

SQL Full Course

What is a database?

A data is a collection of facts or figures, or information that's stored and can be used by a computer or program for calculation, reasoning, or discussion.

A database is a collection of related data stored in a format that can easily be accessed. Databases are used for storing, maintaining and accessing any sort of data.

Example – Phone Contacts



Categorization of DBMS (1/2)

There are two important types of DBMS:


1. Relational or SQL Databases
2. Non-Relational or NoSQL Databases

A relational database is a type of data store organizing data into tables that are related to one another, as per the name.

RDBMS have a predefined schema, meaning data resides in rows (records) and columns (attributes) with a defined structure.

Ex – MS SQL Server, MySQL, Oracle, PostgreSQL, etc.

Emp ID	Dept.	Grade
101	IT	A+
102	HR	B
103	IT	B+



Emp ID	Name	Address	Phone
101	Joe	xxxxxx	xxxxxx
102	Sam	xxxxxx	xxxxxx
103	Bob	xxxxxx	xxxxxx

Categorization of DBMS (2/2)

A Non-relational databases (often called NoSQL databases) are different from traditional relational databases in that they store their data in a non-tabular form. Instead, non-relational databases might be based on data structures like documents.

In NoSQL databases, data may be stored as simple key/value pairs, as JSON documents, or as a graph consisting of edges and vertices.

NoSQL databases grew popular as web applications became more common and more complex.

Ex – MongoDB, Apache Cassandra, Redis, Couchbase and Apache HBase, etc.

Key	Document
1001	<pre>{ "CustomerID": 99, "OrderItems": [{ "ProductID": 2010, "Quantity": 2, "Cost": 520 }, { "ProductID": 4365, "Quantity": 1, "Cost": 18 }], "OrderDate": "04/01/2017" }</pre>
1002	<pre>{ "CustomerID": 220, "OrderItems": [{ "ProductID": 1285, "Quantity": 1, "Cost": 120 }], "OrderDate": "05/08/2017" }</pre>

What is SQL?

S - Structured

Q - Query

L – Language

- Popularly known as Sequel
- SQL is a query language for storing, manipulating, and retrieving data stored in a relational database.
- All RDBMS uses the SQL as a standard database language. It is used to communicate with database.
- Every vendor has its own implementation
- Declarative, not procedural.

Software Installation

Download the SQL Server Developer or Express Edition:

<https://www.microsoft.com/en-in/sql-server/sql-server-downloads>

Download SQL Server Management Studio (SSMS):

<https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms>



Databases

A database is a collection of tables and objects such as views, indexes, stored procedures, and triggers.

System Databases:

SQL Server has the following system databases:

- master
- tempdb
- model
- msdb

Database can be created:

- Graphically
- Using Query – Create Database DBName

DDL and DML in DBMS

DDL (Data Definition Language)	DML (Data Manipulation Language)
The DDL commands help to define the structure of the databases or schema	The DML commands deal with the manipulation of existing records of a database.
Popular Commands: Create, Drop, Alter, Truncate	Popular Commands: Select, Insert, Update, Delete
DDL commands can affect the whole database or table	DML statements only affect single or multiple rows based on the condition specified in a query.
DDL does not use WHERE clause in its statement.	The data in DML statements can be filtered with a WHERE clause.

Data types in SQL Server

In SQL Server, there are various data types available to store different types of data. Here is a list of commonly used data types in SQL Server:

Exact Numeric Data Types:

- BIGINT: Integer values ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
- INT: Integer values ranging from -2,147,483,648 to 2,147,483,647.
- SMALLINT: Integer values ranging from -32,768 to 32,767.
- TINYINT: Integer values ranging from 0 to 255.
- BIT: Represents a single bit value of 0, 1, or NULL.
- DECIMAL(p, s): Fixed precision and scale numeric values.
- NUMERIC(p, s): Same as DECIMAL.

Approximate Numeric Data Types:

- FLOAT: Floating-point numbers with a decimal precision.
- REAL: Floating-point numbers with a decimal precision of 7 digits.

Data types in SQL Server

Date and Time Data Types:

- DATE: Date values ranging from January 1, 0001 to December 31, 9999.
- TIME(p): Time values ranging from 00:00:00.0000000 to 23:59:59.9999999.
- DATETIME: Date and time values ranging from January 1, 1753 to December 31, 9999.
- SMALLDATETIME: Date and time values ranging from January 1, 1900 to June 6, 2079.
- DATETIME2(p): Date and time values with increased precision.
- DATETIMEOFFSET(p): Date and time values with time zone offset.

Character String Data Types:

- CHAR(n): Fixed-length character strings.
- VARCHAR(n): Variable-length character strings.
- TEXT: Variable-length character data up to 2GB.

Unicode Character String Data Types:

- NCHAR(n): Fixed-length Unicode character strings.
- NVARCHAR(n): Variable-length Unicode character strings.
- NTEXT: Variable-length Unicode character data up to 2GB.

Creating a Table

A table is a database object used to store data. Data in a table organized in rows and columns.

A table can be created in a database using the CREATE Table statement.

Ex: Student Table

Column Name	Data Type
StudentID	int
FirstName	Varchar(50)
LastName	Varchar(50)
Branch	Varchar(20)
Semester	int

`sp_help EmployeeDetails`

Data Integrity

Data integrity ensures the consistency and correctness of data stored in a database.

