LEAD	Returns an expression from a later row that is a defined offset from the current row. Returns NULL if no row at specified position.
FIRST_VALUE	Returns the first value in the current window frame. Requires window ordering to be meaningful.
LAST_VALUE	Returns the last value in the current window frame. Requires window ordering to be meaningful.

```
--what is the current balance of the account per a given row..this query did not had good performance
before sql2012
select * from Transactions
where AccountID = 29825

select *
,SUM(Amount) over (
                    partition by AccountID
                    order by TransactionDate, TransactionID
                    rows between unbounded preceding and current row
 ) as FinalBalance
 from Transactions
where AccountID = 29825
order by AccountID , TransactionDate, TransactionID

--i want to 1 step  backwards in the set
select *
,lag(Amount,1,0) over (partition by AccountID order by TransactionDate, TransactionID)
,SUM(Amount) over (
                    partition by AccountID
                    order by TransactionDate, TransactionID
                    rows between unbounded preceding and current row
 ) as FinalBalance
 from Transactions
where AccountID = 29825
order by AccountID , TransactionDate, TransactionID
```

What is pivoting?

Pivoting data is rotating data from a rows-based orientation to a columns-based orientation

Distinct values from a single column are projected across as headings for other columns - may include aggregation

Category	Qty	Orderyear
Dairy Products	12	2006
Grains/Cereals	10	2006
Dairy Products	5	2006
Produce	9	2006
Produce	40	2006
Seafood	10	2006
Produce	35	2006
Condiments	15	2006
Grains/Cereals	6	2006
Grains/Cereals	15	2006
Condiments	20	2006
Confections	40	2006
Dairy Products	25	2006
Dairy Products	40	2006
Dairy Products	20	2006

Pivoted data

Category	2006	2007	2008
Beverages	1842	3996	3694
Condiments	962	2895	1441
Confections	1357	4137	2412
Dairy Products	2086	4374	2689
Grains/Cereals	549	2636	1377
Meat/Poultry	950	2189	1060
Produce	549	1583	858
Seafood	1286	3679	2716

'PIVOT' is T-SQL specific; Standard SQL has as well.

We can pivot using

Pivoting includes three phases:

1. Grouping determines which element gets a row in the result set
2. Spreading provides the distinct values to be pivoted across
3. Aggregation performs an aggregation function (such as SUM)

These 3 phases remain same for both T-SQL specific PIVOT (or) Standard SQL PIVOT.

```
SELECT Category, [2006],[2007],[2008]
FROM ( SELECT Category, Qty, Orderyear
        FROM Sales.CategoryQtyYear) AS D
PIVOT(SUM(QTY) FOR orderyear
             IN([2006],[2007],[2008])
             ) AS pvt;

SELECT VendorID, [250] AS Emp1, [251] AS Emp2, [256] AS Emp3, [257] AS Emp4, [260] AS Emp5
FROM
```

```
(SELECT PurchaseOrderID, EmployeeID, VendorID
FROM Purchasing.PurchaseOrderHeader) p
PIVOT
(
COUNT (PurchaseOrderID)
FOR EmployeeID IN
( [250], [251], [256], [257], [260] )
) AS pvt
ORDER BY pvt.VendorID;
```

```
VendorID  Emp1 Emp2 Emp3  Emp4 Emp5
--------  ----- ----- -----  ----- -----
1492      2    5    4    4    4
1494      2    5    4    5    4
1496      2    4    4    5    5
```

Note that u generally don't use pivot in sql as it mainly in used in excel or reporting services. So pivoting is generally done up in the application layer.

```
CREATE TABLE pvt (VendorID int, Emp1 int, Emp2 int,
    Emp3 int, Emp4 int, Emp5 int);
GO
INSERT INTO pvt VALUES (1,4,3,5,4,4);
INSERT INTO pvt VALUES (2,4,1,5,5,5);
INSERT INTO pvt VALUES (3,4,3,5,4,4);
GO

SELECT VendorID, Employee, Orders
FROM
   (SELECT VendorID, Emp1, Emp2, Emp3, Emp4, Emp5
   FROM pvt) p
UNPIVOT
   (Orders FOR Employee IN
      (Emp1, Emp2, Emp3, Emp4, Emp5)
)AS unpvt;
GO
```

```
VendorID    Employee    Orders
1            Emp1              4
1            Emp2              3
```

Writing queries with grouping sets:

GROUPING SETS subclause builds on T-SQL GROUP BY clause

Allows multiple groupings to be defined in same query

Alternative to use of UNION ALL to combine multiple outputs (each with different GROUP BY) into one result set

```
SELECT <column list with aggregate(s)>
FROM <source>
GROUP BY
GROUPING SETS(
       (<column_name>),--one or more columns
       (<column_name>),--one or more columns
       () -- empty parentheses if aggregating all rows
             );

SELECT TerritoryID, CustomerID, SUM(TotalDue) AS TotalAmountDue
FROM Sales.SalesOrderHeader
GROUP BY
GROUPING SETS((TerritoryID),(CustomerID),());
```

TerritoryID	CustomerID	TotalAmountDue
NULL	30116	211671.2674
NULL	30117	919801.8188
NULL	30118	313671.5352
NULL	NULL	123216786.1159

3	NULL	8913299.2473
6	NULL	18398929.188
9	NULL	11814376.0952
1	NULL	18061660.371
7	NULL	8119749.346

```
--lets say we want to get total sales by customer and territory
SELECT
C.CustomerID
,C.TerritoryID
,SUM(SOD.LineTotal) AS TOTALSALES
FROM Sales.Customer AS C
INNER JOIN Sales.SalesOrderHeader AS SOH ON SOH.CustomerID = C.CustomerID
INNER JOIN Sales.SalesOrderDetail AS SOD ON SOD.SalesOrderID = SOH.SalesOrderID
GROUP BY
       C.CustomerID
       ,C.TerritoryID

--WHAT IS THE TOTAL SALES IN TERRITORY INDEPENDENT OF THE CUSTOMER
SELECT
C.CustomerID
,C.TerritoryID
,SUM(SOD.LineTotal) AS TOTALSALES
FROM Sales.Customer AS C
INNER JOIN Sales.SalesOrderHeader AS SOH ON SOH.CustomerID = C.CustomerID
INNER JOIN Sales.SalesOrderDetail AS SOD ON SOD.SalesOrderID = SOH.SalesOrderID
GROUP BY
       C.CustomerID
       ,C.TerritoryID
UNION ALL
SELECT
       NULL
       ,NULL
,SUM(SOD.LineTotal) AS TOTALSALES
FROM Sales.Customer AS C
INNER JOIN Sales.SalesOrderHeader AS SOH ON SOH.CustomerID = C.CustomerID
INNER JOIN Sales.SalesOrderDetail AS SOD ON SOD.SalesOrderID = SOH.SalesOrderID
ORDER BY CustomerID

--now we are writting a lot of code. instead use grouping sets...EASY AND WE CAN ADD MORE GROUPING SETS
EASILY
SELECT
C.CustomerID
,C.TerritoryID
,SUM(SOD.LineTotal) AS TOTALSALES
FROM Sales.Customer AS C
INNER JOIN Sales.SalesOrderHeader AS SOH ON SOH.CustomerID = C.CustomerID
INNER JOIN Sales.SalesOrderDetail AS SOD ON SOD.SalesOrderID = SOH.SalesOrderID
GROUP BY GROUPING SETS(
       (C.CustomerID,C.TerritoryID)
       ,(C.TerritoryID)
       ,()
       )
ORDER BY CustomerID, TerritoryID
```

CUBE and ROLLUP:

CUBE provides shortcut for defining grouping sets given a list of columns

All possible combinations of grouping sets are created

```
SELECT TerritoryID, CustomerID, SUM(TotalDue) AS TotalAmountDue
FROM Sales.SalesOrderHeader
GROUP BY CUBE(TerritoryID, CustomerID)
ORDER BY TerritoryID, CustomerID;
```

ROLLUP provides shortcut for defining grouping sets, creates combinations assuming input columns form a hierarchy

```
SELECT TerritoryID, CustomerID, SUM(TotalDue) AS TotalAmountDue
FROM Sales.SalesOrderHeader
GROUP BY ROLLUP(TerritoryID, CustomerID)
```

```
ORDER BY TerritoryID, CustomerID;
```

TRUNCATE
MERGE

DML statements (INSERT, UPDATE, and DELETE):

Using INSERT to add data:

The INSERT...VALUES statement inserts a single row by default

```
INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode, ModifiedDate)
VALUES (N'Square Yards', N'Y2', GETDATE());
GO
```

It is standard → Values can be used as replacement for 'Select' of UNION ALL Combinations

Table and row constructors add multi-row capability to INSERT...VALUES

```
INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode, ModifiedDate)
VALUES
      (N'Square Feet', N'F2', GETDATE()),
      (N'Square Inches', N'I2', GETDATE());
```

Values can also be used as a table value constructor to construct a derived table. We can use it as a replacement for UNION ALL, in these cases

Using INSERT with DEFAULT constraint:

DEFAULT constraints are used to assign a value to a column when none is specified in the INSERT statement

Defaults are defined in CREATE or ALTER TABLE statement

INSERT can omit columns which have defaults defined

- Applies to IDENTITY and NULL too

VALUES clause can use DEFAULT keyword

- If no default constraint assigned, NULL inserted

If you added a field called Country with a DEFAULT constraint of 'USA' you could INSERT data using the following command

```
INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode, ModifiedDate, Country)
VALUES (N'Square Miles', N'M2', GETDATE(), DEFAULT);
```

Using INSERT with SELECT and EXEC:

INSERT...SELECT is used to insert the result set of a query into an existing table

```
INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode, ModifiedDate)
SELECT Name, UnitMeasureCode, ModifiedDate
FROM Sales.TempUnitTable
WHERE ModifiedDate < '20080101';
```

There is a BULK INSERT statement for reading data from files.

INSERT...EXEC is used to insert the result of a stored procedure or dynamic SQL expression into an existing table

```
INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode, ModifiedDate)
EXEC Production.Temp_UOM
      @numrows = 5, @catid=1;
```

Using SELECT INTO: → Not Standard.

SELECT...INTO is similar to INSERT...SELECT but SELECT...INTO creates a new table each time the statement is executed

Copies column names, data types, and nullability of identity property.

Does not copy constraints or indexes or triggers or permissions.

```
SELECT Name, UnitMeasureCode, ModifiedDate
INTO Production.TempUOMTable
FROM Production.UnitMeasure
WHERE orderdate < '20080101';
```

Heap is a table without clustered index.
'SELECT INTO' creates a Heap.

Using IDENTITY:

IDENTITY property of a column generates sequential numbers automatically for insertion into a table

Can specify optional set seed and increment values

Only one column in a table may have IDENTITY property defined

IDENTITY column is omitted in INSERT statements

Functions provided to retrieve last generated value

obtain the generated identity value using @@identity & SCOPE_IDENTITY

Creates a table with using the IDENTITY property with a starting number of 100 and incremented by 10 as each row is added

```
CREATE TABLE Production.IdentityProducts(
productid int IDENTITY(100,10) NOT NULL,
productname nvarchar(40) NOT NULL,
categoryid int NOT NULL,
unitprice money NOT NULL)
```

Select productid, $identity
from Production.IdentityProducts

@@identity returns last identity generated in session regardless of scope. So most of times use SCOPE_IDENTITY.

IDENT_CURRENT('Prod.Prods') return last identity value in a table.

Using SEQUENCES:

Sequence objects new in SQL Server 2012
Independent objects in database
More flexible than the IDENTITY property
Can be used as default value for a column
Manage with CREATE/ALTER/DROP statements
Retrieve value with the NEXT VALUE FOR clause

```
-- Define a sequence
CREATE SEQUENCE dbo.InvoiceSeq AS INT START WITH 5 INCREMENT BY 5;

-- Retrieve next available value from sequence
SELECT NEXT VALUE FOR dbo.InvoiceSeq;
```

Using UPDATE to modify data:

Updates all rows in a table or view
    Set can be filtered with a WHERE clause
    Set can be defined with a JOIN clause
Only columns specified in the SET clause are modified
Updates the ModifiedDate using a the GETDATE function for the record that has 'M2' in the UnitMeasureCode

```
UPDATE Production.UnitMeasure
   SET ModifiedDate = (GETDATE())
   WHERE UnitMeasureCode = 'M2'
```

If no WHERE clause is specified, all records in the Production.UnitMeasure will be updated

Using MERGE to modify data:

MERGE modifies data based on one of the following conditions
    When the source matches the target
    When the source has no match in the target
    When the target has no match in the sourace

```
MERGE INTO schema_name.table_name AS TargetTbl
    USING (SELECT <select_list>) AS SourceTbl
    ON (TargetTbl.col1 = SourceTbl.col1)
    WHEN MATCHED THEN
        UPDATE SET col2 = SourceTbl.col2
WHEN NOT MATCHED THEN
    INSERT (<column_list>)
    VALUES (<value_list>);
```

```
--SELECT SOME DATA INTO A NEW TABLE
--SELECT
--ProductID
--,Name
--INTO PRODUCTS
--FROM Production.Product;

--RETURN AS XML
SELECT
*
FROM PRODUCTS
WHERE NAME LIKE 'A%'
FOR XML PATH

--RETURN AS XML
SELECT
*
FROM PRODUCTS
WHERE NAME LIKE 'A%'
FOR XML PATH('PRODUCT'), ROOT('PRODUCTS')


--RETURN AS XML...CHANGE NAMES
SELECT
```

→ Standard

IDENT_CURRENT('dbo.tbl') returns last identity values produced for a table. So in case of an error, this value might not be found stored in the table.

↓ Still not correct

Identity property does not enforce uniqueness.

↓ This is a function.

→ not standard

You might want to have a sequence for every table needing identity. Sequence obj also does not guarantee gaps due to errors.

again, as because of 'all-at-once' concept, you can swap values without needing a 3rd variable:

```
update [dbo.tbl]
set col1 = col2, col2 = col1
```

- Merge is Standard but T-SQL adds some non-Standard extensions.

- As noted earlier, we can run DML against VIEWS. Now view is a table expression. ∴ In general, DML can be run against all table expressions (CTE, UDF, derived tables, view).

- MERGE with WHEN MATCHED clause is a standard alternative to UPDATE based on a JOIN.

```
PRODUCTID AS "@id"--WE HAVE TO USE DOUBLE INVERTED COMMAS AS @ HAS SPECIAL MEANING IN SQL...AND @ IS IS
USED TO SPECIFY THAT
--WE WANT THIS TO BE A ATTRIBUTE RATHER THAN AN ELEMENT
,NAME AS "@name"
FROM PRODUCTS
WHERE NAME LIKE 'A%'
FOR XML PATH('PRODUCT'), ROOT('PRODUCTS')

--UPDATES USING XML...KEEP THE CASING TO LOWER CASE WHEN USING XML
DECLARE @Xml XML = N'
<PRODUCTS>
  <PRODUCT id="1" name="Adjustable Race!" />
  <PRODUCT id="879" name="All-Purpose Bike Stand!" />
  <PRODUCT id="712" name="AWC Logo Cap!" />
  <PRODUCT name="Ramneek!" />
</PRODUCTS>';

SELECT
XT.XC.value('@id','INT') AS PRODUCTID
,XT.XC.value('@name','NVARCHAR(1000)') AS NAME
FROM @Xml.nodes('/PRODUCTS/PRODUCT') AS XT(XC);

WITH SRC AS (
SELECT
XT.XC.value('@id','INT') AS PRODUCTID
,XT.XC.value('@name','NVARCHAR(1000)') AS NAME
FROM @Xml.nodes('/PRODUCTS/PRODUCT') AS XT(XC)
)

--NOW MERGE THIS DATA INTO THE PRODUCTS TABLE
MERGE INTO PRODUCTS AS DEST
       USING SRC ON SRC.PRODUCTID = DEST.PRODUCTID
WHEN NOT MATCHED THEN
       INSERT (NAME)  VALUES(SRC.NAME)
WHEN MATCHED THEN
       UPDATE SET  NAME = SRC.NAME;

SELECT
PRODUCTID AS "@id"--WE HAVE TO USE DOUBLE INVERTED COMMAS AS @ HAS SPECIAL MEANING IN SQL...AND @ IS IS
USED TO SPECIFY THAT
--WE WANT THIS TO BE A ATTRIBUTE RATHER THAN AN ELEMENT
,NAME AS "@name"
FROM PRODUCTS
WHERE NAME LIKE 'A%' OR NAME LIKE 'RaM%'
FOR XML PATH('PRODUCT'), ROOT('PRODUCTS')

--aLSO MERGE FOR DELETES.
DECLARE @Xml XML = N'
<PRODUCTS>
  <PRODUCT id="1" name="Adjustable Race!" />
  <PRODUCT id="879" name="All-Purpose Bike Stand!" />
  <PRODUCT id="712" delete="true"  name="AWC Logo Cap!" />
  <PRODUCT id="1000" name="Ramneek!" />
</PRODUCTS>';

WITH SRC AS (
SELECT
XT.XC.value('@id','INT') AS PRODUCTID
,XT.XC.value('@name','NVARCHAR(1000)') AS NAME
,ISNULL(XT.XC.value('@delete','BIT'),0) AS DODELETE
FROM @Xml.nodes('/PRODUCTS/PRODUCT') AS XT(XC)
)

--NOW MERGE THIS DATA INTO THE PRODUCTS TABLE
MERGE INTO PRODUCTS AS DEST
       USING SRC ON SRC.PRODUCTID = DEST.PRODUCTID
WHEN NOT MATCHED THEN
       INSERT (NAME)  VALUES(SRC.NAME)
WHEN MATCHED  and src.DODELETE=0  THEN
       UPDATE SET  NAME = SRC.NAME
```

XML is Case-Sensitive

```
WHEN MATCHED and src.DODELETE=1 THEN
        delete;

SELECT
PRODUCTID AS "@id"--WE HAVE TO USE DOUBLE INVERTED COMMAS AS @ HAS SPECIAL MEANING IN SQL...AND @ IS IS
USED TO SPECIFY THAT
--WE WANT THIS TO BE A ATTRIBUTE RATHER THAN AN ELEMENT
,NAME AS "@name"
FROM PRODUCTS
WHERE NAME LIKE 'A%' OR NAME LIKE 'RaM%'
FOR XML PATH('PRODUCT'), ROOT('PRODUCTS')
```

you can use NOCOUNT ON if you want that the no. of rows affected should not be reported.

Using DELETE to remove data:

DELETE operates on a set

Set may be filtered with a WHERE clause

Deletion of each row is logged in database's transaction log

DELETE may be rolled back if statement issued within a user-defined transaction or if an error is encountered

```
DELETE FROM Production.UnitMeasure
WHERE UnitMeasureCode = 'Y2';
```

If no WHERE clause is specified, all records in the Production.UnitMeasure will be deleted

Using TRUNCATE TABLE to remove all data:

TRUNCATE TABLE clears the entire table

Storage is physically deallocated, rows not individually removed

Minimally logged

Can be rolled back if TRUNCATE issued within a transaction

TRUNCATE TABLE will fail if the table is referenced by a foreign key constraint in another table

Truncate resets the identity seed.

(even when the table is empty!)

```
TRUNCATE TABLE Production.UnitMeasure
```

Note: if you have a lot of rows, then doing a 'delete from' without a where clause will take a long time. Then use the 'truncate' option

primary key > for. key > unique. key > check & default constraints

PRIMARY KEY constraint:

A ***PRIMARY KEY*** is an important concept of designing a database table as it provides an attribute or set of attributes used to uniquely identify each row in the table

A table can only have one primary key which is created using a primary key constraint and enforced by creating a unique index on the primary key columns. So indexes are not only used to speed up queries by avoiding full table scans but they are also help enforce constraints.

A column that participates in the primary key constraint cannot accept null values

To add a PRIMARY KEY constraint to an existing table use the following command

```
ALTER TABLE Production.TransactionHistoryArchive
ADD CONSTRAINT PK_TransactionHistoryArchive_TransactionID
PRIMARY KEY CLUSTERED (TransactionID);
```

→ do we have to specify this?

FOREIGN KEY constraint:

A ***FOREIGN KEY*** is a column or combination of columns that are used to establish a link between data in two tables. The columns used to create the primary key in one table are also used to create the foreign key constraint and can be used to reference data in the same table or in another table

A foreign key does not have to reference a primary key, it can be defined to reference a unique constraint in either the same table or in another table NULL is allowed in for. key cols. unless restricted otherwise

To add a FOREIGN KEY constraint to an existing table use the following command

when primary or unique key of one table is used to create a relationship or link data in another table (or even in same table)

```
ALTER TABLE Sales.SalesOrderHeaderSalesReason
ADD CONSTRAINT FK_SalesReason
FOREIGN KEY (SalesReasonID)
REFERENCES Sales.SalesReason (SalesReasonID)
ON DELETE CASCADE
ON UPDATE CASCADE ;
```

extra options

instead of CASCADE we could use setnull, set Default & Relationship

UNIQUE constraints:

A ***UNIQUE constraint*** is created to ensure no duplicate values are entered in specific columns that do not participate in a primary key

Creating a UNIQUE constraint automatically creates a corresponding unique index

Both PK & Unique constraint create unique index

A table can have one primary key but multiple unique key constraints.

To create a UNIQUE constraint while creating a table use the following command

```
CREATE TABLE Production.TransactionHistoryArchive4
(TransactionID int NOT NULL,
CONSTRAINT AK_TransactionID UNIQUE(TransactionID) );
```

unique key can be placed on cols that allow NULL. But then that col can have only one NULL value ??

CHECK constraints:

A ***CHECK constraint*** is created in a table to specify the data values that are acceptable in one or more columns

To create a CHECK constraint after creating a table use the following command

```
ALTER TABLE DBO.NewTable
ADD ZipCode int NULL
CONSTRAINT CHK_ZipCode
CHECK (ZipCode LIKE '[0-9][0-9][0-9][0-9][0-9]');
```

Check & for. key constraints can be added with an option called 'WITH NO CHECK' that bypasses constraint for existing data.

DEFAULT constraints:

A ***DEFAUT constraint*** is a special case of a column default that is applied when an INSERT statement doesn't explicitly assign a particular value. In other words, the column default is what the column will get as a value by default

To create a DEFAULT constraint on an existing table use the following command

```
ALTER TABLE Sales.CountryRegionCurrency
ADD CONSTRAINT Default_Country
DEFAULT 'USA' FOR CountryRegionCode
```

other is DDL trigger

DML triggers:

A ***DML trigger*** is a special type of stored procedure that automatically fires when any valid DML event takes place regardless of whether or not any rows are affected

AFTER and INSTEAD OF triggers

DML triggers can include complex T-SQL statements used to enforce business rules and provide data integrity when an INSERT, UPDATE, or DELETE command is executed

INSTEAD OF can also be created on Views

The following trigger will prints a message to the client when anyone tries to add or change data in the Customer table

```
CREATE TRIGGER reminder1 ON Sales.Customer
AFTER INSERT, UPDATE
AS RAISERROR ('Notify Customer Relations', 16, 10);
```

Trigger considered part of the transaction that includes the event that caused the trigger to fire. (Auto Transaction)

Note: we might want to use the new throw statement instead of the raiseerror used here

Note: if the rules are too complex to be enforced by constraints, then use trigger

Note: other use could be to update data in some denormalized tables. But if performance is hampered, then you might want to use some sort of background job instead of a trigger to update the denormalized tables.

Trigger fired even if no rows affected by DML statements

OUTPUT clause:

The ***OUTPUT*** clause is used to return information from, or expressions based on, each row affected by an INSERT, UPDATE, DELETE, or MERGE statement. These results can be returned to the processing application for use in such things as confirmation messages or archiving

The following example deletes all rows in the ShoppingCartItem table. The clause OUTPUT **deleted.*** specifies that all columns in the deleted rows, be returned to the calling application which in this case was the Query Editor

```
DELETE FROM Sales.ShoppingCartItem OUTPUT DELETED.* WHERE ShoppingCartID = 20621;
--Verify the rows in the table matching the WHERE clause have been deleted.
SELECT COUNT(*) AS [Rows in Table]
FROM Sales.ShoppingCartItem
WHERE ShoppingCartID = 20621;
```

```
DECLARE @tmp TABLE (ProductID INT PRIMARY KEY);

UPDATE Production.Product SET
    Name = UPPER(Name)
OUTPUT INSERTED.ProductID INTO @tmp
WHERE ListPrice > 10;

SELECT * FROM @tmp;
```

- u can also store 'OUTPUT' clause output in a table another way to get 'before update' info ??
- There is also

T-SQL batches:

T-SQL batches are collections of one or more T-SQL statements sent to SQL Server as a unit for parsing, optimization, and execution and are terminated with the GO clause

Batches are also boundaries for the scope of a variable

Some statements (e.g., CREATE FUNCTION, CREATE PROCEDURE, CREATE VIEW) may not be combined with others in the same batch

```
CREATE VIEW HumanResources.EmployeeList
AS
SELECT BusinessEntityID, JobTitle, HireDate, VacationHours
FROM HumanResources.Employee;
GO
```

Batches are parsed for syntax as a unit
Syntax errors cause the entire batch to be rejected
Runtime errors may allow the batch to continue after failure, by default
Batches can contain error-handling code

```
--Valid batch
INSERT INTO Production.UnitMeasure (Name, UnitMeasureCode, ModifiedDate)
VALUES (N'Square Footage', N'F4', GETDATE()),
       (N'Square Inches', N'I2', GETDATE());
GO
--Invalid batch
INSERT INTO dbo.t1 VALUE(1,2,N'abc');
INSERT INTO dbo.t1 VALUES(2,3,N'def');
GO
```

T-SQL Variables:
Variables are objects that allow storage of a value for use later in the same batch
Variables are defined with the DECLARE keyword and begin with @
Beginning with SQL Server 2008, variables can be declared and initialized in the same statement
Variables are always local to the batch in which they're declared and go out of scope when the batch ends

```
--Declare,initialize, and use a variable
DECLARE @SalesPerson_id INT = 5;
SELECT OrderYear, COUNT(DISTINCT CustomerID) AS CustCount
FROM (
SELECT YEAR(OrderDate) AS OrderYear, CustomerID
FROM Sales.SalesOrderHeader
WHERE SalesPersonID = @SalesPerson_id
) AS DerivedYear
GROUP BY OrderYear;
```

use 'AS' between var name & datatype, I think only in DDL statements is 'AS' not used before datatype and after col name ??

Working with Variables:
Values can be assigned with a SET command or a SELECT statement
SET can only assign one variable at a time. SELECT can assign multiple variables at a time
When using SELECT to assign a value, make sure that exactly one row is returned by the query

```
--Declare and initialize variables
DECLARE @numrows INT = 3, @catid INT = 2;

--Use variables to pass parameters to procedure
EXEC Production.ProdsByCategory
       @numrows = @numrows, @catid = @catid;
GO
```

Working with Synonyms:
A synonym is an alias or link to an object stored either on the same SQL Server instance or on a linked server
Synonyms can point to tables, views, procedures, and functions
Synonyms can be used for referencing remote objects as though they were located locally, or for providing alternative names to other local objects
Use the CREATE, ALTER, and DROP commands to manage synonyms

```
-- Create a synonym for the Product table in AdventureWorks CREATE SYNONYM dbo.MyProduct
FOR AdventureWorks.Production.Product;
GO
-- Query the Product table by using the synonym.
SELECT ProductID, Name
FROM MyProduct
WHERE ProductID < 5;
GO
```

T-SQL control-of-flow language:

SQL Server provides additional language elements that control the flow and execution of T-SQL statements in batches, stored procedures, and multi-statement functions
Control-of-flow elements allow you to specify statements need to be performed in a specified order or not at all
The default is for statements to execute sequentially, however you can use IF...ELSE, BEGIN...END, WHILE, RETURN, and others to control the flow of your batch files or stored procedures

```
IF OBJECT_ID ('Production.Product', 'U') IS NOT NULL
    PRINT 'I am here and contain data, so don't delete me'
```

IF...ELSE:
IF...ELSE uses a predicate to determine the flow of the code
The code in the IF block is executed if the predicate evaluates to TRUE
The code in the ELSE block is executed if the predicate evaluates to FALSE or UNKNOWN
Very useful when combined with the EXISTS operator

```
IF OBJECT_ID ('Production.Product', 'U') IS NOT NULL
    PRINT 'I am here and contain data, so don't delete me'
ELSE
  PRINT 'Table not found, so feel free to create one'
GO
```

WHILE:
WHILE enables statements to execute in the WHILE block as long as the predicate evaluates to TRUE and doesn't stop executing until the predicate evaluates to FALSE or UNKNOWN
Execution can be altered by BREAK or CONTINUE

```
DECLARE @BusinessEntID AS INT = 1, @Title AS NVARCHAR(50);
WHILE @BusinessEntID <=10
   BEGIN
       SELECT @Title = JobTitle FROM HumanResources.Employee
              WHERE BusinessEntityID = @BusinessEntID;
       PRINT @Title;
       SET @BusinessEntID += 1;
   END;
```

Structured exception handling:
Structured exception handling allows a centralized response to runtime errors
TRY to run a block of commands and CATCH any errors
Execution moves to the CATCH block of commands when an error occurs
No need to check every statement to see if an error occurred
If error you decide whether the transaction should be rolled back, errors logged, etc.
Not all errors can be caught by TRY / CATCH:
Syntax or compile errors
Some name resolution errors

Querying the ERROR object:
Common ERROR object properties and ERROR object functions:

Property	Function to Query	Description
Number	ERROR_NUMBER	Unique number assigned to the error
Message	ERROR_MESSAGE	Error message text
Severity	ERROR_SEVERITY	Severity class (1-25)
Procedure Name	ERROR_PROCEDURE	Name of the procedure or trigger that raised the error
Line Number	ERROR_LINE	Number of the line that raised the error in the batch, procedure, trigger, or function

Values returned correspond to sys.messages view

TRY and CATCH blocks:
TRY block defined by BEGIN TRY...END TRY statements
Place all code that might raise an error between them