**Data Analytics**

**Data analytics** refers to the process of examining, cleaning, transforming, and interpreting data to uncover useful information, support decision-making, and identify patterns or trends. It involves using various techniques, tools, and methodologies to analyze data, derive insights, and solve problems in diverse domains such as business, healthcare, finance, technology, and more.

**Key Components of Data Analytics:**

1. **Data Collection**  
   Gathering data from various sources such as databases, surveys, sensors, or online platforms.
2. **Data Cleaning**  
   Ensuring data quality by removing errors, duplicates, or inconsistencies.
3. **Data Transformation**  
   Converting raw data into a format suitable for analysis through techniques like normalization or aggregation.
4. **Data Analysis**  
   Applying statistical, computational, or machine learning techniques to identify patterns, correlations, and insights.
5. **Data Visualization**  
   Presenting findings through charts, graphs, and dashboards to make insights more comprehensible and actionable.

**Types of Data Analytics:**

1. **Descriptive Analytics**  
   Focuses on summarizing historical data to understand what has happened. *Example: Sales reports showing trends over the past year.*
2. **Diagnostic Analytics**  
   Explores why something happened by identifying causes and correlations. *Example: Analyzing customer churn to find the root cause.*
3. **Predictive Analytics**  
   Uses statistical models and machine learning to forecast future outcomes. *Example: Predicting stock prices or customer behavior.*
4. **Prescriptive Analytics**  
   Recommends actions based on predictive insights to optimize outcomes. *Example: Suggesting marketing strategies to increase sales.*

**Tools Commonly Used:**

* **Programming Languages**: Python, R
* **Data Visualization Tools**: Tableau, Power BI
* **Database Management Systems**: SQL, NoSQL
* **Big Data Platforms**: Hadoop, Spark
* **Statistical Tools**: SAS, SPSS

**Applications of Data Analytics:**

* **Business**: Enhancing customer experience, optimizing operations.
* **Healthcare**: Predicting disease outbreaks, personalizing treatment plans.
* **Finance**: Fraud detection, risk assessment.
* **Sports**: Performance analysis, game strategies.

Data analytics is a cornerstone of decision-making in modern organizations, helping them gain a competitive edge by leveraging data effectively.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Data Analytics**, **Data Science**, and **Artificial Intelligence (AI)** are interrelated fields, but each has distinct goals, methodologies, and applications.

**1. Data Analytics**

**Focus:** Extracting insights from data for decision-making.  
**Goal:** Understand past trends and improve current processes.

**Key Features:**

* **Scope:** Narrower; primarily focused on analyzing existing datasets.
* **Methods:** Statistical analysis, descriptive, diagnostic, predictive, and prescriptive analytics.
* **Tools:** Excel, Tableau, Power BI, SQL, Python (Pandas, Matplotlib), R.
* **Applications:** Business intelligence, performance metrics, reporting, and optimization.

**Example Use Case:**

Analyzing sales data to understand trends and recommend strategies to increase revenue.

**2. Data Science**

**Focus:** A broader discipline that includes data analytics but also involves creating predictive and prescriptive models using advanced techniques.  
**Goal:** Extract knowledge and build systems to solve complex problems.

**Key Features:**

* **Scope:** Broader; includes data collection, cleaning, exploratory data analysis, machine learning, and more.
* **Methods:** Statistics, machine learning, deep learning, data engineering.
* **Tools:** Python (scikit-learn, TensorFlow, PyTorch), R, Hadoop, Spark, Jupyter Notebooks.
* **Applications:** Fraud detection, recommendation systems, natural language processing.

**Example Use Case:**

Building a recommendation system for an e-commerce platform using machine learning algorithms.

**3. Artificial Intelligence (AI)**

**Focus:** Building machines or systems that can mimic human intelligence.  
**Goal:** Create autonomous systems capable of performing tasks without explicit programming.

**Key Features:**

* **Scope:** Encompasses machine learning, deep learning, computer vision, natural language processing, and robotics.
* **Methods:** Neural networks, reinforcement learning, unsupervised and supervised learning.
* **Tools:** TensorFlow, PyTorch, OpenCV, Keras, AI frameworks.
* **Applications:** Autonomous vehicles, virtual assistants, predictive maintenance, and more.

**Example Use Case:**

Developing an AI-powered virtual assistant like Siri or Alexa that understands and responds to user queries.

**Key Differences:**

| **Aspect** | **Data Analytics** | **Data Science** | **Artificial Intelligence** |
| --- | --- | --- | --- |
| **Focus** | Insight generation | Knowledge extraction and modeling | Autonomous intelligence |
| **Methods** | Statistical analysis | Machine learning, data modeling | Neural networks, deep learning |
| **Outcome** | Reports, dashboards | Predictive models, algorithms | Intelligent systems |
| **Complexity** | Low to Medium | Medium to High | High |

**Overlap Between the Fields:**

* **Data Analytics in Data Science:** Data analytics techniques are a subset of the tools data scientists use.
* **Data Science in AI:** Data science provides the data preparation and modeling required to build AI systems.
* **AI and Data Analytics:** AI can enhance data analytics through advanced techniques like predictive modeling and pattern recognition.

**Summary:**

* **Data Analytics** focuses on **what happened** and **why** using existing data.
* **Data Science** answers **what will happen** or **how to solve a problem** using advanced techniques.
* **AI** aims to build systems that **think, learn, and act** autonomously.

All three are interconnected, forming the backbone of modern data-driven industries.

**Role of SQL in Data Analytics**

SQL (Structured Query Language) plays a fundamental role in data analytics by enabling the efficient querying, manipulation, and management of data stored in relational databases. It serves as the backbone for data processing in many analytics workflows

**Importance of Statistics**

**Statistics** is an essential field of study and practice that deals with collecting, analyzing, interpreting, presenting, and organizing data. Its importance lies in its ability to help individuals and organizations make informed decisions based on data rather than intuition or guesswork.

**SQL**

**What is SQL?** SQL is a language used to communicate with databases. It helps you:

* Retrieve data.
* Insert, update, or delete data.
* Create and manage database structures.

**Key Concepts:**

* **Databases**: Structured collections of data.
* **Tables**: Organized collections of rows (records) and columns (fields).
* **Queries**: Instructions written in SQL to interact with data.

**Query Language**

**Query language** is a type of computer programming language used to retrieve, manipulate, and interact with data stored in databases. Query languages are essential tools for managing and accessing structured and unstructured data efficiently.

**Types of Query Languages**

1. **SQL (Structured Query Language)**:
   * Used for relational databases.
   * Examples: MySQL, PostgreSQL, SQLite, SQL Server, Oracle.
   * Common operations:
     + Data Retrieval: SELECT
     + Data Manipulation: INSERT, UPDATE, DELETE
     + Data Definition: CREATE, ALTER, DROP

**Example Query**:

SELECT name, age FROM employees WHERE age > 30;

1. **NoSQL Query Languages**:
   * Used for non-relational databases.
   * Examples: MongoDB Query Language (MQL), Cassandra Query Language (CQL).
   * Designed for flexibility and scalability in handling unstructured or semi-structured data.

**Example Query (MongoDB)**:

db.employees.find({ age: { $gt: 30 } });

1. **Graph Query Languages**:
   * Specialized for graph databases.
   * Examples: Cypher (Neo4j), Gremlin.
   * Used to query nodes, edges, and their relationships.

**Example Query (Cypher)**:

MATCH (e:Employee) WHERE e.age > 30 RETURN e.name, e.age;

1. **XQuery**:
   * A query language designed for XML data.
   * Example Use: Searching and transforming XML documents.

**Example Query**:

for $employee in doc("employees.xml")/employees/employee

where $employee/age > 30

return $employee/name

1. **SPARQL**:
   * Used for querying RDF (Resource Description Framework) data in semantic web applications.
   * Often used with ontologies and linked data.

**Example Query**:

SELECT ?name WHERE {

?employee rdf:type :Employee .

?employee :age ?age .

FILTER(?age > 30)

}

**Common Features of Query Languages**

* **Data Retrieval:** Fetch specific data or datasets.
* **Filtering:** Narrow down results using conditions.
* **Sorting:** Arrange data in ascending or descending order.
* **Aggregation:** Perform calculations like sum, average, count, etc.
* **Joins:** Combine data from multiple sources or tables.

**Importance of Query Languages**

* **Efficient Data Access:** Simplify data retrieval from large datasets.
* **Data Management:** Enable CRUD (Create, Read, Update, Delete) operations.
* **Scalability:** Adapt to different types of databases, from relational to big data systems.
* **Interoperability:** Work with various tools and applications for data analysis and reporting.

Mastering query languages is essential for working with databases and unlocking the power of data analysis. Whether you're working with traditional SQL databases, modern NoSQL solutions, or specialized graph systems, there’s a query language tailored to your needs!

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQL Server**

**SQL Server** is a relational database management system (RDBMS) developed by Microsoft. It is widely used for storing, managing, and retrieving data as requested by software applications. SQL Server supports both structured data in relational databases and semi-structured data formats, making it a versatile platform for a variety of applications.

**Key Features of SQL Server**

1. **Relational Database Engine:**
   * Stores and retrieves data efficiently using tables with rows and columns.
   * Supports **T-SQL (Transact-SQL)**, an extended version of SQL for querying and programming.
2. **Scalability and Performance:**
   * Optimized for handling large-scale, enterprise-level databases.
   * Features such as **in-memory processing**, **indexing**, and **query optimization** enhance performance.
3. **High Availability:**
   * Offers features like **Always On Availability Groups** and **failover clustering** to ensure uptime.
4. **Data Security:**
   * Includes robust security features like **encryption**, **authentication**, and **auditing**.
   * Supports role-based access control to ensure secure data management.
5. **Integration Services:**
   * Built-in tools for **ETL (Extract, Transform, Load)** processes using SQL Server Integration Services (SSIS).
6. **Analytics and Reporting:**
   * Includes tools like SQL Server Reporting Services (SSRS) and SQL Server Analysis Services (SSAS) for data analysis and reporting.
7. **Cross-Platform Support:**
   * SQL Server is available on Windows, Linux, and Docker containers.
8. **Cloud Integration:**
   * Works seamlessly with Azure SQL Database, providing cloud-based database solutions.
9. **Support for Advanced Technologies:**
   * Supports big data clusters, graph databases, JSON, and XML data types.
   * Offers machine learning capabilities with Python and R integration.

**Editions of SQL Server**

SQL Server comes in various editions tailored to different needs:

1. **Enterprise Edition:** For large-scale, mission-critical applications.
2. **Standard Edition:** For medium-sized businesses with moderate database needs.
3. **Express Edition:** Free, lightweight version for small applications.
4. **Developer Edition:** Free version with full functionality for development and testing.

**Core Components of SQL Server**

1. **SQL Server Database Engine:**
   * Core service for processing queries and managing databases.
2. **SQL Server Management Studio (SSMS):**
   * A graphical tool for managing SQL Server databases, writing queries, and monitoring performance.
3. **SQL Server Agent:**
   * Automates tasks like backups, maintenance, and job scheduling.
4. **Integration Services (SSIS):**
   * For data integration and ETL operations.
5. **Reporting Services (SSRS):**
   * For creating and deploying reports.
6. **Analysis Services (SSAS):**
   * For building OLAP cubes and conducting multidimensional data analysis.

**Getting Started with SQL Server**

1. **Install SQL Server:**
   * Download from [Microsoft's SQL Server page](https://www.microsoft.com/en-us/sql-server).
   * Choose an edition based on your requirements (e.g., Developer, Express, Standard).
2. **Install SQL Server Management Studio (SSMS):**
   * Download SSMS from [Microsoft Docs](https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms).
3. **Create Your First Database:**
   * Open SSMS, connect to the server, and use the graphical interface or T-SQL to create databases and tables.

**Use Cases of SQL Server**

* **Enterprise Applications:** Manage large-scale business data with high reliability.
* **Data Warehousing:** Handle analytical workloads for business intelligence.
* **Cloud Solutions:** Integrate with Azure for hybrid and cloud-only scenarios.
* **Web and Mobile Applications:** Provide a robust backend for data storage and retrieval.

**MySQL**

**MySQL** is an open-source relational database management system (RDBMS) based on Structured Query Language (SQL). It is one of the most popular databases globally, used in web applications, data warehousing, and enterprise systems. MySQL is highly favored for its speed, reliability, scalability, and ease of use.

**Key Features of MySQL**

1. **Open Source:**
   * Free to use and open to customization.
   * Offers commercial versions with additional features.
2. **Cross-Platform:**
   * Compatible with various operating systems, including Windows, Linux, and macOS.
3. **High Performance:**
   * Optimized for handling large-scale databases.
   * Supports indexing and query optimization for better speed.
4. **Scalability:**
   * Scales from small projects to large systems with millions of records.
5. **Ease of Use:**
   * Simple to install and manage, with numerous tools and interfaces.
6. **Security:**
   * Features robust security mechanisms, including SSL, encryption, and user management.
7. **Support for Multiple Storage Engines:**
   * Allows flexibility in how data is stored. Common engines include:
     + **InnoDB** (default engine): Supports transactions and foreign keys.
     + **MyISAM**: Fast but lacks support for transactions.
8. **Replication and Clustering:**
   * Supports data replication for high availability and disaster recovery.
   * MySQL Cluster provides distributed databases.
9. **Integration with Programming Languages:**
   * Works well with languages like PHP, Python, Java, and .NET.
10. **Community and Documentation:**
    * Extensive community support and detailed official documentation.

**Editions of MySQL**

1. **Community Edition (Open Source):**
   * Free version for small and medium-scale projects.
2. **MySQL Enterprise Edition:**
   * Paid version with advanced features like data encryption, monitoring, and backup.
3. **MySQL Cluster Edition:**
   * Designed for high availability and real-time performance.
4. **MySQL Embedded (OEM):**
   * For embedding in third-party applications.

**Core Components**

1. **MySQL Server:**
   * Core database server responsible for managing and processing queries.
2. **MySQL Workbench:**
   * A graphical tool for database design, query writing, and administration.
3. **MySQL Command-Line Client:**
   * For executing queries and database operations through a terminal.
4. **Connectors:**
   * APIs to connect MySQL with programming languages and frameworks.

**Getting Started with MySQL**

1. **Download and Install:**
   * Download MySQL from the [official website](https://www.mysql.com/).
   * Install MySQL Server and optionally MySQL Workbench for a GUI-based interface.
2. **Setting Up MySQL:**
   * Configure during installation (e.g., root password, default port 3306).
   * Start the MySQL service.
3. **Connecting to MySQL:**
   * **Command Line:**
   * mysql -u root -p
   * **Workbench:**
     + Open MySQL Workbench and connect using root credentials.

**Use Cases of MySQL**

1. **Web Development:**
   * Powers popular platforms like WordPress, Joomla, and Drupal.
   * Frequently used with PHP in web applications.
2. **E-Commerce Applications:**
   * Manages inventory, customer data, and transactions.
3. **Data Warehousing:**
   * Handles analytical queries for reporting and insights.
4. **Enterprise Applications:**
   * Manages CRM, ERP, and financial systems.

**Comparison with Other Databases**

| **Feature** | **MySQL** | **PostgreSQL** | **SQL Server** |
| --- | --- | --- | --- |
| Open Source | Yes | Yes | No (Paid/Free version) |
| Performance | High for read-heavy | High for complex queries | Enterprise-ready |
| Features | Limited advanced features | Rich features, extensibility | Advanced tools |
| Popular Use Case | Web applications | Complex analytics | Enterprise systems |

**Oracle Database**

Oracle Database is a powerful, multi-model relational database management system (RDBMS) developed by Oracle Corporation. It is widely used for running online transaction processing (OLTP), data warehousing (DW), and mixed database workloads. Oracle Database is known for its robustness, scalability, security features, and extensive support for enterprise-level applications.

**Here are some key features of Oracle Database:**

1. **Multi-Model Support**: Oracle supports not just relational data, but also JSON, XML, spatial data, and more, making it versatile for different use cases.
2. **Scalability**: It can scale from small applications to large enterprise environments. Oracle supports clustering and partitioning to improve performance and availability.
3. **Security**: Oracle provides advanced security features like data encryption, user authentication, and access control to protect sensitive data.
4. **High Availability**: Oracle includes features like Real Application Clusters (RAC), Data Guard, and Flashback technology to ensure continuous uptime and data recovery.
5. **SQL Support**: Oracle uses SQL (Structured Query Language) as its primary language for querying and managing data.
6. **PL/SQL**: A procedural extension of SQL that allows for writing complex scripts, functions, and triggers within the database.
7. **Enterprise Features**: Including advanced backup, recovery, and data replication tools, Oracle is suitable for mission-critical applications.
8. **Cloud Integration**: Oracle provides integration with Oracle Cloud, enabling hybrid cloud environments and offering Oracle Autonomous Database, which uses machine learning to automate routine database tasks.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**SQL\*Plus**

**SQL\*Plus** is an interactive command-line tool provided by Oracle that allows users to execute SQL and PL/SQL queries and commands directly against an Oracle Database. It is one of the most common tools for interacting with Oracle databases, especially for database administrators (DBAs) and developers.

**Key Features of SQL\*Plus:**

1. **Command-Line Interface**:
   * SQL\*Plus is a text-based interface that allows users to run SQL queries, PL/SQL blocks, and scripts.
   * It can be used locally on a machine with Oracle installed or remotely to connect to a database over the network.
2. **Executing SQL Queries**:
   * You can run standard SQL commands such as SELECT, INSERT, UPDATE, DELETE, and others.

**Example of a query**:

SELECT \* FROM employees;

1. **Running PL/SQL Blocks**:
   * SQL\*Plus allows you to execute anonymous PL/SQL blocks or stored procedures.

**Example of an anonymous PL/SQL block**:

DECLARE

v\_employee\_name VARCHAR2(100);

BEGIN

SELECT employee\_name INTO v\_employee\_name FROM employees WHERE employee\_id = 1;

DBMS\_OUTPUT.PUT\_LINE('Employee Name: ' || v\_employee\_name);

END;

1. **Batch Processing**:
   * SQL\*Plus can execute scripts containing multiple SQL or PL/SQL statements. You can run a script file using the @ or START command.

**Example**:

@my\_script.sql

1. **Variables**:
   * SQL\*Plus allows you to define and use variables within scripts. These variables can be used dynamically in queries and scripts.

**Example**:

DEFINE emp\_id = 101;

SELECT \* FROM employees WHERE employee\_id = &emp\_id;

1. **Formatting Output**:
   * SQL\*Plus provides several commands to format the output of your queries. For example, you can set the column width, page size, or whether to show feedback for executed queries.
   * Common formatting commands:
     + SET LINESIZE: Defines the width of the output.
     + SET PAGESIZE: Defines the number of lines per page of output.
     + SET FEEDBACK: Shows or suppresses the message indicating how many rows were returned.