Student ID	First Name	Last Name	Branch ID	Email
1	Manav	Sharma	102	manav@test.com
2	Bobby	Pal	100	bobby@test.com
3	Manish	Mehta	104	manish@test.com

Branch ID	Branch Name	Capacity
100	EE	200
101	CSE	200
102	ECE	150
103	ME	150
104	IT	200

Student Table

- Student ID is unique and not null
- First Name is not null
- Branch ID must be valid in Branch table
- Email must be unique but can be left blank

Branch Table

- Branch ID is unique and not null
- Branch Name can be from EE, CSE, IT, ME, ECE
- The default value for capacity column is 200

Creating Constraints

In SQL Server, a constraint is a rule that is applied to a column or a group of columns in a table to ensure data integrity and consistency. Here are some of the most common types of constraints in SQL Server:

1. **NOT NULL** constraint: This constraint ensures that a column cannot contain a NULL value. It requires that the column must have a value before a row can be inserted into the table.
2. **PRIMARY KEY** constraint: This constraint uniquely identifies each row in a table. It is a combination of the NOT NULL and UNIQUE constraints and is used to enforce data integrity.
3. **UNIQUE** constraint: This constraint ensures that a column or a group of columns contain unique values. It prevents duplicate values from being inserted into the table.
4. **FOREIGN KEY** constraint: This constraint creates a relationship between two tables. It ensures that the values in a column or a group of columns in one table match the values in another table's primary key column.
5. **CHECK** constraint: This constraint defines a condition that must be true for the data to be inserted into the table. It can be used to enforce business rules or other constraints on the data.
6. **DEFAULT** constraint: This constraint sets a default value for a column if no value is specified during an insert operation.

Schema

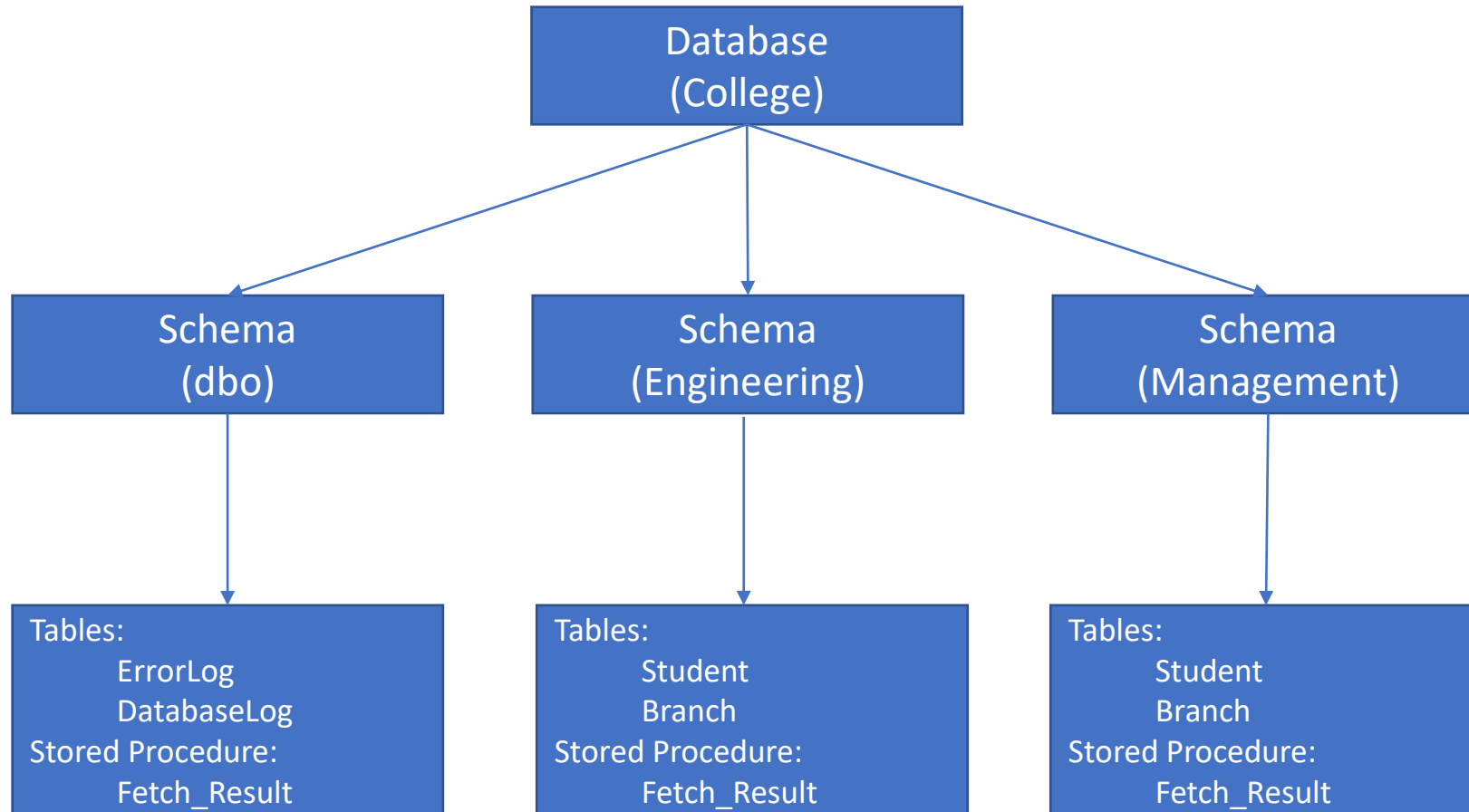
In SQL Server, a schema is a logical container or namespace that holds database objects such as tables, views, stored procedures, functions, and more. It acts as a way to organize and manage database objects within a database.

dbo schema is a default schema created by default in every database

By default, when a user creates an object without specifying a schema, it is created in the default schema associated with the user.

