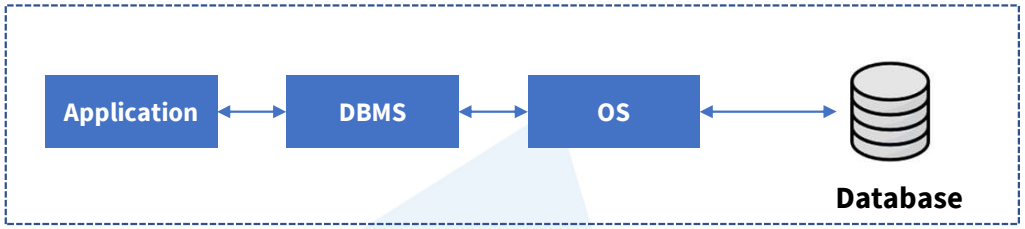
Introduction to Database Concepts

- A database is a collection of related information stored, so that it is available to many users for different purpose. and are used to support Online Transaction Processing (OLTP).
- Database Management Systems (DBMS) store data in the database and enable users and applications to interact with the data.

- A Database Management system (DBMS) is a collection of interrelated data and a set of programs to access those data.



You may also find database characteristics like:

- Security features to ensure the data can only be accessed by authorized users.
- ACID (Atomicity, Consistency, Isolation, Durability) transactions to ensure data integrity.
- Query languages and APIs to easily interact with the data in the database.

Why use a database?

Applications across industries and use cases are built on databases. Many types of data can be stored in databases, including:

- Patient medical records
- Items in an online store
- Financial records
- Online gaming information
- Student grades and scores
- IoT device readings
- Mobile application information etc.

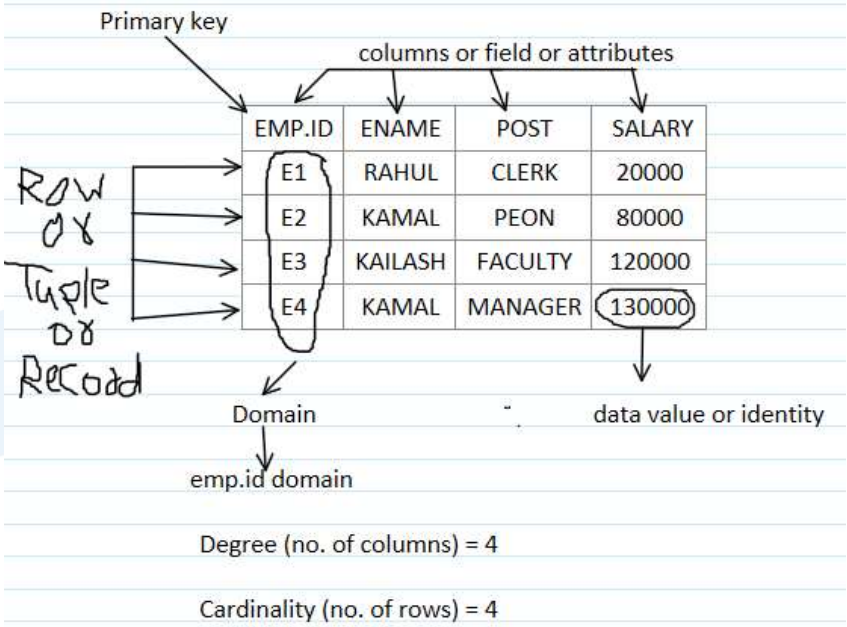
Introduction to Database Concepts

There are many types of Database:

- Relational databases: Oracle, MySQL, Microsoft SQL Server, and PostgreSQL
- Document databases: MongoDB and CouchDB
- Key-value databases: Redis and DynamoDB
- Wide-column stores: Cassandra and HBase
- Graph databases: Neo4j and Amazon Neptune

RDBMS(Relational Database Management system):

- RDBMS stores data in the form of related tables.
- An important feature of relational systems is that a single database can be spread across several tables.
- All modern database management system like MySQL, Oracle, Microsoft SQL Server, PostgreSQL, and SQLite and Microsoft access are based on RDBMS.
- A relational database is the most commonly used database. It contains several tables, and each table has its primary key.
- Due to a collection of an organized set of tables, data can be accessed easily in RDBMS.
- Everything in a relational database is stored in the form of relations.



