_id, schema_id, type, type_desc
FROM sys.objects;
```

Information schema views:
Views stored in the INFORMATION_SCHEMA system schema
Return system metadata per ISO standard, used by third-party tools
Maps standard names (catalog, domain) to SQL Server names (database, user-defined data type)

```
SELECT TABLE_CATALOG, TABLE_SCHEMA,
       TABLE_NAME, TABLE_TYPE
FROM   INFORMATION_SCHEMA.TABLES;

SELECT VIEW_CATALOG, VIEW_SCHEMA, VIEW_NAME,     TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME
FROM INFORMATION_SCHEMA.VIEW_COLUMN_USAGE;

SELECT VIEW_CATALOG, VIEW_SCHEMA, VIEW_NAME,     TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME
FROM INFORMATION_SCHEMA.VIEW_COLUMN_USAGE
WHERE COLUMN_NAME = 'BusinessEntityID'
```

System metadata functions:
Return information about settings, values, and objects in SQL Server
Come in a variety of formats
    Some marked with a @@ prefix, sometimes incorrectly referred to as global variables: @@VERSION
    Some marked with a () suffix, similar to arithmetic or string functions: ERROR_NUMBER()
    Some special functions marked with a $ prefix: $PARTITION
Queried with a standard SELECT statement:

```
SELECT @@VERSION AS SQL_Version;
SELECT SERVERPROPERTY('ProductVersion') AS version;
SELECT SERVERPROPERTY('Collation') AS collation;
```

Querying DMVs and functions:
Dynamic management views are queried like standard views:

```
SELECT session_id, login_time, program_name
FROM sys.dm_exec_sessions
WHERE is_user_process = 1;
```

Dynamic management functions are queried as table-valued functions, including parameters:

```
SELECT referencing_schema_name,    referencing_entity_name,    referencing_class_desc
FROM   sys.dm_sql_referencing_entities(
       'Sales.SalesOrderHeader', 'OBJECT');
```

```
GO
```

About dynamic management objects:
The nearly 200 dynamic management views (DMVs) and functions return server state information
DMVs include catalog information as well as administrative status information, such as object dependencies
DMVs are server-scoped (instance-level) or database-scoped
Requires VIEW SERVER STATE or VIEW DATABASE STATE permission to query DMVs
Underlying structures change over time, so avoid writing SELECT * queries against DMVs
Categories include;

Naming pattern	Description
db	Database-related information
exec	Query execution-related information
io	I/O statistics
os	SQL Server Operating System (SQLOS) information
tran	Transaction-related information

Stored procedures:
Use the EXECUTE or EXEC command before the name of the stored procedure
Pass parameters by position or name, separated by commas when applicable

```
--no parameters so lists all database
EXEC sys.sp_databases;

--single parameter of name of table
EXEC sys.sp_help N'Sales.Customer';

--multiple named parameters
EXEC sys.sp_tables
       @table_name = '%',
       @table_owner = N'Sales';
```

Common system stored procedures:
Database engine procedures can provide general metadata
    sp_help, sp_helplanguage
    sp_who, sp_lock
Catalog procedures can be used as an alternative to system catalog views and functions:

Name	Description
sp_databases	Lists databases in an instance of SQL Server
sp_tables	Returns a list of tables or views, except synonyms
sp_columns	Returns column information for the specified objects

Unlike system views, there is no option to select which columns to return

Executing system stored procedures:
System stored procedures:
    Marked with an sp_ prefix
    Stored in a hidden resource database
    Logically appear in the sys schema of every user and system database
Best practices for execution include:
    Always use EXEC or EXECUTE rather than just calling by name
    Include the sys schema name when executing
    Name each parameter and specify its appropriate data type

```
--This example uses EXEC, includes the sys schema name, --and passes the table name as a named Unicode
parameter --to a procedure accepting an NVARCHAR(776)
--input parameter.
EXEC sys.sp_help @objname = N'Sales.Customer';
```

Creating procedures that return rows:
Stored procedures can be wrappers for simple or complex SELECT statements
Procedures may include input and output parameters as well as return values
Use CREATE PROCEDURE statement:

```
CREATE PROCEDURE <schema_name.proc_name>
(<parameter_list)
 AS
SELECT <body of SELECT statement>;
```

Change procedure with ALTER PROCEDURE statement
No need to drop, recreate

Creating procedures that accept parameters:
Input parameters passed to procedure logically behave like local variables within procedure code
Assign name with @prefix, data type in procedure header
Refer to parameter in body of procedure

