MS SQL Server Simplified

YOUR GATEWAY TO T-SQL

MANZOOR AHMED MOHAMMED
Microsoft Certified Trainer

MS SQL Server Simplified: Your Gateway to T-SQL

Manzoor Ahmed Mohammed

(Microsoft Certified Trainer)

Index

About the Author

Mohammed Manzoor Ahmed is the founder of MTT and a distinguished Microsoft Certified Trainer with over a decade of specialized experience in MS .NET technologies. His career spans more than 15 years as a software development trainer, during which he has established himself as a knowledgeable and engaging educator.

Manzoor's expertise is multifaceted, encompassing both development and education within the MS .NET ecosystem. His passion for teaching complex technologies is evident in his ability to simplify intricate concepts, making them accessible to learners at all levels.

As an author, Manzoor has garnered recognition for his award-winning articles on Code Project, demonstrating his ability to communicate technical concepts effectively through writing. His impact extends to the digital realm, where his YouTube channel has amassed over 2 million views and 13,000 subscribers, further showcasing his talent for delivering valuable educational content.

Manzoor's approach to education is centred on demystifying complex ideas, ensuring that his students gain a deep and practical understanding of the technologies they are learning. His extensive experience, coupled with his commitment to clarity in teaching, makes him a trusted guide for those embarking on their journey in .NET development.

Who This Book Is For

This book is designed for:

1. **IT Graduates:** Recent graduates in Information Technology, Computer Science, or related fields who have a basic understanding of computer operations and are looking to specialize in database management.
2. **Aspiring Database Professionals:** Individuals who aim to start a career in database administration or development and want to build a strong foundation in SQL Server.
3. **Self-Taught Programmers:** Those who have learned programming on their own and want to expand their skills to include database management and SQL.
4. **Career Changers:** Professionals from other IT domains who are looking to transition into database management and need a comprehensive guide to SQL Server.
5. **Students:** Current IT or Computer Science students who want to supplement their coursework with practical knowledge of SQL Server.
6. **IT Enthusiasts:** Anyone with a basic understanding of computers who is interested in learning about database management and SQL Server from the ground up.
7. **Junior Developers:** Software developers who want to improve their database skills to become more well-rounded professionals.

While no prior experience with SQL Server is required, readers should be comfortable with basic computer operations and have a willingness to learn technical concepts. This book will guide you from the fundamentals of databases to advanced querying techniques, making it suitable for beginners while also providing valuable insights for those with some prior exposure to databases.

By the end of this book, readers will have gained the knowledge and skills necessary to work effectively with SQL Server in various professional settings.

What Readers Will Learn:

1. **Getting Started with SQL Server:** Begin your journey by learning how to install Microsoft SQL Server Management Studio (SSMS). You will explore the SSMS interface and learn how to connect to SQL Server instances, setting the foundation for your database management skills.

2. **Fundamentals of Relational Database Management Systems (RDBMS):** Dive into the core concepts of relational databases. You will learn to create databases and tables, implement various types of keys and constraints, and understand the building blocks of database design. This module will give you a solid grasp of RDBMS principles.

3. **SQL Server Data Types and Table Design:** Explore the diverse data types available in SQL Server and master the art of table design. You will learn how to implement relationships between tables using primary and foreign keys, and manage constraints effectively. This knowledge is crucial for creating robust database structures.

4. **Transact-SQL (T-SQL) Essentials:** Get hands-on with T-SQL, the language of SQL Server. You will learn to perform basic data manipulation operations, understand the nuances between different commands, and implement concurrency control. This module will empower you to interact with your database effectively.

5. **Data Retrieval and Manipulation:** Enhance your ability to extract and manipulate data. You will write complex SELECT statements, use aliases and derived columns, sort and filter data efficiently. These skills are fundamental to any data professional and will be used extensively in your database career.

6. **SQL Functions and Data Processing:** Expand your T-SQL toolkit with various built-in functions. You will work with string, mathematical, and datetime functions, as well as powerful aggregate functions. You will also learn to group and summarize data, enabling you to derive meaningful insights from your datasets.

7. **Advanced Querying Techniques:** Take your querying skills to the next level. Master different types of JOINs to combine data from multiple tables, write complex multi-table queries, and implement subqueries and Common Table Expressions. These advanced techniques will allow you to handle complex data retrieval scenarios. Learn the art and science of database design. You will understand key design principles, analyse, and implement complex database structures, and design efficient schemas for real-world scenarios. This module will help you create databases that are performant, scalable, and maintainable.

8. **Views and Stored Procedures:** Discover how to create and manage views and stored procedures. You will learn to perform operations on views and develop stored procedures with parameters. This module also covers implementing control flow, enhancing your ability to create reusable and efficient database code.

9. **Functions, Error Handling, and Transactions:** Delve into creating user-defined functions, implementing robust error handling, and managing transactions. You will understand the critical role of transactions in maintaining data integrity and consistency, preparing you for real-world database management challenges.

10. **Advanced SQL Server Features:** Explore advanced features like triggers, temporary tables, and specialized functions. You will also learn techniques like self-joins and UNION operations. This module rounds out your SQL Server skillset, covering backup and restore operations essential for database administration.

11. **Database Administration Basics:** Cap off your learning with essential database administration tasks. You will generate database scripts, perform backups, and restores, and understand ongoing database design and maintenance tasks. This module provides a glimpse into the world of database administration, preparing you for potential career paths in this field.

By mastering these topics, you will gain a comprehensive understanding of SQL Server, from installation and basic concepts to advanced techniques, preparing you for real-world database management and development tasks.

Chapter 1: Introduction to RDBMS and Getting Started with SQL Server 2022 Express Edition

What is RDBMS?

RDBMS stands for Relational Database Management System. It is a type of database management system that stores data in a structured format using rows and columns, which are organized into tables. Here are some key features and concepts related to RDBMS:

Collection of Tables: An RDBMS consists of a collection of tables, each designed to hold related data.

Tables: A table is composed of rows and columns.

Rows: Each row in a table represents a unique record or entry in that table. For example, in a table of employees, each row would correspond to a different employee.

Columns: Each column represents an attribute or field of the records contained in the table. Continuing with the employee example, columns could include attributes like Employee ID, Name, Job Title, and Salary.

Common RDBMS Examples: Some well-known RDBMSs include:

- Microsoft SQL Server
- Oracle Database
- IBM DB2
- MySQL

Interaction Through SQL: Users interact with RDBMSs using Structured Query Language (SQL). SQL is a powerful language used to query, insert, update, and delete data within the database.