Introduction to Database Concepts

SQL Commands and statements:-

1. Create Database statement:-

Syntax:-

create database <database_name>;

Example:-

create database Demo;

Query:- If you want to use this database

Syntax:-

use Demo;

2. Create table Statement:-

Syntax:-

create table <table_name> (column1 datatype, column2 datatype----------column n datatype);

Example:-

create table student (Name varchar(20), ID int not null primary key, Address varchar(40));

Describe Table:-

desc student;

Field	Type	Null	Key	Default
Name	Varchar(20)	YES		NULL
ID	Int	NO	PRI	NULL
Address	Varchar(40)	YES		NULL

3. Data Insert into Table:-

You can insert a row in the table by using SQL insert into command. It has two ways:-

a. Syntax:-

insert into table name (columnname1,columnname2, columnname3 - - - - -) values(value1, value2,value3 - - - - - - -);

Example:-

insert into table student (Name, ID, Address) values(“Kailash”, 01, “Delhi”)

b. Syntax:-

insert into table-name values(value1, value2, value3 - - - - - - - - - - value n);

Example:-

insert into table student values(“Kailash”, 01, “Delhi”);

Name	ID	Address
Kailash	01	Delhi

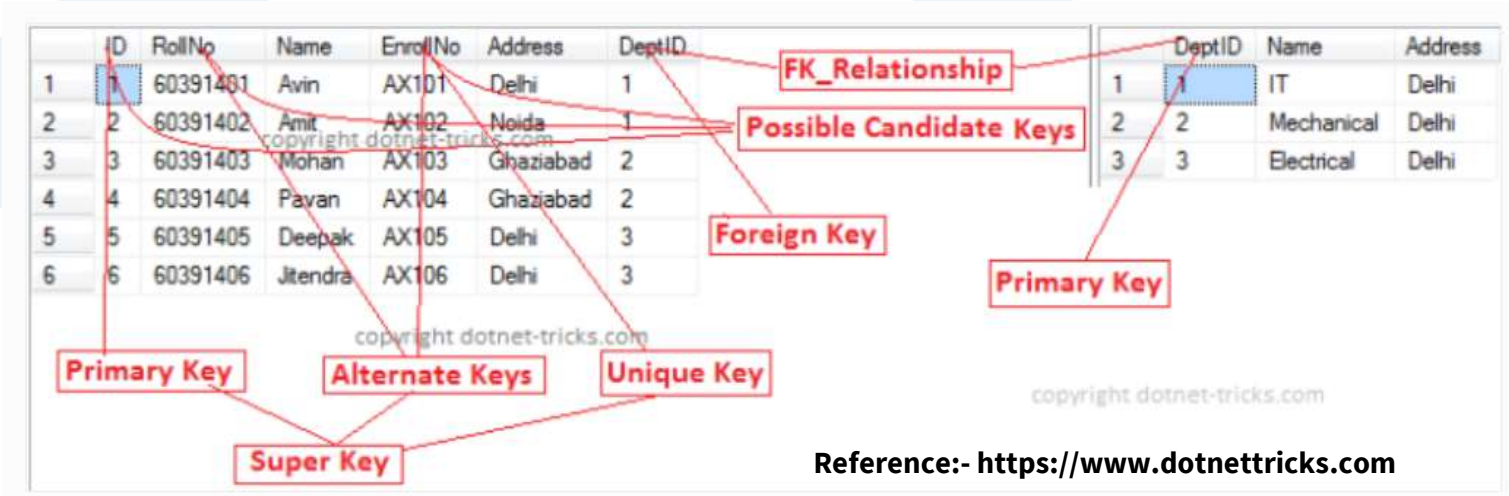
Keys Concept

A key is a single or combination of multiple fields in a table. It is used to fetch or retrieve records/data rows from the data table according to the condition/requirement. Keys are also used to create a relationship among different database tables or views.

Types of Keys:

SQL provides different types of keys such as:

Super Key,
Candidate Key,
Primary Key,
Foreign Key,
Alternate Key,
Compound Key,
Composite Key,



Reference:- https://www.dotnettricks.com

Keys Concept

Super Key:-

A super key is a set of one or more than one key that can be used to identify a record uniquely in a table.
Example: Primary key, Unique key, Alternate key are a subset of Super Keys.

Candidate Key:-

A Candidate Key is a set of one or more fields/columns that can identify a record uniquely in a table.
There can be multiple Candidate Keys in one table. Each Candidate Key can work as a Primary Key.
Example: In the below diagram ID, RollNo and EnrollNo are Candidate Keys since all these three fields can work as Primary Key.

Primary Key:-

A primary key is a set of one or more fields/columns of a table that uniquely identify a record in a database table.
It can not accept null, duplicate values. Only one Candidate Key can be Primary Key.

Foreign Key:-

Foreign Key is a field in a database table that is the Primary key in another table. It can accept multiple nulls and duplicate values.
Example: We can have a DeptID column in the Employee table which is pointing to a DeptID column in a department table where it is a primary key.

Composite Key:-

A composite Key is a combination of more than one field/column of a table. It can be a Candidate key, Primary key.

Unique Key:-

A unique key is a set of one or more fields/columns of a table that uniquely identify a record in a database table.
It is like a Primary key but it can accept only one null value and it can not have duplicate values.

Work with Data Types in SQL

SQL Data Types:-

Data Types are used to represent the nature of the data that can be stored in the database table.

Data Types mainly classified into three categories for every database.