```
CREATE PROCEDURE Production.ProdsByProductLine
(@numrows AS int, @ProdLine AS nchar)
 AS
SELECT TOP(@numrows) ProductID,
       Name, ListPrice
FROM   Production.Product
WHERE  ProductLine = @ProdLine;

--Retrieve top 50 products with product line = M
EXEC Production.ProdsByProductLine 50, 'M'
```

NOTE: `SET NOCOUNT ON`
Nocount on so that after each statement we don't return number of rows affected back to the client. Reduces the data sent back to the client.

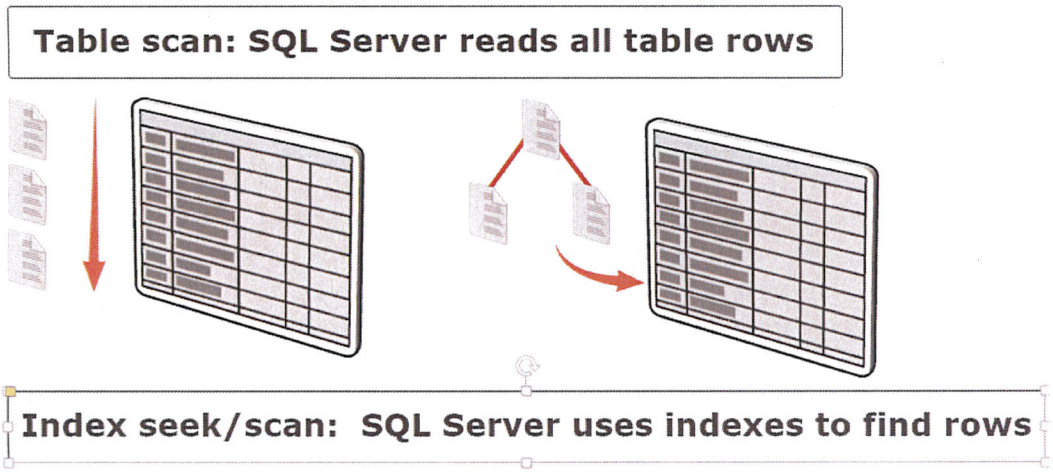
Writing well-performing queries:
Only retrieve what you need
- In the SELECT clause, only use needed columns – avoid *
- Use a WHERE clause, filter to return only needed rows

Improve search performance of WHERE clause
- Avoid expressions that manipulate columns in the predicate

Minimize use of temporary tables or table variables
- Use windowing functions or other set-based operations when possible

Avoid cursors and other iterative approaches
Work with your DBA to arrange good indexes to support filters, joins, and ordering
Learn how to address tasks with different query approaches to compare performance

Indexing in SQL Server:
SQL Server accesses data by using indexes or by scanning all rows in a table
Indexes also supports ordering operations such as grouping, joining, and ORDER BY clauses



SQL Server indexes: performance considerations:
Check query execution plans to see if indexes are present and being used as expected
For query writers who are not DBAs or database developers, the ability to recognize problems with indexes, such as the use of table scans when you expect an index to be used, can be very helpful in tuning an application

Note: while indexes improve the searching of data, there is a slight hit while inserting data.

Distribution statistics:
Distribution statistics describe the distribution and the uniqueness, or selectivity, of data
Statistics, by default, are created and updated automatically
Statistics are used by the query optimizer to estimate the selectivity of data, including the size of the results
Large variances between estimated and actual values might indicate a problem with the estimates, which may be addressed through updating statistics

Avoiding cursors:
Cursors contradict the relational model, which operates on sets