In summary, an RDBMS is crucial for organizing and managing data in a way that allows for easy access and manipulation, making it a foundational technology in data management systems.

What is MS SQL Server?

Microsoft SQL Server is a relational database management system (RDBMS) developed by Microsoft. It is designed to store, retrieve, and manage data for various applications, from small applications to large enterprise systems. Here are some key features and components of MS SQL Server:

Key Features

1. **Relational Database Management:**
   - MS SQL Server organizes data into tables, allowing for easy management and retrieval using SQL (Structured Query Language).
2. **High Performance:**
   - It is optimized for high performance and can handle large volumes of transactions, making it suitable for mission-critical applications.
3. **Scalability:**
   - MS SQL Server can scale from small applications to large data warehouses, accommodating growing data needs.

4. **Security:**
    - It offers robust security features, such as authentication, encryption, and fine-grained access control, to protect sensitive data.
5. **Data Integrity:**
    - MS SQL Server ensures data integrity through constraints, triggers, and referential integrity, guaranteeing that the data remains accurate and reliable.
6. **Advanced Analytics:**
    - Built-in support for advanced analytics and data processing, including support for data mining and machine learning integration.
7. **Business Intelligence:**
    - It includes tools for business intelligence (BI), such as SQL Server Reporting Services (SSRS) and SQL Server Analysis Services (SSAS), to analyse and visualize data.
8. **Integration Services:**
    - SQL Server Integration Services (SSIS) allows for data extraction, transformation, and loading (ETL) operations, enabling data integration from various sources.
9. **Multiple Editions:**
    - MS SQL Server offers various editions tailored for different needs, including Developer, Standard, and Enterprise editions.

Usage Scenarios

- Web Applications: Used as the backend database for web applications, ensuring data persistence and accessibility.
- Enterprise Applications: Supports large-scale enterprise applications requiring robust data handling capabilities.
- Data Warehousing: Utilized for building data warehouses that aggregate data from multiple sources for reporting and analytics.

Conclusion

Overall, MS SQL Server is a powerful and versatile RDBMS that is widely used in different industries and applications. Its combination of performance, security, and advanced analytics capabilities makes it a preferred choice for organizations looking to manage their data effectively.

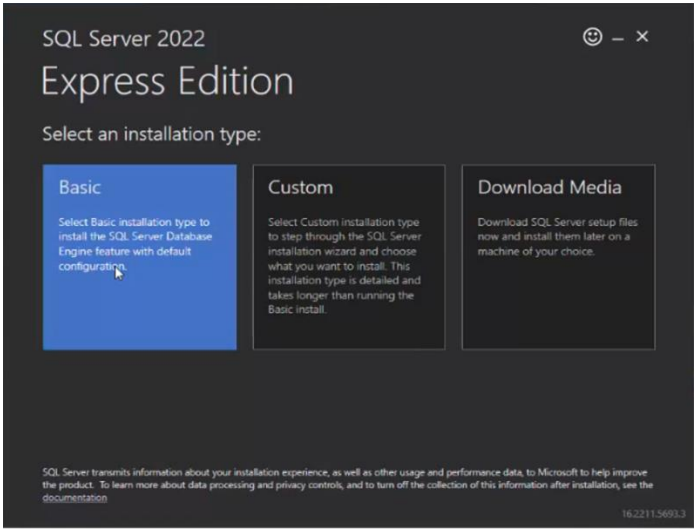
Now we will explore how to install and begin working with SQL Server 2022 Express Edition, a free version of Microsoft's SQL Server. This edition is perfect for learning and small projects.

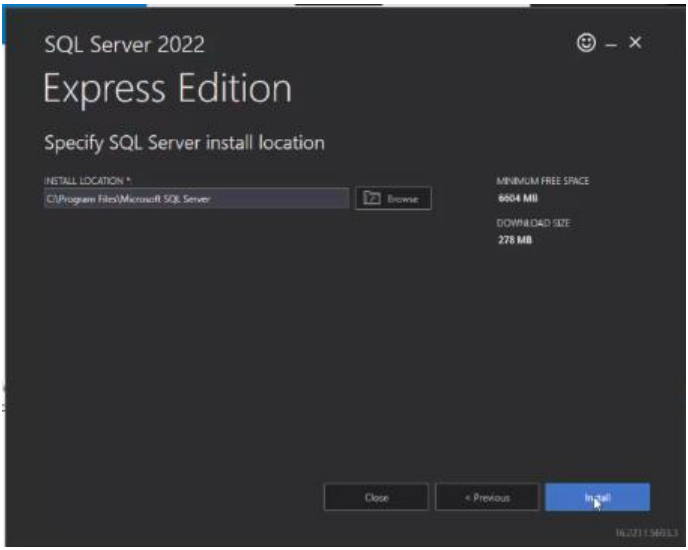
Downloading SQL Server 2022 Express Edition

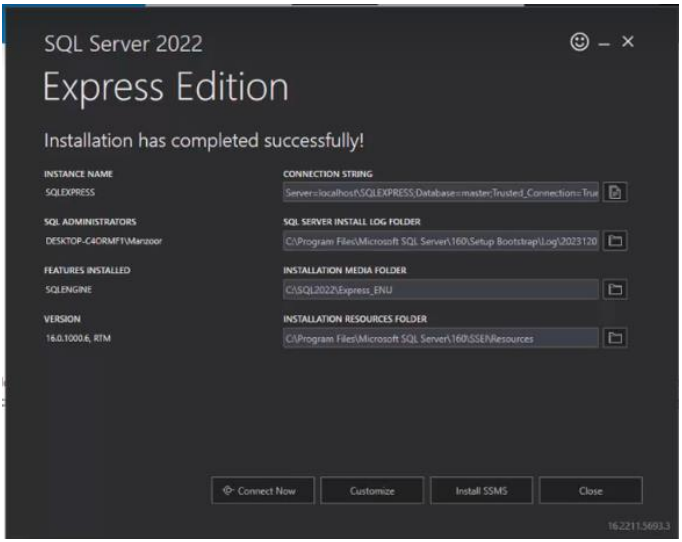
To start, we will search for the latest SQL Server version and proceed to the SQL Server Downloads page. Here, we will specifically look for the SQL Server 2022 Express Edition. Once located, click on the download link to begin the process.