1. String Data Types
2. Numeric Data Types
3. Date and time Data types

1. String Data Types

Data type	Description	Max size	Storage
char(n)	Fixed width character string	8,000 characters	Defined width
varchar(n)	Variable width character string	8,000 characters	2 bytes + number of chars
varchar(max)	Variable width character string	1,073,741,824 characters	2 bytes + number of chars
text	Variable width character string	2GB of text data	4 bytes + number of chars
nchar	Fixed width Unicode string	4,000 characters	Defined width x 2
nvarchar	Variable width Unicode string	4,000 characters	
nvarchar(max)	Variable width Unicode string	536,870,912 characters	
ntext	Variable width Unicode string	2GB of text data	
binary(n)	Fixed width binary string	8,000 bytes	
varbinary	Variable width binary string	8,000 bytes	
varbinary(max)	Variable width binary string	2GB	
image	Variable width binary string	2GB	

Work with Data Types in SQL

2. Numeric Data Types:-

Data type	Description	Storage
bit	Integer that can be 0, 1, or NULL	
tinyint	Allows whole numbers from 0 to 255	1 byte
smallint	Allows whole numbers between -32,768 and 32,767	2 bytes
int	Allows whole numbers between -2,147,483,648 and 2,147,483,647	4 bytes
bigint	Allows whole numbers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807	8 bytes
decimal(p,s)	Fixed precision and scale numbers.Allows numbers from -10^38 +1 to 10^38 –1. The p parameter indicates the maximum total number of digits that can be stored (both to the left and to the right of the decimal point). p must be a value from 1 to 38. Default is 18. The s parameter indicates the maximum number of digits stored to the right of the decimal point. s must be a value from 0 to p. Default value is 0	5-17 bytes
numeric(p,s)	Fixed precision and scale numbers.Allows numbers from -10^38 +1 to 10^38 –1. The p parameter indicates the maximum total number of digits that can be stored (both to the left and to the right of the decimal point). p must be a value from 1 to 38. Default is 18. The s parameter indicates the maximum number of digits stored to the right of the decimal point. s must be a value from 0 to p. Default value is 0	5-17 bytes
smallmoney	Monetary data from -214,748.3648 to 214,748.3647	4 bytes
money	Monetary data from -922,337,203,685,477.5808 to 922,337,203,685,477.5807	8 bytes
float(n)	Floating precision number data from -1.79E + 308 to 1.79E + 308.The n parameter indicates whether the field should hold 4 or 8 bytes. float(24) holds a 4-byte field and float(53) holds an 8-byte field. Default value of n is 53.	4 or 8 bytes
real	Floating precision number data from -3.40E + 38 to 3.40E + 38	4 bytes

Work with Data Types in SQL

3. Date and Time Data Types:-

Data type	Description	Storage
datetime	From January 1, 1753 to December 31, 9999 with an accuracy of 3.33 milliseconds	8 bytes
datetime2	From January 1, 0001 to December 31, 9999 with an accuracy of 100 nanoseconds	6-8 bytes
smalldatetime	From January 1, 1900 to June 6, 2079 with an accuracy of 1 minute	4 bytes
date	Store a date only. From January 1, 0001 to December 31, 9999	3 bytes
time	Store a time only to an accuracy of 100 nanoseconds	3-5 bytes
datetimeoffset	The same as datetime2 with the addition of a time zone offset	8-10 bytes
timestamp	Stores a unique number that gets updated every time a row gets created or modified. The timestamp value is based upon an internal clock and does not correspond to real time. Each table may have only one timestamp variable	

Select Concept

In SQL, SELECT is a statement used to retrieve data from a database table. It allows you to specify which columns you want to retrieve and which rows you want to retrieve based on certain conditions. Here's a basic example of how to use the SELECT statement:

```
CREATE TABLE Employees (
  EmployeeID INT PRIMARY KEY,
  FirstName VARCHAR(50),
  LastName VARCHAR(50),
  Department VARCHAR(50),
  Salary DECIMAL(10, 2)
);
```

Now, suppose you want to retrieve the first and last names of all employees in the "HR" department.
You would use the SELECT statement like this:

```
SELECT FirstName, LastName
FROM Employees
WHERE Department = 'HR';
```

How to Modify Data

Certainly! Below is an example of how you can create a simple table in SQL Server, and then perform various operations such as renaming, deleting, updating, dropping, altering, and truncating.

```
-- Create a new database (if not already created)
CREATE DATABASE MyDatabase;
USE MyDatabase;

-- Create a sample table
CREATE TABLE Employee (
   EmployeeID INT PRIMARY KEY,
   FirstName VARCHAR(50),
   LastName VARCHAR(50),
   Salary DECIMAL(10, 2)
);
```

```
-- Insert some data
INSERT INTO Employee (EmployeeID, FirstName, LastName, Salary)
VALUES (1, 'John', 'Doe', 50000),
     (2, 'Jane', 'Smith', 60000),
     (3, 'Bob', 'Johnson', 55000);
```

```
-- Delete a record
DELETE FROM EmployeeInfo
WHERE EmployeeID = 3;
```

```
-- Rename the table
EXEC sp_rename
'Employee', 'EmployeeInfo';
```

```
-- Update data
UPDATE EmployeeInfo
SET Salary = 65000
WHERE EmployeeID = 2;
```

```
-- Alter the table (add a new column)
ALTER TABLE EmployeeInfo
ADD Department VARCHAR(50);
```

```
-- Truncate the table (remove all data)
TRUNCATE TABLE EmployeeInfo;
```

```
-- Drop the table
DROP TABLE EmployeeInfo;
```

```
-- Drop the database
USE master;
DROP DATABASE MyDatabase;
```