Cursors typically require more code than set-based approach
Cursors typically incur more overhead during execution than a comparable set-based operation
Alternatives to cursors:
- Windowing functions
- Aggregate functions

Appropriate uses for cursors:
- Generating dynamic SQL code
- Performing administrative tasks

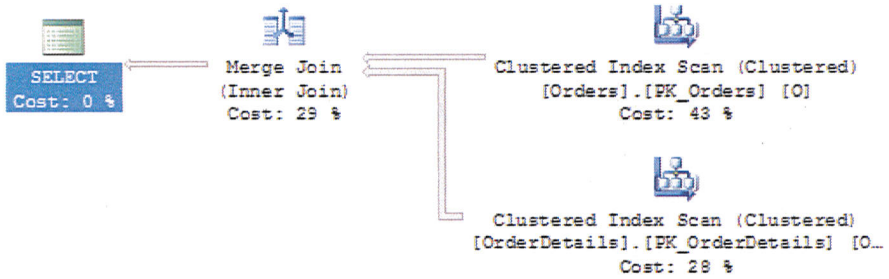
What is an execution plan?
Review of the process of executing a query:
- Parse, resolve, optimize, execute

An execution plan includes information on which tables to access, which indexes, what joins to perform
- If statistics exist for a relevant column or index, then the optimizer will use them in its calculations

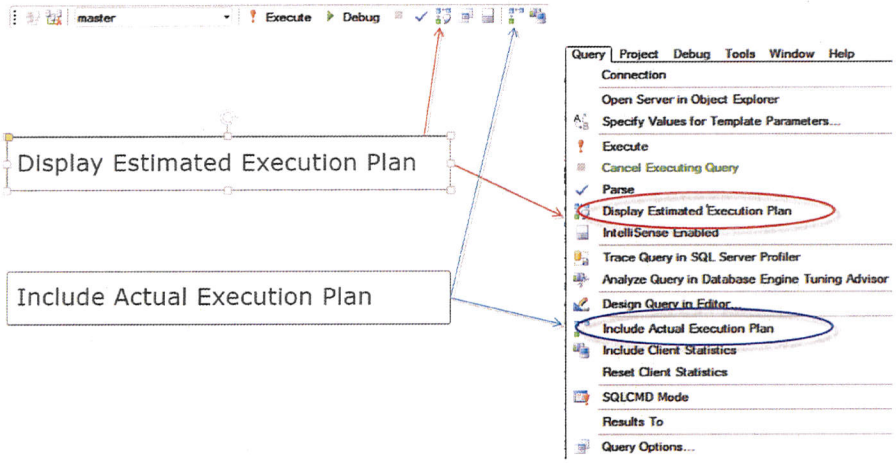
SQL Server tools provide access to execution plans to show how a query was executed or how it would be executed
- Plans available in text format (deprecated), XML format, and graphical renderings of XML

Plan viewer accessible in results pane of SSMS

Actual and estimated execution plans:
Execution plans graphically represent the methods that SQL Server uses to execute the statements in a T-SQL query
SSMS provides access to two forms of execution plans:
Estimated execution plans do not execute the query. Instead, they display the plan that SQL Server would likely use if the query were run.
Actual execution plans are returned the next time the query is executed. They display the plan that was actually used by SQL Server



Viewing graphical execution plans:
Enable execution plan viewers in SSMS



Interpreting the execution plan:

Read the plan right to left, top to bottom

Hover the mouse pointer over an item to see additional information

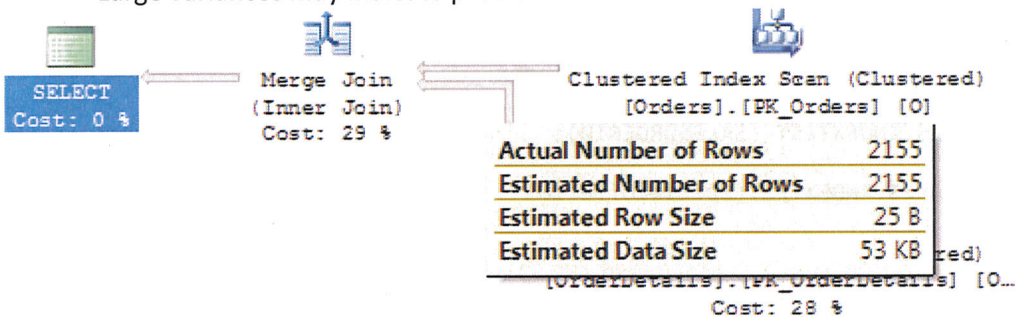
Percentages indicate cost of operator relative to total query

Thickness of lines between operators indicates relative number of rows passing through

For issues, look for thick lines leading into high-cost operators

In an actual execution plan, note any differences between estimated and actual values

Large variances may indicate problems with estimates



Displaying Query Statistics:

SQL Server provides detailed runtime information about the execution of a query

STATISTICS TIME will show time spent parsing and compiling a query

```
SET STATISTICS TIME ON;
```

STATISTICS IO will show amount of disk activity generated by a query

```
SET STATISTICS IO ON;
```

Query performance:

```
SET STATISTICS TIME ON;

SET STATISTICS IO ON;

--WITHOUT ANY INDEX, A TABLE IS CALLED A HEAP. IT DOES NOT HAVE A CLUSTERED INDEX ON THE TABLE
--INSQ SERVER 2014, A TABLE CAN BE OF 3 TYPES WHILE IN SQL SERVER 2012 IT CAN BE OF 2 TYPES(HEAP OR
CLUSTERED INDEX)
--SELECT  *
--INTO SALES.INDEXTEST
--FROM SALES.SalesOrderDetail;

EXEC sp_helpindex 'SALES.INDEXTEST';

--lets say we want this: by order we want the sum of unitprice*orderqty, groupby sale orderid for a
certain specialoffer
--include the actual execution plan and run the query
SELECT
       SalesOrderID
       ,SUM(UnitPrice*OrderQty) AS TOTAL
FROM SALES.INDEXTEST
WHERE SpecialOfferID=2
GROUP BY SalesOrderID
ORDER BY TOTAL;
```

```
--1499 pages,table scan,hash aggr. :(

--has match aggregate is the grouping ..for grouping either sql server can do a sort or can do a hash
operation...typically sorting should be used
--for agregates. so when u see a hash match in a execution plan, it means u don't have right indexing in
place.
--the execution plan also contains a hint that u probably need to create a nonclustered index. You can
even generate the indexing code by right clicking the
--execution plan...now run the query again after switching on the stats 'SET STATISTICS IO ON;' because
time stats depend on the locks that happened during the
--execution and how busy the server is. Now messages tell me per table how much work we did..logical
reads tell me how many pages i had to read to get the data
--physical reads suggest for how many reads did i have to go the disk to fetch(vs in-memory reads if the
data is already in memory)

--clustered index means u r sorting the table and adding a balanced tree structure on top of the table
CREATE CLUSTERED INDEX CluIdx ON SALES.INDEXTEST (SpecialOfferID);

--and then again run the query. it actually does more page reads...as the table is very big and
restructring it actually caused it to use more space
--1525 pages,clustered index seek,hash aggr. :( but for query for specialofferid=2 has a reduction in
the number of page reads and is down to 49
--but we still have a hash aggr.
--if a table already has a clustered index, we have to create non-clustered indexes. take all the cols
in the non-clust. index, create a new table with those cols and shovel
--all the data for those cols into the new table and athen add a pointer back to original table
--sometime the query analyzer does not use the index that it should have used. then (generally only for
testing purposes), you can force the query to use a particular index
DROP INDEX CluIdx ON SALES.INDEXTEST;
CREATE CLUSTERED INDEX CluIdx ON SALES.INDEXTEST (SALESORDERID);

CREATE NONCLUSTERED INDEX NCIdx ON SALES.INDEXTEST(SpecialOfferID)
SELECT
       SalesOrderID
       ,SUM(UnitPrice*OrderQty) AS TOTAL
FROM SALES.INDEXTEST WITH(INDEX(NCIdx))
WHERE SpecialOfferID=2
GROUP BY SalesOrderID
ORDER BY TOTAL;
--10517 pages,non clustered index seek + table lookup,hash aggr. :((( but for query for specialofferid=2
has a reduction in the number of page reads and is down to 49
--to avoid lookups, include additional cols in the nonclustered index that are being used in query
DROP INDEX CluIdx ON SALES.INDEXTEST;
DROP INDEX NCIdx ON SALES.INDEXTEST;
CREATE CLUSTERED INDEX CluIdx ON SALES.INDEXTEST (SALESORDERID);
CREATE NONCLUSTERED INDEX NCIdx ON SALES.INDEXTEST(SpecialOfferID) INCLUDE
(SalesOrderID,UnitPrice,OrderQty)
SELECT
       SalesOrderID
       ,SUM(UnitPrice*OrderQty) AS TOTAL
FROM SALES.INDEXTEST WITH(INDEX(NCIdx))
WHERE SpecialOfferID=2
GROUP BY SalesOrderID
ORDER BY TOTAL;
--17 pages,non clustered index seek, sort for grouping .
--now since i group on saleseorderid, if i include that in the sort order, it would become faster
DROP INDEX CluIdx ON SALES.INDEXTEST;
DROP INDEX NCIdx ON SALES.INDEXTEST;
CREATE CLUSTERED INDEX CluIdx ON SALES.INDEXTEST (SALESORDERID);
CREATE NONCLUSTERED INDEX NCIdx ON SALES.INDEXTEST(SpecialOfferID,SalesOrderID) INCLUDE
(UnitPrice,OrderQty)
SELECT
       SalesOrderID
       ,SUM(UnitPrice*OrderQty) AS TOTAL
FROM SALES.INDEXTEST WITH(INDEX(NCIdx))
WHERE SpecialOfferID=2
GROUP BY SalesOrderID
ORDER BY TOTAL;
--17 pages,non clustered index seek,no sort for grouping ....the sort operation has now disappeared
```