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

After downloading, you will find the setup file in your File Explorer's 'Downloads' folder. Open this folder, locate the executable file, and run it to begin the installation. Choose the basic installation option and accept the license terms to install the Express Edition.




SQL Server 2022
Express Edition
Specify SQL Server install location
INSTALL LOCATION *
C:\Program Files\Microsoft SQL Server
Browse
MINIMUM FREE SPACE
6604 MB
DOWNLOAD SIZE
278 MB
Close
< Previous
Install

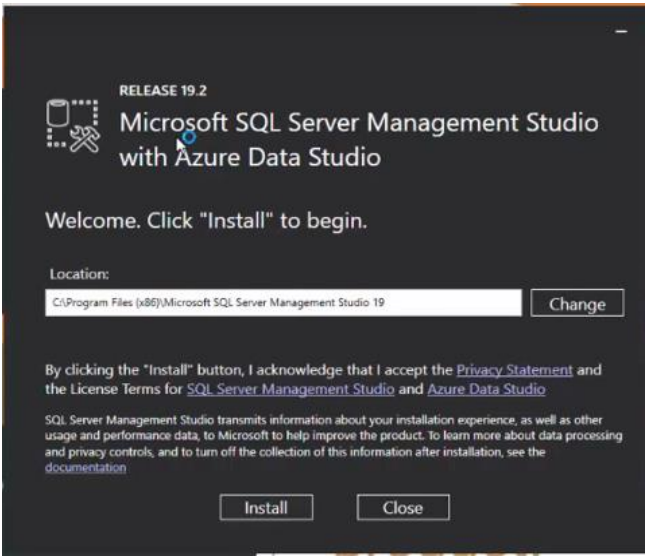

SQL Server 2022
Express Edition
Installation has completed successfully!
INSTANCE NAME
SQLEXPRESS
CONNECTION STRING
SQL ADMINISTRATORS
DESKTOP-C4ORMFT\Marzoor
SQL SERVER INSTALL LOG FOLDER
C:\Program Files\Microsoft SQL Server\160\Setup Bootstrap\Log\20231120
FEATURES INSTALLED
SQLENGINE
INSTALLATION MEDIA FOLDER
C:\SQL2022\Express_ENU
VERSION
16.0.1000.6, RTM
INSTALLATION RESOURCES FOLDER
C:\Program Files\Microsoft SQL Server\160\SSEI\Resources
Connect Now
Customize
Install SSMS
Close

Installing SQL Server Management Studio (SSMS)

With SQL Server Express Edition ready, the next step is to install SQL Server Management Studio (SSMS), an essential tool for managing your SQL Server instances. Search online for the latest version of SSMS, which should be available from Microsoft's website.

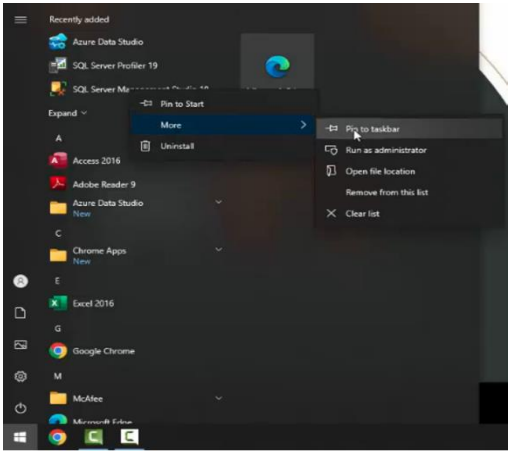
https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16

Download the SSMS setup file, and once complete, open the file from your downloads. The installation wizard will guide you through a straightforward process: simply click 'Next', and then 'Install'.



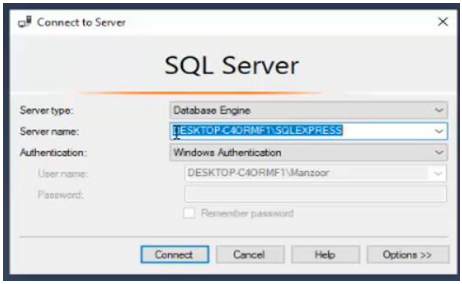
Launching and Pinning SSMS

After installing SSMS, you can locate it from your 'Start' menu under SQL Server Tools. Open SSMS to ensure it is correctly installed. To facilitate quick access, pin SSMS to your taskbar by right-clicking its icon and selecting 'Pin to Taskbar'.



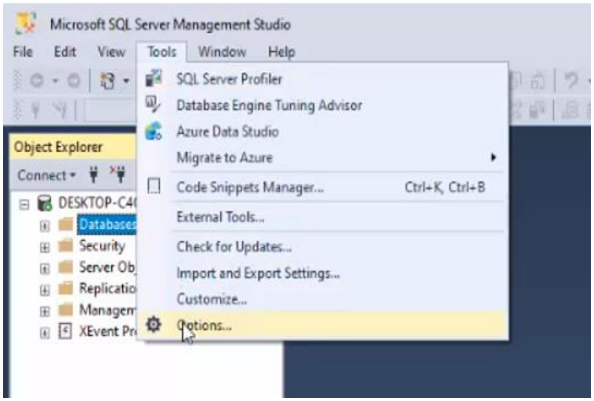
Connecting to the SQL Server

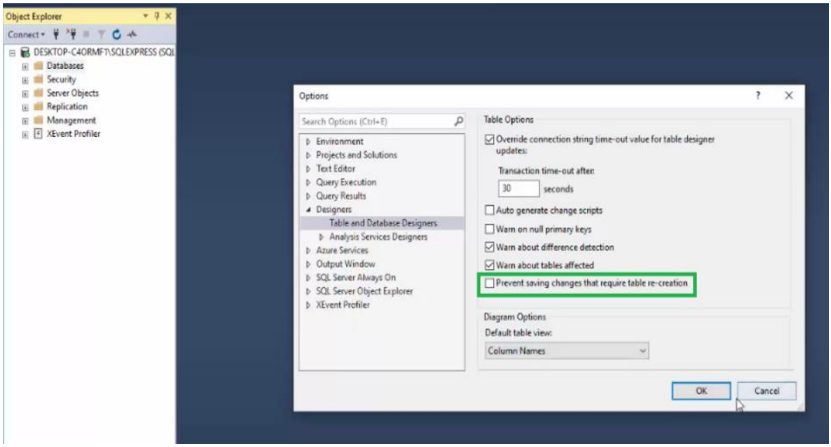
Upon running SSMS, connect to your SQL Server instance. If the server's name does not appear by default, browse local servers, and select yours. Click 'Connect' to access the database server. From here, you can begin creating and managing databases.