Transact-SQL DDL Statements

- A SQL statement is an atomic unit of work and either completely succeeds or completely fails.
- A SQL statement is a set of instruction that consists of identifiers, parameters, variables, names, data types, and SQL reserved words that compiles successfully.

- **DDL Statements:-**

  - Data Definition Language (DDL) statements defines data structures.
  - Use these statements to create, alter, or drop data structures in a database.

These statements include:

- ALTER
- Collations
- CREATE
- DROP
- RENAME
- UPDATE STATISTICS
- TRUNCATE TABLE

Use Functions and Aggregate Data

In SQL Server, aggregate functions perform a calculation on a set of values and return a single value. Aggregate functions are often used in conjunction with the GROUP BY clause to group rows that have the same values in specified columns into summary rows.

Common Aggregate Functions:

1. COUNT()

Returns the number of rows in a set

```
SELECT COUNT(column_name) FROM table_name;
```

2. SUM()

Returns the sum of the values in a numeric column.

```
SELECT SUM(column_name) FROM table_name;
```

3. AVG()

Returns the average value of a numeric column.

```
SELECT AVG(column_name) FROM table_name;
```

4. MIN() and MAX()

Return the minimum and maximum values in a set, respectively.

```
SELECT MIN(column_name), MAX(column_name) FROM table_name;
```

5. GROUP_CONCAT() / STRING_AGG()

Concatenates strings from multiple rows into a single string. In SQL Server, this is achieved using the 'STRING _AGG' function.

```
SELECT STRING_AGG(column_name, separator) FROM table_name;
```

Use Functions and Aggregate Data

Example with Aggregate Functions:

Let's consider a table named Sales with the following structure:

Creating Tables:-

```
CREATE TABLE Sales (
                SaleID INT PRIMARY KEY,
                ProductID INT,
                Quantity INT,
                SaleDate DATE
);
```

Inserting Data:-

```
INSERT INTO Sales (SaleID, ProductID, Quantity, SaleDate)
VALUES
(1, 101, 5, '2023-09-01'),
(2, 102, 3, '2023-09-02'),
(3, 101, 2, '2023-09-03'),
(4, 103, 7, '2023-09-04');
```

Using Aggregate Functions:

```
-- Count the number of sales
SELECT COUNT(SaleID) AS NumberOfSales FROM Sales;

-- Sum the quantity of products sold
SELECT SUM(Quantity) AS TotalQuantitySold FROM Sales;

-- Find the average quantity sold per sale
SELECT AVG(Quantity) AS AverageQuantityPerSale FROM Sales;

-- Find the minimum and maximum quantity sold in a single sale
SELECT MIN(Quantity) AS MinQuantity, MAX(Quantity) AS MaxQuantity FROM Sales;

-- Concatenate ProductIDs sold, separated by a comma
SELECT STRING_AGG(CAST(ProductID AS VARCHAR), ',') AS ProductsSold FROM Sales;
```

Use Functions and Aggregate Data

Using GROUP BY with Aggregate Functions:

The GROUP BY clause is used to arrange similar data into groups, based on one or more columns. It is often used with aggregate functions to perform calculations on each group of data.

Example:-

```
-- Find the total quantity sold for each product
SELECT ProductID, SUM(Quantity) AS TotalQuantitySold
FROM Sales
GROUP BY ProductID;

-- Count the number of sales and find the average quantity sold per product on each sale date
SELECT SaleDate, COUNT(SaleID) AS NumberOfSales, AVG(Quantity) AS AverageQuantityPerSale
FROM Sales
GROUP BY SaleDate;
```

Using HAVING with GROUP BY:

The HAVING clause is used to filter the results of a GROUP BY clause. It is similar to the WHERE clause but is used to filter groups.

Example:

```
-- Find the product IDs of products that have been sold in a quantity greater than 5
SELECT ProductID
FROM Sales
GROUP BY ProductID
HAVING SUM(Quantity) > 5;
```

SQL Joins Concepts

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.
Let's look at a selection from the "Orders" table:

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

Then, look at a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.
Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

and it will produce something like this:

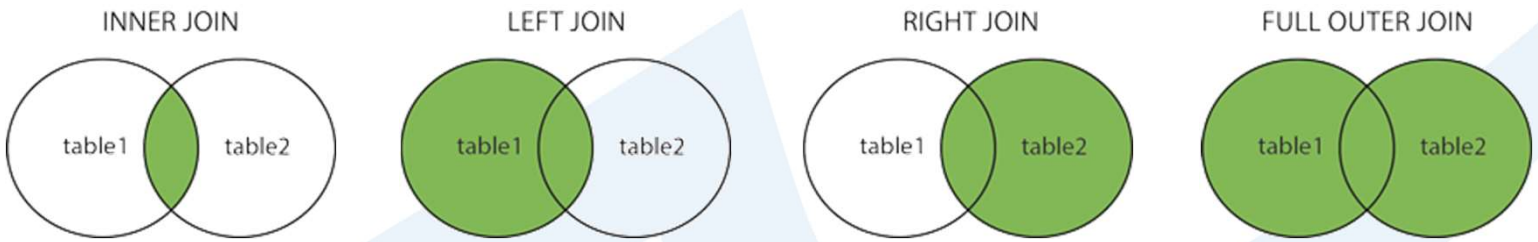
OrderID	CustomerName	OrderDate
10308	Ana Trujillo Emparedados y helados	9/18/1996
10365	Antonio Moreno Taquería	11/27/1996
10383	Around the Horn	12/16/1996
10355	Around the Horn	11/15/1996
10278	Berglunds snabbköp	8/12/1996

SQL Joins Concepts

Different Types of SQL JOINs

Here are the different types of the JOINs in SQL:

- (INNER) JOIN: Returns records that have matching values in both tables
- LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
- RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
- FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table



SQL Joins Concepts

Let us first create two tables "Student" and "Fee" using the following statement:

Create Tables:-

```
CREATE TABLE Student (
 id int PRIMARY KEY IDENTITY,
 admission_no varchar(45) NOT NULL,
 first_name varchar(45) NOT NULL,
 last_name varchar(45) NOT NULL,
 age int,
 city varchar(25) NOT NULL
);

CREATE TABLE Fee (
 admission_no varchar(45) NOT NULL,
 course varchar(45) NOT NULL,
 amount_paid int,
);
```

Inserting the data into Tables:-

```
INSERT INTO Student (admission_no, first_name, last_name, age, city)
VALUES (3354,'Luisa', 'Evans', 13, 'Texas'),
(2135, 'Paul', 'Ward', 15, 'Alaska'),
(4321, 'Peter', 'Bennett', 14, 'California'),
(4213,'Carlos', 'Patterson', 17, 'New York'),
(5112, 'Rose', 'Huges', 16, 'Florida'),
(6113, 'Marielia', 'Simmons', 15, 'Arizona'),
(7555,'Antonio', 'Butler', 14, 'New York'),
(8345, 'Diego', 'Cox', 13, 'California');


INSERT INTO Fee (admission_no, course, amount_paid)
VALUES (3354,'Java', 20000),
(7555, 'Android', 22000),
(4321, 'Python', 18000),
(8345,'SQL', 15000),
(5112, 'Machine Learning', 30000);
```

id	admission_no	first_name	last_name	age	city
1	3354	Luisa	Evans	13	Texas
2	2135	Paul	Ward	15	Alaska
3	4321	Peter	Bennett	14	California
4	4213	Carlos	Patterson	17	New York
5	5112	Rose	Huges	16	Florida
6	6113	Marielia	Simmons	15	Arizona
7	7555	Antonio	Butler	14	New York
8	8345	Diego	Cox	13	California

admission_no	course	amount_paid
3354	Java	20000
7555	Android	22000
4321	Python	18000
8345	SQL	15000
5112	Machine Learning	30000

SQL Joins Concepts

Inner Join:-

```
SELECT Student.admission_no, Student.first_name, Student.last_name, Fee.course, Fee.amount_paid
FROM Student
INNER JOIN Fee
ON Student.admission_no = Fee.admission_no;
```

admission_no	first_name	last_name	course	amount_paid
3354	Luisa	Evans	Java	20000
4321	Peter	Bennett	Python	18000
5112	Rose	Huges	Machine Learning	30000
7555	Antonio	Butler	Android	22000
8345	Diego	Cox	SQL	15000

Left Outer Join:-

```
SELECT Student.admission_no, Student.first_name, Student.last_name, Fee.course, Fee.amount_paid
FROM Student
LEFT OUTER JOIN Fee
ON Student.admission_no = Fee.admission_no;
```

admission_no	first_name	last_name	course	amount_paid
3354	Luisa	Evans	Java	20000
2135	Paul	Ward	NULL	NULL
4321	Peter	Bennett	Python	18000
4213	Carlos	Patterson	NULL	NULL
5112	Rose	Huges	Machine Learning	30000
6113	Marielia	Simmons	NULL	NULL
7555	Antonio	Butler	Android	22000
8345	Diego	Cox	SQL	15000

Right Outer Join:-

```
SELECT Student.admission_no, Student.first_name, Student.last_name, Fee.course, Fee.amount_paid

FROM Student
RIGHT OUTER JOIN Fee
ON Student.admission_no = Fee.admission_no;
```

admission_no	first_name	last_name	course	amount_paid
3354	Luisa	Evans	Java	20000
7555	Antonio	Butler	Android	22000
4321	Peter	Bennett	Python	18000
8345	Diego	Cox	SQL	15000
5112	Rose	Huges	Machine Learning	30000

SQL Joins Concepts

Full Outer Join:-

```
SELECT Student.admission_no, Student.first_name, Student.last_name, Fee.course, Fee.amount_paid
FROM Student
FULL OUTER JOIN Fee
ON Student.admission_no = Fee.admission_no;
```

admission_no	first_name	last_name	course	amount_paid
3354	Luisa	Evans	Java	20000
2135	Paul	Ward	NULL	NULL
4321	Peter	Bennett	Python	18000
4213	Carlos	Patterson	NULL	NULL
5112	Rose	Huges	Machine Learning	30000
6113	Marielia	Simmons	NULL	NULL
7555	Antonio	Butler	Android	22000
8345	Diego	Cox	SQL	15000

Merge Statement

In SQL Server, the MERGE statement, also known as an "upsert," is used to perform insert, update, or delete operations on a target table based on the results of a join with a source table.
You can synchronize the rows in two tables by inserting, updating, or deleting rows in one table based on differences found in the other table.

Here are the different merge options available in SQL Server:

INSERT

If a record from the source table does not exist in the target table, the MERGE statement can insert it into the target table.

```
WHEN NOT MATCHED BY TARGET THEN
        INSERT (column_list) VALUES (value_list)
```

UPDATE

If a record from the source table matches a record in the target table, the MERGE statement can update the record in the target table.

```
WHEN MATCHED THEN UPDATE SET column1 = value1, column2 = value2, ...
```

DELETE

If a record in the target table does not have a corresponding match in the source table, the MERGE statement can delete the record from the target table.

```
WHEN NOT MATCHED BY SOURCE THEN DELETE
```

Merge Statement

Example of MERGE Statement:-

```
MERGE INTO TargetTable AS Target
USING SourceTable AS Source
ON Target.ID = Source.ID

-- Update records
WHEN MATCHED THEN
        UPDATE SET Target.Name = Source.Name, Target.Age = Source.Age

-- Insert records WHEN NOT MATCHED BY TARGET THEN
        INSERT (ID, Name, Age)
        VALUES (Source.ID, Source.Name, Source.Age)
-- Delete records WHEN NOT MATCHED BY SOURCE THEN
        DELETE;
```

Creating Tables:-

Employees Table (Target Table)

```
CREATE TABLE Employees (
        EmployeeID INT PRIMARY KEY,
        FirstName VARCHAR(50),
        LastName VARCHAR(50),
        Salary INT
);
```

Update Table (Source Table)

```
CREATE TABLE Updates (
        EmployeeID INT PRIMARY KEY,
        FirstName VARCHAR(50),
        LastName VARCHAR(50),
        Salary INT
);
```

a. Inserting Data into Employees Table

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, Salary)
VALUES
(1, 'John', 'Doe', 50000),
(2, 'Jane', 'Smith', 55000),
(3, 'Mike', 'Jordan', 60000);
```

b. Inserting Data into Updates Table

```
INSERT INTO Updates (EmployeeID, FirstName, LastName, Salary)
VALUES
(2, 'Jane', 'Smith', 58000), -- Existing employee with updated salary
(3, 'Mike', 'Jordan', 60000), -- Existing employee with no change
(4, 'Chris', 'Evans', 62000); -- New employee
```

Merge Statement

3. Performing MERGE Operation:-

```
MERGE INTO Employees AS Target
USING Updates AS Source
ON Target.EmployeeID = Source.EmployeeID

-- Update existing records
WHEN MATCHED THEN
  UPDATE SET Target.FirstName = Source.FirstName,
        Target.LastName = Source.LastName,
        Target.Salary = Source.Salary

-- Insert new records
WHEN NOT MATCHED BY TARGET THEN
  INSERT (EmployeeID, FirstName, LastName, Salary)
  VALUES (Source.EmployeeID, Source.FirstName, Source.LastName, Source.Salary)

-- Delete records not present in Source table
WHEN NOT MATCHED BY SOURCE THEN
  DELETE;

-- Output the changes
OUTPUT $action, inserted.*, deleted.*;
```

Expected Outcome:-

- ❑ The salary of Jane Smith will be updated to 58000 in the Employees table.
- ❑ Chris Evans will be added to the Employees table.
- ❑ John Doe will be deleted from the Employees table as he is not present in the Updates table.
- ❑ Mike Jordan will remain unchanged in the Employees table.

EmployeeID	FirstName	LastName	Salary
2	Jane	Smith	58000
3	Mike	Jordan	60000
4	Chris	Evans	62000

Work with Null Values in SQL

In SQL there may be some records in a table that do not have values or data for every field and those fields are termed as a NULL value.
NULL values could be possible because at the time of data entry information is not available.
So, NULL values are those values in which there is no data value in the particular field in the table.

Importance of NULL Value:-

It is important to understand that a NULL value differs from a zero value.

A NULL value is used to represent a missing value, but it usually has one of three different interpretations:

- The value unknown (value exists but is not known)
- Value not available (exists but is purposely withheld)
- Attribute not applicable (undefined for this tuple)

How To Test for NULL Values?

SQL allows queries that check whether an attribute value is NULL. Rather than using = or to compare an attribute value to NULL, SQL uses IS and IS NOT.
This is because SQL considers each NULL value as being distinct from every other NULL value, so equality comparison is not appropriate.

Now, consider the following Employee Table:-

Work with Null Values in SQL


```
CREATE TABLE Employee (
  Fname VARCHAR(50),
  Lname VARCHAR(50),
  SSN VARCHAR(11),
  Phoneno VARCHAR(15),
  Salary FLOAT
);

INSERT INTO Employee (Fname, Lname, SSN, Phoneno, Salary)
VALUES
  ('Shubham', 'Thakur', '123-45-6789', '9876543210', 50000.00),
  ('Aman', 'Chopra', '234-56-7890', NULL, 45000.00),
  ('Aditya', 'Arpan', NULL, '8765432109', 55000.00),
  ('Naveen', 'Patnaik', '345-67-8901', NULL, NULL),
  ('Nishant', 'Jain', '456-78-9012', '7654321098', 60000.00);
```

Fname	Lname	SSN	Phoneno	Salary
Shubham	Thakur	123-45-6789	9876543210	50000
Aman	Chopra	234-56-7890		45000
Aditya	Arpan		8765432109	55000
Naveen	Patnaik	345-67-8901		
Nishant	Jain	456-78-9012	7654321098	60000

Suppose we find the Fname and Lname of the Employee having no Super_ssn then the query will be:

```
SELECT Fname, Lname FROM Employee WHERE SSN IS NULL;
```

Fname	Lname
Aditya	Arpan

Work with Null Values in SQL

Now if we find the Count of number of Employees having SSNs.

```
SELECT COUNT(*) AS Count FROM Employee WHERE SSN IS NOT NULL;
```

Output:-

Count
4

Updating NULL Values in a Table:-

We can update the NULL values present in a table using the UPDATE statement in SQL. To do so, we can use the IS NULL operator in the WHERE clause to select the rows with NULL values and then we can set the new value using the SET keyword.

```
UPDATE Employee
SET SSN = '789-01-2345'
WHERE Fname = 'Aditya' AND Lname = 'Arpan';
select* from Employee;
```

Fname	Lname	SSN	Phoneno	Salary
Shubham	Thakur	123-45-6789	9876543210	50000
Aman	Chopra	234-56-7890		45000
Aditya	Arpan	789-01-2345	8765432109	55000
Naveen	Patnaik	345-67-8901		
Nishant	Jain	456-78-9012	7654321098	60000

Explore Set Operators

In SQL Server, set operations allow you to perform operations on two or more result sets to combine, compare, or filter the data based on set theory. The primary set operations in SQL Server are UNION, INTERSECT, and EXCEPT.

1. UNION and UNION ALL

- ❑ UNION combines the result sets of two or more SELECT statements and removes duplicate rows from the combined result set.
- ❑ UNION ALL combines the result sets of two or more SELECT statements and includes all rows, even if there are duplicates.

```
SELECT column1, column2 FROM table1
UNION [ALL]
SELECT column1, column2 FROM table2;
```

2. INTERSECT

- ❑ INTERSECT returns only the rows that are common to the result sets of two or more SELECT statements.

```
SELECT column1, column2 FROM table1
INTERSECT
SELECT column1, column2 FROM table2;
```

3. EXCEPT

- ❑ EXCEPT returns only the rows from the first SELECT statement that do not appear in the result set of the second SELECT statement.

```
SELECT column1, column2 FROM table1
EXCEPT
SELECT column1, column2 FROM table2;
```

Explore Set Operators

Example:

Let's consider two tables: Employees and Contractors. Both tables have similar structures, with columns ID, FirstName, and LastName.

Creating Tables:-

```
CREATE TABLE Employees (
                ID INT PRIMARY KEY,
                FirstName VARCHAR(50),
                LastName VARCHAR(50)
);
CREATE TABLE Contractors (
                ID INT PRIMARY KEY,
                FirstName VARCHAR(50),
                LastName VARCHAR(50)
);
```

Inserting Data:-

```
INSERT INTO Employees (ID, FirstName, LastName)
VALUES (1, 'John', 'Doe'), (2, 'Jane', 'Smith'), (3, 'Mike', 'Jordan');

INSERT INTO Contractors (ID, FirstName, LastName)
VALUES (2, 'Jane', 'Smith'), (4, 'Chris', 'Evans'), (5, 'Robert', 'Downey');
```

Performing Set Operations:-

```
-- UNION: Get all unique individuals from both tables
SELECT FirstName, LastName FROM Employees
UNION
SELECT FirstName, LastName FROM Contractors;

-- UNION ALL: Get all individuals from both tables, including duplicates
SELECT FirstName, LastName FROM Employees
UNION ALL
SELECT FirstName, LastName FROM Contractors;

 -- INTERSECT: Get individuals that are common in both tables
SELECT FirstName, LastName FROM Employees
INTERSECT
SELECT FirstName, LastName FROM Contractors;

-- EXCEPT: Get individuals that are in Employees but not in Contractors
SELECT FirstName, LastName FROM Employees
EXCEPT
SELECT FirstName, LastName FROM Contractors;
```

Work with Subqueries and Apply

In SQL Server, subqueries and the APPLY operator are powerful tools that allow you to perform more complex queries and data manipulations.

1. Subqueries

A subquery is a SQL query nested inside another SQL query. Subqueries can return a single value (scalar subquery), a single row, a single column, or a table (table subquery).

a. Scalar Subquery

Returns a single value.

```
SELECT column_name
FROM table_name
WHERE column_name = (SELECT column_name FROM another_table WHERE condition);
```

b. Table Subquery

Returns a table.

```
SELECT column_name
FROM (SELECT column_name FROM table_name WHERE condition) AS alias_name;
```

c. Correlated Subquery

Refers to columns from the outer query.

```
SELECT column_name
FROM table_name t1
WHERE column_name = (SELECT column_name FROM another_table t2 WHERE t1.id = t2.id);
```

Work with Subqueries and Apply

Example of Subquery:

```
-- Find the average quantity sold for the product that has the maximum quantity sold in a single sale

SELECT AVG(Quantity) AS AverageQuantity
FROM Sales
WHERE ProductID = (SELECT ProductID FROM Sales ORDER BY Quantity DESC LIMIT 1);
```

2. APPLY Operator

The APPLY operator is used to invoke a table-valued function for each row returned by an outer table expression.
SQL Server supports two types of APPLY: CROSS APPLY and OUTER APPLY.

a. CROSS APPLY

Returns only rows that produce a result set from the table-valued function.

```
SELECT column_name
FROM table_name
CROSS APPLY table_valued_function(column_name);
```

b. OUTER APPLY

Returns all rows from the outer table expression and returns NULL for rows that do not produce a result set from the table-valued function.

```
SELECT column_name
FROM table_name
OUTER APPLY table_valued_function(column_name);
```

Work with Subqueries and Apply

Example of APPLY Operator:

Assume we have a table-valued function dbo.GetProductSales that takes a ProductID and returns all sales for that product.

```
-- Using CROSS APPLY to get all sales for each product
SELECT p.ProductID, s.SaleID, s.Quantity
FROM Products p
CROSS APPLY dbo.GetProductSales(p.ProductID) s;

 -- Using OUTER APPLY to get all products and their sales, including products with no sales
SELECT p.ProductID, s.SaleID, s.Quantity
FROM Products p
OUTER APPLY dbo.GetProductSales(p.ProductID) s;
```

Example:-

For Subqueries

```
-- Create a new database (if not already created)
CREATE DATABASE MyDatabase;
USE MyDatabase;

-- Create a sample table
CREATE TABLE Students (
   StudentID INT PRIMARY KEY,
   FirstName VARCHAR(50),
   LastName VARCHAR(50),
   Age INT,
   GPA DECIMAL(3, 2)
);
```

```
-- Insert some data
INSERT INTO Students (StudentID, FirstName, LastName, Age, GPA)
VALUES (1, 'John', 'Doe', 20, 3.75),
     (2, 'Jane', 'Smith', 22, 3.90),
     (3, 'Bob', 'Johnson', 21, 3.60),
     (4, 'Alice', 'Williams', 19, 3.85);
```

Work with Subqueries and Apply


```
-- Example of a subquery: Find students with GPA greater than the average GPA
SELECT FirstName, LastName
FROM Students
WHERE GPA > (SELECT AVG(GPA) FROM Students);
```

```
-- Example of a correlated subquery: Find the youngest student in each GPA range
SELECT FirstName, LastName, Age, GPA
FROM Students s1
WHERE Age = (SELECT MIN(Age) FROM Students s2 WHERE s1.GPA = s2.GPA);
```

```
-- Example of an EXISTS subquery: Find students with a certain condition
SELECT FirstName, LastName
FROM Students
WHERE EXISTS (SELECT 1 FROM Students WHERE Age < 20);
```

```
-- Example of a subquery with aggregation: Find the student with the highest GPA
SELECT FirstName, LastName, GPA
FROM Students
WHERE GPA = (SELECT MAX(GPA) FROM Students);
```

```
-- Drop the table
DROP TABLE Students;
```

```
-- Drop the database
USE master;
DROP DATABASE MyDatabase;
```