```sql
--THESE QUERIES ARE SAME
SELECT
Color
FROM Production.Product
WHERE Color IS NOT NULL
GROUP BY Color;

SELECT
DISTINCT Color
FROM Production.Product
WHERE Color IS NOT NULL


--THESE QUERIES ARE SAME ARE NOT SAME!!
SELECT
COUNT(Color)
FROM Production.Product
WHERE Color IS NOT NULL
GROUP BY Color;

SELECT
COUNT(DISTINCT Color)
FROM Production.Product
WHERE Color IS NOT NULL
```

→ Table variables (@) are not performant as statistics about them are maintained but temp variables do maintain that & thus give good performance as a good query plan can be selected. But table vars (@) are also stored in tempdb!.

→ Using same CTE multiple times in the same query might have performance repercussions. Think about storing CTE result in table var & temp table.

→ equi join & sorted inputs both sides = Merge JOIN. equi join & none of the inputs sorted (indexed) = Hash JOIN. Nested loop JOIN for equi or Non-equi joins for smaller rowsets??

→ Logical order scans (same order as embedded in index).

→ Physical order scans (Same order as of rows inserted into the table). Called ALLOCATION ORDER SCAN

→ (Called INDEX ORDER SCAN)

→ optimizer hints (table, query & join hints) are directives, not hints.

- Turn the setting to use double quotes as identifier delimiter.
Exec sp_helptext 'dbo.sales' (or) select OBJECT_DEFINITION(OBJECT_ID('dbo.sales'));
- Differences between 'ORDER BY' used for filtering purposes or windowing purpose Vs 'ORDER BY' for presentation purpose
- Running totals
→ @mytbl is table variable. #mytbl is local temp. table
##mytbl is global temp table. @@ system funcs
→ PK & FK inner join gains performance if we also declare a non-clustered index on the foreign key.
→ CHECK constraint evaluates to 'TRUE' with NULL values (usually NULL causes expressions to evaluate to 'NOT TRUE')
→ 'UDF' Vs 'IF' → How does DEFAULT constraint behave when one explicitly specifies a NULL value Vs when no value is specified.
→ To turn values into a table expression, use @ SELECT without a FROM clause (b) VALUES table value constructor.
→ Even with 'IDENTITY INSERT ON', you can only INSERT identity values, not update them. For 'UPDATE', u need sequence objects
→ Use table hint 'HOLDLOCK' or 'SERIALIZABLE' to prevent issues arising from checking for value before insertion.
→ [ ] should be used instead of double quotes, even though double quotes are ANSI standard for identifiers. Reason is that SQL server has an option which can turn off the use of double quotes for identifiers.
→ BEGIN-END block not required in Stored PROCS
→ To SET the variable to. scalar value from a subquery, the subquery should be in round brackets. Can also be done using '=' inside query (@var = age)
→ UDF can't do DDL & DML operations
→ Scan Vs seek. 8KB page (table, indexed & indexed views) . 8 pages = 1 extent. Heap vs Balanced trees (clustered index). RID Vs KEY lookup for non-clustered indexes. Non cluster index on a clustered table (heap / balanced tree) uses KEY lookup.
→ How to allow multiple NULLs but unique NOT-NULL values in a col.
→ You can accomodate more rows in each page of the non-clustered index on the clustered-index table (balanced tree) than on the heap because clustering key (key lookup) is shorter than RID.
→ There can be only one clustered index as it is the table itself organized as balanced tree. (So what is the type of index created by primary key & unique keys?)
→ Columns used not only in WHERE clause but also JOIN, GROUPBY & ORDERBY are candidates for index creation