**Example**:

SET LINESIZE 100;

SET PAGESIZE 20;

SELECT employee\_name FROM employees;

1. **User Authentication**:
   * SQL\*Plus requires a valid Oracle username and password to connect to the database. You can connect with the CONNECT command:

CONNECT username/password@hostname:port/servicename

1. **Spooling Output**:
   * SQL\*Plus allows you to save the output of a session or query to a file using the SPOOL command. This is useful for generating reports or saving query results.

**Example**:

SPOOL output.txt;

SELECT \* FROM employees;

SPOOL OFF;

1. **Describing Database Objects**:
   * SQL\*Plus provides the DESCRIBE command to view the structure of a table, view, or other database objects.

**Example**:

DESCRIBE employees;

1. **Exiting SQL\*Plus**:
   * To exit SQL\*Plus, you can use the EXIT or QUIT command:

EXIT;

**Example SQL\*Plus Session:**

CONNECT hr/hr\_password@localhost:1521/xe

Connected.

SET LINESIZE 120

SET PAGESIZE 15

SELECT \* FROM employees;

EMPLOYEE\_ID EMPLOYEE\_NAME HIRE\_DATE

------------ ----------------- ----------

1 John Doe 2023-01-01

2 Jane Smith 2022-05-15

...

EXIT;

Disconnected from Oracle Database 19c

**Differences between SQL\*Plus and Oracle SQL Developer:**

* **SQL\*Plus** is a command-line tool, meaning it's text-based and typically used for running scripts or interacting with Oracle databases in a terminal window.
* **Oracle SQL Developer**, on the other hand, is a graphical user interface (GUI) that allows users to interact with Oracle databases with more advanced features, such as database management, reporting, and visual query design.

SQL\*Plus remains a powerful tool for automating tasks, batch processing, and script execution, particularly for administrators or users comfortable working with the command line.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Connect with System User & Create a new Database User & Log into new User**

In **Microsoft SQL Server (MSSQL)**, there is no direct equivalent to **SQL\*Plus** as it is specific to Oracle databases. However, I will show you how to achieve the tasks you mentioned using **SQLCMD**, which is the SQL Server command-line tool, and how to create a new user, connect as a system user, and log in as the new user in SQL Server.

Here are the steps:

**Steps for Achieving the Tasks in SQL Server:**

1. **Connect with a System User (e.g., sa) in SQL Server**
2. **Create a New Database User**
3. **Log in as the New User**

**1. Connect with System User (e.g., sa) in SQL Server**

To connect to SQL Server using **SQLCMD** as a system user (e.g., sa), use the following command:

sqlcmd -S <server\_name> -U sa -P <password>

* Replace <server\_name> with the name of the SQL Server instance.
* Replace <password> with the password for the sa user (or any user with sufficient privileges).

**Example**:

sqlcmd -S localhost -U sa -P 'your\_password'

This will open a **SQLCMD** session connected to SQL Server as the sa user (system administrator).

**2. Create a New Database User**

To create a new database user in SQL Server, you need to:

1. Create a new login (for server authentication).
2. Create a user in a specific database and associate it with the login.

**Create a Login:**

A login in SQL Server allows a user to access the SQL Server instance. Use the CREATE LOGIN statement to create a new login.

CREATE LOGIN new\_user WITH PASSWORD = 'user\_password';

GO

* Replace new\_user with the desired login name.
* Replace 'user\_password' with a strong password.

**Example:**

CREATE LOGIN HR\_USER WITH PASSWORD = 'HR\_password123';

GO

**Create a Database User:**

Next, create a user in a specific database and associate the new login with that user.

USE <database\_name>;

GO

CREATE USER new\_user FOR LOGIN new\_user;

GO

* Replace <database\_name> with the name of the database where the user will be created.
* Replace new\_user with the login name you created earlier.

**Example:**

USE HRDatabase;

GO

CREATE USER HR\_USER FOR LOGIN HR\_USER;

GO

**3. Grant Permissions to the New User**

Now, grant the necessary permissions or roles to the new user. For example, you can grant read-only or read-write permissions.

* **Granting read-only access:**

ALTER ROLE db\_datareader ADD MEMBER HR\_USER;

GO

* **Granting read and write access:**

ALTER ROLE db\_datawriter ADD MEMBER HR\_USER;

GO

You can also assign specific permissions on tables, views, etc., if needed.

**4. Log in as the New User**

After creating the new user, you can log in to SQL Server using that user's credentials to test the connection. To do this, open a new **SQLCMD** session and use the new user's login and password:

sqlcmd -S <server\_name> -U HR\_USER -P 'HR\_password123'

This will connect to SQL Server as the HR\_USER and allow you to execute queries based on the permissions granted to that user.

**Complete Example:**

1. **Connect as sa (System Administrator):**

sqlcmd -S localhost -U sa -P 'your\_password'

1. **Create a New Login and User:**

-- Create a new login (for server authentication)

CREATE LOGIN HR\_USER WITH PASSWORD = 'HR\_password123';

GO

-- Switch to the target database

USE HRDatabase;

GO

-- Create a new user in the HRDatabase database

CREATE USER HR\_USER FOR LOGIN HR\_USER;

GO

-- Grant roles (optional)

ALTER ROLE db\_datareader ADD MEMBER HR\_USER;

GO

ALTER ROLE db\_datawriter ADD MEMBER HR\_USER;

GO

1. **Log in as the New User:**

sqlcmd -S localhost -U HR\_USER -P 'HR\_password123'

**Summary:**

* **SQL\*Plus** is specific to Oracle, while **SQLCMD** is used for interacting with SQL Server databases.
* You can connect to SQL Server as a system user (e.g., sa) via **SQLCMD**.
* After creating a login and user, you can grant appropriate roles or permissions.
* Once the user is created, you can log in as that user to test the connection and the permissions granted.

**SQL DATABASE**

**1. Create a Database**

To create a new database in MSSQL, use the CREATE DATABASE command.

**SQL Syntax:**

CREATE DATABASE <database\_name>;

GO

**Example:**

CREATE DATABASE EmployeeDB;

GO

This creates a database named EmployeeDB. You can run this command in **SQLCMD** or within SQL Server Management Studio (**SSMS**).

**2. Use a Database**

To use or switch to a specific database, use the USE statement.

**SQL Syntax:**

USE <database\_name>;

GO

**Example:**

USE EmployeeDB;

GO

This switches the context to the EmployeeDB database so subsequent commands will be executed in that database.

**3. Delete or Drop a Database**

To delete (drop) a database, use the DROP DATABASE command. Be cautious, as this will permanently delete the database and all its contents.

**SQL Syntax:**

DROP DATABASE <database\_name>;

GO

**Example:**

DROP DATABASE EmployeeDB;

GO

**Note:**

* Ensure no users are connected to the database before attempting to drop it.
* If you encounter an error due to active connections, use this sequence:

ALTER DATABASE EmployeeDB SET SINGLE\_USER WITH ROLLBACK IMMEDIATE;

GO

DROP DATABASE EmployeeDB;

GO

**4. Rename a Database**

Renaming a database in SQL Server requires using the ALTER DATABASE command. Only one database can be renamed at a time.

**SQL Syntax:**

ALTER DATABASE <old\_database\_name> MODIFY NAME = <new\_database\_name>;

GO

**Example:**

ALTER DATABASE EmployeeDB MODIFY NAME = HRDatabase;

GO

This renames the EmployeeDB database to HRDatabase.

**Note:**

* Ensure no connections are active while renaming a database.
* Use the following to set the database to single-user mode before renaming:

ALTER DATABASE EmployeeDB SET SINGLE\_USER WITH ROLLBACK IMMEDIATE;

GO

ALTER DATABASE EmployeeDB MODIFY NAME = HRDatabase;

GO

ALTER DATABASE HRDatabase SET MULTI\_USER;

GO

**5. Verifying Database Creation**

To verify the list of databases, use the sys.databases system view.

**SQL Syntax:**

SELECT name AS DatabaseName FROM sys.databases;

GO

This will list all databases in the SQL Server instance.

**Summary Table of Commands:**

| **Task** | **Command** |
| --- | --- |
| **Create Database** | CREATE DATABASE <database\_name>; GO |
| **Use Database** | USE <database\_name>; GO |
| **Delete Database** | DROP DATABASE <database\_name>; GO |
| **Rename Database** | ALTER DATABASE <old\_database\_name> MODIFY NAME = <new\_database\_name>; GO |
| **List Databases** | SELECT name AS DatabaseName FROM sys.databases; GO |

**Syntax**

The standard SQL commands to interact with relational databases are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP.

These commands can be classified into groups based on their nature:

**DDL - Data Definition Language:**

CREATE - Creates a new table, a view of a table, or other object in database

ALTER - Modifies an existing database object, such as a table.

DROP - Command Deletes an entire table, a view of a table or other object in the database.

**DML - Data Manipulation Language:**

INSERT - Creates a record UPDATE Modifies records DELETE Command Deletes records

**DCL - Data Control Language**:

GRANT - Gives a privilege to user REVOKE Command Takes back privileges granted from user

**DQL - Data Query Language**:

SELECT - Retrieves certain records from one or more tables

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQL CREATE TABLE Statement:**

The SQL CREATE TABLE statement is used to create a new table.

**Syntax**:

Basic syntax of CREATE TABLE statement is as follows:

CREATE TABLE table\_name(

column1 datatype,

column2 datatype,

column3 datatype,

.....

columnN datatype,

PRIMARY KEY( one or more columns )

);

CREATE TABLE is the keyword telling the database system what you want to do. In this case, you want to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement.

Then in brackets comes the list defining each column in the table and what sort of data type it is. The syntax becomes clearer with an example below.

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement. You can check complete details at Create Table Using another Table.

**Create Table Using another Table**

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement.

The new table has the same column definitions. All columns or specific columns can be selected.

When you create a new table using existing table, new table would be populated using existing values in the old table.

**Syntax**:

The basic syntax for creating a table from another table is as follows:

CREATE TABLE NEW\_TABLE\_NAME AS

SELECT [ column1, column2...columnN ]

FROM EXISTING\_TABLE\_NAME

[ WHERE ]

Here, column1, column2...are the fields of existing table and same would be used to create fields of new table.

**Example**:

Following is an example, which would create a table SALARY using CUSTOMERS table and having fields

customer ID and customer SALARY:

CREATE TABLE SALARY AS SELECT ID, SALARY FROM CUSTOMERS;

This would create new table SALARY, which would have the following records:

+----+----------+

| ID | SALARY |

+----+----------+

| 1 | 2000.00 |

| 2 | 1500.00 |

| 3 | 2000.00 |

| 4 | 6500.00 |

| 5 | 8500.00 |

| 6 | 4500.00 |

| 7 | 10000.00 |

+----+----------+

Example:

Following is an example, which creates a CUSTOMERS table with ID as primary key and NOT NULL are the constraints showing that these fileds can not be NULL while creating records in this table:

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

You can verify if your table has been created successfully by looking at the message displayed by the SQL server, otherwise you can use DESC command as follows:

DESC CUSTOMERS;

+---------+---------------+------+-----+---------+-------+

| Field | Type | Null | Key | Default | Extra |

+---------+---------------+------+-----+---------+-------+

| ID | int(11) | NO | PRI | | |

| NAME | varchar(20) | NO | | | |

| AGE | int(11) | NO | | | |

| ADDRESS | char(25) | YES | | NULL | |

| SALARY | decimal(18,2) | YES | | NULL | |

+---------+---------------+------+-----+---------+-------+

5 rows in set (0.00 sec)

Now, you have CUSTOMERS table available in your database which you can use to store required information related to customers.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQL DROP TABLE Statement**:

SQL DROP TABLE statement is used to remove a table definition and all data, indexes, triggers,

constraints, and permission specifications for that table.

**NOTE**: You have to be careful while using this command because once a table is deleted then all the information available in the table would also be lost forever.

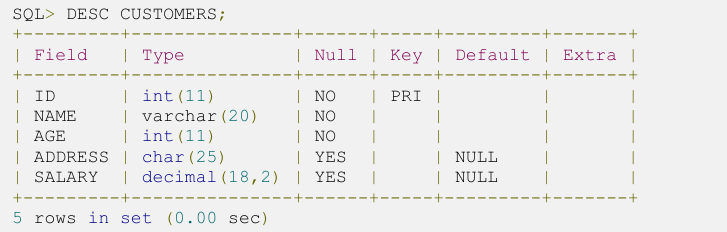
**Syntax**:

Basic syntax of DROP TABLE statement is as follows:

DROP TABLE table\_name;

**Example**:

Let us first verify CUSTOMERS table and then we would delete it from the database:



This means CUSTOMERS table is available in the database, so let us drop it as follows:

DROP TABLE CUSTOMERS;

Query OK, 0 rows affected (0.01 sec)

Now, if you would try DESC command, then you would get error as follows:

DESC CUSTOMERS;

ERROR 1146 (42S02): Table 'TEST.CUSTOMERS' doesn't exist

Here, TEST is database name which we are using for our examples.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQL TRUNCATE TABLE Statement**:

TRUNCATE TABLE command is used to delete complete data from an existing table.

You can also use DROP TABLE command to delete complete table but it would remove complete table structure form the database and you would need to re-create this table once again if you wish you store some data.

**Syntax**:

The basic syntax of TRUNCATE TABLE is as follows:

TRUNCATE TABLE table\_name;

**Example**:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

Following is the example to truncate:

TRUNCATE TABLE CUSTOMERS;

Now, CUSTOMERS table is truncated and following would be the output from SELECT statement:

SELECT \* FROM CUSTOMERS;

Empty set (0.00 sec) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQL INSERT INTO Statement**:

There are two basic syntaxes of INSERT INTO statement as follows:

INSERT INTO TABLE\_NAME (column1, column2, column3,...columnN)]

VALUES (value1, value2, value3,...valueN);

Here, column1, column2,...columnN are the names of the columns in the table into which you want to insert data.

You may not need to specify the column(s) name in the SQL query if you are adding values for all the columns of the table. But make sure the order of the values is in the same order as the columns in the table. The SQL INSERT

INTO syntax would be as follows:

INSERT INTO TABLE\_NAME VALUES (value1,value2,value3,...valueN);

**Example**:

Following statements would create six records in CUSTOMERS table:

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (6, 'Komal', 22, 'MP', 4500.00 );

You can create a record in CUSTOMERS table using second syntax as follows:

INSERT INTO CUSTOMERS

VALUES (7, 'Muffy', 24, 'Indore', 10000.00 );

All the above statements would produce the following records in CUSTOMERS table:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

**Populate one table using another table:**

You can populate data into a table through select statement over another table provided another table has a set of fields, which are required to populate first table. Here is the syntax:

INSERT INTO first\_table\_name [(column1, column2, ... columnN)]

SELECT column1, column2, ...columnN

FROM second\_table\_name

[WHERE condition];

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Select Distinct**

**DISTINCT** keyword is used in conjunction with **SELECT** statement to eliminate all the duplicate

records and fetching only unique records.

There may be a situation when you have multiple duplicate records in a table. While fetching such records, it

makes more sense to fetch only unique records instead of fetching duplicate records.

**Syntax**:

The basic syntax of DISTINCT keyword to eliminate duplicate records is as follows:

SELECT DISTINCT column1, column2,.....columnN

FROM table\_name

WHERE [condition]

**Example**:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

First, let us see how the following SELECT query returns duplicate salary records:

SELECT SALARY FROM CUSTOMERS

ORDER BY SALARY;

This would produce the following result where salary 2000 is coming twice which is a duplicate record from the original table.

+----------+

| SALARY |

+----------+

| 1500.00 |

| 2000.00 |

| 2000.00 |

| 4500.00 |

| 6500.00 |

| 8500.00 |

| 10000.00 |

+----------+

Now, let us use DISTINCT keyword with the above SELECT query and see the result:

SELECT DISTINCT SALARY FROM CUSTOMERS

ORDER BY SALARY;

This would produce the following result where we do not have any duplicate entry:

+----------+

| SALARY |

+----------+

| 1500.00 |

| 2000.00 |

| 4500.00 |

| 6500.00 |

| 8500.00 |

| 10000.00 |

+----------+

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Select Top**

TOP clause is used to fetch a TOP N number or X percent records from a table.

Note: All the databases do not support TOP clause. For example MySQL supports LIMIT clause to fetch limited number of records and Oracle uses ROWNUM to fetch limited number of records.

**Syntax:**

The basic syntax of TOP clause with SELECT statement would be as follows:

SELECT TOP number|percent column\_name(s)

FROM table\_name

WHERE [condition]

**Example**:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

Following is an example on SQL server, which would fetch top 3 records from CUSTOMERS table:

SELECT TOP 3 \* FROM CUSTOMERS;

This would produce the following result:

+----+---------+-----+-----------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+---------+-----+-----------+---------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

+----+---------+-----+-----------+---------+

If you are using MySQL server, then here is an equivalent example:

SELECT \* FROM CUSTOMERS

LIMIT 3;

This would produce the following result:

+----+---------+-----+-----------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+---------+-----+-----------+---------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

+----+---------+-----+-----------+---------+

If you are using Oracle server, then here is an equivalent example:

SELECT \* FROM CUSTOMERS

WHERE ROWNUM <= 3;

This would produce the following result:

+----+---------+-----+-----------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+---------+-----+-----------+---------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

+----+---------+-----+-----------+---------+

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Where Clause**

WHERE clause is used to specify a condition while fetching the data from single table or joining

with multiple tables.

If the given condition is satisfied, then only it returns specific value from the table. You would use WHERE clause to filter the records and fetching only necessary records.

The WHERE clause is not only used in SELECT statement, but it is also used in UPDATE, DELETE statement.

**Syntax**:

The basic syntax of SELECT statement with WHERE clause is as follows:

SELECT column1, column2, columnN

FROM table\_name

WHERE [condition]

You can specify a condition using comparison or logical operators like >, <, =, LIKE, NOT etc. Below examples would make this concept clear.

**Example**:

Consider the CUSTOMERS table having the following records:

| 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 |

+----+----------+-----+-----------+----------+

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table where salary is greater than 2000:

SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE SALARY > 2000;

This would produce the following result:

+----+----------+----------+

| ID | NAME | SALARY |

+----+----------+----------+

| 4 | Chaitali | 6500.00 |

| 5 | Hardik | 8500.00 |

| 6 | Komal | 4500.00 |

| 7 | Muffy | 10000.00 |

+----+----------+----------+

Following is an example, which would fetch ID, Name and Salary fields from the CUSTOMERS table for a customer with name Hardik. Here, it is important to note that all the strings should be given inside single quotes ('') where as numeric values should be given without any quote as in above example:

SELECT ID, NAME, SALARY

FROM CUSTOMERS

WHERE NAME = 'Hardik';

This would produce the following result:

+----+----------+----------+

| ID | NAME | SALARY |

+----+----------+----------+

| 5 | Hardik | 8500.00 |

+----+----------+----------+

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Aliases**

rename a table or a column temporarily by giving another name known as alias.

The use of table aliases means to rename a table in a particular SQL statement. The renaming is a temporary change and the actual table name does not change in the database.

The column aliases are used to rename a table's columns for the purpose of a particular SQL query.

**Syntax**:

The basic syntax of table alias is as follows:

SELECT column1, column2....

FROM table\_name AS alias\_name

WHERE [condition];

The basic syntax of column alias is as follows:

SELECT column\_name AS alias\_name

FROM table\_name

WHERE [condition];

**Example**:

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

(b) Another table is ORDERS as follows:

Simply Easy Learning

+-----+---------------------+-------------+--------+

|OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now, following is the usage of table alias:

SELECT C.ID, C.NAME, C.AGE, O.AMOUNT

FROM CUSTOMERS AS C, ORDERS AS O

WHERE C.ID = O.CUSTOMER\_ID;

This would produce the following result:

+----+----------+-----+--------+

| ID | NAME | AGE | AMOUNT |

+----+----------+-----+--------+

| 3 | kaushik | 23 | 3000 |

| 3 | kaushik | 23 | 1500 |

| 2 | Khilan | 25 | 1560 |

| 4 | Chaitali | 25 | 2060 |

+----+----------+-----+--------+

Following is the usage of column alias:

SELECT ID AS CUSTOMER\_ID, NAME AS CUSTOMER\_NAME

FROM CUSTOMERS

WHERE SALARY IS NOT NULL;

This would produce the following result:

+-------------+---------------+

| CUSTOMER\_ID | CUSTOMER\_NAME |

+-------------+---------------+

| 1 | Ramesh |

| 2 | Khilan |

| 3 | kaushik |

| 4 | Chaitali |

| 5 | Hardik |

| 6 | Komal |

| 7 | Muffy |

+-------------+---------------+

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Update Table**

UPDATE Query is used to modify the existing records in a table.

You can use WHERE clause with UPDATE query to update selected rows, otherwise all the rows would be affected.

**Syntax**:

The basic syntax of UPDATE query with WHERE clause is as follows:

UPDATE table\_name

SET column1 = value1, column2 = value2...., columnN = valueN

WHERE [condition];

You can combine N number of conditions using AND or OR operators.

**Example**:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

Following is an example, which would update ADDRESS for a customer whose ID is 6:

UPDATE CUSTOMERS

SET ADDRESS = 'Pune'

WHERE ID = 6;

Now, CUSTOMERS table would have the following records:

+----+----------+-----+-----------+----------+

| 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 | Pune | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

If you want to modify all ADDRESS and SALARY column values in CUSTOMERS table, you do not need to use WHERE clause and UPDATE query would be as follows:

UPDATE CUSTOMERS

SET ADDRESS = 'Pune', SALARY = 1000.00;

Now, CUSTOMERS table would have the following records:

+----+----------+-----+---------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+---------+---------+

| 1 | Ramesh | 32 | Pune | 1000.00 |

| 2 | Khilan | 25 | Pune | 1000.00 |

| 3 | kaushik | 23 | Pune | 1000.00 |

| 4 | Chaitali | 25 | Pune | 1000.00 |

| 5 | Hardik | 27 | Pune | 1000.00 |

| 6 | Komal | 22 | Pune | 1000.00 |

| 7 | Muffy | 24 | Pune | 1000.00 |

+----+----------+-----+---------+---------+

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Order By**

ORDER BY clause is used to sort the data in ascending or descending order, based on one or

more columns. Some database sorts query results in ascending order by default.

**Syntax**:

The basic syntax of ORDER BY clause is as follows:

SELECT column-list

FROM table\_name

[WHERE condition]

[ORDER BY column1, column2, .. columnN] [ASC | DESC];

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort, that column should be in column-list.

**Example**:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

Following is an example, which would sort the result in ascending order by NAME and SALARY:

SELECT \* FROM CUSTOMERS

ORDER BY NAME, SALARY;

This would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

+----+----------+-----+-----------+----------+

Following is an example, which would sort the result in descending order by NAME:

SELECT \* FROM CUSTOMERS

ORDER BY NAME DESC;

This would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

+----+----------+-----+-----------+----------+

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Group By**

GROUP BY clause is used in collaboration with the SELECT statement to arrange identical data

into groups.

The GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

**Syntax**:

The basic syntax of GROUP BY clause is given below. The GROUP BY clause must follow the conditions in the WHERE clause and must precede the ORDER BY clause if one is used.

SELECT column1, column2

FROM table\_name

WHERE [ conditions ]

GROUP BY column1, column2

ORDER BY column1, column2

**Example**:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

If you want to know the total amount of salary on each customer, then GROUP BY query would be as follows:

SELECT NAME, SUM(SALARY) FROM CUSTOMERS

GROUP BY NAME;

This would produce the following result:

+----------+-------------+

| NAME | SUM(SALARY) |

+----------+-------------+

| Chaitali | 6500.00 |

| Hardik | 8500.00 |

| kaushik | 2000.00 |

| Khilan | 1500.00 |

| Komal | 4500.00 |

| Muffy | 10000.00 |

| Ramesh | 2000.00 |

+----------+-------------+

Now, let us have following table where CUSTOMERS table has the following records with duplicate names:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Ramesh | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | kaushik | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

Now again, if you want to know the total amount of salary on each customer, then GROUP BY query would be as follows:

SELECT NAME, SUM(SALARY) FROM CUSTOMERS

GROUP BY NAME;

This would produce the following result:

+---------+-------------+

| NAME | SUM(SALARY) |

+---------+-------------+

| Hardik | 8500.00 |

| kaushik | 8500.00 |

| Komal | 4500.00 |

| Muffy | 10000.00 |

| Ramesh | 3500.00 |

+---------+-------------+

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Having**

HAVING clause enables you to specify conditions that filter which group results appear in the final

results.

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

**Syntax**:

The following is the position of the HAVING clause in a query:

SELECT

FROM

WHERE

GROUP BY

HAVING

ORDER BY

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used. The following is the syntax of the SELECT statement, including the HAVING clause:

SELECT column1, column2

FROM table1, table2

WHERE [ conditions ]

GROUP BY column1, column2

HAVING [ conditions ]

ORDER BY column1, column2

**Example**:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

Following is the example, which would display record for which similar age count would be more than or equal to 2:

SELECT \*

FROM CUSTOMERS

GROUP BY age

HAVING COUNT(age) >= 2;

This would produce the following result:

+----+--------+-----+---------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+--------+-----+---------+---------+

| 2 | Khilan | 25 | Delhi | 1500.00 |

+----+--------+-----+---------+---------+

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Rank**

The RANK() function is a powerful window function in SQL Server used to assign a rank to each row within a partition of a result set. Unlike other ranking functions, the RANK() function introduces gaps in the ranking sequence for rows with duplicate values, making it particularly useful for scenarios requiring rank differentiation.

**Syntax:**

The Syntax to use the RANK function in SQL Server is

RANK() OVER (  
   [PARTITION BY expression, ]  
   ORDER BY expression (ASC | DESC) );

**SQL RANK Function Example**

Let’s look at some examples of the RANK function in SQL server to understand how it’s working.

Let’s create a demo table on which we will perform the RANK statement. Write the following queries to create a table “demo”.

**CREATE TABLE** demo (Name VARCHAR(10) );

**INSERT INTO** demo (Name)

**VALUES** ('A'), ('B'), ('B'), ('C'), ('C'), ('D'), ('E');

**SELECT** \* **FROM** demo;

**Output**

| **Name** |
| --- |
| A |
| B |
| B |
| C |
| C |
| D |
| E |

In this example, we will use RANK() to assign ranks to the rows in the result set of the demo table.

**Query:**

**SELECT** Name,

**RANK () OVER** (

**ORDER BY** Name

) **AS** Rank\_no

**FROM** demo;

**Output**

| **Name** | **Rank\_no** |  |
| --- | --- | --- |
| A | 1 |  |
| B | 2 |  |
| B | 2 |  |
| C | 4 |  |
| C | 4 |  |
| D | 6 |  |
| E | 7 |  |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Dense\_Rank**

The **DENSE\_RANK()** function in SQL server serves the purpose of **assigning ranks to rows** in a dataset according to specific conditions. Much like the RANK() function, it orders the data based on certain criteria. However, what sets it apart is that it ensures there are no gaps between ranks in cases where multiple rows share the same values. This means that **tied rows receive consecutive ranks** without any interruptions. DENSE\_RANK() is particularly useful when you need a continuous and unbroken sequence of ranks. It's especially valuable in situations where you require a clear and uninterrupted ordering of data, particularly when dealing with tied values. This function is widely employed in scenarios where a seamless and sequential arrangement of data is essential for accurate analysis and reporting.

**Syntax:**

DENSE\_RANK() OVER (PARTITION BY column1, column2, ... ORDER BY sort\_column1, sort\_column2, ...)

**DENSE\_RANK():** The function itself.

**PARTITION BY:** This optional clause divides the result set into partitions or groups, and the ranking is applied within each partition separately. If omitted, the entire result set is treated as a single partition.

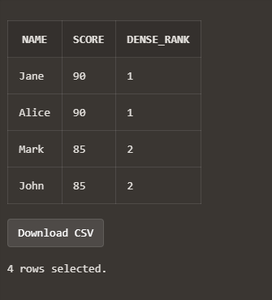
**ORDER BY:** This clause specifies the columns by which the data is sorted to determine the ranking order.

**Example:**

Let's assume we have same table called 'Students' with the following data. In this table, Alice has a score of 90.

| Name | **Score** |
| --- | --- |
| John | 85 |
| Jane | 90 |
| Mark | 85 |
| Alice | 90 |

SELECT  
 Name,  
 Score,  
 DENSE\_RANK() OVER (ORDER BY Score DESC) AS Dense\_Rank  
FROM Students;

**Output:  
**

**Explanation:**

* Both Alice and Jane have the highest score (90), so they share the top dense rank (1).
* John and Mark both have the same score (85), so they share dense rank 2. The next dense rank is 3.

**Differnce between Rank() and Dense Rank()**

| **Criteria** | **Rank()** | **Dense\_Rank()** |
| --- | --- | --- |
| Definition | Assigns a unique rank to each row, leaving no gaps between ranks. If there are ties, the next rank will be skipped. | Assigns a unique rank to each row, leaving no gaps between ranks. If there are ties, the next rank will not be skipped. |
| Example | Data: 10, 20, 20, 30, 40 Rank: 1, 2, 2, 4, 5 | Data: 10, 20, 20, 30, 40 Dense Rank: 1, 2, 2, 3, 4 |
| Behavior with ties | Skips the next rank after a tie. | Does not skip the next rank after a tie. |
| Example with ties | Data: 10, 20, 20, 30, 40 Rank: 1, 2, 2, 4, 5 | Data: 10, 20, 20, 30, 40 Dense Rank: 1, 2, 2, 3, 4 |
| Gaps between ranks | Leaves gaps between ranks after ties. | Does not leave gaps between ranks after ties. |
| Example with gaps | Data: 10, 20, 20, 30, 40 Rank: 1, 2, 2, 4, 5 | Data: 10, 20, 20, 30, 40 Dense Rank: 1, 2, 2, 3, 4 |
| Effect of skipping ranks | May lead to non-sequential ranks. | Always maintains sequential ranks. |
| Application | Useful when you want to differentiate between tied values distinctly. | Useful when you want to maintain a sequential rank without gaps. |
| Syntax | RANK() OVER (PARTITION BY ... ORDER BY ...) | DENSE\_RANK() OVER (PARTITION BY ... ORDER BY ...) |

**CASE Statement**

The CASE expression goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

If there is no ELSE part and no conditions are true, it returns NULL.

**Syntax**

CASE  
    WHEN condition1 THEN result1  
    WHEN condition2 THEN result2  
    WHEN conditionN THEN resultN  
    ELSE result  
END;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_

SELECT OrderID, Quantity,  
CASE  
    WHEN Quantity > 30 THEN 'The quantity is greater than 30'  
    WHEN Quantity = 30 THEN 'The quantity is 30'  
    ELSE 'The quantity is under 30'  
END AS QuantityText  
FROM OrderDetails;

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Select Into**

The SELECT INTO statement copies data from one table into a new table.

**Syntax**

Copy all columns into a new table:

SELECT \*

INTO newtable [IN externaldb]

FROM oldtable

WHERE condition;

Copy only some columns into a new table:

SELECT column1, column2, column3, ...

INTO newtable [IN externaldb]

FROM oldtable

WHERE condition;

The new table will be created with the column-names and types as defined in the old table. You can create new column names using the AS clause.

**SQL SELECT INTO Examples**

The following SQL statement creates a backup copy of Customers:

SELECT \* INTO CustomersBackup2017

FROM Customers;

The following SQL statement uses the IN clause to copy the table into a new table in another database:

SELECT \* INTO CustomersBackup2017 IN 'Backup.mdb'

FROM Customers;

The following SQL statement copies only a few columns into a new table:

SELECT CustomerName, ContactName INTO CustomersBackup2017

FROM Customers;

The following SQL statement copies only the German customers into a new table:

SELECT \* INTO CustomersGermany

FROM Customers

WHERE Country = 'Germany';

The following SQL statement copies data from more than one table into a new table:

SELECT Customers.CustomerName, Orders.OrderID

INTO CustomersOrderBackup2017

FROM Customers

LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;

Tip: SELECT INTO can also be used to create a new, empty table using the schema of another. Just add a WHERE clause that causes the query to return no data:

SELECT \* INTO newtable

FROM oldtable

WHERE 1 = 0;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Alter Table**

ALTER TABLE command is used to add, delete or modify columns in an existing table.

You would also use ALTER TABLE command to add and drop various constraints on an existing table.

**Syntax**:

The basic syntax of ALTER TABLE to add a new column in an existing table is as follows:

ALTER TABLE table\_name ADD column\_name datatype;

The basic syntax of ALTER TABLE to DROP COLUMN in an existing table is as follows:

ALTER TABLE table\_name DROP COLUMN column\_name;

The basic syntax of ALTER TABLE to change the DATA TYPE of a column in a table is as follows:

ALTER TABLE table\_name ALTER COLUMN column\_name datatype;

The basic syntax of ALTER TABLE to add a NOT NULL constraint to a column in a table is as follows:

ALTER TABLE table\_name ALTER column\_name datatype NOT NULL;

The basic syntax of ALTER TABLE to ADD UNIQUE CONSTRAINT to a table is as follows:

ALTER TABLE table\_name

ADD CONSTRAINT MyUniqueConstraint UNIQUE(column1, column2...);

The basic syntax of ALTER TABLE to ADD CHECK CONSTRAINT to a table is as follows:

ALTER TABLE table\_name

ADD CONSTRAINT MyUniqueConstraint CHECK (CONDITION);

The basic syntax of ALTER TABLE to ADD PRIMARY KEY constraint to a table is as follows:

ALTER TABLE table\_name

ADD CONSTRAINT MyPrimaryKey PRIMARY KEY (column1, column2...);

The basic syntax of ALTER TABLE to DROP CONSTRAINT from a table is as follows:

ALTER TABLE table\_name

DROP CONSTRAINT MyUniqueConstraint;

If you're using MySQL, the code is as follows:

ALTER TABLE table\_name

DROP INDEX MyUniqueConstraint;

The basic syntax of ALTER TABLE to DROP PRIMARY KEY constraint from a table is as follows:

ALTER TABLE table\_name

DROP CONSTRAINT MyPrimaryKey;

If you're using MySQL, the code is as follows:

ALTER TABLE table\_name

DROP PRIMARY KEY;

**Example**:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

Following is the example to ADD a new column in an existing table:

ALTER TABLE CUSTOMERS ADD SEX char(1);

Now, CUSTOMERS table is changed and following would be output from SELECT statement:

+----+---------+-----+-----------+----------+------+

| ID | NAME | AGE | ADDRESS | SALARY | SEX |

+----+---------+-----+-----------+----------+------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 | NULL |

| 2 | Ramesh | 25 | Delhi | 1500.00 | NULL |

| 3 | kaushik | 23 | Kota | 2000.00 | NULL |

| 4 | kaushik | 25 | Mumbai | 6500.00 | NULL |

| 5 | Hardik | 27 | Bhopal | 8500.00 | NULL |

| 6 | Komal | 22 | MP | 4500.00 | NULL |

| 7 | Muffy | 24 | Indore | 10000.00 | NULL |

+----+---------+-----+-----------+----------+------+

Following is the example to DROP sex column from existing table:

ALTER TABLE CUSTOMERS DROP SEX;

Now, CUSTOMERS table is changed and following would be output from SELECT statement:

+----+---------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+---------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Ramesh | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | kaushik | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+---------+-----+-----------+----------+

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Order of execution of a Query**

An SQL query comprises various clauses like SELECT, FROM, WHERE, GROUPBY, HAVING, and ORDERBY clauses. Each clause has a specific role in the query. Let’s understand each of them briefly.

When you write any query, your query is processed in the following steps:

1. Getting Data (FROM/JOIN)
2. Row Filter (WHERE)
3. Grouping (GROUP BY)
4. Group Filter (HAVING)
5. Return Expression (SELECT)
6. Order & Paging (ORDER BY & LIMIT/OFFSET)

|  |  |
| --- | --- |
| **Clause** | **Function** |
| **FROM /** [**JOIN**](https://www.shiksha.com/online-courses/articles/sql-joins/) | When you write any query, SQL starts by identifying the tables for the data retrieval and how they are connected. |
| [**WHERE**](https://www.shiksha.com/online-courses/articles/how-to-use-where-clause-in-sql/) | It acts as a filter; it filters the record based on the conditions specified by the users. |
| [**GROUP BY**](https://www.shiksha.com/online-courses/articles/how-to-use-group-by-in-sql/) | The filtered data is grouped based on the specified condition. |
| [**HAVING**](https://www.shiksha.com/online-courses/articles/how-to-use-having-clause-in-sql/) | It is similar to the WHERE clause but applied after grouping the data. |
| [**SELECT**](https://www.shiksha.com/online-courses/articles/select-statement-in-sql/) | The clause selects the columns to be included in the final result. |
| [**DISTINCT**](https://www.shiksha.com/online-courses/articles/how-to-use-distinct-in-sql/) | Remove the duplicate rows from the result. Once you apply this clause, you are only left with distinct records. |
| [**ORDER BY**](https://www.shiksha.com/online-courses/articles/how-to-use-order-by-clause-in-sql/) | It sorts the results (increasing/decreasing/A->Z/Z->A) based on the specified condition. |
| [**LIMIT**](https://www.shiksha.com/online-courses/articles/how-to-use-limit-clause-in-sql/) **/ OFFSET** | It determines the number of records to return and from where to start. |

You have clearly understood the theoretical aspect of the order of execution in SQL until now. Let’s take an example to better understand the concept.

Let’s consider a simple dataset with two tables: **Customers and Orders.**

* The Customers table has 5 columns: **customer\_id, first\_name, last\_name, age, and country.**
* Orders Table has 4 columns: **order\_id, item, amount, customer\_id**

**Customers Table**

| **customer\_id** | **first\_name** | **last\_name** | **age** | **country** |
| --- | --- | --- | --- | --- |
| 1 | John | Doe | 31 | USA |
| 2 | Robert | Luna | 22 | USA |
| 3 | David | Robinson | 22 | UK |
| 4 | John | Reinhardt | 25 | UK |
| 5 | Betty | Doe | 28 | UAE |

**Orders Table**

| **order\_id** | **item** | **amount** | **customer\_id** |
| --- | --- | --- | --- |
| 1 | Keyboard | 400 | 4 |
| 2 | Mouse | 300 | 4 |
| 3 | Monitor | 12000 | 3 |
| 4 | Keyboard | 400 | 1 |
| 5 | Mousepad | 250 | 2 |

**Problem Statement: Find the amount spent by each customer in the USA.**

SELECT Customers.first\_name, Customers.last\_name, SUM(Orders.Amount) as Amount

FROM Customers

JOIN Orders ON Customers.customer\_id = Orders.customer\_id

WHERE Customers.country = 'USA'

GROUP BY Customers.first\_name, Customers.last\_name

ORDER BY Amount DESC;

**Output**

| **first\_name** | **last\_name** | **Amount** |
| --- | --- | --- |
| John | Doe | 400 |
| Robert | Luna | 250 |

**Explanation**

* **FROM**and**JOIN:** We start by identifying the ‘**Customers**‘ and ‘**Orders**‘ tables and joining them on ‘**customer\_id**‘.
* **WHERE:** It will filter the record to include only those where ‘**country**‘ = ‘**USA**‘.
* **GROUP BY:** Group the remaining entries (after filtering by WHERE clause) by ‘**first\_name**‘ and ‘**last\_name**‘.
* **SELECT:** SELECT the ‘**first\_name**‘, ‘**last\_name**‘, and the sum of ‘**Amount**‘ for each group.
* **ORDER BY:** Finally, the result is sorted by ‘**Amount**‘ in descending order.

Now, let’s take an example and reshuffle the order of execution in sql.

**Case-1: Let you want to filter the record based on the ‘Amount’ using the WHERE clause.**

SELECT Customers.first\_name, Customers.last\_name, SUM(Orders.Amount) as Amount

FROM Customers

JOIN Orders ON Customers.customer\_id = Orders.customer\_id

WHERE Orders.Amount >300

GROUP BY Customers.first\_name, Customers.last\_name

ORDER BY Amount DESC;

**Output**

|  |  |  |
| --- | --- | --- |
| **first\_name** | **last\_name** | **Amount** |
| David | Robinson | 12000 |
| John | Doe | 400 |
| John | Reinhardt | 400 |

**Case-2: Filter the record based on the ‘Amount’ using the HAVING clause.**

SELECT Customers.first\_name, Customers.last\_name, SUM(Orders.Amount) as Amount

FROM Customers

JOIN Orders ON Customers.customer\_id = Orders.customer\_id

GROUP BY Customers.first\_name, Customers.last\_name

HAVING Amount > 300

ORDER BY Amount DESC;

**Output**

|  |  |  |
| --- | --- | --- |
| **first\_name** | **last\_name** | **Amount** |
| David | Robinson | 12000 |
| John | Reinhardt | 700 |
| John | Doe | 400 |

Now, let’s see what happened in both cases:

Since the WHERE clause is processed before the SELECT clause in the Order of Execution, so, in the first case, SQL won’t recognize the Amount and will give the error.  
It just filters out the record of the customer who purchased orders greater than 300.

However, the best way to filter the aggregate function is to use the HAVING clause.  
Since the HAVING clause is processed after the GROUP BY clause. So, in the second case, the HAVING clause filters the group to include only those where the total Amount is greater than 300.

**Tips for Writing Efficient SQL Queries**

* The first thing you must know while writing the SQL queries is the correct order of SQL query execution.
  + Since a lot of people think SQL processes queries from top to bottom as they have written.
  + But SQL processes queries in the order: FROM, JOIN, WHERE, GROUP BY, HAVING, SELECT, DISTINCT, ORDER BY, and finally, LIMIT/OFFSET.
* One of the common mistakes is using aliases defined in the SELECT clause within the WHERE clause.
  + Because SQL processes the WHERE clause before the SELECT clause.
* Use the HAVING clause if you need to filter your query based on the result of an aggregate function.
* While joining multiple tables, start with the smallest table or the table that allows you to filter out the most data early on.

**Conclusion**

An SQL query comprises various clauses, such as SELECT, FROM, WHERE, GROUPBY, HAVING, and ORDERBY clauses. Each clause has a specific role in the query.

In this article, we have learned the importance and implications of order of execution (or SQL order of operation) with an example. Understanding the order of operation allows you to write more efficient and accurate queries. It also helps you predict the outcome of queries, troubleshoot issues, and optimize performance.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

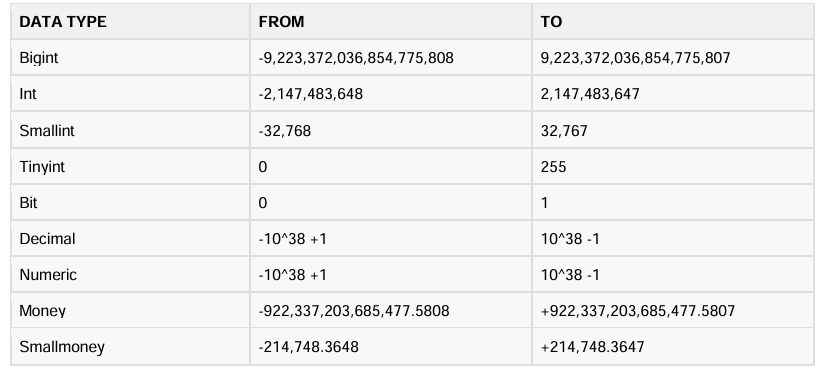
**Data Types**

SQL data type is an attribute that specifies type of data of any object. Each column, variable and expression has related data type in SQL.

You would use these data types while creating your tables. You would choose a particular data type for a table column based on your requirement.

SQL Server offers six categories of data types for your use:

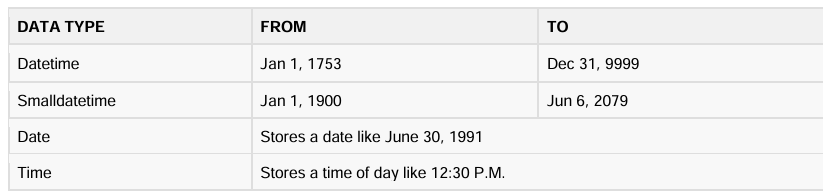
**Exact Numeric Data Types:**



**Approximate Numeric Data Types:**

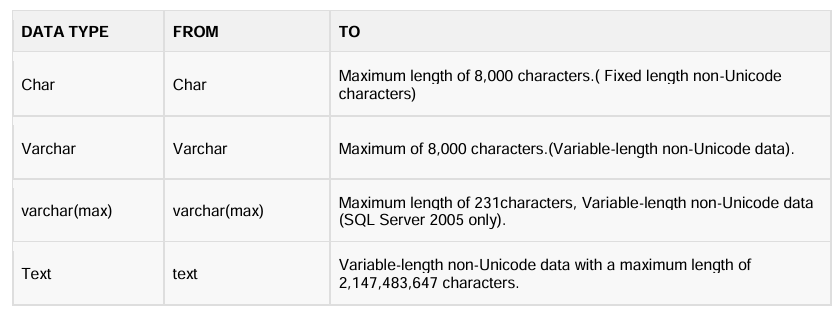


**Date and Time Data Types:**

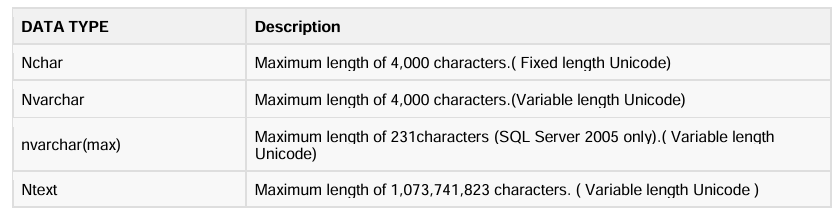


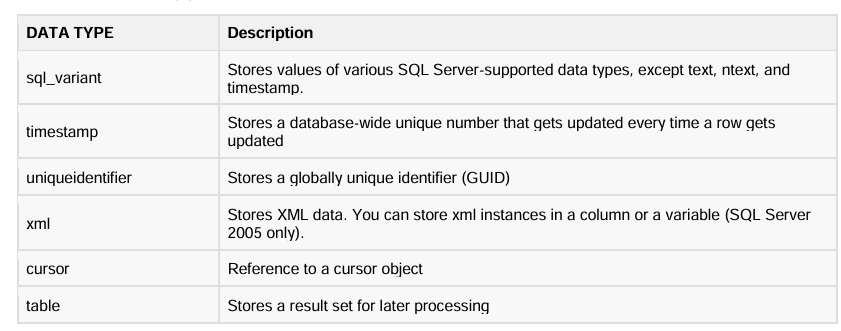
Note: Here, datetime has 3.33 milliseconds accuracy where as smalldatetime has 1 minute accuracy.

**Character Strings Data Types:**

****

**Unicode Character Strings Data Types:**

****

**Misc Data Types:** 

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**SQL Constraints**

Constraints are the rules enforced on data columns on table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Constraints could be column level or table level. Column level constraints are applied only to one column, whereas table level constraints are applied to the whole table.

Following are commonly used constraints available in SQL:

* **NOT NULL Constraint**: Ensures that a column cannot have NULL value.
* **DEFAULT Constraint**: Provides a default value for a column when none is specified.
* **UNIQUE Constraint**: Ensures that all values in a column are different.
* **PRIMARY Key**: Uniquely identified each rows/records in a database table.
* **FOREIGN Key**: Uniquely identified a rows/records in any another database table.
* **CHECK Constraint**: The CHECK constraint ensures that all values in a column satisfy certain conditions.
* **INDEX**: Use to create and retrieve data from the database very quickly.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**NOT NULL Constraint:**

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column.

A NULL is not the same as no data, rather, it represents unknown data.

**Example**:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns, three of which, ID and NAME and AGE, specify not to accept NULLs:

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

If CUSTOMERS table has already been created, then to add a NOT NULL constraint to SALARY column in Oracle and MySQL, you would write a statement similar to the following:

ALTER TABLE CUSTOMERS

ALTER COLUMN SALARY DECIMAL (18, 2) NOT NULL;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**DEFAULT Constraint:**

The DEFAULT constraint provides a default value to a column when the INSERT INTO statement does not provide a specific value.

**Example**:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, SALARY column is set to 5000.00 by default, so in case INSERT INTO statement does not provide a value for this column. then by default this column would be set to 5000.00.

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2) DEFAULT 5000.00,

PRIMARY KEY (ID)

);

If CUSTOMERS table has already been created, then to add a DFAULT constraint to SALARY column, you would write a statement similar to the following:

ALTER TABLE CUSTOMERS

ALTER COLUMN SALARY DECIMAL (18, 2) DEFAULT 5000.00;

**Drop Default Constraint**:

To drop a DEFAULT constraint, use the following SQL:

ALTER TABLE CUSTOMERS

ALTER COLUMN SALARY DROP DEFAULT;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**UNIQUE Constraint:**

The UNIQUE Constraint prevents two records from having identical values in a particular column. In the CUSTOMERS table, for example, you might want to prevent two or more people from having identical age.

**Example**:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, AGE column is set to UNIQUE, so that you can not have two records with same age:

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL UNIQUE,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

If CUSTOMERS table has already been created, then to add a UNIQUE constraint to AGE column, you would write a statement similar to the following:

ALTER TABLE CUSTOMERS

ALTER COLUMN AGE INT NOT NULL UNIQUE;

You can also use following syntax, which supports naming the constraint in multiple columns as well:

ALTER TABLE CUSTOMERS

ADD CONSTRAINT myUniqueConstraint UNIQUE(AGE, SALARY);

**DROP a UNIQUE Constraint**:

To drop a UNIQUE constraint, use the following SQL:

ALTER TABLE CUSTOMERS

DROP CONSTRAINT myUniqueConstraint;

If you are using MySQL, then you can use the following syntax:

ALTER TABLE CUSTOMERS

DROP INDEX myUniqueConstraint; \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PRIMARY Key:**

A primary key is a field in a table which uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values.

A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a composite key.

If a table has a primary key defined on any field(s), then you can not have two records having the same value of that field(s).

Note: You would use these concepts while creating database tables.

Create Primary Key:

Here is the syntax to define ID attribute as a primary key in a CUSTOMERS table.

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

To create a PRIMARY KEY constraint on the "ID" column when CUSTOMERS table already exists, use the following SQL syntax:

ALTER TABLE CUSTOMER ADD PRIMARY KEY (ID);

NOTE: If you use the ALTER TABLE statement to add a primary key, the primary key column(s) must already have been declared to not contain NULL values (when the table was first created).

For defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID, NAME)

);

To create a PRIMARY KEY constraint on the "ID" and "NAMES" columns when CUSTOMERS table already exists, use the following SQL syntax:

ALTER TABLE CUSTOMERS

ADD CONSTRAINT PK\_CUSTID PRIMARY KEY (ID, NAME);

**Delete Primary Key**:

You can clear the primary key constraints from the table, Use Syntax:

ALTER TABLE CUSTOMERS DROP PRIMARY KEY ;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**FOREIGN Key:**

A foreign key is a key used to link two tables together. This is sometimes called a referencing key.

Foreign Key is a column or a combination of columns whose values match a Primary Key in a different table.

The relationship between 2 tables matches the Primary Key in one of the tables with a Foreign Key in the second table.

If a table has a primary key defined on any field(s), then you can not have two records having the same value of that field(s).

**Example**:

Consider the structure of the two tables as follows:

CUSTOMERS table:

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

ORDERS table:

CREATE TABLE ORDERS (

ID INT NOT NULL,

DATE DATETIME,

CUSTOMER\_ID INT references CUSTOMERS(ID),

AMOUNT double,

PRIMARY KEY (ID)

);

If ORDERS table has already been created, and the foreign key has not yet been set, use the syntax for specifying a foreign key by altering a table.

ALTER TABLE ORDERS

ADD FOREIGN KEY (Customer\_ID) REFERENCES CUSTOMERS (ID);

**DROP a FOREIGN KEY Constraint:**

To drop a FOREIGN KEY constraint, use the following SQL:

ALTER TABLE ORDERS

DROP FOREIGN KEY;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CHECK Constraint:**

The CHECK Constraint enables a condition to check the value being entered into a record. If the condition

evaluates to false, the record violates the constraint and isn’t entered into the table.

**Example**:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns. Here, we add a CHECK with AGE column, so that you can not have any CUSTOMER below 18 years:

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL CHECK (AGE >= 18),

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

If CUSTOMERS table has already been created, then to add a CHECK constraint to AGE column, you would write a statement similar to the following:

ALTER TABLE CUSTOMERS

MODIFY AGE INT NOT NULL CHECK (AGE >= 18 );

You can also use following syntax, which supports naming the constraint in multiple columns as well:

ALTER TABLE CUSTOMERS

ADD CONSTRAINT myCheckConstraint CHECK(AGE >= 18);

**DROP a CHECK Constraint:**

To drop a CHECK constraint, use the following SQL. This syntax does not work with MySQL:

ALTER TABLE CUSTOMERS

DROP CONSTRAINT myCheckConstraint;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**INDEX:**

The INDEX is used to create and retrieve data from the database very quickly. Index can be created by using single or group of columns in a table. When index is created, it is assigned a ROWID for each row before it sorts out the data.

Proper indexes are good for performance in large databases, but you need to be careful while creating index.

Selection of fields depends on what you are using in your SQL queries.

**Example**:

For example, the following SQL creates a new table called CUSTOMERS and adds five columns:

CREATE TABLE CUSTOMERS(

ID INT NOT NULL,

NAME VARCHAR (20) NOT NULL,

AGE INT NOT NULL,

ADDRESS CHAR (25) ,

SALARY DECIMAL (18, 2),

PRIMARY KEY (ID)

);

Now, you can create index on single or multiple columns using the following syntax:

CREATE INDEX index\_name

ON table\_name ( column1, column2.....);

To create an INDEX on AGE column, to optimize the search on customers for a particular age, following is the SQL

**syntax**:

CREATE INDEX idx\_age

ON CUSTOMERS ( AGE );

DROP an INDEX Constraint:

To drop an INDEX constraint, use the following SQL:

ALTER TABLE CUSTOMERS

DROP INDEX idx\_age;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQL Server Auto Increment**

In SQL Server, the Auto Increment feature is implemented using the IDENTITY property. This allows the database to automatically generate unique numbers for each new record in the table. Here,

starting\_value – specifies where the numbering starts (in this case, 101).

increment\_value –determines how much the value will increase for each new record (here, it’s 1).

**Example**

We will create a Student table with fields Student\_ID, First\_Name, and Last\_Name, we will auto-generate Student\_ID by using auto-increment and will make it the Primary Key for the table. Let the starting value of IDENTITY be 101 and we will increment the auto-generated key by 1 for each new record.

CREATE TABLE Students(

Student\_ID int IDENTITY(101, 1) PRIMARY KEY,

First\_Name varchar(255),

Last\_Name varchar(255)

);

**Inserting Data with Auto Increment:**

To insert a new record into students table, we will not specify the value of Student\_ID as it will be added automatically.

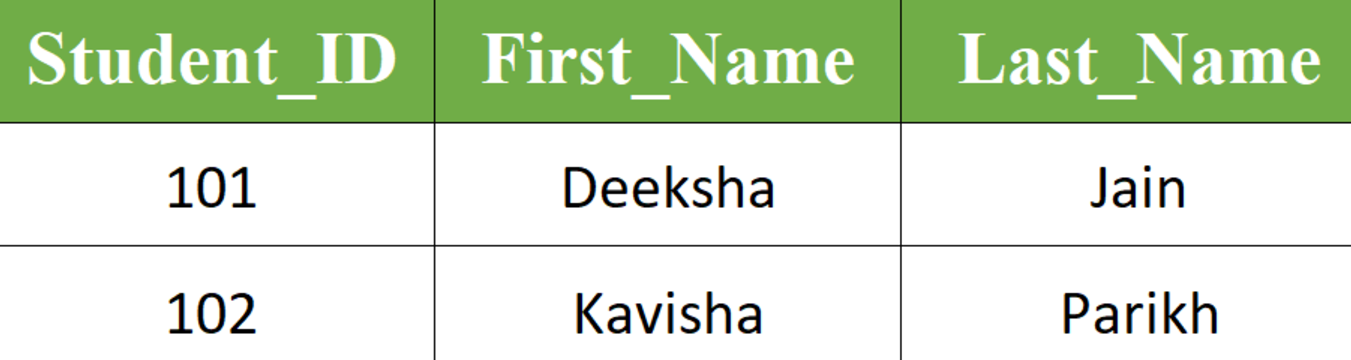
INSERT INTO Students(First\_Name, Last\_Name )

VALUES ('Deeksha', 'Jain');

INSERT INTO Students(First\_Name, Last\_Name )

VALUES ('Kavisha', 'Parikh');

Output



**MySQL Auto Increment**

In MySQL, the AUTO\_INCREMENT keyword is used to set a column as Auto Increment. By default, the counter starts at 1 and increases by 1 for each new row inserted.

**Example**

We will create Students table with fields Student\_ID, First\_Name, Last\_Name, we will auto generate Student\_ID by using auto increment and will make it Primary Key for the table.

CREATE TABLE Students(

Student\_ID int AUTO\_INCREMENT PRIMARY KEY,

First\_Name varchar(255),

Last\_Name varchar(255)

);

**Inserting Data with Auto Increment:**

To insert a new record into students table, we will not specify the value of Student\_ID as it will be added automatically and the first record will have key as 1 and key for every subsequent record will increase by 1.

INSERT INTO Students(First\_Name, Last\_Name )

VALUES ('Anish', 'Jain');

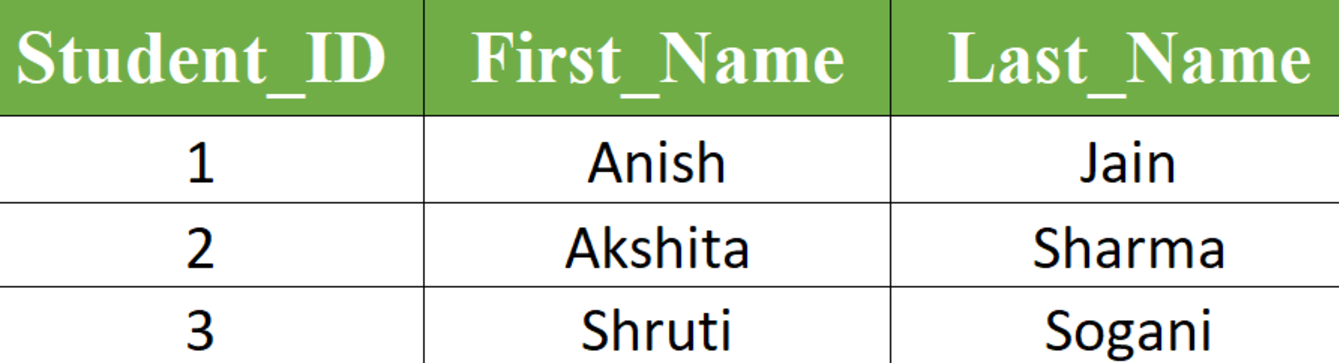
INSERT INTO Students(First\_Name, Last\_Name )

VALUES ('Akshita', 'Sharma');

INSERT INTO Students(First\_Name, Last\_Name )

VALUES ('Shruti', 'Sogani');

Output



Changing Auto Increment Start Value

To change the default starting value we can use ALTER TABLE command as follows:

ALTER TABLE Students AUTO\_INCREMENT = new\_value;

Setting the Interval for AUTO\_INCREMENT in MySQL

In MySQL, we can also modify the interval at which the AUTO\_INCREMENT value increases. By default, it increments by 1, but we can change it using the auto\_increment\_increment system variable.

To change the increment value (i.e., the step by which the value increments), we can set the auto\_increment\_increment variable and new\_interval\_value is the interval value we would like to use.

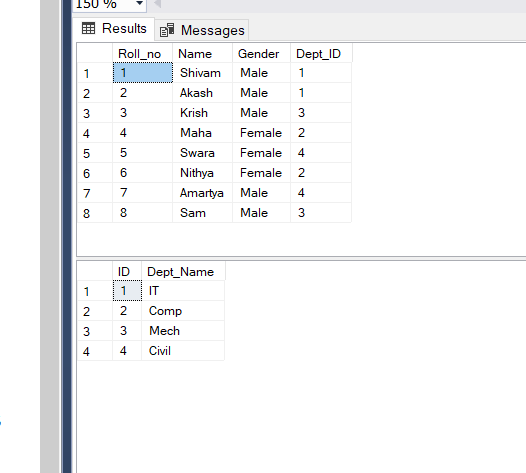
SET @@auto\_increment\_increment = new\_interval\_value;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Cascading Referential Integrity Constraints in SQL Server Management Studio**

In the Microsoft SQL server if we want to delete any record or column from one table but that record or column is a foreign key for another table then we will get the error to solve this problem we use **Cascading referential integrity constraint.**

It allows the actions that SQL Server should take when a user tries to delete or update a key to which an existing foreign key points. Suppose we have two tables, the first table’s name is**“Student”**and the second is **“Department”**as follows,



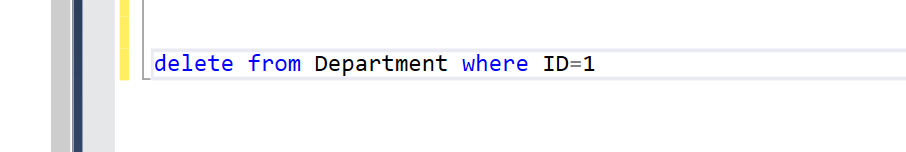
*Student and Department Table*

In the **Student**table **“Roll\_no”** is the **primary key** which identifies each record uniquely and in the **Department**table “**ID**” is the**primary key**. Here the **foreign key**is **Dept\_ID**in the **Student**table get the reference from the **primary key ID**from the **Department**table.

In the **Department**table, if you delete the row with **ID=1** then the records with **Roll\_no=1 and 2** from the **Student**table become an unsupported records which is also called as **Orphan Record.**Consequently, you won’t be able to determine the Department of this row. Therefore, Cascading Referential Integrity Constraint (CRI) can be used to specify what SQL Server should do in this situation. By default, the DELETE or UPDATE statement is rolled back and we receive an error.

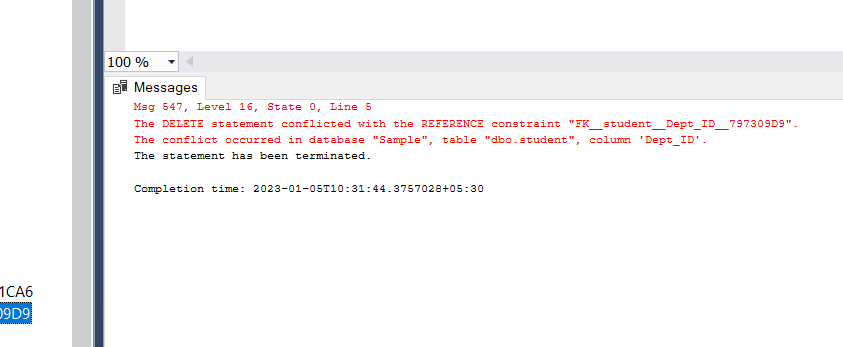
**Query:**

DELETE from Department WHERE ID =1



*Query*

**Output**:



*output*

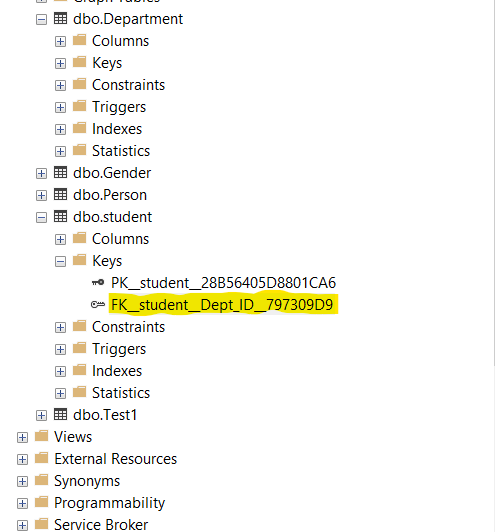
To handle this type of error by using Cascading Referential Integrity constraint following actions can be performed.

Here are the options when setting up Cascading referential integrity constraints:

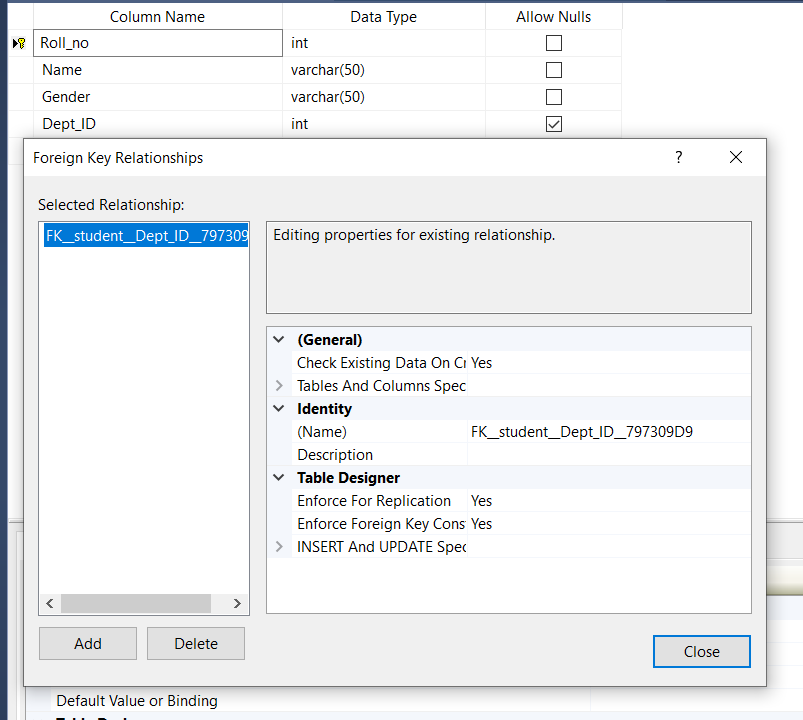
* **No  Action:** This is the default behaviour. An error is raised and the DELETE or UPDATE is rolled back if we attempt to delete or update a row whose key is referenced with existing rows in other tables.
* **Cascade:** Specifies that all rows containing those foreign keys are removed  if we attempt to delete or update a row with a key that is referenced by existing rows in other tables.
* **Set NULL:** Specifies that all rows containing those foreign keys are set to NULL if we attempt to delete or update a row with a key that is referenced by existing rows in other tables.
* **Set Default:** Specifies that all rows containing those foreign keys are set to a default value if an attempt is made to delete or update a row with a key referenced by existing rows in other tables.

**By using the following steps we can perform the above operations or actions on the tables:**

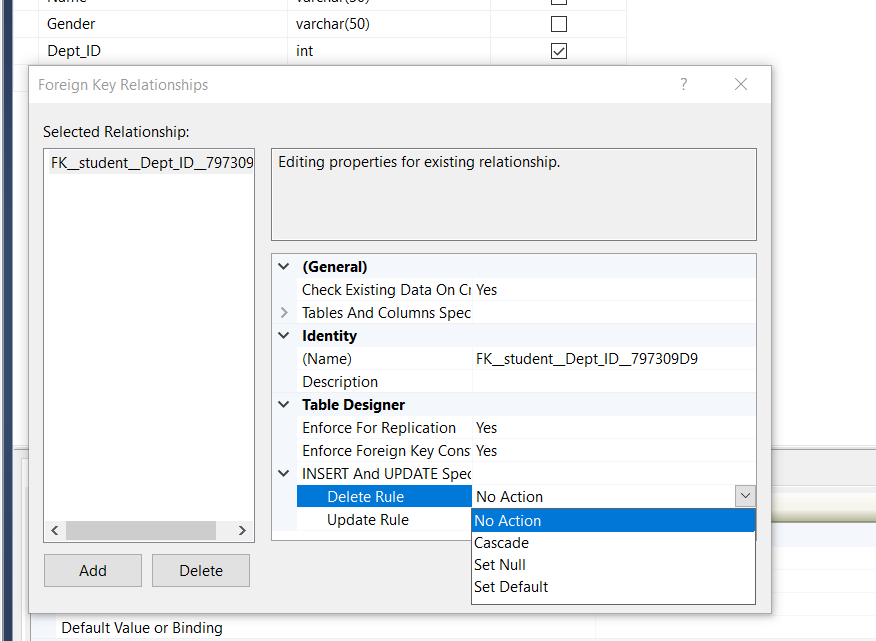
**Step 1:**In SQL server management studio go to the Keys of Table and select Foreign  key :



**Step 2:** After clicking “**FK\_\_student\_\_Dept\_ID\_\_797309D9″** go to the Foreign  key Relationship:



**Step 3:** In Foreign key Relationship click on INSERT And UPDATE Specification, and you will get the options such as No Action, Cascade, Set NULL, and Set Default.



From the above steps, we can use the designer of SQL Server Management Studio to set Cascading Referential Integrity Constraints for all the above mention actions. Also by using query you can you can specify cascading referential integrity constraints when creating a foreign key in SQL Server:

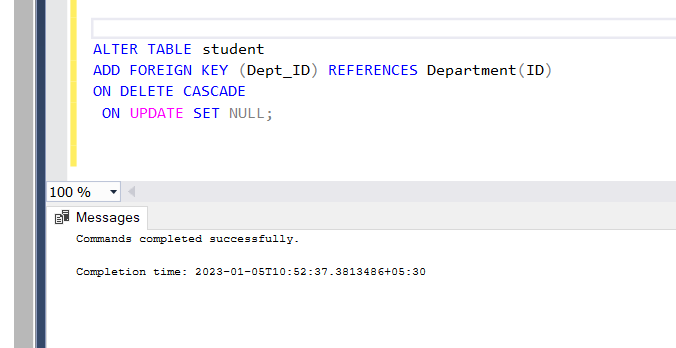
ALTER TABLE student

ADD FOREIGN KEY (Dept\_ID)

REFERENCES Department(ID)

ON DELETE CASCADE

ON UPDATE SET NULL;

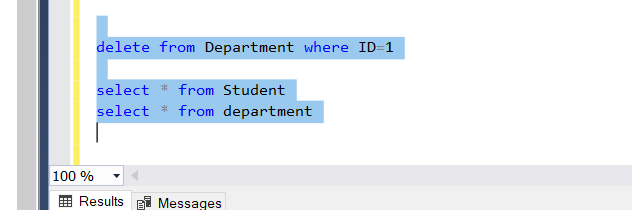


Now if we set the Delete rule of Insert and update specification as **cascade**and perform the DELETE operation on the Gender table then,

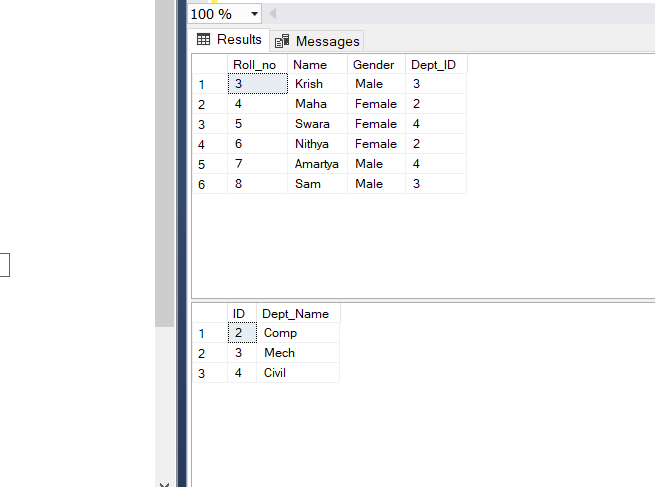
delete from Department where ID=1

select \* from Student

select \* from department



**Output:**After Successful Completion of the Delete Query, the tables  will be:



Here you can see that the row with**ID =1** from **Department** Table has been deleted and also the consecutive rows whose **Dept\_ID**is **1**from the studenttable have also been Deleted. Finally **Cascading Referential Integrity Constraints** helps the user to delete or update the foreign key column data from one table and also change the consecutive table record also.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQL Operators**

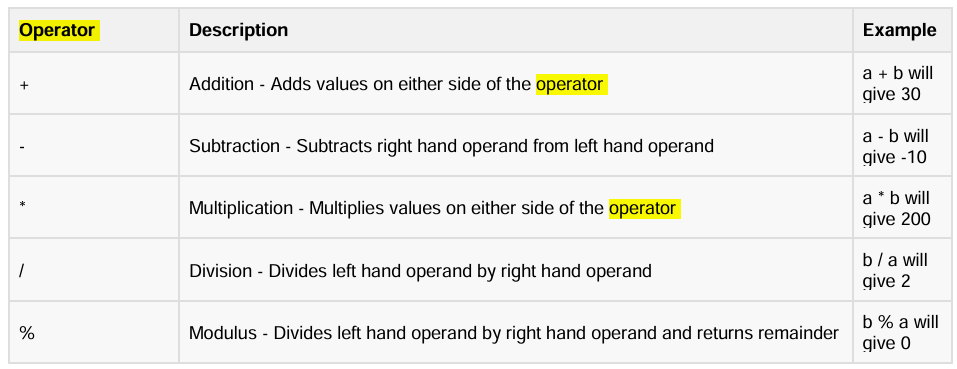
Operator is a reserved word or a character used primarily in an SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations.

Operators are used to specify conditions in an SQL statement and to serve as conjunctions for multiple conditions in a statement.

* Arithmetic operators
* Comparison operators
* Logical operators

**SQL Arithmetic Operators**:

Assume variable a holds 10 and variable b holds 20, then:



Here are simple examples showing usage of SQL Arithmetic Operators:

select 10+ 20;

+--------+

| 10+ 20 |

+--------+

| 30 |

+--------+

1 row in set (0.00 sec)

select 10 \* 20;

+---------+

| 10 \* 20 |

+---------+

| 200 |

+---------+

1 row in set (0.00 sec)

select 10 / 5;

+--------+

| 10 / 5 |

+--------+

| 2.0000 |

+--------+

1 row in set (0.03 sec)

select 12 % 5;

+---------+

| 12 % 5 |

+---------+

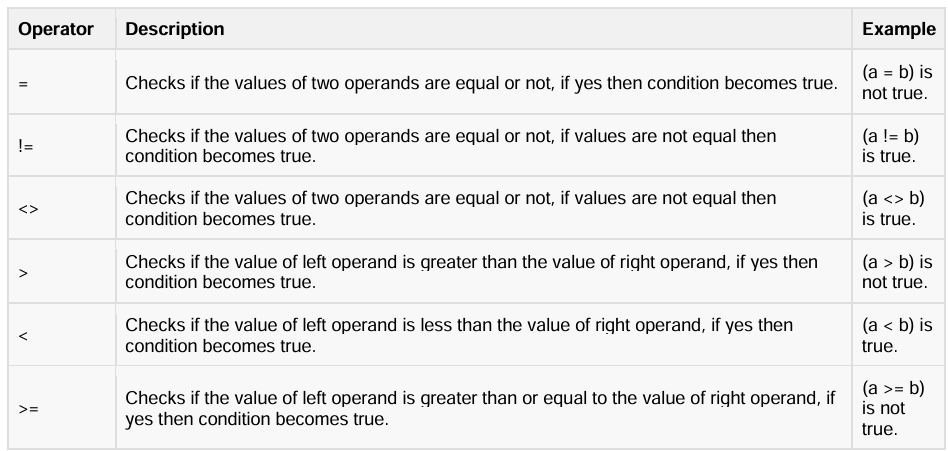
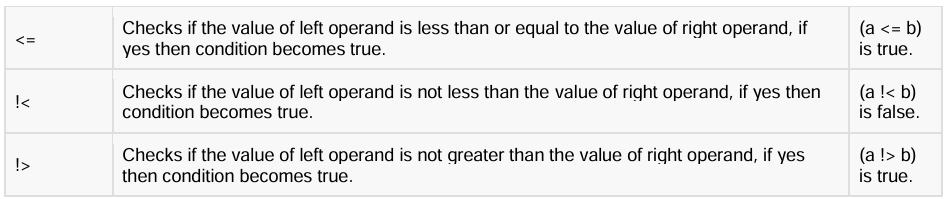
| 2 |

+---------+

1 row in set (0.00 sec)

**SQL Comparison Operators**:

Assume variable a holds 10 and variable b holds 20, then:

Consider the CUSTOMERS table having the following records:

SELECT \* FROM CUSTOMERS;

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

7 rows in set (0.00 sec)

Here are simple examples showing usage of SQL Comparison Operators:

SELECT \* FROM CUSTOMERS WHERE SALARY > 5000;

+----+----------+-----+---------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+---------+----------+

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+---------+----------+

3 rows in set (0.00 sec)

SELECT \* FROM CUSTOMERS WHERE SALARY = 2000;

+----+---------+-----+-----------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+---------+-----+-----------+---------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

+----+---------+-----+-----------+---------+

2 rows in set (0.00 sec)

SELECT \* FROM CUSTOMERS WHERE SALARY != 2000;

+----+----------+-----+---------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+---------+----------+

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+---------+----------+

5 rows in set (0.00 sec)

SELECT \* FROM CUSTOMERS WHERE SALARY <> 2000;

+----+----------+-----+---------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+---------+----------+

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+---------+----------+

5 rows in set (0.00 sec)

SELECT \* FROM CUSTOMERS WHERE SALARY >= 6500;

+----+----------+-----+---------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+---------+----------+

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

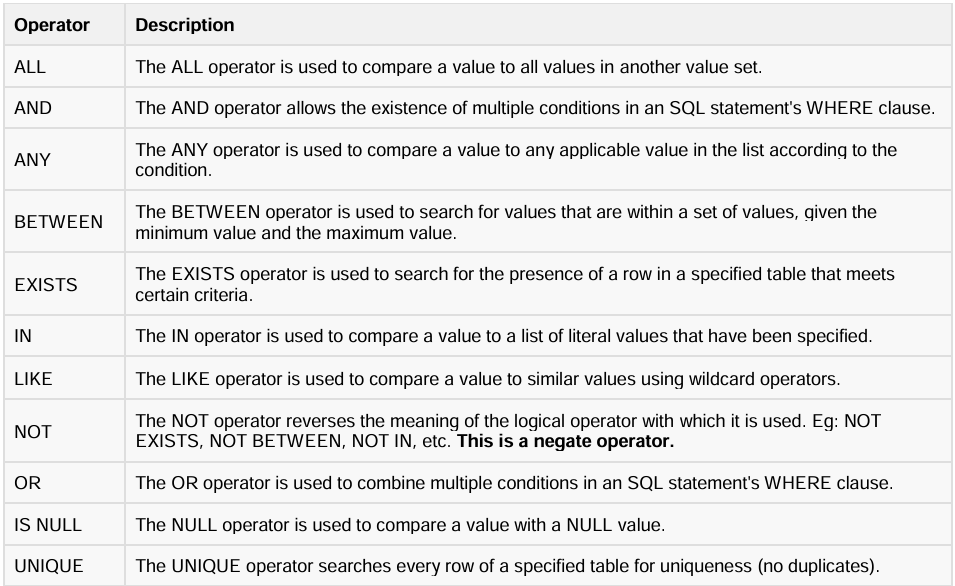
+----+----------+-----+---------+----------+

3 rows in set (0.00 sec)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQL Logical Operators**:

Here is a list of all the logical operators available in SQL.



Consider the CUSTOMERS table having the following records:

SELECT \* FROM CUSTOMERS;

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

7 rows in set (0.00 sec)

Here are simple examples showing usage of SQL Comparison Operators:

SELECT \* FROM CUSTOMERS WHERE AGE >= 25 AND SALARY >= 6500;

+----+----------+-----+---------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+---------+---------+

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

+----+----------+-----+---------+---------+

2 rows in set (0.00 sec)

SELECT \* FROM CUSTOMERS WHERE AGE >= 25 OR SALARY >= 6500;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

5 rows in set (0.00 sec)

SELECT \* FROM CUSTOMERS WHERE AGE IS NOT NULL;

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

7 rows in set (0.00 sec)

SELECT \* FROM CUSTOMERS WHERE NAME LIKE 'Ko%';

+----+-------+-----+---------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+-------+-----+---------+---------+

| 6 | Komal | 22 | MP | 4500.00 |

+----+-------+-----+---------+---------+

1 row in set (0.00 sec)

SELECT \* FROM CUSTOMERS WHERE AGE IN ( 25, 27 );

+----+----------+-----+---------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+---------+---------+

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

+----+----------+-----+---------+---------+

3 rows in set (0.00 sec)

SELECT \* FROM CUSTOMERS WHERE AGE BETWEEN 25 AND 27;

+----+----------+-----+---------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+---------+---------+

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

+----+----------+-----+---------+---------+

3 rows in set (0.00 sec)

SELECT AGE FROM CUSTOMERS

WHERE EXISTS (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);

+-----+

| AGE |

+-----+

| 32 |

| 25 |

| 23 |

| 25 |

| 27 |

| 22 |

| 24 |

+-----+

7 rows in set (0.02 sec)

SELECT \* FROM CUSTOMERS

WHERE AGE > ALL (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);

+----+--------+-----+-----------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+--------+-----+-----------+---------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

+----+--------+-----+-----------+---------+

1 row in set (0.02 sec)

SELECT \* FROM CUSTOMERS

WHERE AGE > ANY (SELECT AGE FROM CUSTOMERS WHERE SALARY > 6500);

+----+----------+-----+-----------+---------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+---------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

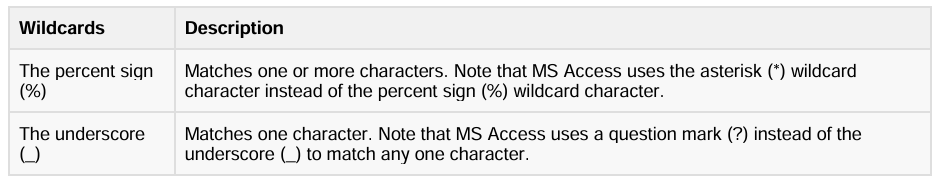
+----+----------+-----+-----------+---------+

4 rows in set (0.00 sec)

**Wildcard Characters**

We already have discussed SQL LIKE operator, which is used to compare a value to similar values using wildcard operators.

SQL supports following two wildcard operators in conjunction with the LIKE operator:



The percent sign represents zero, one, or multiple characters. The underscore represents a single number or character. The symbols can be used in combinations.

**Syntax**:

The basic syntax of ‘%’ and ‘\_’ is as follows:

SELECT FROM table\_name

WHERE column LIKE 'XXXX%'

or

SELECT FROM table\_name

WHERE column LIKE '%XXXX%'

or

SELECT FROM table\_name

WHERE column LIKE 'XXXX\_'

or

SELECT FROM table\_name

WHERE column LIKE '\_XXXX'

or

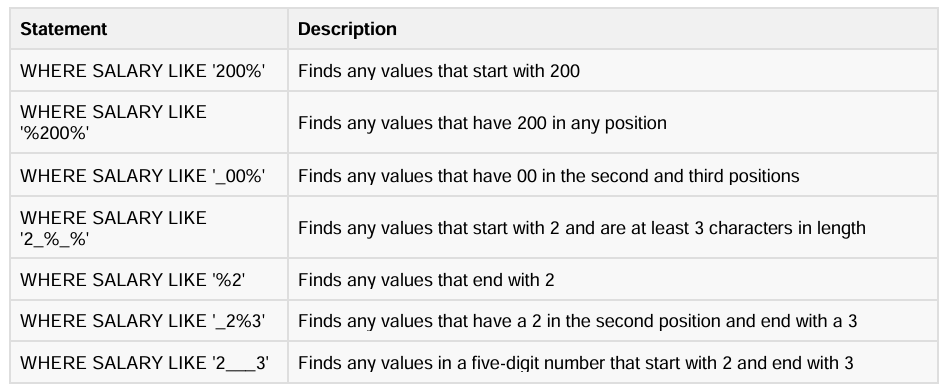
SELECT FROM table\_name

WHERE column LIKE '\_XXXX\_'

You can combine N number of conditions using AND or OR operators. Here, XXXX could be any numeric or string value.

**Example**:

Here are number of examples showing WHERE part having different LIKE clause with '%' and '\_' operators:



Let us take a real example, consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

Following is an example, which would display all the records from CUSTOMERS table where SALARY starts with

200:

SQL> SELECT \* FROM CUSTOMERS

WHERE SALARY LIKE '200%';

This would produce the following result:

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

+----+----------+-----+-----------+----------

**SQL JOINS**

SQL Joins clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

(b) Another table is ORDERS as follows:

+-----+---------------------+-------------+--------+

|OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now, let us join these two tables in our SELECT statement as follows:

SELECT ID, NAME, AGE, AMOUNT

FROM CUSTOMERS, ORDERS

WHERE CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

+----+----------+-----+--------+

| ID | NAME | AGE | AMOUNT |

+----+----------+-----+--------+

| 3 | kaushik | 23 | 3000 |

| 3 | kaushik | 23 | 1500 |

| 2 | Khilan | 25 | 1560 |

| 4 | Chaitali | 25 | 2060 |

+----+----------+-----+--------+

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal symbol.

**SQL Join Types**:

There are different types of joins available in SQL:

INNER JOIN: returns rows when there is a match in both tables.

LEFT JOIN: returns all rows from the left table, even if there are no matches in the right table.

RIGHT JOIN: returns all rows from the right table, even if there are no matches in the left table.

FULL JOIN: returns rows when there is a match in one of the tables.

SELF JOIN: is used to join a table to itself as if the table were two tables, temporarily renaming at least one

table in the SQL statement.

CARTESIAN JOIN: returns the Cartesian product of the sets of records from the two or more joined tables.

**INNER JOIN**

The most frequently used and important of the joins is the INNER JOIN. They are also referred to as an EQUIJOIN.

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

**Syntax**:

The basic syntax of INNER JOIN is as follows:

SELECT table1.column1, table2.column2...

FROM table1

INNER JOIN table2

ON table1.common\_filed = table2.common\_field;

**Example**:

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

(b) Another table is ORDERS as follows:

+-----+---------------------+-------------+--------+

| OID | DATE | ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now, let us join these two tables using INNER JOIN as follows:

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

INNER JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

+----+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+----+----------+--------+---------------------+

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+----+----------+--------+---------------------+

**LEFT JOIN**

The SQL LEFT JOIN returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in right table, the join will still return a row in the result, but with NULL in each column from right table.

This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

**Syntax**:

The basic syntax of LEFT JOIN is as follows:

SELECT table1.column1, table2.column2...

FROM table1

LEFT JOIN table2

ON table1.common\_filed = table2.common\_field;

Here given condition could be any given expression based on your requirement.

**Example**:

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

(b) Another table is ORDERS as follows:

+-----+---------------------+-------------+--------+

| OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now, let us join these two tables using LEFT JOIN as follows:

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

+----+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+----+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

+----+----------+--------+---------------------+

**RIGHT JOIN**

The SQL RIGHT JOIN returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in left table, the join will still return a row in the result, but with NULL in each column from left table.

This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

**Syntax**:

The basic syntax of RIGHT JOIN is as follows:

SELECT table1.column1, table2.column2...

FROM table1

RIGHT JOIN table2

ON table1.common\_filed = table2.common\_field;

**Example**:

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

(b) Another table is ORDERS as follows:

+-----+---------------------+-------------+--------+

|OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now, let us join these two tables using RIGHT JOIN as follows:

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

**FULL JOIN**

The SQL FULL JOIN combines the results of both left and right outer joins.

The joined table will contain all records from both tables, and fill in NULLs for missing matches on either side.

**Syntax**:

The basic syntax of FULL JOIN is as follows:

SELECT table1.column1, table2.column2...

FROM table1

FULL JOIN table2

ON table1.common\_filed = table2.common\_field;

Here given condition could be any given expression based on your requirement.

**Example**:

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

(b) Another table is ORDERS as follows:

+-----+---------------------+-------------+--------+

|OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now, let us join these two tables using FULL JOIN as follows:

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

FULL JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID;

This would produce the following result:

+------+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+------+----------+--------+---------------------+

| 1 | Ramesh | NULL | NULL |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | NULL | NULL |

| 6 | Komal | NULL | NULL |

| 7 | Muffy | NULL | NULL |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

+------+----------+--------+---------------------+

If your Database does not support FULL JOIN like MySQL does not support FULL JOIN, then you can use UNION ALL clause to combine two JOINS as follows:

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

LEFT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

UNION ALL

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS

RIGHT JOIN ORDERS

ON CUSTOMERS.ID = ORDERS.CUSTOMER\_ID

**SELF JOIN**

The SQL SELF JOIN is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

**Syntax**:

The basic syntax of SELF JOIN is as follows:

SELECT a.column\_name, b.column\_name...

FROM table1 a, table1 b

WHERE a.common\_filed = b.common\_field;

Here, WHERE clause could be any given expression based on your requirement.

**Example**:

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

Now, let us join this table using SELF JOIN as follows:

SELECT a.ID, b.NAME, a.SALARY

FROM CUSTOMERS a, CUSTOMERS b

WHERE a.SALARY < b.SALARY;

This would produce the following result:

+----+----------+---------+

| ID | NAME | SALARY |

+----+----------+---------+

| 2 | Ramesh | 1500.00 |

| 2 | kaushik | 1500.00 |

| 1 | Chaitali | 2000.00 |

| 2 | Chaitali | 1500.00 |

| 3 | Chaitali | 2000.00 |

| 6 | Chaitali | 4500.00 |

| 1 | Hardik | 2000.00 |

| 2 | Hardik | 1500.00 |

| 3 | Hardik | 2000.00 |

| 4 | Hardik | 6500.00 |

| 6 | Hardik | 4500.00 |

| 1 | Komal | 2000.00 |

| 2 | Komal | 1500.00 |

| 3 | Komal | 2000.00 |

| 1 | Muffy | 2000.00 |

| 2 | Muffy | 1500.00 |

| 3 | Muffy | 2000.00 |

| 4 | Muffy | 6500.00 |

| 5 | Muffy | 8500.00 |

| 6 | Muffy | 4500.00 |

+----+----------+---------+

**CARTESIAN JOIN**

The CARTESIAN JOIN or CROSS JOIN returns the cartesian product of the sets of records from the two or more joined tables. Thus, it equates to an inner join where the join-condition always evaluates to True or where the join condition is absent from the statement.

**Syntax**:

The basic syntax of INNER JOIN is as follows:

SELECT table1.column1, table2.column2...

FROM table1, table2 [, table3 ]

Example:

Consider the following two tables, (a) CUSTOMERS table is as follows:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

(b) Another table is ORDERS as follows:

+-----+---------------------+-------------+--------+

|OID | DATE | CUSTOMER\_ID | AMOUNT |

+-----+---------------------+-------------+--------+

| 102 | 2009-10-08 00:00:00 | 3 | 3000 |

| 100 | 2009-10-08 00:00:00 | 3 | 1500 |

| 101 | 2009-11-20 00:00:00 | 2 | 1560 |

| 103 | 2008-05-20 00:00:00 | 4 | 2060 |

+-----+---------------------+-------------+--------+

Now, let us join these two tables using INNER JOIN as follows:

SELECT ID, NAME, AMOUNT, DATE

FROM CUSTOMERS, ORDERS;

This would produce the following result:

+----+----------+--------+---------------------+

| ID | NAME | AMOUNT | DATE |

+----+----------+--------+---------------------+

| 1 | Ramesh | 3000 | 2009-10-08 00:00:00 |

| 1 | Ramesh | 1500 | 2009-10-08 00:00:00 |

| 1 | Ramesh | 1560 | 2009-11-20 00:00:00 |

| 1 | Ramesh | 2060 | 2008-05-20 00:00:00 |

| 2 | Khilan | 3000 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1500 | 2009-10-08 00:00:00 |

| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |

| 2 | Khilan | 2060 | 2008-05-20 00:00:00 |

| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |

| 3 | kaushik | 1560 | 2009-11-20 00:00:00 |

| 3 | kaushik | 2060 | 2008-05-20 00:00:00 |

| 4 | Chaitali | 3000 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 1500 | 2009-10-08 00:00:00 |

| 4 | Chaitali | 1560 | 2009-11-20 00:00:00 |

| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

| 5 | Hardik | 3000 | 2009-10-08 00:00:00 |

| 5 | Hardik | 1500 | 2009-10-08 00:00:00 |

| 5 | Hardik | 1560 | 2009-11-20 00:00:00 |

| 5 | Hardik | 2060 | 2008-05-20 00:00:00 |

| 6 | Komal | 3000 | 2009-10-08 00:00:00 |

| 6 | Komal | 1500 | 2009-10-08 00:00:00 |

| 6 | Komal | 1560 | 2009-11-20 00:00:00 |

| 6 | Komal | 2060 | 2008-05-20 00:00:00 |

| 7 | Muffy | 3000 | 2009-10-08 00:00:00 |

| 7 | Muffy | 1500 | 2009-10-08 00:00:00 |

| 7 | Muffy | 1560 | 2009-11-20 00:00:00 |

| 7 | Muffy | 2060 | 2008-05-20 00:00:00 |

+----+----------+--------+---------------------+

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQL VIEW**

view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are kind of virtual tables, allow users to do the following:

* Structure data in a way that users or classes of users find natural or intuitive.
* Restrict access to the data such that a user can see and (sometimes) modify exactly what they need and no more.
* Summarize data from various tables which can be used to generate reports.

**Creating Views:**

Database views are created using the CREATE VIEW statement. Views can be created from a single table, multiple tables, or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic CREATE VIEW syntax is as follows:

CREATE VIEW view\_name AS

SELECT column1, column2.....

FROM table\_name

WHERE [condition];

You can include multiple tables in your SELECT statement in very similar way as you use them in normal SQL SELECT query.

Example:

Consider the CUSTOMERS table having the following records:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

Now, following is the example to create a view from CUSTOMERS table. This view would be used to have customer name and age from CUSTOMERS table:

CREATE VIEW CUSTOMERS\_VIEW AS

SELECT name, age

FROM CUSTOMERS;

Now, you can query CUSTOMERS\_VIEW in similar way as you query an actual table. Following is the example:

SELECT \* FROM CUSTOMERS\_VIEW;

This would produce the following result:

+----------+-----+

| name | age |

+----------+-----+

| Ramesh | 32 |

| Khilan | 25 |

| kaushik | 23 |

| Chaitali | 25 |

| Hardik | 27 |

| Komal | 22 |

| Muffy | 24 |

+----------+-----+

The WITH CHECK OPTION:

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following is an example of creating same view CUSTOMERS\_VIEW with the WITH CHECK OPTION:

CREATE VIEW CUSTOMERS\_VIEW AS

SELECT name, age

FROM CUSTOMERS

WHERE age IS NOT NULL

WITH CHECK OPTION;

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

**Updating a View**:

A view can be updated under certain conditions:

* The SELECT clause may not contain the keyword DISTINCT.
* The SELECT clause may not contain summary functions.
* The SELECT clause may not contain set functions.
* The SELECT clause may not contain set operators.
* The SELECT clause may not contain an ORDER BY clause.
* The FROM clause may not contain multiple tables.
* The WHERE clause may not contain subqueries.
* The query may not contain GROUP BY or HAVING.
* Calculated columns may not be updated.
* All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So if a view satisfies all the abovementioned rules then you can update a view. Following is an example to update the age of Ramesh:

UPDATE CUSTOMERS\_VIEW

SET AGE = 35

WHERE name='Ramesh';

This would ultimately update the base table CUSTOMERS and same would reflect in the view itself. Now, try to query base table, and SELECT statement would produce the following result:

+----+----------+-----+-----------+----------+

| 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 |

+----+----------+-----+-----------+----------+

**Inserting Rows into a View:**

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Here, we can not insert rows in CUSTOMERS\_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in similar way as you insert them in a table.

**Deleting Rows into a View**:

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE= 22.

DELETE FROM CUSTOMERS\_VIEW

WHERE age = 22;

This would ultimately delete a row from the base table CUSTOMERS and same would reflect in the view itself.

Now, try to query base table, and SELECT statement would produce the following result:

+----+----------+-----+-----------+----------+

| 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 |

| 7 | Muffy | 24 | Indore | 10000.00 |

+----+----------+-----+-----------+----------+

**Dropping Views**:

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple as given below:

DROP VIEW view\_name;

Following is an example to drop CUSTOMERS\_VIEW from CUSTOMERS table:

DROP VIEW CUSTOMERS\_VIEW;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Stored Procedure**

**Creating a simple stored procedure**

The following SELECT statement returns a list of products from the products table in the BikeStores sample database:

SELECT

product\_name,

list\_price

FROM

production.products

ORDER BY

product\_name;

To create a stored procedure that wraps this query, you use the CREATE PROCEDURE statement as follows:

CREATE PROCEDURE uspProductList

AS

BEGIN

SELECT

product\_name,

list\_price

FROM

production.products

ORDER BY

product\_name;

END;

In this syntax:

The uspProductList is the name of the stored procedure.

The AS keyword separates the heading and the body of the stored procedure.

If the stored procedure has one statement, the BEGIN and END keywords surrounding the statement are optional. However, it is a good practice to include them to make the code clear.

Note that in addition to the CREATE PROCEDURE keywords, you can use the CREATE PROC keywords to make the statement shorter.

**Executing a stored procedure**

To execute a stored procedure, you use the EXECUTE or EXEC statement followed by the name of the stored procedure:

EXECUTE sp\_name;

Or

EXEC sp\_name;

where sp\_name is the name of the stored procedure that you want to execute.

For example, to execute the uspProductList stored procedure, you use the following statement:

EXEC uspProductList;

**Modifying a stored procedure**

To modify an existing stored procedure, you use the ALTER PROCEDURE statement.

1. First, open the stored procedure to view its contents by right-clicking the stored procedure name and select Modify menu item:
2. Second, change the body of the stored procedure by sorting the products by list prices instead of product names:

ALTER PROCEDURE uspProductList

AS

BEGIN

SELECT

product\_name,

list\_price

FROM

production.products

ORDER BY

list\_price

END;

1. Third, click the Execute button, SQL Server modifies the stored procedure and returns the following output:

Commands completed successfully.

Now, if you execute the stored procedure again, you will see the changes taking effect:

EXEC uspProductList;

**Deleting a stored procedure**

To delete a stored procedure, you use the DROP PROCEDURE or DROP PROC statement:

DROP PROCEDURE sp\_name;

or

DROP PROC sp\_name;

where sp\_name is the name of the stored procedure that you want to delete.

For example, to remove the uspProductList stored procedure, you execute the following statement:

DROP PROCEDURE uspProductList;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQL Functions**

SQL provides much in-build function to perform operation of table data, these functions can be with-in SQL statements or queries, and can also be used within the programming environment provided by the SQL Server (Transact-SQL) database, such as stored procedures, functions, triggers, etc.

SQL Built-In functions are generally used to perform string concatenations, mathematical calculations etc.

SQL functions are categorized into the following two categories:

1. SQL Built-In Functions
2. SQL User Defined Function

**SQL Built-In Functions**

SQL provides many built-in functions for performing processing on string or numeric data, the following are the list of categories of Built-In functions based on the type of data used in operation

1. Aggregate Functions
2. Scalar functions

**Aggregate Functions:** Aggregate functions operates on set of numeric values and return a single value, SQL Aggregate function are used as the select list of SQL SELECT statement and also can be used in combination with the GROUP BY clause to calculate the aggregation on categories of rows

**List of Built-In SQL Aggregate Functions**

1. AVG
2. CHECKSUM\_AGG
3. COUNT
4. COUNT\_BIG
5. GROUPING
6. GROUPING\_ID
7. MAX
8. MIN
9. STDEV
10. STDEVP
11. STRING\_AGG
12. SUM
13. VAR
14. VARP

**Examples of SQL Aggregate Functions**

Consider School result management system to perform practical examples on SQL functions

**Student Table**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| student\_id | studentname | admissionno | admissiondate | enrollmentno | date\_of\_birth | email | city | class\_id |
| 101 | reema | 10001 | 02-02-2000 | e15200002 | 02-02-1990 | reema@gmail.com | surat | 2 |
| 102 | kriya | 10002 | 04-05-2001 | e16200003 | 04-08-1991 | kriya@gmail.com | surat | 1 |
| 103 | meena | 10003 | 06-05-1999 | e15200004 | 02-09-1989 | meena@gmail.com | vadodara | 3 |
| 104 | carlin | 2001 | 04-01-1998 | e14200001 | 04-04-1989 | carli@gmail.com | vapi | 1 |
| 105 | dhiren | 2002 | 02-02-1997 | e13400002 | 02-02-1987 | dhiru@gmail.com | vapi | 2 |
| 106 | hiren | 2003 | 01-01-1997 | e13400001 | 03-03-1887 | hiren@gmail.com | surat | 2 |
| 107 | mahir | 10004 | 06-09-2000 | e15200003 | 07-09-1990 | mahi@gmail.com | vapi | 3 |
| 108 | nishi | 2004 | 02-04-2001 | e16200001 | 03-02-1991 | nishi@gmail.com | vadodara | 1 |

**Result Table**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| result\_id | student\_id | examname | examdate | subject | obtainmark | totalmarks | percentage | grade | status |
| 3001 | 101 | sem1 | 07-08-2001 | 1 | 80 | 100 | 80 | A+ | pass |
| 3002 | 101 | sem1 | 08-08-2001 | 2 | 76 | 100 | 76 | A+ | pass |
| 3003 | 102 | sem3 | 05-05-2000 | 3 | 67 | 100 | 67 | A | pass |
| 3004 | 102 | sem3 | 06-05-2000 | 4 | 89 | 100 | 89 | A+ | pass |
| 3005 | 102 | sem3 | 07-05-2000 | 5 | 90 | 100 | 90 | A+ | pass |
| 3006 | 103 | sem5 | 08-09-1998 | 6 | 55 | 100 | 55 | B | pass |
| 3007 | 103 | sem5 | 09-09-1998 | 7 | 30 | 100 | 30 | D | fail |
| 3008 | 103 | sem5 | 10-09-1998 | 8 | 34 | 100 | 34 | D | fail |

**Subject Table**

|  |  |  |  |
| --- | --- | --- | --- |
| subjectid | facultyname | subjectname | subjectcode |
| 1 | krishna | c | 1003 |
| 2 | rahul | cpp | 1004 |
| 3 | radha | asp | 1005 |
| 4 | meera | sql | 1006 |
| 5 | yasoda | cloud | 1007 |
| 6 | nadan | cg | 1008 |

**SQL AVG() Function with Example**

SQL AVG() function is used to calculate the average value of given numeric values

**Example 1**:

Write SQL query to calculate average obtain marks of each semester student

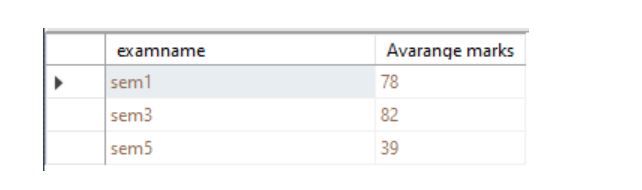
SELECT examname, AVG(obtainmark) AS 'Avarange marks'

FROM tblresult

GROUP BY examname

* In the above query, SQL AVG() function is applied on obtain marks column of result table to calculate average value of obtain mark values
* SQL Group By clause is used make group of records based on examname column value

**OUTPUT:**



**SQL CHECKSUM\_AGG Function with Example**

The CHECKSUM\_AGG() function is used to calculates a checksum value based on a group of records

**Example 2:**

Write SQL query to calculate checksum value of semester 3 student’s obtain mark

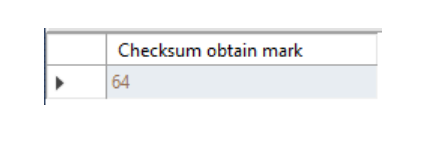
SELECT CHECKSUM\_AGG(obtainmark) AS 'Checksum obtain mark'

FROM tblresult

WHERE (examname = 'sem3')

* In the above query, SQL CHECKSUM\_AGG function is used with obtainmark column to calculate checksum value
* SQL where clause is used to conditionally retrieve sem 3 records from result table

**OUTPUT:**



**SQL COUNT() Function with Example**

SQL COUNT() function is used to count the total number of records which matches with given condition.

**Example 3:**

Write SQL query to count total number of students citywise

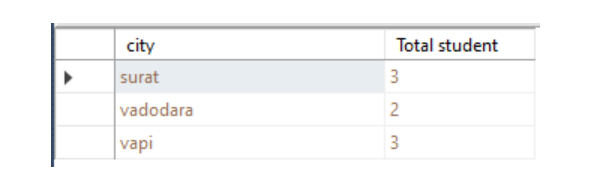
SELECT city, COUNT(\*) AS 'Total student'

FROM tblstudent

GROUP BY city

* In the above query, SQL COUNT() Aggregate function is used to count total number of students lived in each city
* SQL Group By clause is used to make a group of records based on city name

**OUTPUT:**



**SQL COUNT\_BIG() Function with Example**

The COUNT\_BIG() function is used to count the number of records including records with NULL values

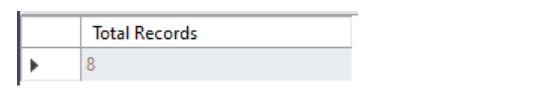
**Example 4** : Write SQL query to count total number of records in result table

SELECT COUNT\_BIG(obtainmark) AS 'Total Records'

FROM tblresult

In the above query, SQL COUNT\_BIG() function is used to count total records in result table

**OUTPUT:**

****

**SQL GROUPING () Function with Example**

SQL GROUPING function is used to identify whether a specified column expression in a GROUP BY list is aggregated or not

**Example 5:** Write SQL query to check grouping is performed on city column or on class ID column

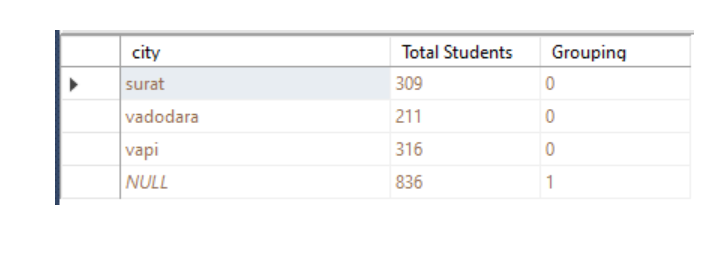
SELECT city, SUM(student\_id) AS 'Total Students', GROUPING(city) AS 'Grouping'

FROM tblstudent

GROUP BY city WITH ROLLUP

* In the above query, SQL GROUPING function is applied on city column to check whether group has performed on city column with GROUP BY clause or not
* The result set shows NULL value in the last summery record added by ROLLUP operation
* The summery record shows total number of student of all groups and it indicates 1 in grouping result

**OUTPUT:**



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQL GROUPING\_ID() Function with Example**

* SQL GROUPING\_ID() function is use to concatenates the output of the GROUPING functions applied to all the columns specified in the GROUP BY clause
* GROUPING\_ID can be used only in the SELECT <select> list, HAVING, or ORDER BY clauses when GROUP BY is specified

**Example 6:** Write SQL query to concatenate grouping value of two columns city and class\_id

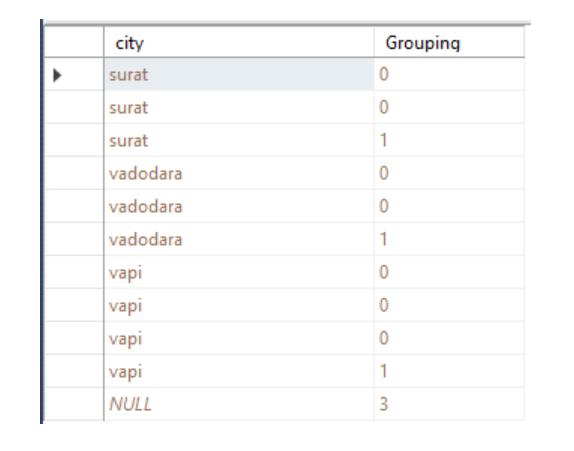
SELECT city, GROUPING\_ID(city,class\_id) AS 'Grouping'

FROM tblstudent

GROUP BY city WITH ROLLUP

* In the above query , SQL GROUPING\_ID function is used to concate grouping output of two columns city and class\_id
* The result set shows NULL value in the last summery record added by ROLLUP operation
* The summery record shows total number of student of all groups and it indicates 1 in grouping result

**OUTPUT:**



**SQL MAX() Function with Example**

SQL MAX() function is used to find Maximum value from the list of values

**Example 7:** Write SQL query to find maximum value of obtain mark for sem3 students

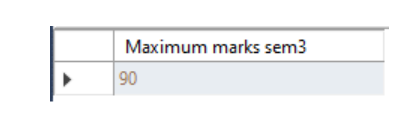
SELECT MAX(obtainmark) AS 'Maximum marks sem3'

FROM tblresult

WHERE (examname = 'sem3')

* In the above query, SQL MAX () function is applied on obtain mark column to find maximum mark from all students
* SQL Where clause is used to conditionally retrieve records of sem3 students

**OUTPUT**:



**SQL MIN() Function with Example**

SQL MIN() function is used to find Minimum value from the list of values

**Example 8**: Write SQL query to find minimum value of obtain mark for sem3 students

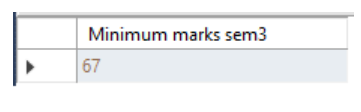
SELECT MIN(obtainmark) AS 'Minimum marks sem3'

FROM tblresult

WHERE (examname = 'sem3')

* In the above query, SQL MIN () function is applied on obtain mark column to find minimum mark from all students
* SQL Where clause is used to conditionally retrieve records of sem3 students

**OUTPUT**:



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQL STDEV() Function with Example**

SQL STDEV() function is used to calculate the Standard Deviation of total records retrieved by the SQL SELECT Statement

**Example 9:** Write SQL query to calculate standard deviation of obtain marks of semester 3 students

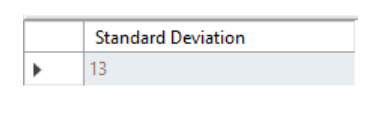
SELECT STDEV(obtainmark) AS 'Standard Deviation'

FROM tblresult

WHERE (examname = 'sem3')

* In the above query, SQL STDEV() function is applied on obtainmarks column to calculate standard deviation of obtain marks value of semester 3 student

**OUTPUT**:



**SQL STDEVP() Function with Example**

SQL STDEVP() function is used to calculates the Statistical Standard Deviation for the population of total records selected by the SELECT Statement

**Example 10**: Write SQL query to calculate statistical standard deviation of obtain marks of all semester 3 students

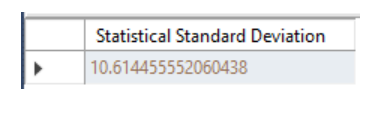
SELECT STDEVP(obtainmark) AS 'Statistical Standard Deviation '

FROM tblresult

WHERE (examname = 'sem3')

* In the above query, SQL STDEVP() function is applied on obtainmarks column to calculate statistical standard deviation of obtain marks value of semester 3 student

**OUTPUT**:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**SQL STRING\_AGG() Function with Example**

SQL STRING\_AGG() function is used to concatenates the string expressions retrieved as the select list with SQL select statement , and places a specified separator in-between them

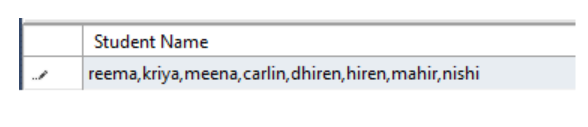
**Example 11:** Write SQL query to concatenate student name value of all records separated by comma(,)

SELECT STRING\_AGG(studentname,' , ') AS 'Student Name '

FROM tblstudent

* In the above query, SQL STRING\_AGG function is used with two arguments , studentname and ‘,’
* The first argument specifies column name, and second argument separator symbol in the single quotation

**OUTPUT**:



SQL SUM () Function with Example

SQL SUM() function is used to calculate the total of given numeric values

Example 12: Write SQL query to make total of all obtain marks of each semester

sql

SELECT examname, SUM(obtainmark) AS 'Total obtain marks'

FROM tblresult

GROUP BY examname

In the above query SQL SUM() is used with SQL select statement to make total of obtain marks

SQL Group by clause is used to make group of records based on examname

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL VAR() Function with Example

SQL VAR() is used to calculate the statistical variance of sample records selected by the SELECT Statement

SQL VAR() function can only be used with numeric columns , Null values are ignored

Example 13: Write SQL query to find variance value of obtainmark column semester wise

sql

SELECT examname, VAR(obtainmark) AS 'Variance value'

FROM tblresult

GROUP BY examname

In the above query, SQL VAR() function is applied on obtainmark column to calculate variance value of each semester student marks based on sample obtain mark

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL VARP() Function with Example

SQL VARP() function is used to calculates the statistical variance for the population of total rows selected by the SQL SELECT Statement

SQL VARP() function can be used with numeric columns only,Null values are ignored

ADVERTISEMENT

Difference between VAR() and VARP() Functions

VarP() evaluates a population, whereas the Var() function evaluates a population sample

Example 14: Write SQL query to find variance of total evaluated obtain marks of all students semsterwise

sql

SELECT examname, VARP(obtainmark) AS 'Variance value'

FROM tblresult

GROUP BY examname

In the above SQL query , SQL VARP() function is applied on obtainmark column with grouping of semester

SQL VARP() function calculate variance of total obtain marks of each semester

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL Scalar functions

Scalar functions operate on a single value and then return a single value, Scalar functions can be used as an expression or as select list of SQL SELECT statement

List of Build-in Scalar Functions

1. STRING FUNCTIONS

RIGHT

LEFT

LEN

RTRIM

LTRIM

REPLACE

REVERSE

SUBSTRING

LOWER

UPPER

2. DATE FUNCTIONS

DATEADD

DATEDIFF

DAY

MONTH

YEAR

GETDATE

3. NUMERIC FUNCTIONS

FLOOR

CEILING

ROUND

4. CONVERSION FUNCTIONS

CAST

CONVERT

5. NULL-RELATED FUNCTIONS

ISNULL

SQL STRING FUNCTIONS Examples

SQL RIGHT() FUNCTION WITH Example

SQL RIGHT() function returns specified number of characters from the right side of given string

Example 15: Write SQL query to retrieve last 8 characters from given string ‘SQL Tutorial'

sql

SELECT RIGHT('SQL Tutorial', 8) AS 'RIGHT FUNCTION'

In the above query, SQL RIGHT() function is used with two arguments, the first argument specifies the string from which the characters will retrieve and second argument specifies the number of characters to be retrieved

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL LEFT() FUNCTION WITH Example

SQL LEFT() function returns specified number of characters from left side of given string

Example 16: Write SQL query to retrieve first 3 characters from given string 'SQL Tutorial'

ADVERTISEMENT

sql

SELECT LEFT('SQL Tutorial', 3) AS 'LEFT FUNCTION'

In the above query, SQL LEFT() function is used with two arguments, the first argument specifies the string from which the characters will retrieve and second argument specifies the number of characters to be retrieved

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL LEN() FUNCTION WITH Example

SQL LEN() function is used to count number of characters in the given string

Example 17: Write SQL query to count number of characters of a given string ‘SQL Tutorial'

sql

SELECT LEN('SQL Tutorial') AS 'Length'

In the above query, SQL LEN() function specified with one argument of a string ‘SQL Tutorial’ to count number of characters in a string

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL RTRIM() FUNCTION WITH Example

SQL RTRIM() function is used to remove all trailing blanks from the right side of given string

Example 18: Write SQL query to remove all trailing blank space from right side of given string ‘ SQL Tutorial ’

sql

SELECT RTRIM(' SQL Tutorial ') AS 'RTRIM'

In the above query, SQL RTRIM() function specified with one argument of string ‘ SQL Tutorial ‘ with the trailing blank space in the right and left side of a string

SQL RTRIM() function will remove the space from the right side of a string

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL LTRIM() FUNCTION WITH Example

SQL LTRIM() function is used to remove all trailing blanks from the left side of given string

ADVERTISEMENT

Example 19: Write SQL query to remove all trailing blank space from left side of given string ‘ SQL Tutorial ’

sql

SELECT LTRIM(' SQL Tutorial ') AS 'LTRIM'

In the above query, SQL LTRIM() function specified with one argument of string ‘ SQL Tutorial ‘ with the trailing blank space in the right and left side of a string

SQL LTRIM() function will remove the space from the left side of a string

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL REPLACE() FUNCTION WITH Example

SQL REPLACE() function is used to replace all occurrence of specified character in a given string with new character

Example 20: Write SQL query to replace character ‘T’ in the given string with new character ’$’

sql

SELECT REPLACE('SQL Tutorial', 't', '$') AS 'REPLACE'

In the above query, SQL REPLACE() function specified used with three arguments

The first argument is the string in which we want to perform replacement

The second argument is the character to be replaced

The third argument is the new character with which character will be replaced

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL REVERSE() FUNCTION WITH Example

SQL REVERSE() function is used to reverse the specified string

Example 21: Write SQL query to reverse the given string ‘SQL Tutorial’

sql

SELECT REVERSE('SQL Tutorial') AS 'REVERSE'

In the above query, SQL REVERSE () function specified with one argument of string to be revered

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL SUBSTRING() FUNCTION WITH Example

SQL SUBSTRING() function is used to extract part of string

Example 22: Write SQL query to extract string ‘Tutorial’ from the given string ‘SQL Tutorial Videos’

ADVERTISEMENT

sql

SELECT SUBSTRING('SQL Tutorial Videos', 5, 8) AS 'SUBSTRING'

In the above query, SQL SUBSTRING () function specified with three arguments to extract part of inputted string ‘'SQL Tutorial Videos'

The First argument specifies the input string from which a part is going to be extracted

The Second argument specified the starting index

The Third argument specified number of characters

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL LOWER() FUNCTION WITH Example

SQL LOWER() function is used to convert all upper case character to lower case characters

Example 23: Write SQL query to convert all upper-case letters in a given string to lower-case letters

sql

SELECT LOWER('SQL Tutorial') AS 'LOWER()'

In the above query, SQL LOWER () function is applied on a string” SQL Tutorial” to convert all letters of given string into lower case

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL UPPER() FUNCTION WITH Example

SQL UPPER() function is used to convert all lower case characters to upper case characters

Example 24: Write SQL query to convert all lower-case letters in a given string to upper-case letters

sql

SELECT UPPER('SQL Tutorial') AS 'UPPER()'

In the above query, SQL UPPER () function is applied on a string” SQL Tutorial” to convert all letters of given string into upper case

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL DATE FUNCTIONS with Examples

SQL DATEADD() FUNCTION with Example

SQL DATEADD() function is used to add specified number interval to the specified date part of a given date

Example 25: Write SQL query to add two months to the given date '02-02-1990'

ADVERTISEMENT

sql

SELECT DATEADD(mm, 2, '1990-02-02') AS [DATEADD()]

In the above query, SQL DATEADD() function specifies with 3 arguments to add 2 months to given date

The first argument specifies the part of date ‘mm’ represents month, ‘dd’ represents date and ‘yy’ represents year

The second argument specifies the numeric value to be added to the given date value

The third argument is date in which months to be added

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL DATEDIFF() FUNCTION with Example

SQL DATEDIFF() function is used to find the difference between two given dates based on specified datepart

Example 26: Write SQL query to fund month difference between two date ’01-01-1990’ & ’03-03-1990’

sql

SELECT DATEDIFF(mm, '1990-01-02', '1990-03-03') AS [DATEDIFF()]

In the above query, SQL DATEDIFF() function specifies with 3 arguments to find the difference between two given dates

The first argument specifies the part of date ‘mm’ represents month, ‘dd’ represents date and ‘yy’ represents year

The second argument specifies the first date

The third argument specifies the second date

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL DAY() FUNCTION with Example

SQL DAY() function is used to retrieve the day of a month in integer number from the given date value

Example 27: Write SQL query to retrieve day number from the date ’02-01-1990’

sql

SELECT DAY('1990-01-02') AS [DAY()]

In the above query, SQL DAY() function specified with one argument of date to retrieve day of a month in integer number

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL MONTH() FUNCTION with Example

SQL MONTH() function is used to retrieve month from given date value in an integer number

Example 28: Write SQL query to retrieve month from the date ’02-01-1990’

sql

SELECT MONTH('1990-01-02') AS [MONTH()]

In the above query, SQL MONTH() function specified with one argument of date to retrieve month in integer number

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL YEAR() FUNCTION with Example

SQL YEAR() function is used to retrieve year from given date value

Example 29: Write SQL query to retrieve year from the date ’02-01-1990’

sql

SELECT YEAR('1990-01-02') AS [YEAR()]

In the above query, SQL YEAR() function specified with one date argument to retrieve year from given date value

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL GETDATE() FUNCTION with Example

SQL GETDATE() function is used to find the current database system date

Example 30 : Write SQL query to retrieve current date of system

SELECT GETDATE() AS [CURRENT DATE]

In the above query, SQL GETDATE() function is used with SQL select statement to retrieve current date

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL NUMERIC FUNCTIONS

SQL FLOOR() FUNCTION with Example

SQL FLOOR() function is used to find largest integer which is less than or equal to given numerical value

Example 31: Write SQL query to find integer number of 29.90

sql

SELECT FLOOR(29.90) AS 'FLOOR()'

In the above query, SQL FLOOR() function specified with one argument of floating number 29.90 to find integer floor value

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL CEILING() FUNCTION with Example

SQL CEILING() function is used to find smallest integer which is greater than or equal to given numerical value

Example 32: Write SQL query to find integer number of 29.90

sql

SELECT CEILING(29.90) AS CEILING()'

In the above query, SQL CEILING() function specified with one argument of floating number 29.90 to find integer floor value

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL ROUND() FUNCTION with Example

SQL ROUND() function is used to find roundup value of given numeric value

EXAMPLE 33: Write SQL query to find roundup value of given numeric value 29.90

sql

SELECT ROUND(29.90, 0) AS 'ROUND()'

In the above query, SQL ROUND() function specified with two argument

The first argument is a numeric value to be roundup

The second argument is a precision digit

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL CONVERSION FUNCTIONS

SQL CAST() FUNCTION with Example

SQL CAST() function is used to convert given value into another datatype

Example 34: Write SQL query to cast given date datatype value ’01-02-1990’ to varchar datatype

sql

SELECT CAST('1990-02-01' AS VARCHAR) AS 'VARCHAR Datatype'

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL CONVERT() FUNCTION with Example

SQL CONVERT() function is used to convert given value to another datatype

Example 35: Write SQL query to convert given date datatype value ’01-02-1990’ to varchar datatype

sql

SELECT CONVERT(VARCHAR, '1990-02-01') AS 'VARCHAR Datatype'

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

SQL NULL-Related Functions

SQL ISNULL() Function with Example

SQL ISNULL() function is used to replace NULL value with specified value

Example 36: Write SQL query to replace NULL with value ‘ISNULL Function’

sql

SELECT ISNULL(NULL, 'ISNULL Function') AS 'ISNULL Function'

OUTPUT:

35+ SQL Functions Explained in Detail [Practical Examples]

-\*\*\*\*\*\*\*\*\*\*##############\*\*\*\*\*\*\*\*\*\*\*\*\*\*-

SQL User Defined Function

SQL Server allows to create user-defined function using CREATE FUNCTION statement, SQL Server user-defined functions are routines that accept parameters, perform an action, such as a complex calculation, and return the result of that action as a value

There are two main types of user-defined functions in SQL based on the data they return:

1.User-defined Scalar functions

These types of functions return a single value like float, int, varchar, datetime and so on..

2. Table-Valued functions

These functions return tables

Syntax to create User-define Scalar function

sql

CREATE FUNCTION scalar\_function\_name

(

@parameter1 AS datatype

@parameter2 AS datatype

)

RETURNS <return datatype>

AS

BEGIN

RETURN <return value/expression >

END;

Example of User-define Scalar function

Example 37: Create User-define Scalar function to calculate percentage of all students

sql

CREATE FUNCTION percentage\_student

(

@totalmarks int,

@obtainmark int

)

returns int

as

begin

return round(@obtainmark \*100/@totalmarks,0)

end

In the above User-defined SQL function, CREATE FUNCTION statement is used to create user-defined scalar function as percentage\_student

The function contains two parameters @totalmarks and @obtainmarks of int datatype

Return type of the function is int

The function will return percentage of all students based on the records of result table

OUTPUT:

To see the result of above created function we need to call this function within SQL SELECT statement

sql

SELECT student\_id AS 'Student ID', examname AS Semester, dbo.percentage\_student(obtainmark, totalmarks) AS Percentage

FROM tblresult

35+ SQL Functions Explained in Detail [Practical Examples]

Syntax to create User-define Table-valued function

sql

CREATE FUNCTION function\_name

(

@parameter1 dataype,

)

RETURNS TABLE

AS

RETURN

SELECT column1,column2…

FROM

table\_name

WHERE

Condition

Example of User-define Table-valued function

Example 38: Create user-defined function to generate table contains result of semester 3 students

sql

CREATE FUNCTION result\_semester3

(

@semester varchar

)

RETURNS TABLE

AS

RETURN

SELECT examname,totalmarks,obtainmark,pecentage FROM tblresult WHERE examname=@semester

In the above User-defined SQL function, CREATE FUNCTION statement is used to create user-defined scalar function as result\_semester3

The function contains one parameters @semester of varchar datatype

The function will return a table as result set

OUTPUT:

To see the result of above created function we need to call this function with SQL SELECT statement

sql

SELECT examname, totalmarks, obtainmark, pecentage

FROM dbo.result\_semester3('sem3') AS Result

35+ SQL Functions Explained in Detail [Practical Examples]

Summary

In this article of SQL function, we have covered overview of SQL functions with the types of SQL function: Built-in SQL functions and User-defined functions, In the built-in type SQL function we have explained Aggregate functions and Scalar functions with practical examples, also we have explained SQL User-defined function with two types: Scalar User-defined function and Table-valued User-defined function with practical examples

Multi-Statement Table-Valued Function

A user-defined multi-statement table-valued function returns a table variable due to actions performed by the function. In this, a table variable must be explicitly declared and defined whose value can be derived from multiple SQL statements.

--Create function for EmpID,FirstName and Salary of Employee

Create function fnGetMulEmployee()

returns @Emp Table

(

EmpID int,

FirstName varchar(50),

Salary int

)

As

begin

Insert into @Emp Select e.EmpID,e.FirstName,e.Salary from Employee e;

--Now update salary of first employee

update @Emp set Salary=25000 where EmpID=1;

--It will update only in @Emp table not in Original Employee table

return

end

Creates a function called "fnGetMulEmployee" that returns a custom table variable called "@Emp" that has columns for employee ID, first name, and salary. Fills the "@Emp" table with data from the "Employee" table. Modify the salary of the first employee in the "@Emp" database, but without affecting the original "Employee" table, guaranteeing data integrity. As the function's output, it returns the changed "@Emp" table.

User-Defined Function - Multi-Statement Table-Valued Function

--Now call the above created function

Select \* from fnGetMulEmployee()

This query uses the "fnGetMulEmployee" method to retrieve a custom table with employee IDs, first names, and salaries, with the first employee's pay modified to 25000.

User-Defined Function - Multi-Statement Table-Valued Function

--Now see the original table. This is not affected by above function update command

Select \* from Employee

This query gets & displays the original "Employee" table, confirming that its data is unaltered because function modifications were limited to the temporary table.

User-Defined Function - Multi-Statement Table-Valued Function

Difference between Function and Procedure in SQL Server

Feature Function Procedure

Basic principles Functions use the input provided to calculate the output of a program. Procedures use the inputs provided to determine which tasks to complete in what order.

Try-Catch Blocks The try-catch Blocks are not supported by functions. Try-catch Blocks are supported by procedures.

SQL Query Within an SQL query, we can invoke a function. An SQL query cannot call a procedure.

SELECT Function calls may be present in the SELECT statements. Procedure calls cannot ever be made in the SELECT statements.

Return To the caller function or code, a function would return the returned value or control. In contrast, a procedure would return control but neither the caller function nor the code would receive any value from it.

Statements in DML DML statements cannot be used in a function (this includes insert, delete, and update functions). The DML statements are always available in the event of a procedure.

Call A procedure can be used to call a function. No function can be used to call a procedure.

Compilation When we call a function within a program, it gets compiled. The procedures must be compiled once, but if more than one is needed, they can be called repeatedly; a compilation is not required each time.

Expression Expressions must be handled by a function. Expressions do not have to be handled by a process.

Note

Unlike the Stored Procedure, the Function returns only a single value.

Unlike the Stored Procedure, the Function accepts only input parameters.

Unlike the Stored Procedure, the Function is not used to Insert, Update, or Delete data in a database table(s).

Like the Stored Procedure, the Function can be nested up to 32 levels.

User Defined Function can have up to 1023 input parameters while a Stored Procedure can have up to 2100 input parameters.

User Defined Function can't return XML Data Type.

User Defined Function doesn't support Exception handling.

User Defined Function can call only Extended Stored Procedure.

User Defined Function doesn't support set options like set ROWCOUNT etc.