Schema ownership can be transferred from one user to another user in the same database.

Schema



Identity Column

Identity is used for those columns that need automatically generated unique system values. To create an identity column for a table, we use the identity property as follows:

Syntax:

IDENTITY[(seed, increment)]

Seed: It is the starting or the initial value for the IDENTITY column

Increment: It is the step value used to generate the next value for the column. This value can also be negative.

Ex: Use College

```
CREATE TABLE Sports
(
SportsID int IDENTITY(100,1) NOT NULL,
SportsName varchar(50) NOT NULL,
EnrolledStudentID INT
)
```


Delete Table

- The DELETE command is used to remove specific rows from a table based on specified conditions.
- It allows you to selectively delete data while leaving the table structure intact.
- The DELETE command is typically used with a WHERE clause to specify the criteria for row deletion.
- For example, to delete all rows from a table named "MyTable" where the "Status" column is set to 'Inactive,' you would use the following syntax:

```
DELETE FROM MyTable WHERE Status = 'Inactive';
```

TRUNCATE Table

- The TRUNCATE command is used to remove all rows from a table quickly.
- It effectively deletes all the data in the table but keeps the table structure intact, including column definitions, indexes, and constraints.
- TRUNCATE doesn't log individual row deletions and is generally faster when dealing with large tables.
- The syntax for using TRUNCATE is as follows:

Truncate Table Table_Name

DROP Table

To delete a table in SQL, you can use the DROP TABLE statement.

Syntax:

```
DROP TABLE table_name;
```

Note that when you delete a table using DROP TABLE, all data in the table will be permanently deleted and cannot be recovered. Therefore, it is important to make sure that you really want to delete the table before executing the DROP TABLE statement. Additionally, if the table has any dependent objects such as views, stored procedures or triggers, they may also need to be dropped before the table can be deleted.

Drop Vs Delete Vs Truncate Table

	Delete	Truncate	Drop
Command	DML	DDL	DDL
Uses	Deletes few or all rows from a table	All rows from a table	Table structure: data, privileges, constraints, indexes, triggers
Speed	Slow	Fastest	Faster
Constraints	Does not remove	Does not remove	Remove
Free tablespace	No	No	Yes
Can be rolled back	Yes	No	No
Execute triggers	Yes	No	No
Identity Column	Does not reset	Reset	NA



Select * shop
From

Select Apples
Papaya / w/m
From shop



DISTINCT

Where Clause with Comparison and Logical Operators

The WHERE clause in SQL Server is used to filter the rows that are returned in the result set of a SELECT, UPDATE, or DELETE statement.

Comparison Operators	Description	Example
=	Equal to	column_name = value
<>	Not equal to	column_name <> value
>	Greater than	column_name > value
<	Less than	column_name < value
>=	Greater than or equal to	column_name >= value
<=	Less than or equal to	column_name <= value

Logical Operators	Description	Example
AND	True if both conditions are true	condition1 AND condition2
OR	True if at least one condition is true	condition1 OR condition2
NOT	Inverts the result of a single condition	NOT condition

Range and List Operators

Range Operators	Description	Example
BETWEEN	True if a value falls within a range	value BETWEEN start_range AND end_range
NOT BETWEEN	True if a value is outside a range	value NOT BETWEEN start_range AND end_range

List Operators	Description	Example
IN	True if a value matches any item in the list	value IN (item1, item2, item3, ...)
NOT IN	True if a value does not match any item in the list	value NOT IN (item1, item2, item3, ...)

NULL, NOT NULL, Wildcard Operators

Operator	Description	Example
IS NULL	True if the value is NULL	column_name IS NULL
IS NOT NULL	True if the value is not NULL	column_name IS NOT NULL

Wildcard	Description	Example
%	Matches any sequence of characters	column_name LIKE 'abc%'
_	Matches any single character	column_name LIKE 'a_c'

Order By Clause

It allows you to sort the retrieved data in either ascending or descending order based on one or more specified columns.

The syntax for the ORDER BY clause is as follows:

```
SELECT column1, column2, ...  
FROM table_name  
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;
```

By default, if you do not specify a sorting direction, it will use ASC, sorting the results in ascending order.

Example:

```
SELECT first_name, last_name, salary  
FROM employees  
ORDER BY salary DESC, last_name ASC;
```

Aggregate functions

Aggregate functions in SQL Server are used to perform calculations on sets of rows and **return a single result.** They include:

1. COUNT: Returns the number of rows or non-null values in a column.
2. SUM: Calculates the sum of values in a column.
3. AVG: Computes the average value of a column.
4. MIN: Retrieves the minimum value from a column.
5. MAX: Retrieves the maximum value from a column.

EMP ID	F. Name	Salary
1	Amit	10000
2	Ajay	5000
3	Bala	NULL
4	Madhu	9000
5	Neel	10000
6	Prabha	11000
7	Ravi	6000

Group By and Having Clause

In SQL Server, the GROUP BY clause is used to group rows with the same values in one or more columns together. It is often used in combination with aggregate functions (e.g., SUM, COUNT, AVG, MIN, MAX) to perform calculations on the grouped data.

```
SELECT column1, column2, aggregate_function(column3)
FROM table_name
GROUP BY column1, column2;
```

The HAVING clause in SQL Server is used to filter the results of a GROUP BY query based on a specified condition.

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1
HAVING condition;
```

String Functions

In SQL Server, string functions are a set of built-in functions that allow you to manipulate and work with strings (character data) in SQL queries. Here are some commonly used string functions in SQL Server:

1. **CONCAT():** Concatenates two or more strings into a single string.

Example: ``SELECT CONCAT('Hello', ' ', 'World') AS Result; -- Output: "Hello World"``

2. **LEN():** Returns the length (number of characters) of a string.

Example: ``SELECT LEN('SQL Server') AS Length; -- Output: 11``

3. **UPPER():** Converts a string to all uppercase characters.

Example: ``SELECT UPPER('hello') AS Uppercase; -- Output: "HELLO"``

4. **LOWER():** Converts a string to all lowercase characters.

Example: ``SELECT LOWER('WORLD') AS Lowercase; -- Output: "world"``

5. **SUBSTRING():** Extracts a substring from a given string based on the starting position and length.

Example: ``SELECT SUBSTRING('Hello World', 7, 5) AS Substring; -- Output: "World"``

String Functions

6. REPLACE(): Replaces occurrences of a substring within a string with a specified new substring.

Example: ``SELECT REPLACE('Hello, Hello, Hello', 'Hello', 'Hi') AS Replaced; -- Output: "Hi, Hi, Hi"``

7. CHARINDEX(): Returns the starting position of a substring within a string.

Example: ``SELECT CHARINDEX('SQL', 'Welcome to SQL Server') AS Position; -- Output: 13``

8. LTRIM(): Removes leading spaces from a string.

Example: ``SELECT LTRIM(' Trim Spaces') AS Trimmed; -- Output: "Trim Spaces"``

9. RTRIM(): Removes trailing spaces from a string.

Example: ``SELECT RTRIM('Trim Spaces ') AS Trimmed; -- Output: "Trim Spaces"``

10. LEFT(): Returns the left part of a string with a specified length.

Example: ``SELECT LEFT('Hello World', 5) AS LeftPart; -- Output: "Hello"``

11. RIGHT(): Returns the right part of a string with a specified length.

Example: ``SELECT RIGHT('Hello World', 5) AS RightPart; -- Output: "World"``

Date Functions & Data Conversion Functions

In SQL Server, date functions are a set of built-in functions that allow you to work with dates and perform various operations related to date and time. These functions help you extract, manipulate, and format date values within SQL queries. Here are some commonly used date functions in SQL Server:

1. GETDATE(): Returns the current system date and time.

Example: ``SELECT GETDATE() AS CurrentDateTime;``

2. DATEPART(): Extracts a specific part of a date (e.g., year, month, day) as an integer value.

Example: ``SELECT DATEPART(YEAR, '2023-07-28') AS YearValue;``

3. DATEDIFF(): Calculates the difference between two dates, returning the result in a specified date part (e.g., days, months, years).

Example: ``SELECT DATEDIFF(DAY, '2023-07-01', '2023-07-15') AS DaysDifference;``

4. DATEADD(): Adds or subtracts a specified time interval to a date.

Example: ``SELECT DATEADD(MONTH, 3, '2023-07-01') AS FutureDate;``

5. EOMONTH(): Returns the last day of the month for a given date.

Example: ``SELECT EOMONTH('2023-07-15') AS LastDayOfMonth;``

Date Functions & Data Conversion Functions

6. **CONVERT():** Converts an expression to a specified data type, with optional style formatting for date and time values

Example: ``SELECT CONVERT(VARCHAR, '2023-07-28', 101) AS FormattedDate;``

7. **CAST():** Converts an expression to a specified data type.

Example: `SELECT CAST('123' AS INT);` -- Converts '123' to an integer

8. **FORMAT():** Formats a date into a specific format using a .NET Framework format string.

Example: ``SELECT FORMAT(CAST('2023-07-28' AS DATE), 'MM/dd/yyyy') AS FormattedDate;``

9. **DAY(), MONTH(), YEAR():** Extracts the day, month, and year parts from a date, respectively.

Example: ``SELECT DAY('2023-07-28') AS DayValue;``

These date functions in SQL Server can be extremely useful when working with date-related data and performing date manipulations in your SQL queries. Remember to refer to the SQL Server documentation for more details and additional date functions available in the system.

COALESCE function

COALESCE(): Returns the first non-null expression from a list of expressions.

StudentID	Email	PhoneNo
1	aarav.kumar@example.com	1234567890
2	aditi.sharma@example.com	NULL
3	NULL	9876543210
4	aishwarya.singh@example.com	9988776655
5	NULL	NULL
6	ananya.choudhary@example.com	9090909090
7	aniket.yadav@example.com	NULL
8	NULL	9999999999
9	anuj.gupta@example.com	8888888888
10	NULL	NULL



StudentID	Contact Details
1	aarav.kumar@example.com
2	aditi.sharma@example.com
3	9876543210
4	aishwarya.singh@example.com
5	NULL
6	ananya.choudhary@example.com
7	aniket.yadav@example.com
8	9999999999
9	anuj.gupta@example.com
10	NULL

STRING_AGG function

STRING_AGG allows you to concatenate values from multiple rows into a single string, separated by a specified delimiter.

StudentID	FirstName
1	Aarav
2	Aditi
3	Advait
4	Aishwarya
5	Akash
6	Ananya
7	Aniket
8	Anisha
9	Anuj
10	Arjun
11	Aryan
12	Ayush
13	Bhavya
14	Chetan
15	Daksh
16	Diya
17	Tanvi



The basic syntax of the **STRING_AGG** function is as follows:
`STRING_AGG(expression, delimiter)`

All Names
Aarav, Aditi, Advait, Aishwarya, Akash, Ananya, Aniket, Anisha, Anuj, Arjun, Aryan, Ayush, Bhavya, Chetan, Daksh, Diya, Tanvi

Joins

In SQL Server, joins are used to combine rows from two or more tables based on a related column between them.

Here are some common types of joins in SQL Server:

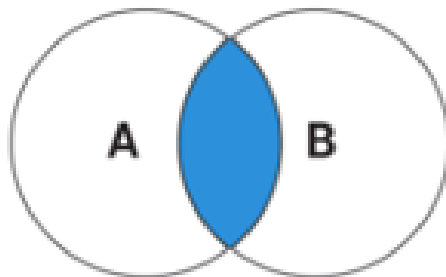
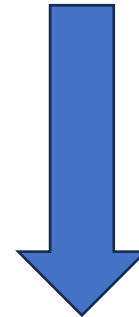
- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

Inner Join

INNER JOIN: Returns only the rows that have matching values in both tables.

StudentID	FirstName	LastName	BranchID	Email
1	Manav	Sharma	100	manish@test.com
2	Manish	Mehta	101	manav@test.com
3	Bobby	Pal	104	bobby@test.com
4	Mohan	NULL	100	NULL
5	Amit	NULL	200	amit@test.com
6	Shivam	Gupta	200	shivam@test.com

BranchID	BranchName	Capacity
100	EE	200
101	CSE	200
102	ECE	150
103	ME	150
104	IT	200



INNER JOIN

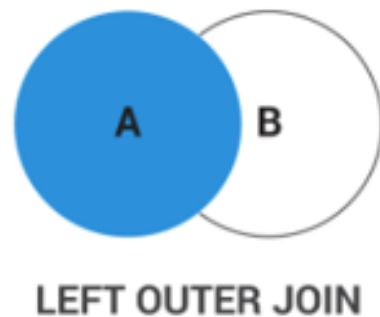
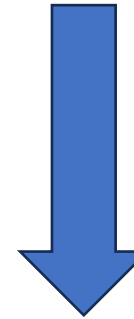
StudentID	FirstName	LastName	BranchName	Capacity
1	Manav	Sharma	EE	200
2	Manish	Mehta	CSE	200
3	Bobby	Pal	IT	200
4	Mohan	NULL	EE	200

Left Join

LEFT JOIN (or LEFT OUTER JOIN): Returns all the rows from the left table and the matching rows from the right table. If there is no match, NULL values are returned for the right table.

StudentID	FirstName	LastName	BranchID	Email
1	Manav	Sharma	100	manish@test.com
2	Manish	Mehta	101	manav@test.com
3	Bobby	Pal	104	bobby@test.com
4	Mohan	NULL	100	NULL
5	Amit	NULL	200	amit@test.com
6	Shivam	Gupta	200	shivam@test.com

BranchID	BranchName	Capacity
100	EE	200
101	CSE	200
102	ECE	150
103	ME	150
104	IT	200



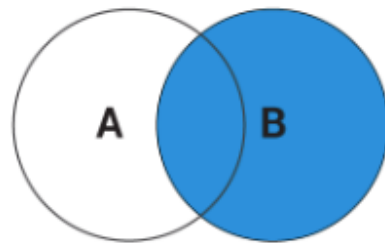
StudentID	FirstName	LastName	BranchName	Capacity
1	Manav	Sharma	EE	200
2	Manish	Mehta	CSE	200
3	Bobby	Pal	IT	200
4	Mohan	NULL	EE	200
5	Amit	NULL	NULL	NULL
6	Shivam	Gupta	NULL	NULL

Right Join

RIGHT JOIN (or RIGHT OUTER JOIN): Returns all the rows from the right table and the matching rows from the left table. If there is no match, NULL values are returned for the left table.

StudentID	FirstName	LastName	BranchID	Email
1	Manav	Sharma	100	manish@test.com
2	Manish	Mehta	101	manav@test.com
3	Bobby	Pal	104	bobby@test.com
4	Mohan	NULL	100	NULL
5	Amit	NULL	200	amit@test.com
6	Shivam	Gupta	200	shivam@test.com

BranchID	BranchName	Capacity
100	EE	200
101	CSE	200
102	ECE	150
103	ME	150
104	IT	200



RIGHT OUTER JOIN

StudentID	FirstName	LastName	BranchName	Capacity
1	Manav	Sharma	EE	200
4	Mohan	NULL	EE	200
2	Manish	Mehta	CSE	200
NULL	NULL	NULL	ECE	150
NULL	NULL	NULL	ME	150
3	Bobby	Pal	IT	200

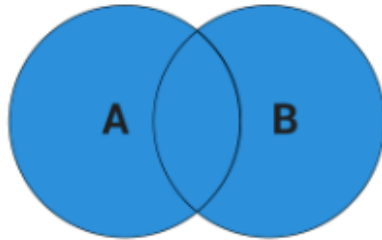
Full Outer Join

FULL JOIN (or FULL OUTER JOIN): Returns all the rows from both tables and combines the results of both the LEFT JOIN and RIGHT JOIN. If there is no match, NULL values are returned for the non-matching side.

StudentID	FirstName	LastName	BranchID	Email
1	Manav	Sharma	100	manish@test.com
2	Manish	Mehta	101	manav@test.com
3	Bobby	Pal	104	bobby@test.com
4	Mohan	NULL	100	NULL
5	Amit	NULL	200	amit@test.com
6	Shivam	Gupta	200	shivam@test.com



BranchID	BranchName	Capacity
100	EE	200
101	CSE	200
102	ECE	150
103	ME	150
104	IT	200



FULL OUTER JOIN

StudentID	FirstName	LastName	BranchName	Capacity
1	Manav	Sharma	EE	200
2	Manish	Mehta	CSE	200
3	Bobby	Pal	IT	200
4	Mohan	NULL	EE	200
5	Amit	NULL	NULL	NULL
6	Shivam	Gupta	NULL	NULL
NULL	NULL	NULL	ECE	150
NULL	NULL	NULL	ME	150

Self Join

SELF JOIN: A self-join is a special type of join in which a table is joined with itself. It allows you to combine rows within a single table based on related values.

```
SELECT e1.EmployeeName, e2.ManagerName  
FROM Employees e1  
INNER JOIN Employees e2 ON e1.ManagerID = e2.EmployeeID;
```

Show the manager's name for each of the employee?

Employee ID	Employee Name	Division	Salary	Manager ID
1	Amit	HR	4000	3
2	Vikas	Finance	5000	4
3	Neha	IT	3900	2
4	Prashant	IT	4000	3



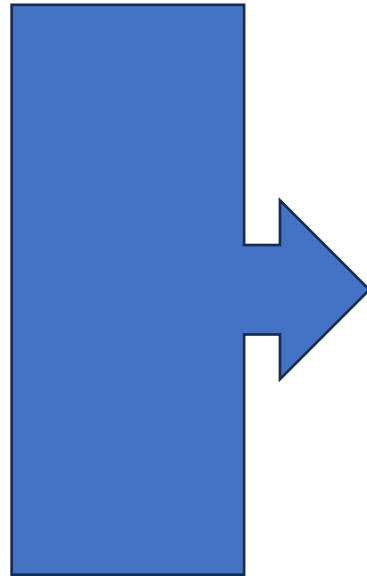
Emp Name	Manager Name
Neha	Vikas
Amit	Neha
Prashant	Neha
Vikas	Prashant

Cross Join

CROSS JOIN: Returns the Cartesian product of both tables, meaning it combines each row from the first table with every row from the second table. It does not require a matching condition.

T Shirt Color
Red
Green
Yellow
Blue

Size Chart
Small
Medium
Large



T Shirt Color	Size Chart
Red	Small
Green	Medium
Yellow	Large
Blue	Small
Red	Medium
Green	Large
Yellow	Small
Blue	Medium
Red	Large
Green	Small
Yellow	Medium
Blue	Large

Set Operator

Set operations in SQL enable you to execute various operations, including merging rows, eliminating duplicates, discovering shared elements, or detecting discrepancies between datasets. These powerful operations facilitate data manipulation and analysis, providing valuable insights and streamlining data processing tasks.

The commonly used set operations in SQL Server are:

- UNION
- UNION ALL
- INTERSECT
- EXCEPT (or MINUS in some databases)

Set Operator – UNION and UNION ALL

UNION:

- The UNION operator is used to combine the result sets of two or more SELECT statements into a single result set.
- It removes duplicate rows from the final result set. If there are any identical rows in the result sets, only one instance of the duplicate row will be included in the combined result.

UNION ALL:

- The UNION ALL operator is used to combine the result sets of two or more SELECT statements into a single result set.
- It retains all rows from the result sets, including duplicates. If there are any identical rows in the result sets, all instances of the duplicate row will be included in the combined result.

Set Operator – INTERSECT and EXCEPT

INTERSECT:

The INTERSECT operator is used to retrieve the common rows between the result sets of two or more SELECT statements. It returns only the rows that appear in all SELECT statements involved in the operation.

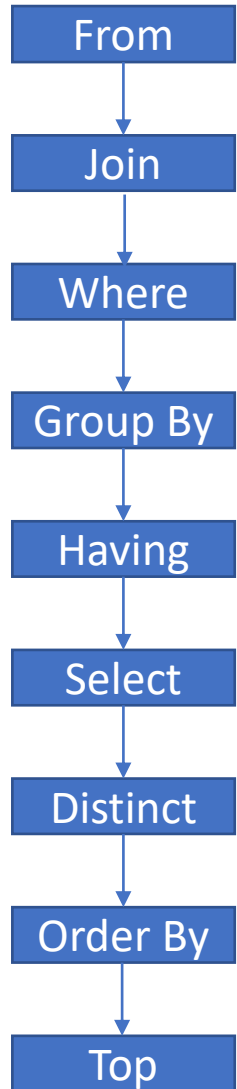
EXCEPT or MINUS:

The EXCEPT operator is used to retrieve the distinct rows from the result set of the first SELECT statement that do not appear in the result set of the second SELECT statement. It effectively removes the rows that exist in both result sets.

Important Notes:

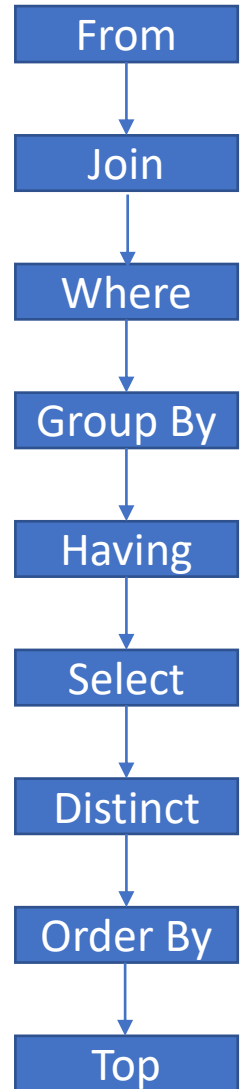
- The Order and data types of corresponding columns in the SELECT statements used with Set Operators must be the same.
- Every SELECT statement within Set Operators must have the same number of columns
- Both INTERSECT and EXCEPT operators remove duplicate rows from the final result set.

Order of execution



1. FROM: The FROM clause specifies the tables or views from which the data will be retrieved.
2. JOIN: If there are any JOIN clauses in the query, the join conditions are evaluated and the tables are combined based on the specified join type (e.g., INNER JOIN, LEFT JOIN, etc.).
3. WHERE: The WHERE clause is used to filter the rows based on specified conditions.
4. GROUP BY: The rows are grouped based on the specified column(s) or expressions.
5. HAVING: The HAVING clause is used to filter the grouped rows based on specified conditions.
6. SELECT: The SELECT clause is used to specify the columns to be included in the result set.
7. DISTINCT: If the DISTINCT keyword is used, duplicate rows are removed from the result set.
8. ORDER BY: It is used to sort the result set based on specified column(s) or expressions.
9. Top: It is used to retrieve a specified number of rows.

Order of execution



Retrieve the top state with the highest number of employees (with more than or equal to 2 employees) where the salary exceeds \$5000.

EMP ID	Salary	State
100	6000	Florida
101	5500	New Jersey
102	4000	Texas
104	4900	New Jersey
105	6500	California
106	6500	Florida
107	6200	New Jersey
108	5500	New Jersey
109	4000	Virginia