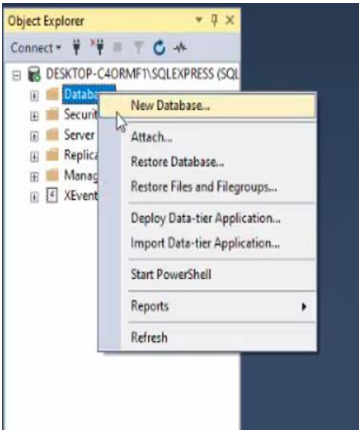
Configuring SQL Server and Basic Usage

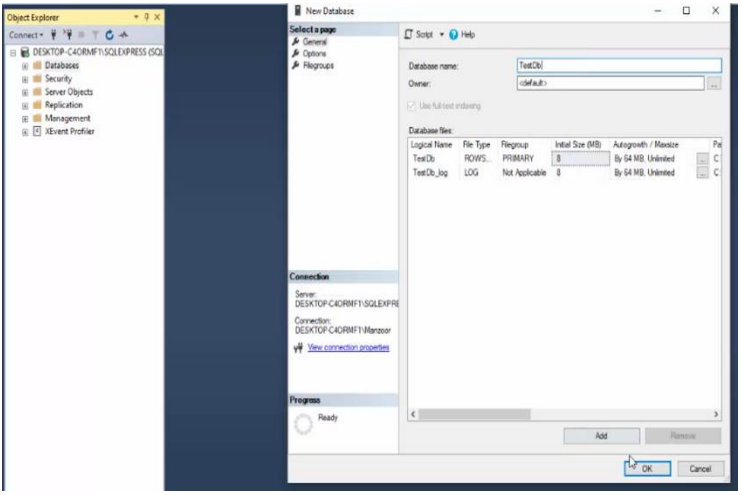
Before creating a database, navigate to Tools > Options > Designers in SSMS. Uncheck the option that prevents changes from being saved to table designs. This configuration allows you to make and save changes to your database structures.

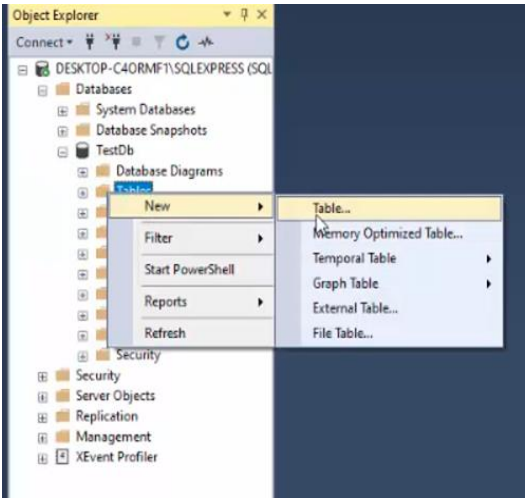


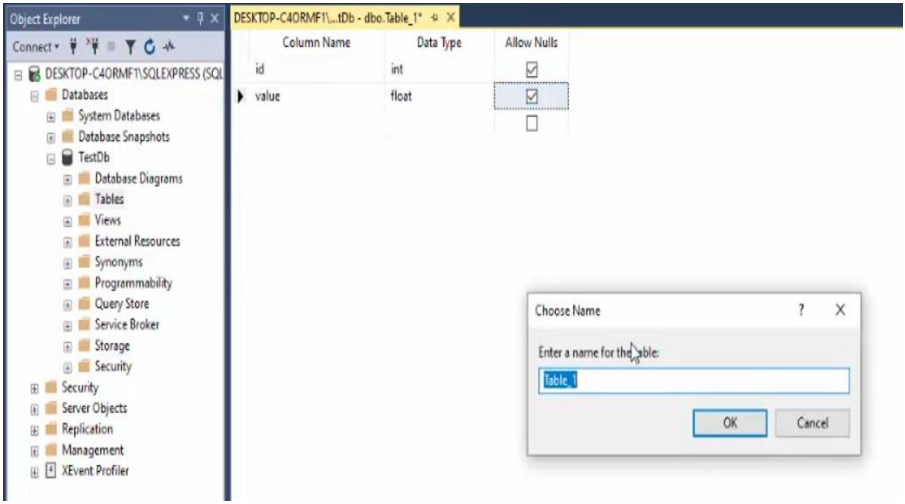


Create a new database named '**TestDB**', then proceed to create a new table within it. To test the server's functionality, define two columns: 'id' (integer) and 'value' (float). Save this table as '**ValueTable**'.



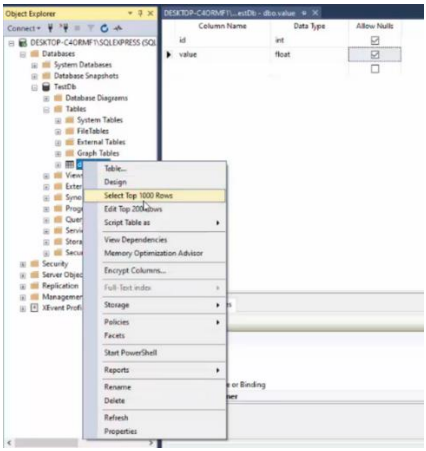


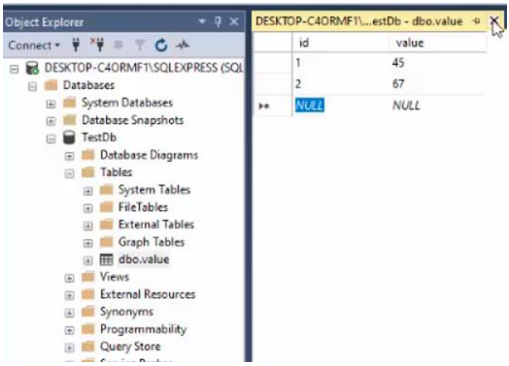

Object Explorer
Connect
DESKTOP-C4ORMF1\SQLEXPRESS (SQL
Databases
System Databases
Database Snapshots
TestDb
Database Diagrams
Tables
New
Filter
Start PowerShell
Reports
Refresh
Table...
Memory Optimized Table...
Temporal Table
Graph Table
External Table...
File Table...
Security
Security
Server Objects
Replication
Management
XEvent Profiler


