

# Database Training Microsoft

## SQL Server (Day 2)



Authorized & published by Summitworks Technologies Inc



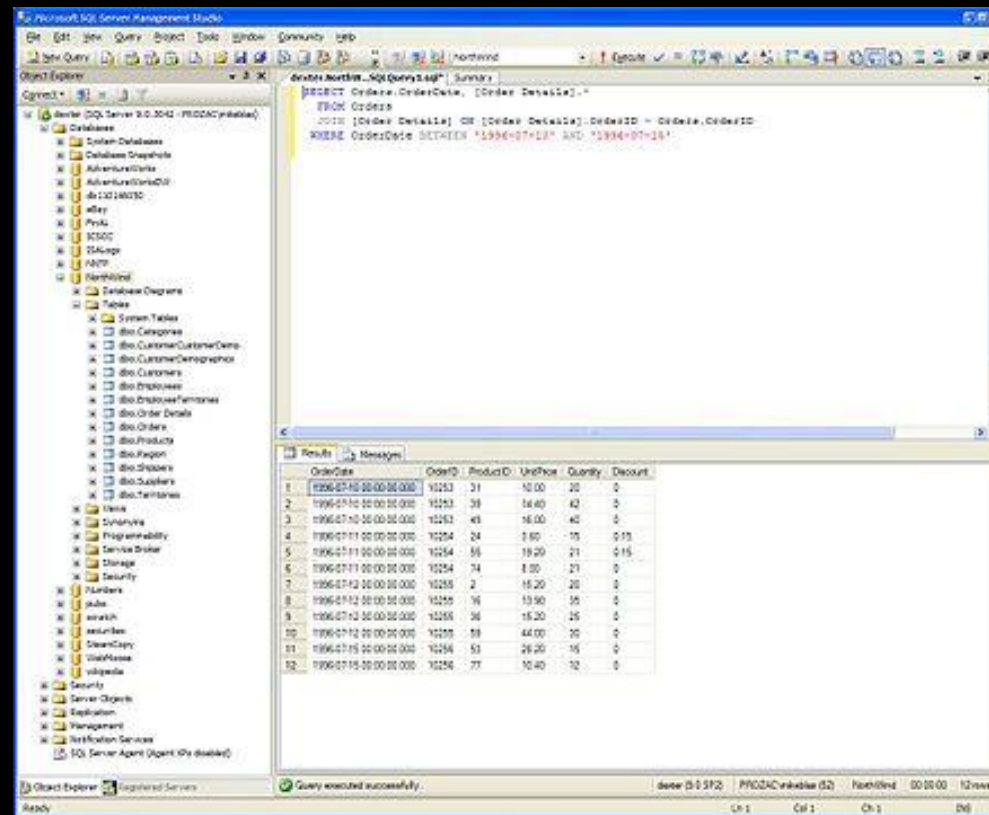
# Agenda Day 1

## T-SQL Part 2

- Introduction to various SQL server tools and SQL Server Management Studio
- Introduction to T-SQL
- DDL Queries (Create, Alter, Drop, Truncate)
- DML Queries (Insert, Update, Delete, Select)
- DCL Data Control Language
- TCL Transaction Control Language
- Joins
- Sub-Queries
- Union/ Union All
- Built-in functions: null(), Numeric, String and Date functions
- Aggregate functions

# SQL Server Management Studio

**SQL Server Management Studio (SSMS)** is a software application first launched with the [Microsoft SQL Server 2005](#) that is used for configuring, managing, and administering all components within Microsoft SQL Server. The tool includes both script editors and graphical tools which work with objects and features of the server.



# Introduction to T-SQL

- Transact – Structure Query Language (T-SQL) is Microsoft's (& Sybase's) proprietary extension to SQL.
- Its used for querying, altering and defining databases.
- Transact-SQL is central to using Microsoft SQL Server.
- All applications that communicate with an instance of SQL Server do so by sending Transact-SQL statements to the server, regardless of the user interface of the application.
- Although you can often avoid writing SQL, but using SQL GUI tools, there are still many situations where knowing basic T-SQL code allow to achieve tasks difficult or impossible with the GUI.

# SQL statements Categories

SQL statements are divided into two major categories:

- **Data Definition Language (DDL)**
- **Data Manipulation Language (DML)**
- **Data Control Language (DCL)**

DCL statements control the level of access that users have on database objects.

**GRANT** – allows users to read/write on certain database objects

**REVOKE** – keeps users from read/write permission on database objects

- **Transaction Control Language (TCL)**

TCL statements allow you to control and manage transactions to maintain the integrity of data within SQL statements.

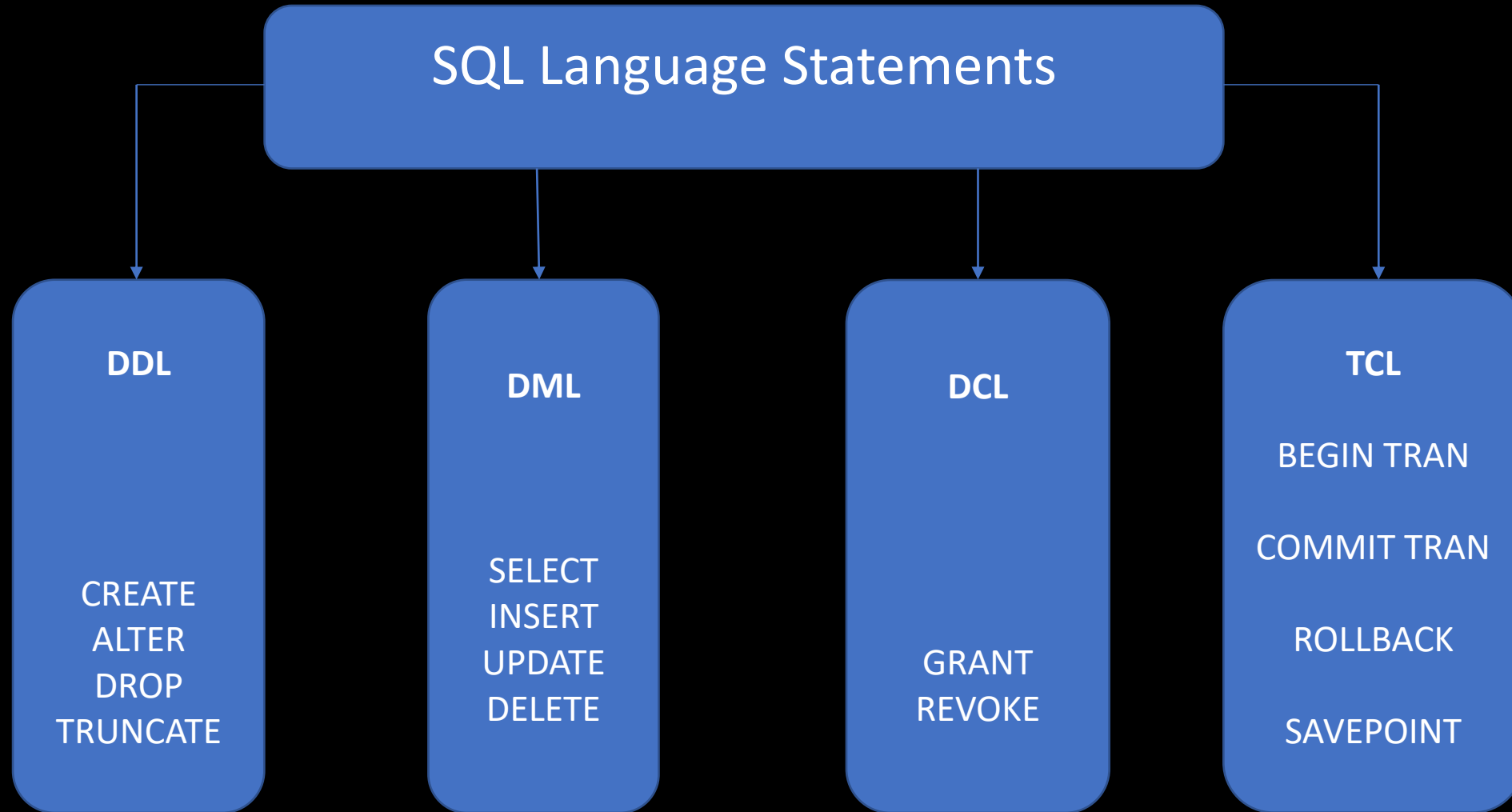
**BEGIN Transaction** – opens a transaction

**COMMIT Transaction** – commits a transaction

**ROLLBACK Transaction** – ROLLBACK a transaction in case of any error

**SAVE Transaction** – sets a Savepoint within a transaction

# SQL statements Categories



DDL STATEMENT

# Create, Alter, Drop Database

Data Definition Language (DDL) is a vocabulary used to define data structures in SQL Server. Use these statements to create, alter, or drop data structures in an instance of SQL Server. CREATE statements are used to define new entities.

Rules to create a database:

- Database names must be unique within an instance of SQL Server.
- Database names can be a maximum of 128 characters.
- The CREATE DATABASE statement must run in an auto-commit mode.
- A maximum of 32,767 databases can be specified on an instance of SQL Server.

Create Database :

```
CREATE DATABASE database_name;
```

Alter Database :

```
ALTER DATABASE Database_name  
MODIFY NAME = New Name
```

Drop Database :

```
DROP DATABASE Database_name
```

Restore Database :

```
restore Database database_name from disk = '<Backup file location + filename>'
```

System Databases cannot be deleted.

Create, Alter & Drop: All operations are case insensitive. We can use both upper and lower case as a syntax.



# Create, Alter and Drop table

- Tables are used to store data in the database.
- Tables are uniquely named within a [database](#) and [schema](#).
- Each table contains one or more columns and each column has an associated [data type](#) that defines the kind of data it can store e.g., numbers, strings, or temporal data.

To create a new table

```
CREATE TABLE [database_name.][schema_name.]table_name (  
    pk_column data_type PRIMARY KEY,  
    column_1 data_type NOT NULL,  
    column_2 data_type,  
    ...,  
    table_constraints  
);
```

Alter the table

```
Alter TABLE <Table name> ADD Column1 datatype, Column2 datatype;
```

Drop the table

```
DROP TABLE <tableName>;
```

# Truncate Table

Truncate deletes all the rows from the table without using the where clause

Create a new table customer\_group and insert some rows

```
CREATE TABLE sales.customer_groups (  
    group_id INT PRIMARY KEY IDENTITY,  
    group_name VARCHAR (50) NOT NULL  
);  
  
INSERT INTO sales.customer_groups (group_name)  
VALUES  
    ('Intercompany'),  
    ('Third Party'),  
    ('One time');
```

```
DELETE FROM sales.customer_groups;  
  
TRUNCATE TABLE sales.customer_groups;
```

TRUNCATE is similar to DELETE statement without using a WHERE clause but TRUNCATE executes faster and uses fewer system and transaction log resources.

# Truncate Vs Delete

Truncate	Delete
1. Use less Transaction log: Deletes the data by deallocating the data pages used to store the table data and inserts only the page deallocations in the transaction logs.	Removes rows one at a time and inserts an entry in the transaction log for each removed row.
2. Use fewer locks: Truncate locks the table and pages, not each row	Delete uses row lock, each row in the table is locked for removal
3. Identity reset: Reset the Identity Column to the seed value when data is deleted	Will not reset the Identity column to the seed value when data is deleted

# Restrictions (Truncate)

You cannot use TRUNCATE TABLE on tables that:

- Are referenced by a FOREIGN KEY constraint. (You can't truncate a table that has a foreign key that references itself.)
- Participate in an indexed view.
- Are published by using transactional replication or merge replication.
- TRUNCATE TABLE cannot activate a trigger because the operation does not log individual row deletions.

# DML Statement

# Data Manipulation Language (DML) Statements

- DML statements are used to work with the data in the tables.
- DML statements affect records in a table. These are basic operations we perform on data such as selecting a few records from a table, inserting new records, deleting unnecessary records, and updating/modifying existing records.
- SELECT Statement is considered to be part of DML even though it just retrieves data rather than modifying it.

SELECT – select records from a table

INSERT – insert new records

UPDATE – update/Modify existing records

DELETE – delete existing records

# INSERT (Transact-SQL)

INSERT statement is used to add one or more rows into a table

```
INSERT INTO table_name (column_list)
VALUES (value_list);
```

- Each column in the column list must have a corresponding value in the value list.
- The value of the identity Column in the table is automatically populated by the SQL Server when you add a new row to the table.
- When inserting records into a table using the SQL Server INSERT statement, you must provide a value for every NOT NULL column.
- You can omit a column from the SQL Server INSERT statement if the column allows NULL values.
- To capture the inserted values, you use the OUTPUT clause

# INSERT (Transact-SQL)

Example: Create a new table promotions and insert a row into the table

```
CREATE TABLE sales.promotions (  
    promotion_id INT PRIMARY KEY IDENTITY (1, 1),  
    promotion_name VARCHAR (255) NOT NULL,  
    discount NUMERIC (3, 2) DEFAULT 0,  
    start_date DATE NOT NULL,  
    expired_date DATE NOT NULL  
);
```

```
INSERT INTO sales.promotions (promotion_name, discount, start_date, expired_date)  
VALUES('2018 Summer Promotion', 0.15, '20180601', '20180901');
```

Messages

(1 row affected)



# INSERT (Transact-SQL)

Example to insert multiple rows in a table

```
INSERT INTO sales.promotions(promotion_name,discount,start_date,expired_date)  
VALUES('2019 Summer Promotion',0.15,'20190601', '20190901'),  
      ('2019 Fall Promotion',0.20,'20191001','20191101'  ),  
      ('2019 Winter Promotion',0.25,'20191201','20200101' );
```

Messages

(3 rows affected)


# INSERT (Transact-SQL)

Example to insert data from other tables into a table using INSERT INTO SELECT statement

Create a address table and insert all address from the customer table into the address table

```
CREATE TABLE sales.addresses (  
    address_id INT IDENTITY PRIMARY KEY,  
    street VARCHAR (255) NOT NULL,  
    city VARCHAR (50),  
    state VARCHAR (25),  
    zip_code VARCHAR (5));
```

```
INSERT INTO sales.addresses (street, city, state, zip_code)  
SELECT street, city, state, zip_code FROM sales.customers
```

 Messages

(1445 rows affected)

# DELETE Statement

To remove one or more rows from a table completely, we use the DELETE statement.

To Delete all the rows from a table we use the following syntax

```
DELETE FROM target_table;
```

To specify the number of rows that will be deleted, we use the TOP clause

```
DELETE TOP 10 FROM target_table;
```

To DELETE the rows by using WHERE clause

```
DELETE FROM table_name WHERE condition;
```

```
DELETE FROM table_name WHERE condition1 AND condition2;
```

# DELETE Statement

Example to Delete the number of random rows using Top clause

```
DELETE TOP (21) FROM production.product_history;
```

Messages

(21 rows affected)

Example to Delete some rows with a condition using Where clause

```
DELETE FROM production.product_history WHERE model_year = 2017;
```

Messages

(75 rows affected)

# UPDATE (TRANSACTION-SQL)

To modify existing data in a table we use UPDATE statement

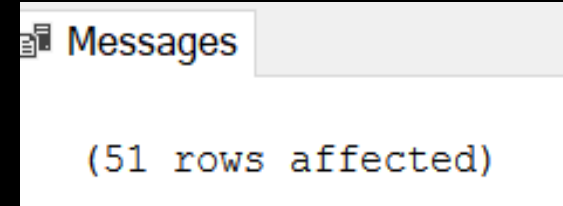
- First, specify the name of the table from which the data is to be updated.
- Second, specify a list of column c1, c2, ..., cn and values v1, v2, ... vn to be updated.
- Third, specify the conditions in the WHERE clause for selecting the rows that are updated.
- The WHERE clause is optional if you skip all rows in the table are updated.

```
UPDATE table_name  
SET c1 = v1, c2 = v2, ... cn = vn  
[WHERE condition]
```

# UPDATE (TRANSACTION-SQL)

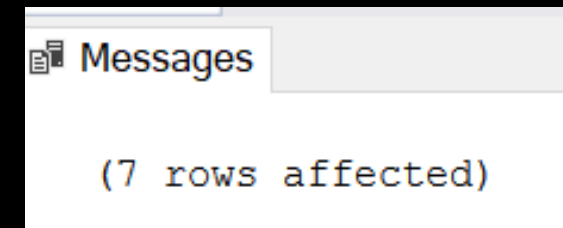
Example to UPDATE a single column in all rows

```
UPDATE sales.taxes SET updated_at = GETDATE();
```



Example to UPDATE multiple columns in a table

```
UPDATE sales.taxes  
SET max_local_tax_rate += 0.02,  
    avg_local_tax_rate += 0.01  
WHERE  
    max_local_tax_rate = 0.01;
```



# Select Statement

Database tables are objects that stores all the data in a database. In a table, data is logically organized in a row-and-column format which is similar to a spreadsheet.

In a table, each row represents a unique record and each column represents a field in the record.

To query the data from the table we use the SELECT statement

- When processing the SELECT statement SQL Server processes the FROM clause first and then the SELECT clause even though the SELECT clause appears first in the query.



```
SELECT first_name, last_name FROM sales.customers;
```

- Example to remove duplicate records from a table using the DISTINCT clause

```
SELECT DISTINCT city, state, zip_code FROM sales.customers;
```

# SELECT (Transact-SQL)

Example to retrieve some columns of a table

```
SELECT first_name,last_name FROM sales.customers;
```

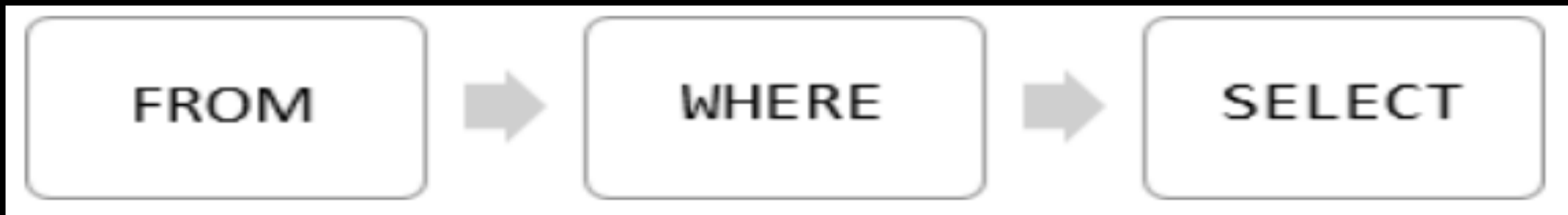
Example to retrieve all columns from a table

```
SELECT * FROM sales.customers;
```

Example to retrieve some rows using the WHERE clause

```
SELECT * FROM sales.customers WHERE state = 'CA';
```

SQL Server processes the above clauses of the query in the following sequence:



To passing more than one condition in the WHERE clause, use Logical Operators AND|OR|NOT

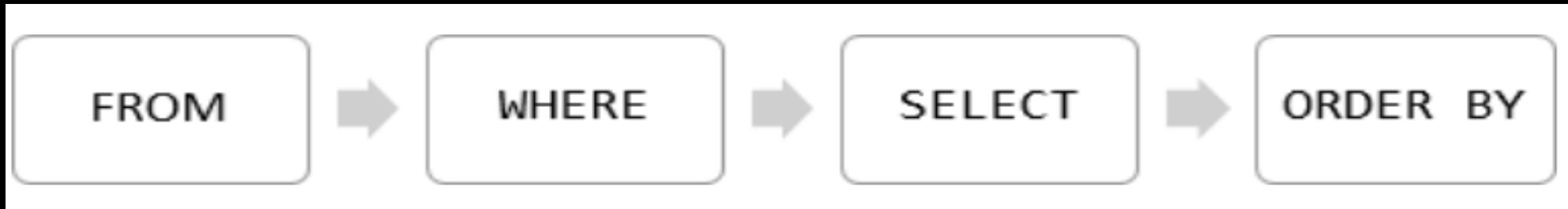


# SELECT (Transact-SQL)

To sort the result set based on one or more columns, use the ORDER BY clause

```
SELECT * FROM sales.customers WHERE state = 'CA' ORDER BY first_name;
```

SQL Server processes the clauses of the query in the following sequence:



# SELECT (Transact-SQL)

To group rows into groups, use the GROUP BY clause

```
SELECT city, COUNT (*) FROM sales.customers WHERE state = 'CA' GROUP BY city ORDER BY city;
```

SQL Server processes the clauses in the following sequence:



To filter groups based on one or more conditions, use the HAVING clause

```
SELECT city, COUNT (*) FROM sales.customers WHERE state = 'CA' GROUP BY city HAVING COUNT (*) > 10 ORDER BY city;
```

Note: The WHERE clause filters rows while the HAVING clause filter groups.

# SELECT Statements

## Operators (to pass some value)

=	→equal to [value]
!=	→!= not equal to [value]
<	→Less than [value]
>	→Greater than [value]
<=	→less than or equal to [value]
>=	→greater than or equal to [value]
LIKE	[wildcards]
BETWEEN	[VALUES AND VALUES]
IN	[VALUE1, VALUE2, ...]

# SELECT Statements

- WILDCARDS

- 'any text' → text/string enclosed in single quotation marks; for numerical values, NO NEED to enclose in ' ' marks.
- ^ → not
- '%a', 'a%', '%a%' (enclosed in ' ' marks)
  - a% → starts with a
  - %a → ends with a
  - %a% → with a in between

```
SELECT * as QueryResults  
WHERE ColumnName LIKE 'a%'
```

```
SELECT * as QueryResults  
WHERE ColumnName LIKE '[abcde]%'
```

```
SELECT * as QueryResults  
WHERE ColumnName BETWEEN [1 AND  
25]
```

```
SELECT * as QueryResults  
WHERE ColumnName IN [1,4,6,10]
```

# GROUP BY Clause

- GROUP BY clause allows you to arrange the rows of a [query](#) in groups.
- The groups are determined by the columns that you specify in the GROUP BY clause.
- GROUP BY clause is often used with [aggregate functions](#) for generating summary reports.
- HAVING clause is often used with the GROUP BY clause to filter groups based on a specified list of conditions.

# AGGREGATE FUNCTIONS

An aggregate function performs a calculation on one or more values and returns a single value. The aggregate function is often used with the GROUP BY clause and HAVING clause of the SELECT statement

STDEVP	The <code>STDEVP()</code> function also returns the standard deviation for all values in the provided expression, but does so based on the entire data population.
SUM	The <code>SUM()</code> aggregate function returns the summation of all non-NULL values in a set.
VAR	The <code>VAR()</code> function returns the statistical variance of values in an expression based on a sample of the specified population.
VARP	The <code>VARP()</code> function returns the statistical variance of values in an expression but does so based on the entire data population.

Aggregate function	Description
AVG	The <code>AVG()</code> aggregate function calculates the average of non-NULL values in a set.
CHECKSUM_AGG	The <code>CHECKSUM_AGG()</code> function calculates a checksum value based on a group of rows.
COUNT	The <code>COUNT()</code> aggregate function returns the number of rows in a group, including rows with NULL values.
COUNT_BIG	The <code>COUNT_BIG()</code> aggregate function returns the number of rows (with BIGINT data type) in a group, including rows with NULL values.
MAX	The <code>MAX()</code> aggregate function returns the highest value (maximum) in a set of non-NULL values.
MIN	The <code>MIN()</code> aggregate function returns the lowest value (minimum) in a set of non-NULL values.
STDEV	The <code>STDEV()</code> function returns the statistical standard deviation of all values provided in the

# AGGREGATE FUNCTIONS

- First, specify the name of an aggregate function that you want to use such as AVG, SUM, and MAX
- Second, use DISTINCT if you want only distinct values are considered in the calculation
- Third, the expression can be a column of a table or an expression that consists of multiple columns with arithmetic operators.
- Example for the AVG() function to return the average list price of all products in the products table:

```
SELECT
    AVG(list_price) avg_product_price
FROM
    production.products;
```

JOINS



# Joins

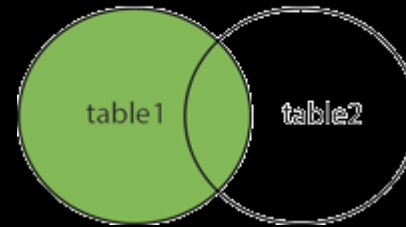
- In a relational database, data is distributed in multiple logical tables. To get a complete meaningful set of data, you need to query data from these tables by using joins.
- SQL joins are used to combine rows from two or more tables, based on a common field between them.
- SQL Server supports many kinds of joins including
  - [inner join](#)
  - [Left join](#)
  - [right join](#)
  - [full outer join](#)
  - [cross join](#)
- Each join type specifies how SQL Server uses data from one table to select rows in another table.

# Types of Joins

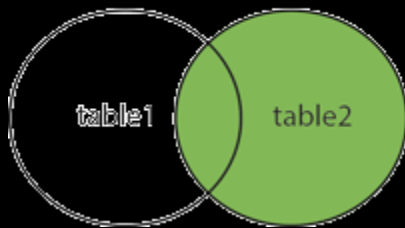
INNER JOIN



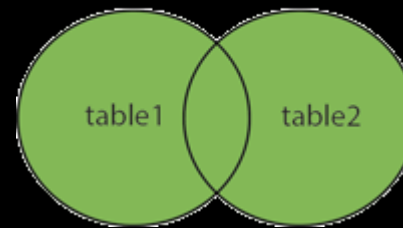
LEFT JOIN



RIGHT JOIN

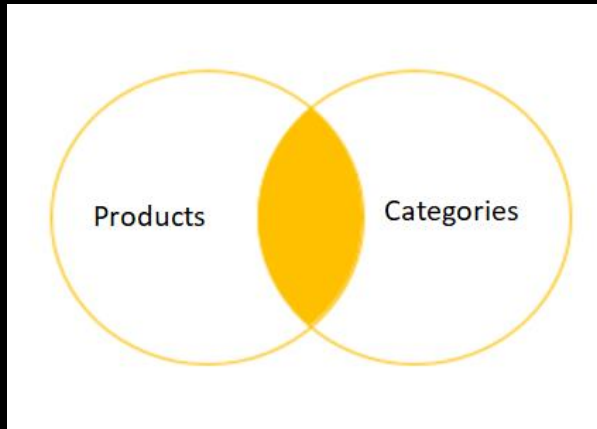


FULL OUTER JOIN



# INNER Join

The inner join is one of the most commonly used joins in SQL Server. The inner join clause allows you to query data from two or more related tables.

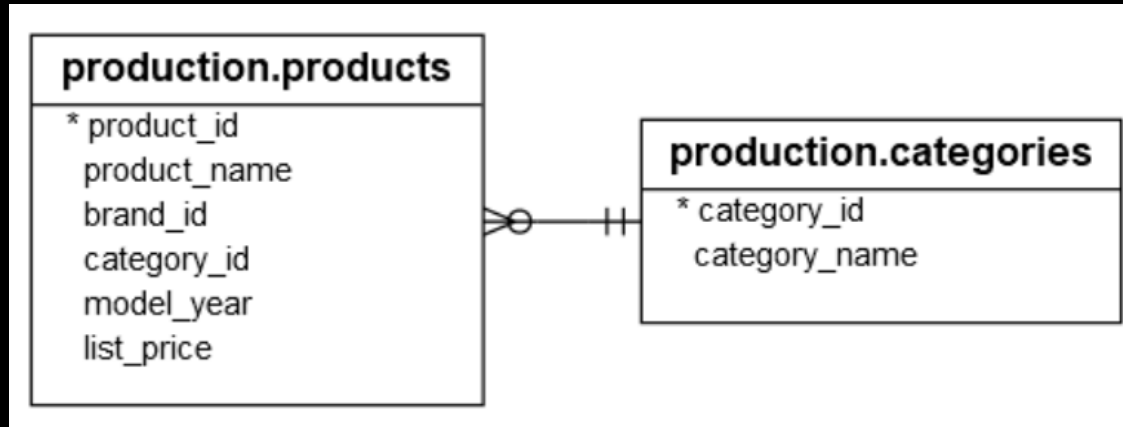


```
SELECT
    select_list
FROM
    T1
INNER JOIN T2 ON join_predicate;
```

In this syntax, the query retrieved data from both T1 and T2 tables:

- First, specify the main table (T1) in the FROM clause
- Second, specify the second table in the INNER JOIN clause (T2) and a join predicate. Only rows that cause the join predicate to evaluate to TRUE are included in the result set.

# INNER Join



To include the category names in the result set, use the INNER clause as follows:

```
SELECT product_name, category_name, list_price FROM production.products p
      INNER JOIN production.categories c ON c.category_id = p.category_id
      ORDER BY product_name DESC;
```

# Sample tables

## Employee table

LastName	DepartmentID
Rafferty	31
Jones	33
Heisenberg	33
Robinson	34
Smith	34
Williams	NULL

## Department table

DepartmentID	DepartmentName
31	Sales
33	IT
34	Clerical
35	Marketing

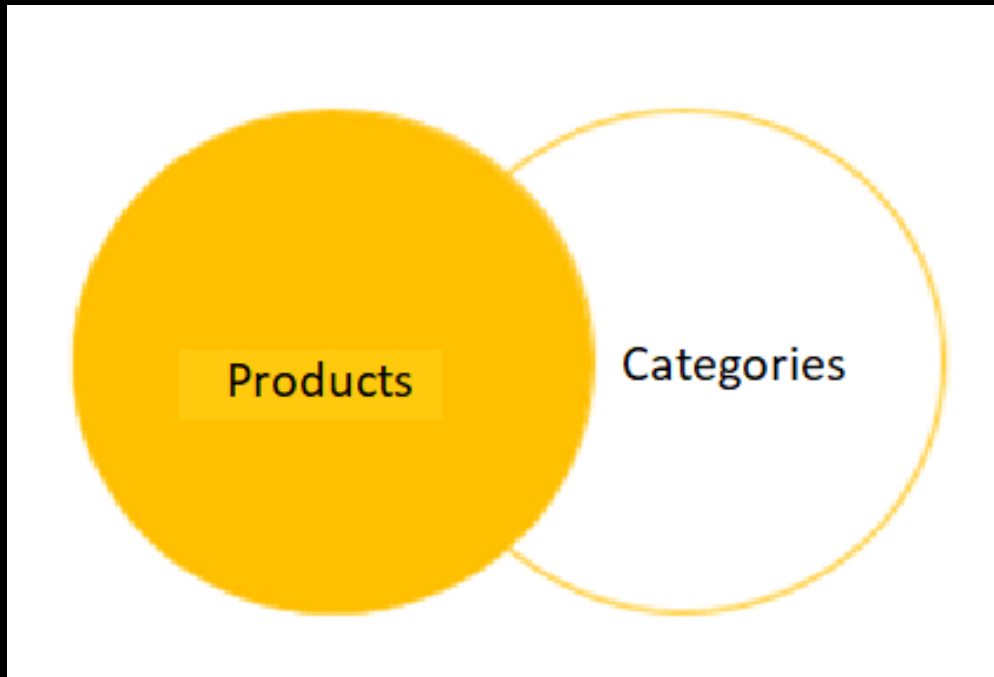
# Inner Join

```
SELECT E.Lastname,E.DepartmentID, D.DepartmentName,D.DepartmentID
FROM employee E
INNER JOIN department D
ON E.DepartmentID = D.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Robinson	34	Clerical	34
Jones	33	IT	33
Smith	34	Clerical	34
Heisenberg	33	IT	33
Rafferty	31	Sales	31

# LEFT Joins

It returns all rows from the left table and the matching rows from the right table. If no matching rows found in the right table NULL are used.



```
SELECT
    select_list
FROM
    T1
LEFT JOIN T2 ON
    join_predicate;
```

# Left Outer Join

```
SELECT * FROM Employee E  
LEFT OUTER JOIN Department D  
ON E.DepartmentID = D.DepartmentID;
```

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Jones	33	IT	33
Rafferty	31	Sales	31
Robinson	34	Clerical	34
Smith	34	Clerical	34
Williams	NULL	NULL	NULL
Heisenberg	33	IT	33



# Right Outer Join

**SELECT \* FROM** employee **RIGHT OUTER JOIN** department **ON** employee.DepartmentID = department.DepartmentID;

Employee.LastName	Employee.DepartmentID	Department.Department Name	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	IT	33
Robinson	34	Clerical	34
Heisenberg	33	IT	33
Rafferty	31	Sales	31
NULL	NULL	Marketing	35

# Full Outer Join

**SELECT \* FROM** employee **FULL OUTER JOIN** department **ON** employee.DepartmentID = department.DepartmentID;

Employee.LastName	Employee.DepartmentID	Department.Department Name	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	IT	33
Robinson	34	Clerical	34
Williams	NULL	NULL	NULL
Heisenberg	33	IT	33
Rafferty	31	Sales	31
NULL	NULL	Marketing	35

# CROSS JOIN

CROSS JOIN to Join two or more unrelated tables.

The CROSS JOIN joined every row from the first table (T1) with every row from the second table (T2). In other words, the cross join returns a Cartesian product of rows from both tables.

Unlike INNER JOIN and LEFT JOIN, the cross join does not establish a relationship between the joined tables.

```
SELECT
  select_list
FROM
  T1
CROSS JOIN T2;
```

```
SELECT product_id, product_name, store_id, 0 AS quantity
FROM production.products CROSS JOIN sales.stores
ORDER BY product_name, store_id;
```

## A self-join is joining a table to itself

```
SELECT F EmployeeID F LastName S EmployeeID S LastName F Country
FROM Employee F
INNER JOIN Employee S
ON F Country = S Country
WHERE F EmployeeID < S EmployeeID
ORDER BY F EmployeeID S EmployeeID
```

Employee Table (Sample table)

EmployeeID	LastName	Country	DepartmentID
123	Rafferty	Australia	31
124	Jones	Australia	33
145	Heisenberg	Australia	33
201	Robinson	United States	34
305	Smith	Germany	34
306	Williams	Germany	NULL

Employee Table after Self-join by Country

EmployeeID	LastName	EmployeeID	LastName	Country
123	Rafferty	124	Jones	Australia
123	Rafferty	145	Heisenberg	Australia
124	Jones	145	Heisenberg	Australia
305	Smith	306	Williams	Germany

# SUB-QUERIES

- A Subquery or Inner query or Nested query is a query within another SQL query and embedded within the WHERE clause.
- A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
- Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN etc.

# SUB-QUERIES

**There are a few rules that subqueries must follow:**

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators, such as the IN operator.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery; however, the BETWEEN operator can be used within the subquery.

# Subquery

```
SELECT column_name [, column_name ]  
FROM table1 [, table2 ]  
WHERE column_name OPERATOR (SELECT column_name [,  
column_name ]  
FROM table1 [, table2 ] [WHERE])
```

```
SQL> SELECT *  
      FROM CUSTOMERS  
      WHERE ID IN (SELECT ID  
                  FROM CUSTOMERS  
                  WHERE SALARY > 4500) ;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Result

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
7	Muffy	24	Indore	10000.00

# UNION/ UNION ALL

The UNION operator is used to combine the result-set of two or more SELECT statements.

Each SELECT statement within the UNION must have the same number of columns.

The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.

The UNION operator selects only distinct values by default. To allow duplicate values, use the **ALL** keyword with **UNION**.

**Select Columnname from table 1**

**Union**

**Select Columnname from table 2**

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

SupplierID	SupplierName	ContactName	Address	City	PostalCode	Country
1	Exotic Liquid	Charlotte Cooper	49 Gilbert St.	Londona	EC1 4SD	UK
2	New Orleans Cajun Delights	Shelley Burke	P.O. Box 78934	New Orleans	70117	USA
3	Grandma Kelly's Homestead	Regina Murphy	707 Oxford Rd.	Ann Arbor	48104	USA

Result

Results will get all the data from both the tables



# Built-In Functions

- **Numeric Functions**
- **String Functions**
- **String / Number Conversion Functions**
- **Formats for TO\_CHAR Function**
- **Group Functions**
- **Date and Time Functions**
- **Date Conversion Functions**
- **Date Formats**

# Number Functions

## Numeric Functions

Function	Input Argument	Value Returned
ABS ( m )	m = value	Absolute value of m
MOD ( m, n )	m = value, n = divisor	Remainder of m divided by n
POWER ( m, n )	m = value, n = exponent	m raised to the nth power
ROUND ( m [, n ] )	m = value, n = number of decimal places, default 0	m rounded to the nth decimal place
TRUNC ( m [, n ] )	m = value, n = number of decimal places, default 0	m truncated to the nth decimal place
SIN ( n )	n = angle expressed in radians	sine (n)
COS ( n )	n = angle expressed in radians	cosine (n)
TAN ( n )	n = angle expressed in radians	tan (n)
ASIN ( n )	n is in the range -1 to +1	arc sine of n in the range $-\pi/2$ to $+\pi/2$
ACOS ( n )	n is in the range -1 to +1	arc cosine of n in the range 0 to $\pi$
ATAN ( n )	n is unbounded	arc tangent of n in the range $-\pi/2$ to $+\pi/2$
SINH ( n )	n = value	hyperbolic sine of n
COSH ( n )	n = value	hyperbolic cosine of n
TANH ( n )	n = value	hyperbolic tangent of n
SQRT ( n )	n = value	positive square root of n
EXP ( n )	n = value	e raised to the power n
LN ( n )	n > 0	natural logarithm of n
LOG ( n2, n1 )	base n2 any positive value other than 0 or 1, n1 any positive value	logarithm of n1, base n2
CEIL ( n )	n = value	smallest integer greater than or equal to n
FLOOR ( n )	n = value	greatest integer smaller than or equal to n
SIGN ( n )	n = value	-1 if n < 0, 0 if n = 0, and 1 if n > 0

# String Functions

## String Functions

Function	Input Argument	Value Returned
INITCAP ( s )	s = character string	First letter of each word is changed to uppercase and all other letters are in lower case.
LOWER ( s )	s = character string	All letters are changed to lowercase.
UPPER ( s )	s = character string	All letters are changed to uppercase.
CONCAT ( s1, s2 )	s1 and s2 are character strings	Concatenation of s1 and s2. Equivalent to <i>s1    s2</i>
LPAD ( s1, n [, s2] )	s1 and s2 are character strings and n is an integer value	Returns s1 right justified and padded left with n characters from s2; s2 defaults to space.
RPAD ( s1, n [, s2] )	s1 and s2 are character strings and n is an integer value	Returns s1 left justified and padded right with n characters from s2; s2 defaults to space.
LTRIM ( s [, set ] )	s is a character string and <i>set</i> is a set of characters	Returns s with characters removed up to the first character not in set; defaults to space
RTRIM ( s [, set ] )	s is a character string and <i>set</i> is a set of characters	Returns s with final characters removed after the last character not in set; defaults to space
REPLACE ( s, search_s [, replace_s ] )	s = character string, search_s = target string, replace_s = replacement string	Returns s with every occurrence of search_s in s replaced by replace_s; default removes search_s
SUBSTR ( s, m [, n ] )	s = character string, m = beginning position, n = number of characters	Returns a substring from s, beginning in position m and n characters long; default returns to end of s.
LENGTH ( s )	s = character string	Returns the number of characters in s.
INSTR ( s1, s2 [, m [, n ] ] )	s1 and s2 are character strings, m = beginning position, n = occurrence of s2 in s1	Returns the position of the nth occurrence of s2 in s1, beginning at position m, both m and n default to 1.

# String and Number Conversion

## String / Number Conversion Functions

Function	Input Argument	Value Returned
NANVL ( n2, n1 )	n1, n2 = value	if (n2 = NaN) returns n1 else returns n2
TO_CHAR ( m [, fmt ] )	m = numeric value, fmt = format	Number m converted to character string as specified by the format
TO_NUMBER ( s [, fmt ] )	s = character string, fmt = format	Character string s converted to a number as specified by the format

## Formats for TO\_CHAR Function

Symbol	Explanation
9	Each 9 represents one digit in the result
0	Represents a leading zero to be displayed
\$	Floating dollar sign printed to the left of number
L	Any local floating currency symbol
.	Prints the decimal point
,	Prints the comma to represent thousands

## Group Functions

Function	Input Argument	Value Returned
AVG ( [ DISTINCT   ALL ] col )	col = column name	The average value of that column
COUNT ( * )	none	Number of rows returned including duplicates and NULLs
COUNT ( [ DISTINCT   ALL ] col )	col = column name	Number of rows where the value of the column is not NULL
MAX ( [ DISTINCT   ALL ] col )	col = column name	Maximum value in the column
MIN ( [ DISTINCT   ALL ] col )	col = column name	Minimum value in the column
SUM ( [ DISTINCT   ALL ] col )	col = column name	Sum of the values in the column
CORR ( e1, e2 )	e1 and e2 are column names	Correlation coefficient between the two columns after eliminating nulls
MEDIAN ( col )	col = column name	Middle value in the sorted column, interpolating if necessary
STDDEV ( [ DISTINCT   ALL ] col )	col = column name	Standard deviation of the column ignoring NULL values
VARIANCE ( [ DISTINCT   ALL ] col )	col = column name	Variance of the column ignoring NULL values

# Date and Time Function

CURRENT\_TIMESTAMP

## 1. CURRENT\_TIMESTAMP: Will give the current time as result set

The syntax for the CURRENT\_TIMESTAMP function in SQL Server (Transact-SQL) is:

There are no parameters or arguments for the CURRENT\_TIMESTAMP function.

### **Note:**

- The CURRENT\_TIMESTAMP function returns the system date and time in the format '*yyyy-mm-dd hh:mi:ss.mmm*'.
- Do not put parentheses () after the CURRENT\_TIMESTAMP function.

# Date and Time Function

2. DATEADD(): In SQL Server (Transact-SQL), this function returns a date after which a certain time/date interval has been added.

The syntax for the DATEADD function in SQL Server (Transact-SQL) is:

```
DATEADD( interval, number, date )
```

## Parameters or Arguments

Interval: The time/date interval that you wish to add. It can be one of the following values:

number: The number of intervals that you wish to add

Date: The date to which the interval should be added.

Value (any one of)	Explanation
year, yyyy, yy	Year interval
quarter, qq, q	Quarter interval
month, mm, m	Month interval
dayofyear	Day of year interval
day, dy, y	Day interval
week, ww, wk	Week interval
weekday, dw, w	Weekday interval
hour, hh	Hour interval
minute, mi, n	Minute interval
second, ss, s	Second interval
millisecond, ms	Millisecond interval

# Date and Time Function

## Examples:

```
SELECT DATEADD(year, 1, '2014/04/28');  
Result: '2015-04-28 00:00:00.000'
```

```
SELECT DATEADD(yyyy, 1, '2014/04/28');  
Result: '2015-04-28 00:00:00.000'
```

```
SELECT DATEADD(yy, 1, '2014/04/28');  
Result: '2015-04-28 00:00:00.000'
```

```
SELECT DATEADD(year, -1, '2014/04/28');  
Result: '2013-04-28 00:00:00.000'
```

```
SELECT DATEADD(month, 1, '2014/04/28');  
Result: '2014-05-28 00:00:00.000'
```

```
SELECT DATEADD(month, -1, '2014/04/28');  
Result: '2014-03-28 00:00:00.000'
```

```
SELECT DATEADD(day, 1, '2014/04/28');  
Result: '2014-04-29 00:00:00.000'
```

```
SELECT DATEADD(day, -1, '2014/04/28');  
Result: '2014-04-27 00:00:00.000'
```

# Date and Time Function

3. DATEDIFF(): The DATEDIFF function returns the difference between two date values, based on the interval specified.

Syntax:

```
DATEDIFF( interval, date1, date2 )
```

Interval: The time/date interval that you wish to add.

Date1, Date2: The two dates to calculate the difference between.

Examples:

```
SELECT DATEDIFF(year, '2012/04/28', '2014/04/28');  
Result: 2
```

```
SELECT DATEDIFF(yyyy, '2012/04/28', '2014/04/28');  
Result: 2
```

```
SELECT DATEDIFF(yy, '2012/04/28', '2014/04/28');  
Result: 2
```

```
SELECT DATEDIFF(month, '2014/01/01', '2014/04/28');  
Result: 3
```

```
SELECT DATEDIFF(day, '2014/01/01', '2014/04/28');  
Result: 117
```

```
SELECT DATEDIFF(hour, '2014/04/28 08:00', '2014/04/28 10:45');  
Result: 2
```

```
SELECT DATEDIFF(minute, '2014/04/28 08:00', '2014/04/28 10:45');  
Result: 165
```



# Date Functions

## Date Formats

Format Code	Description	Range of Values
DD	Day of the month	1 - 31
DY	Name of the day in 3 uppercase letters	SUN, ..., SAT
DAY	Complete name of the day in uppercase, padded to 9 characters	SUNDAY, ..., SATURDAY
MM	Number of the month	1 - 12
MON	Name of the month in 3 uppercase letters	JAN, ..., DEC
MONTH	Name of the month in uppercase padded to a length of 9 characters	JANUARY, ..., DECEMBER
RM	Roman numeral for the month	I, ..., XII
YY or YYYY	Two or four digit year	71 or 1971
HH:MI:SS	Hours : Minutes : Seconds	10:28:53
HH 12 or HH 24	Hour displayed in 12 or 24 hour format	1 - 12 or 1 - 24
MI	Minutes of the hour	0 - 59
SS	Seconds of the minute	0 - 59
AM or PM	Meridian indicator	AM or PM
SP	A suffix that forces the number to be spelled out.	e.g. TWO THOUSAND NINE
TH	A suffix meaning that the ordinal number is to be added	e.g. 1st, 2nd, 3rd, ...
FM	Prefix to DAY or MONTH or YEAR to suppress padding	e.g. MONDAY with no extra spaces at the end

# Sample build-in functions

## SAMPLE BUILT-IN FUNCTION:

```
select round (83.28749, 2) as QueryResults;
```

```
select sqrt (3.67) as QueryResults;
```

```
select power (2.512, 5) as QueryResults;
```

```
select to_char ( sysdate, 'MON DD, YYYY' ) as QueryResults;
```

```
select to_char ( sysdate, 'HH12:MI:SS AM' ) as QueryResults;
```

```
select to_char ( new_time ( sysdate, 'CDT', 'GMT'), 'HH24:MI' ) as QueryResults;
```

```
select greatest ( to_date ( 'JAN 19, 2000', 'MON DD, YYYY' ), to_date ( 'SEP 27, 1999', 'MON DD, YYYY' ), to_date ( '13-Mar-2009', 'DD-Mon-YYYY' ) ) as QueryResults;
```

```
select next_day ( sysdate, 'FRIDAY' ) as QueryResults;
```

```
select last_day ( add_months ( sysdate, 1 ) ) as QueryResults;
```

```
select concat ('Alan', 'Turing') as "NAME" as QueryResults;
```

```
select 'Alan' || 'Turing' as "NAME" as QueryResults;
```

```
select initcap ("now is the time for all good men to come to the aid of the party") as "SLOGAN" as QueryResults;
```

```
select substr ('Alan Turing', 1, 4) as "FIRST" as QueryResults;
```

# AGGREGATE FUNCTIONS

- SQL aggregate functions return a single value, calculated from values in a column.
- Useful aggregate functions:
  - AVG() - Returns the average value
  - COUNT() - Returns the number of rows
  - FIRST() - Returns the first value
  - LAST() - Returns the last value
  - MAX() - Returns the largest value
  - MIN() - Returns the smallest value
  - SUM() - Returns the sum

# AGGREGATE FUNCTIONS

- GROUP BY column\_name
  - Used to group a selected set of rows into summary of rows by the values of one or more columns or expression
  - ALWAYS used in conjunction with one or more AGGREGATE function
  - Columns that does not have an aggregate function must be contain in a GROUP BY clause
- HAVING search\_condition
  - Pass condition after the AGGREGATE function takes place

# AGGREGATE FUNCTIONS

Sample:

```
SELECT COUNT(name) as total_names  
FROM TraineeList
```

```
SELECT Column1, Column2, AVG(Column3)  
FROM TableName  
GROUP BY Column1, Column2
```

```
SELECT ID, NAME, AGE, ADDRESS, SALARY  
FROM CUSTOMERS  
GROUP BY age  
HAVING COUNT(ID) = 2;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

result

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500.00

# Null Functions

- Null Functions- used to handle NULL values in database. The objective of the general NULL handling functions is to replace the NULL values with an alternate value.
  - ISNULL() function is used to specify how we want to treat NULL values.
  - ```
SELECT ProductName,UnitPrice*(UnitsInStock+ISNULL(UnitsOnOrder,0))  
FROM Products
```
  - \* if "UnitsOnOrder" is NULL it will not harm the calculation, because ISNULL() returns a zero if the value is NULL
- NULLIF()-The NULLIF function compares two arguments expr1 and expr2. If expr1 and expr2 are equal, it returns NULL; else, it returns expr1. Unlike the other null handling function, first argument can't be NULL.
- NULLIF (expr1, expr2);

# NVL

- NVL()  
substitutes an alternate value for a NULL value. both the parameters are mandatory. Note that NVL function works with all types of data types. And also that the data type of original string and the replacement must be in compatible state. If arg1 is a character value, then converts replacement string to the data type compatible with arg1 before comparing them and returns VARCHAR in the character set of expr1. If arg1 is numeric, then determines the argument with highest numeric precedence, implicitly converts the other argument to that data type, and returns that data type.

```
NVL( Arg1, replace_with )
```

```
SELECT first_name, NVL(JOB_ID, 'n/a')  
FROM employees;
```