```
SELECT TOP 1 State, COUNT(*) AS EmpCount
FROM Employees
WHERE Salary > 5000
GROUP BY State
HAVING COUNT(*) >= 2
ORDER BY EmpCount DESC;
```

CASE Statement

The CASE statement is a conditional statement which is used to implement conditional logic within a query. The CASE statement can be used in various parts of a SQL statement, such as SELECT, UPDATE, DELETE, and more.

CASE Statement	Syntax	Example
Simple CASE Expression (compares an expression with a set of predetermined values and returns a result based on the matching condition.)	<pre>CASE expression WHEN value1 THEN result1 WHEN value2 THEN result2 ... ELSE result END</pre>	<pre>SELECT OrderID, Quantity, CASE Quantity WHEN 0 THEN 'Out of Stock' WHEN 1 THEN 'Low Stock' WHEN 2 THEN 'Medium Stock' ELSE 'High Stock' END AS StockStatus FROM Orders;</pre>
Searched CASE Expression (evaluates multiple Boolean conditions and returns a result based on the first condition that evaluates to true.)	<pre>CASE WHEN condition1 THEN result1 WHEN condition2 THEN result2 ... ELSE result END</pre>	<pre>SELECT OrderID, Quantity, CASE WHEN Quantity = 0 THEN 'Out of Stock' WHEN Quantity > 0 AND Quantity <= 5 THEN 'Low Stock' WHEN Quantity > 5 AND Quantity <= 10 THEN 'Medium Stock' ELSE 'High Stock' END AS StockStatus FROM Orders;</pre>

Subqueries

Subqueries are nested queries within another query, used to retrieve data from a table, which is then utilized in the main query as a condition to further refine the data retrieval; they are primarily employed in the SELECT, FROM, WHERE, and HAVING clauses of a SQL statement and can retrieve either a single value or a set of values.

Types	Description	Example
Single Row Subquery	Retrieve a single value from a subquery	<pre>SELECT * FROM Orders WHERE TotalAmount = (SELECT MAX(TotalAmount) FROM Orders);</pre>
Multiple Rows Subquery	Retrieve multiple values from a subquery.	<pre>SELECT OrderID, CustomerID FROM Orders WHERE CustomerID IN (SELECT CustomerID FROM Customers WHERE Country IN ('INDIA', 'Australia', 'Japan'));</pre>
Correlated Subquery	Reference outer query values in the subquery.	<pre>SELECT StudentName, Grade FROM Students s WHERE Grade > (SELECT AVG(Grade) FROM Grades WHERE Subject = s.Subject);</pre>

CTE (WITH Clause)

A Common Table Expression (CTE) is a temporary named result set that enhances query organization and reuse. It's used in conjunction with a single SELECT, INSERT, UPDATE, or DELETE statement referencing its columns. CTEs allow you to define a query block and apply it multiple times within a larger query.

CTEs are established using the 'WITH' keyword, where you assign a name to the CTE along with its columns if needed. This is followed by 'AS' and a SELECT statement that defines the CTE's result set.

```
WITH CTE_Name (column1, column2, ..., columnN) AS (  
    -- CTE query definition  
    SELECT column1, column2, ..., columnN  
    FROM TableName  
    WHERE condition  
)  
SELECT column1, column2, ..., columnN  
FROM CTE_NAME;
```

In essence, CTEs streamline query design and execution by providing a concise way to create a temporary result set that can be used effectively throughout your broader query.

CTE (WITH Clause)

- Usage: CTEs can be used within the same query where they are defined. They can be referenced like a table or view, allowing for easy integration of complex logic or recursive queries.
- Recursive CTEs: One of the powerful use cases of CTEs is handling recursive queries. A recursive CTE is a CTE that references itself within its definition. It enables querying hierarchical or self-referencing data structures.
- Scoping: CTEs have a local scope, meaning they are only visible within the query where they are defined. They do not persist beyond the execution of the query.
- Reusability: CTEs provide the advantage of defining a complex query block once and referencing it multiple times within the same query, improving query readability and maintainability.

CTEs provide a flexible and concise way to organize and reuse complex queries, making them a valuable feature in SQL Server.

Window Function

- A window function in SQL lets you do calculations on a group of rows called as 'Window' and shows the results alongside each row. It's like looking at a small group of rows at a time and doing math on them without changing the main list of rows you're looking at. The window is defined by the OVER clause and can be further partitioned and ordered based on specific columns
- Here's the basic syntax of a window function:

```
<Window Function> OVER (  
    [PARTITION BY partition_expression, ... ]  
    [ORDER BY sort_expression [ASC | DESC], ... ]  
    [ROWS or RANGE Clause]  
)
```

- ☐ <Window Function>: The function you want to apply, like SUM, AVG, ROW_NUMBER, etc.
- ☐ OVER: This keyword indicates that you're using a window function. Or It defines a window
- ☐ PARTITION BY: Optional. It helps you group rows into partitions based on specific columns.
- ☐ ORDER BY: Optional. It defines the order of rows within each partition.
- ☐ ROWS or RANGE: It limits the rows within the partition by specifying start and end points within the partition.

Window Function

Types of Window Functions:

- ❑ **Aggregate Window Functions:** An aggregate window function in SQL is like a magic calculator that helps you quickly find out something about a group of rows, without changing the list of rows you're looking at. It gives you a special number that summarizes a specific aspect of those rows, and you can see that number next to each row in the group. These are powerful tools that work with multiple rows. They include functions like SUM(), MAX(), MIN(), AVG(), COUNT(), and others.
- ❑ **Ranking Window Functions:** These functions give each row a special rank within a group of rows. Think of them as assigning a unique number to each row. Examples are RANK(), DENSE_RANK(), ROW_NUMBER(), NTILE(), and others. They're useful when you want to arrange data based on certain rules.
- ❑ **Value Window Functions:** These functions let you peek at neighboring rows. You can compare data in a row with data in other rows nearby. Examples are LAG(), LEAD(), FIRST_VALUE(), LAST_VALUE(), and more. They're handy for comparing and predicting data patterns.

Window Function

In SQL's Window Functions, think of the "ROWS BETWEEN" clause as a way to pick a group of rows for your calculations. This group is like a window frame, and you can adjust where it starts and ends.

- ❑ UNBOUNDED PRECEDING: This means your window starts from the very beginning of your data and goes up to the current row.
- ❑ UNBOUNDED FOLLOWING: Your window starts from the current row and goes all the way to the end of your data.
- ❑ N PRECEDING: You choose a fixed number (N), and your window includes N rows before the current row.
- ❑ N FOLLOWING: Similar to the previous one, but your window includes N rows after the current row.
- ❑ CURRENT ROW: Your window is just the current row itself.

By using the "ROWS BETWEEN" clause, you can customize your window to match what you need. It helps you do calculations on a group of rows around the current one. For example, you can calculate things like running totals or averages.

Thank you & Happy Quering!!!