Object Explorer
Connect
DESKTOP-C4ORMF1\SQLEXPRESS (SQL
Databases
System Databases
Database Snapshots
TestDb
Database Diagrams
Tables
Views
External Resources
Synonyms
Programmability
Query Store
Service Broker
Storage
Security
Security
Server Objects
Replication
Management
XEvent Profiler
DESKTOP-C4ORMF1\...tDb - dbo.Table_1*
Column Name
Data Type
Allow Nulls
id
int
value
float
Choose Name
Enter a name for the table:
Table_1
OK
Cancel

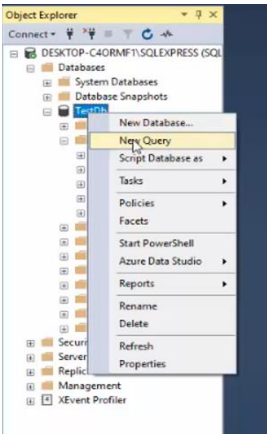
Inserting Data and Running Queries

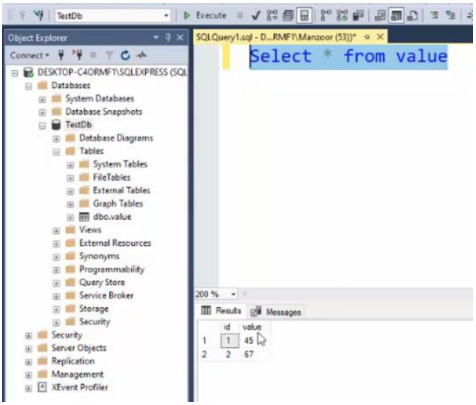
Insert some sample data into your table by right-clicking 'ValueTable', selecting 'Edit Top 200 Rows', and entering your records. After inserting the data, access the query designer by right-clicking on your database and choosing 'New Query'.





Write a simple query, such as `**SELECT * FROM ValueTable**`. Execute this query to view your inserted data. If needed, adjust the font size for better visibility via the settings at the bottom of the query window.





Conclusion

By following these steps, you have successfully installed SQL Server 2022 Express Edition and SQL Server Management Studio. You should now be able to create, manage, and query databases, setting the foundation for more advanced SQL Server tasks.

Chapter 2: Introduction to SQL Server and Database Management

Introduction

This chapter introduces SQL Server and the foundational concepts of database management. It provides insights into creating databases, tables, and understanding key database concepts including primary keys, unique keys, and data types.

The Role of SQL Server

The ultimate goal of any application development is to store and retrieve information efficiently. SQL Server plays a vital role in this process by hosting databases that range from simple to complex structures.

Database Development

You should aim to develop skills to query databases effectively. This involves learning to write queries that can extract information from databases with multiple tables. Understanding database architecture is also crucial, as it enables one to not only query efficiently but eventually design databases to meet specific requirements.

Database Querying

By mastering SQL queries, even within structures containing thousands of tables, you can pull necessary data efficiently. The importance of understanding and designing databases is stressed—once you comprehend database requirements, you can contribute significantly to application development.

Tips on Understanding SQL

You need to understand the structure of a database and what SQL Server solutions entail, including the importance of relational database management systems (RDBMS), their architecture, and how they differ from flat files in terms of handling data complexity.

RDBMS stands for Relational Database Management System. It is a type of database management system (DBMS) that is based on the relational model introduced by E.F. Codd. An RDBMS stores data in a structured format, using rows and columns. It uses SQL (Structured Query Language) for accessing and managing the data within the database.

RDBMS Vs Flat Files

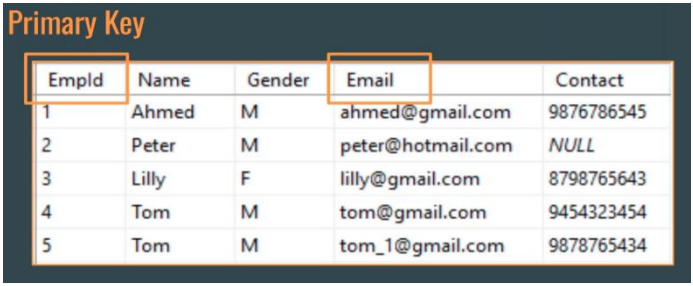
Flat files are like simple spreadsheets where data is stored in a straightforward, text-based format without any connections between the entries, making them easy to use for basic lists or logs but tricky when you need to find specific information quickly or relate different pieces of data. In contrast, an RDBMS (Relational Database Management System) is like a well-organized library where data is stored in tables that can interact with each other, allowing for efficient searching and management, which is especially useful when dealing with large amounts of complex, related information.

Key Concepts in SQL

Primary and Unique Keys

Identifying rows uniquely using primary keys ensures data integrity and helps maintain organized data retrieval systems. Primary keys are unique to each record, are not nullable, and there can be only one primary key per table. Unique keys, however, can have one nullable value and there may be multiple unique keys per table.

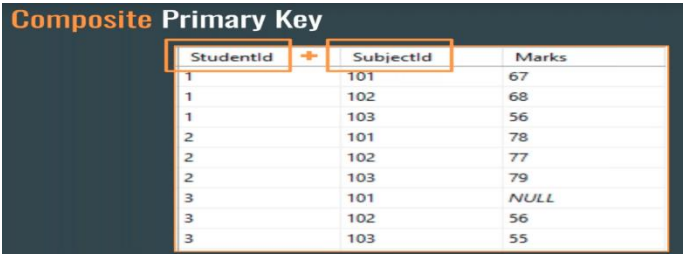
Here **EmpId** is Primary Key and **Email** Is Unique Key



EmpId	Name	Gender	Email	Contact
1	Ahmed	M	ahmed@gmail.com	9876786545
2	Peter	M	peter@hotmail.com	*NULL*
3	Lilly	F	lilly@gmail.com	8798765643
4	Tom	M	tom@gmail.com	9454323454
5	Tom	M	tom_1@gmail.com	9878765434

Composite Keys

In scenarios where a single attribute does not suffice to uniquely identify a record, composite keys—comprising two or more attributes—are used. Understanding composite keys is crucial for handling relationships in complex databases.



StudentId	SubjectId	Marks
1	101	67
1	102	68
1	103	56
2	101	78
2	102	77
2	103	79
3	101	*NULL*
3	102	56
3	103	55

Practical Exercises

Database and Table Management

The chapter includes step-by-step instructions on creating databases and tables, setting primary keys, and introducing constraints to enforce business rules within the data. Emphasis is placed on using SQL Server Management Studio's (SSMS) GUI for these tasks.

Setting Default Values and Constraints

Participants will also work on setting default values for columns and enforcing constraints to ensure data validity, which is critical for maintaining data integrity.

Assignments and Exercises

Assignments are given to solidify learning. Participants are encouraged to experiment with SQL Server Management Studio, navigating and manipulating database objects, and understanding the importance of database options and settings.

Conclusion

The session concludes with a recap of essential database concepts covered, encouraging participants to review SQL tutorials and practice assignments to enhance their familiarity with SQL Server tools and techniques. This ensures participants are well-prepared for more advanced topics in subsequent modules.

Chapter 3: Data Types in MS SQL Server

Understanding the data types available in MS SQL Server is essential for effective database design and manipulation. Each data type defines the kind of data that can be stored in a field, ensuring that the data is consistent and accurate. In this chapter, we will explore the various data types offered by MS SQL Server, categorized into strings, numbers, date and time, unique identifiers, and Boolean values.

1. String Data Types

String data types in MS SQL Server are used to store character data. Depending on the application's needs, you can choose fixed-length or variable-length data types, and whether they support Unicode characters.

1.1. char(n)

Description: The char(n) data type is used for fixed-length non-Unicode character strings. The "n" specifies the string length, which means that a char (10) will always store 10 characters. If the string is shorter than the specified length, it will be padded with spaces.

Use Case: Ideal for storing values that have a consistent length, such as fixed-length codes (e.g., country codes or product ID).

1.2. varchar(n)

Description: The varchar(n) data type stands for variable-length non-Unicode character strings. The "n" specifies the maximum length of the string. Unlike char, varchar only uses

as much space as needed to store the string, plus 2 bytes to store its length.

Use Case: Useful for storing strings with varying lengths, such as names or addresses where the length can differ significantly.

1.3. nchar(n)

Description: Similar to char, but nchar(n) is used for fixed-length Unicode character strings. This allows for storage of characters from various languages and scripts.

Use Case: Ideal for cases where you need a fixed-length string with international character support, such as names in different languages.

1.4. nvarchar (n)

Description: The nvarchar (n) data type is similar to varchar, but it is used for variable-length Unicode character strings. It allows the storage of multiple languages and scripts.

Use Case: Recommended for storing user input that may contain characters from different languages, such as user names or comments.

2. Number Data Types

MS SQL Server provides various numeric data types to store integer and floating-point numbers. Below are descriptions of the most used number data types:

2.1. int, bigint, smallint, and tinyint

int: A 4-byte integer data type that can store values from -2,147,483,648 to 2,147,483,647.

bigint: An 8-byte integer data type that can hold larger integer values, with a range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

smallint: A 2-byte integer data type with a range of -32,768 to 32,767, suitable for smaller integer values.

tinyint: A 1-byte integer data type that can store values from 0 to 255, most useful for storing small numbers.

2.2. decimal and numeric

decimal (p, s) and numeric (p, s): Both types store fixed-point numbers. The "p" defines the precision (total number of digits), and "s" defines the scale (number of digits to the right of the decimal point). Decimal data types maintain exact precision, making them ideal for financial calculations.

2.3. float and real

float: A floating-point data type, where the precision can vary. It can accommodate a wide range of values, but there may be some precision loss.

real: Similar to float, but with less precision (4 bytes). It's used when less precision is acceptable.

2.4. money and smallmoney

money: A data type for storing currency values, with four decimal places. It can hold values from -2^63 to 2^63–1.

smallmoney: Similar to money but can only store values from -214748.3648 to 214748.3647. It is useful for scenarios where you deal with smaller currency amounts.

3. Date and Time Data Types

Date and time data types in SQL Server are crucial for managing temporal data. They provide flexibility in how date and time values are stored and manipulated.

3.1. date

Description: Stores date values (year, month, day) without time, taking up 3 bytes. It can hold dates ranging from 0001-01-01 to 9999-12-31.

Use Case: Suitable for scenarios where only the date is relevant, such as birthdays or event dates.

3.2. time

Description: Stores time values (hours, minutes, seconds, and fractions of a second) without date, using 3 to 5 bytes depending on precision (fractions of a second).

Use Case: Useful when you need to capture time of day without the associated date, such as clock-in and clock-out times.

3.3. datetime

Description: Combines date and time into one data type, using 8 bytes. It can represent dates between 1753-9999 and precise to 3.33 milliseconds.

Use Case: Commonly used for timestamps in applications where both date and time are important, such as order entries or logging.

3.4. datetime2

Description: An extension of the datetime type, allowing for a larger range of dates (0001-01-01 to 9999-12-31) and greater precision (up to 7 decimal places for seconds).

Use Case: Preferred over datetime in scenarios requiring high precision and a broader range of dates.

3.5. smalldatetime

Description: Combines date and time but with a limited range. It takes 4 bytes and can represent dates from 1900-2079, with minute precision (no seconds).

Use Case: Useful for applications where high precision in time is not necessary, but date and time are still relevant, like scheduling.

4. Other Data Types

4.1. Uniqueidentifier

Description: This data type is used to store globally unique identifiers (GUIDs). It takes 16 bytes and is often used for primary keys in distributed systems.

Use Case: Useful when you need a unique identifier across multiple databases or systems, such as in distributed applications.

4.2. Bit

Description: A Boolean data type that can store one of three states: 0 (false), 1 (true), or NULL. It uses only 1 bit of storage.

Use Case: Ideal for representing binary choices or flags, such as whether an account is active or inactive.

Conclusion

Understanding the various data types in MS SQL Server is crucial for effective database design. Choosing the appropriate data type can greatly affect performance, storage efficiency, and data integrity. From string data types for textual information to specialized types for date and time, MS SQL Server offers a comprehensive set of options to meet a wide range of application requirements. For ranges of the various data types, you can refer to additional resources such as W3Schools SQL Data Types.

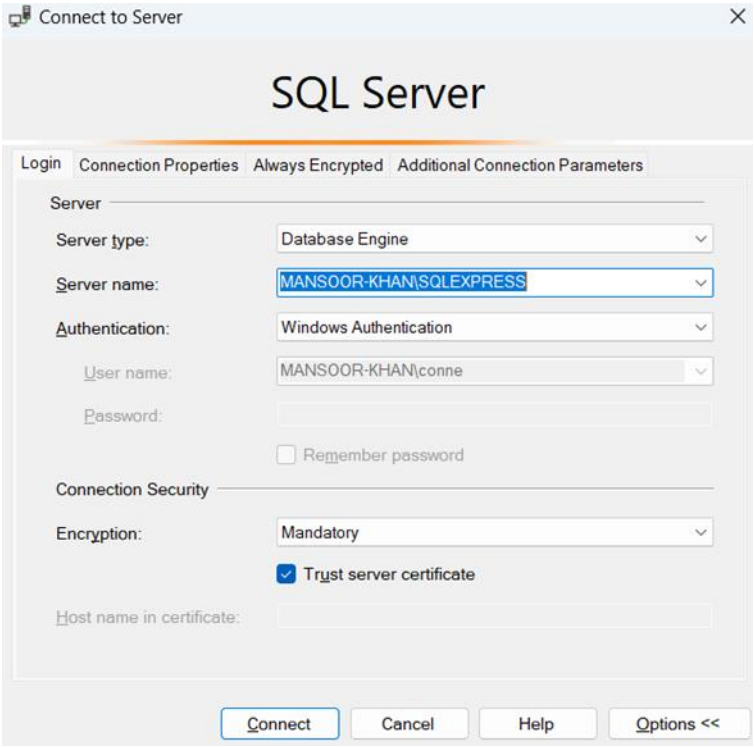
Chapter 4: Database Creation in MS SQL

Creating 'OrgDb' Database with Department and Employee Tables Using SSMS GUI

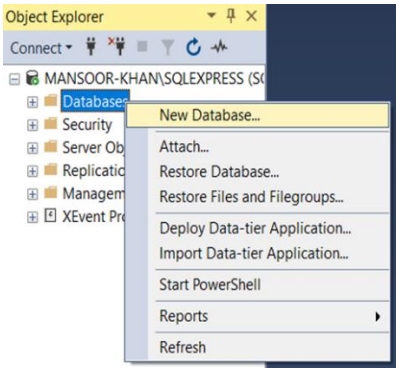
Creating a database and tables using the SQL Server Management Studio (SSMS) user interface without writing SQL scripts can be done in the following steps:

Step 1: Create a Database (OrgDb)

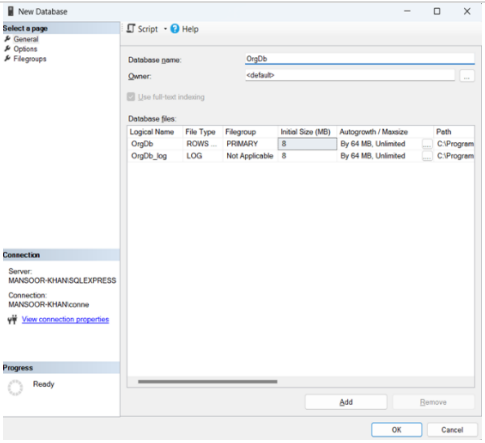
1. **Open SSMS**: Launch SQL Server Management Studio and connect to your database server.



2. **Create a New Database**:
   - In the "Object Explorer", right-click on the "Databases" folder.
   - Select "New Database..." from the context menu.

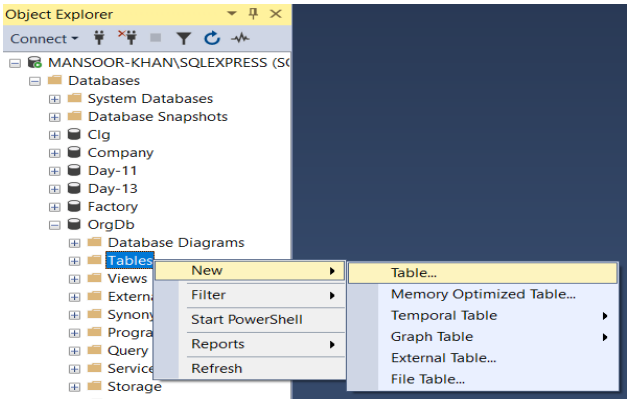


3. **Define the Database**:
   - In the "New Database" dialog, enter OrgDb in the "Database name" field.
   - Configure any other settings if needed, though default settings should suffice for a basic setup.
   - Click OK to create the database.



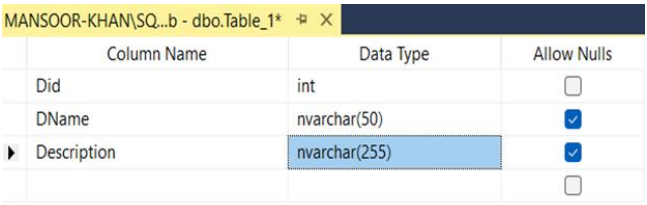
Step 2: Create the Department Table

1. **Navigate to the Database**:
    - In "Object Explorer", expand the newly created OrgDb database.
    - Right-click on the "Tables" folder.
    - Select "New Table".

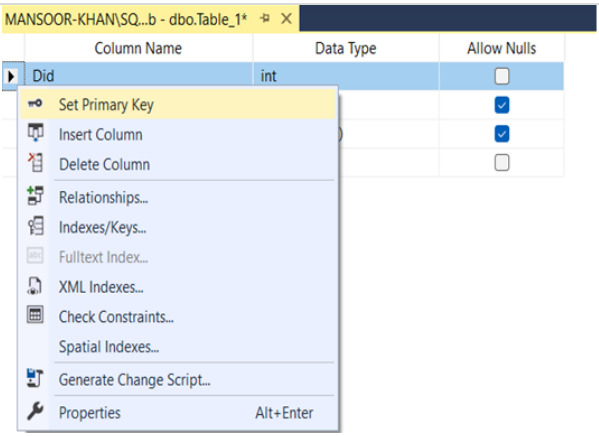


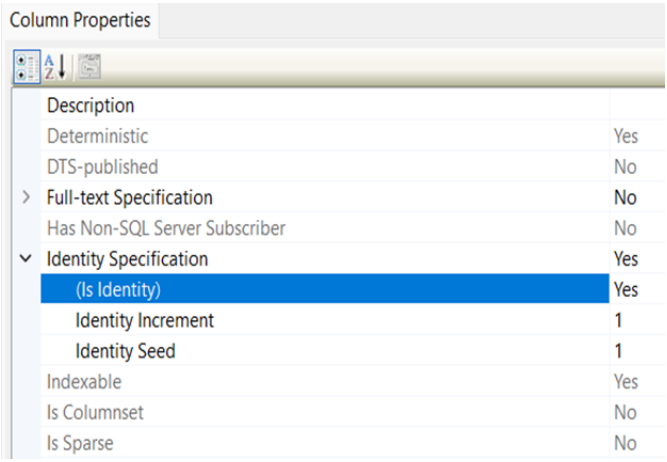
2. **Define Columns for Department Table**:

- **Did**: In the first row under "Column Name", type Did. Change the data type to int, and uncheck the "Allow Nulls" checkbox.
- **DName**: In the next row, type DName and set the data type to nvarchar(50) to allow strings up to 50 characters.
- **Description**: Finally, add a Description column with the nvarchar (255) data type to store detailed text.

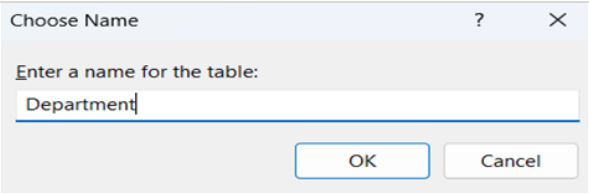


3. **Set Primary Key and Identity for Department Table**:
   - Right-click the row selector for the Did column.
   - Select "Set Primary Key".
   - With Did column selected, scroll to "Column Properties" at the bottom and expand "Identity Specification".
   - Set "Is Identity" to Yes, which auto-increments integer values for this column.



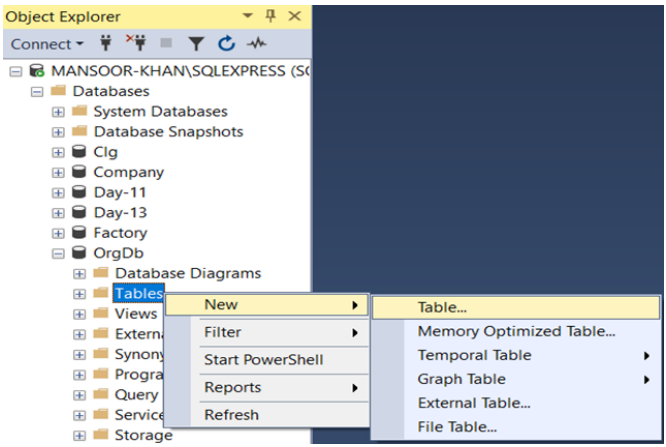


4. **Save the Department Table**:
   - Click the Save icon (or press Ctrl + S).
   - Enter Department as the table name and click OK.



Step 3: Create the Employee Table

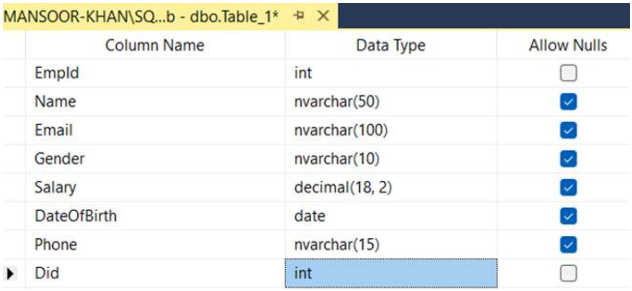
1. **Create New Table**:
   - Right-click on the "Tables" folder under the OrgDb database.
   - Select "New Table".



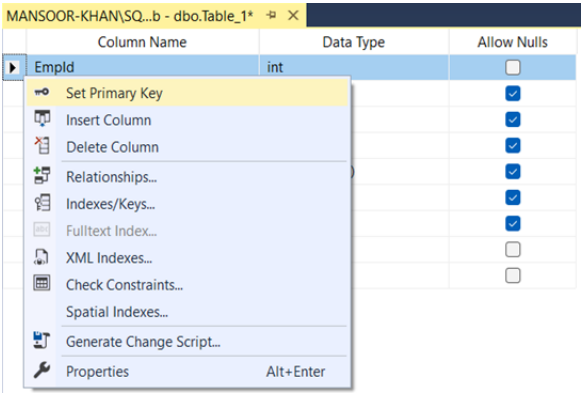
2. **Define Columns for Employee Table**:

   - **EmpId**: Type EmpId, set its data type to int, and ensure "Allow Nulls" is unchecked.
   - **Name**: Enter Name, with a data type of nvarchar (50).
   - **Email**: Add Email, with a data type of nvarchar (100).
   - **Gender**: Enter Gender, using a data type such as char(1) or nvarchar(10).

- **Salary**: Add Salary, using a data type of decimal (18, 2).
- **Date Of Birth**: Enter DateOfBirth, and set the data type to date.
- **Phone**: Add Phone, with a data type of nvarchar (15).
- **DId**: Finally, type DId, and set the data type to int to reference the department.

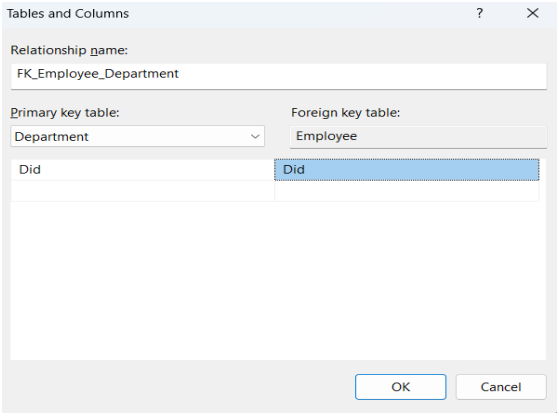


3. **Set Primary Key for Employee Table**:
   - Right-click the row selector for the EmpId column.
   - Choose "Set Primary Key".



4. **Define Foreign Key**:
   - After saving the table, right-click the DId column row selector.

- Choose "Relationships...".
- In the Foreign Key Relationships dialog, click "Add".
- Select "Tables and Columns Specification", and then click the ellipsis (...) button.
- In the dialog, set the "Primary key table" to Department and "Foreign key table" to Employee.
- Map DId in the Employee table to Did in the Department table.
- Click OK to confirm and close the dialog.



5. **Save the Employee Table**:
   - Click the Save icon (or press Ctrl + S).
   - Name the table Employee and click OK.

This process will set up your OrgDb database with the **Department** and **Employee** tables using the SSMS GUI.

Creating 'OrgDb' Database with Department and Employee Tables Using Script

To create a database and tables using a script in SQL Server, you can use the following SQL statements. This script will create the OrgDb database, along with the Department and Employee tables, and include appropriate primary and foreign key constraints.

```
 -- