<https://mva.microsoft.com/en-US/training-courses/querying-microsoft-sql-server-2012-databases-jump-start-8241>

--ddl or data definition language: create alter drop

--dml or data manipulation language: select insert update delete (CRUD operations)

--dcl or data control language: grant revoke deny

**T-SQL enforces operator precedence**

|  |  |
| --- | --- |
| **Elements:** | **Predicates and Operators:** |
| Predicates | IN, BETWEEN, LIKE |
| Comparison Operators | =, >, <, >=, <=, <>, !=, !>, !< |
| Logical Operators | AND, OR, NOT |
| Arithmetic Operators | +, -, \*, /, % |
| Concatenation | + |

**T-SQL functions**:

String functions: SUBSTRING,LEFT, RIGHT,LEN,DATALENGTH, REPLACE,REPLICATE,UPPER, LOWER,RTRIM, LTRIM

Date and time functions: GETDATE,SYSTDATETIME,GETUTCDATE,DATEADD,DATEDIFF,YEAR,MONTH,DAY

Aggregate functions: SUM,MIN,MAX,AVG,COUNT

**T-SQL variables**:

Local variables in T-SQL temporarily store a value of a specific data type

Name begins with single @ sign

@@ reserved for system functions

Assigned a data type

Must be declared and used within the same batch

In SQL Server 2008 and later, can declare and initialize in the same statement

**DECLARE @MyVar int = 30;**

**T-SQL expressions**:

Combination of identifiers, values, and operators evaluated to obtain a single result

Can be used in SELECT statements

SELECT clause

WHERE clause

Can be single constant, single-valued function, or variable

Can be combined if expressions have same the data type

**SELECT YEAR(OrderDate) + 1 ...**

**SELECT OrderQty \* UnitPrice ...**

**T-SQL batch separators**:

Batches are sets of commands sent to SQL Server as a unit

Batches determine variable scope, name resolution

To separate statements into batches, use a separator:

SQL Server tools use the GO keyword

GO is not a SQL Server T-SQL command

**T-SQL flow control, errors and transactions**:

Used in programmatic code objects such as stored procedures, triggers, statement blocks

Flow control: IF...ELSE,WHILE,BREAK,CONTINUE,BEGIN...END

Error handling: TRY…CATCH

Transaction control: BEGIN TRANSACTION,COMMIT TRANSACTION,ROLLBACK TRANSACTION

The order in which a query is written is not the order in which it is evaluated by the server. The order is:

5: SELECT <select list>

1: FROM <table source>

2: WHERE <search condition>

3: GROUP BY <group by list>

4: HAVING <search condition>

6: ORDER BY <order by list>

USE AdventureWorks2014;

SELECT SalesPersonID, YEAR(OrderDate) AS OrderYear

FROM Sales.SalesOrderHeader

WHERE CustomerID = 29974

GROUP BY SalesPersonID, YEAR(OrderDate)

HAVING COUNT(\*) > 1

ORDER BY SalesPersonID, OrderYear;

When performance tuning, using SELECT 1 gives you stats for just speaking to sql server on network(rather than what happens inside the database engine)

**Advanced SELECT clauses (DISTINCT, aliases, CASE, and scalar functions)**

Distinct: Specifies that only unique rows can appear in the result set

Removes duplicates based on column list results, not source table

Provides uniqueness across set of selected columns

Removes rows already operated on by WHERE, HAVING, and GROUP BY clauses

Some queries may improve performance by filtering out duplicates prior to execution of SELECT clause

SELECT DISTINCT <column list>

FROM <table or view>

**SELECT DISTINCT StoreID**

**FROM Sales.Customer;**

Using aliases to refer to columns: only ‘AS’ is the standard. Rest are here for legacy reasons

**SELECT SalesOrderID, UnitPrice, OrderQty AS Quantity**

**FROM Sales.SalesOrderDetail;**

**SELECT SalesOrderID, UnitPrice, Quantity =** **OrderQty**

**FROM Sales.SalesOrderDetail;**

**SELECT SalesOrderID, UnitPrice** **Quantity**

**FROM Sales.SalesOrderDetail;**

Using aliases to refer to tables: using alias makes name shorter and also then u can use same table multiple times(with different aliases) in joins

**SELECT SalesOrderID, ProductID FROM Sales.SalesOrderDetail AS SalesOrders;**

**SELECT SalesOrderID, ProductID FROM Sales.SalesOrderDetail SalesOrders;**

**SELECT SalesOrders.SalesOrderID, SalesOrders.ProductID FROM Sales.SalesOrderDetail AS SalesOrders;**

**T-SQL case expressions**:

Simple CASE

Compares one value to a list of possible values and returns first match

If no match, returns value found in optional ELSE clause

If no match and no ELSE, returns NULL

Searched CASE

Evaluates a set of predicates, or logical expressions

Returns value found in THEN clause matching first expression that evaluates to TRUE

T-SQL CASE expressions return a single (scalar) value

CASE expressions may be used in:

SELECT column list (behaves as calculated column requiring an alias)

WHERE or HAVING clauses

ORDER BY clause

**SELECT ProductID, Name, ProductSubCategoryID,**

**CASE ProductSubCategoryID**

**WHEN 1 THEN 'Beverages'**

**ELSE 'Unknown Category'**

**END**

**FROM Production.Product**

**Joins**:

|  |  |
| --- | --- |
| **Join Type** | **Description** |
| Cross | Combines all rows in both tables (creates Cartesian product). |
| Inner(the default join) | Starts with Cartesian product; applies filter to match rows between tables based on predicate. |
| Outer | Starts with Cartesian product; all rows from designated table preserved, matching rows from other table retrieved. Additional NULLs inserted as placeholders. |

Inner Join: As it is the default join, just specify join for inner join

Returns only rows where a match is found in both tables

Matches rows based on attributes supplied in predicate

ON clause in SQL-92 syntax

Why filter in ON clause?

Logical separation between filtering for purposes of JOIN and filtering results in WHERE

‘on’ filter is applied before the ‘where’ filter

Typically no difference to query optimizer

If JOIN predicate operator is =, also known as equi-join

**SELECT SOH.SalesOrderID, SOH.OrderDate, SOD.ProductID, SOD.UnitPrice, SOD.OrderQty**

**FROM Sales.SalesOrderHeader AS SOH**

**JOIN Sales.SalesOrderDetail AS SOD**

**ON SOH.SalesOrderID = SOD.SalesOrderID;**

Outer Join:

Returns all rows from one table and any matching rows from second table

One table’s rows are “preserved”

Designated with LEFT, RIGHT, FULL keyword

All rows from preserved table output to result set

Matches from other table retrieved

Additional rows added to results for non-matched rows

NULLs added in place where attributes do not match

Example: Return all customers and for those who have placed orders, return order information. Customers without matching orders will display NULL for order details.

Customers that did not place orders

SELECT CUST.CustomerID, CUST.StoreID, ORD.SalesOrderID, ORD.OrderDate

FROM Sales.Customer AS CUST

LEFT OUTER JOIN Sales.SalesOrderHeader AS ORD

ON CUST.CustomerID = ORD.CustomerID

WHERE ORD.SalesOrderID IS NULL;

Cross join:

Combine each row from first table with each row from second table

All possible combinations are displayed

Logical foundation for inner and outer joins

INNER JOIN starts with Cartesian product, adds filter

OUTER JOIN takes Cartesian output, filtered, adds back non-matching rows (with NULL placeholders)

Due to Cartesian product output, not typically a desired form of JOIN

Some useful exceptions:

Generating a table of numbers for testing

Example:

Create test data by returning all combinations of two inputs:

SELECT EMP1.BusinessEntityID, EMP2.JobTitle

FROM HumanResources.Employee AS EMP1

CROSS JOIN HumanResources.Employee AS EMP2;

Self Join:

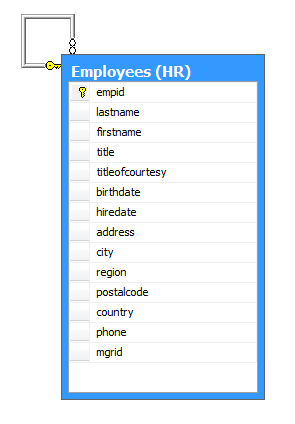
Why use self-joins?

Compare rows in same table to each other

Create two instances of same table in FROM clause

At least one alias required

Example: Return all employees and the name of the employee’s manager



Return all employees with ID of employee’s manager when a manager exists (INNER JOIN):

SELECT EMP.EmpID, EMP.LastName, EMP.JobTitle, EMP.MgrID, MGR.LastName

FROM HR.Employees AS EMP

INNER JOIN HR.Employees AS MGR

ON EMP.MgrID = MGR.EmpID;

Return all employees with ID of manager (OUTER JOIN). This will return NULL for the CEO:

SELECT EMP.EmpID, EMP.LastName,

EMP.Title, MGR.MgrID

FROM HumanResources.Employee AS EMP

LEFT OUTER JOIN HumanResources.Employee AS MGR

ON EMP.MgrID = MGR.EmpID;

Order By clause:

ORDER BY sorts rows in results for presentation purposes

Use of ORDER BY guarantees the sort order of the result

Last clause to be logically processed(processed after SELECT)

Sorts all NULLs together

ORDER BY can refer to:

Columns by name, alias or ordinal position (not recommended)

Columns not part of SELECT list unless DISTINCT clause specified

Declare sort order with ASC or DESC

SELECT SalesOrderID, CustomerID, OrderDate

FROM Sales.SalesOrderHeader

ORDER BY OrderDate;

SELECT SalesOrderID, CustomerID, YEAR(OrderDate) AS OrderYear

FROM Sales.SalesOrderHeader

ORDER BY OrderYear;

SELECT SalesOrderID, CustomerID, OrderDate

FROM Sales.SalesOrderHeader

ORDER BY OrderDate DESC;

WHERE clause:

WHERE clauses use predicates

Must be expressed as logical conditions

Only rows for which predicate evaluates to TRUE are accepted

Values of FALSE or UNKNOWN are filtered out

WHERE clause follows FROM, precedes other clauses

Can’t see aliases declared in SELECT clause

Can be optimized by SQL Server to use indexes

SELECT CustomerID, TerritoryID

FROM Sales.Customer

WHERE TerritoryID = 6;

SELECT CustomerID, TerritoryID

FROM Sales.Customer

WHERE TerritoryID >= 6;

SELECT CustomerID, TerritoryID, StoreID

FROM Sales.Customer

WHERE StoreID >= 1000 AND StoreID <= 1200;

Filtering data in the SELECT clause:

TOP allows you to limit the number or percentage of rows returned

Works with ORDER BY clause to limit rows by sort order

If ORDER BY list is not unique, results are not deterministic (no single correct result set)

Modify ORDER BY list to ensure uniqueness, or use TOP WITH TIES

Added to SELECT clause:

SELECT TOP (N) | TOP (N) Percent

With percent, number of rows rounded up

SELECT TOP (N) WITH TIES

Retrieve duplicates where applicable (nondeterministic)

TOP is proprietary to Microsoft SQL Server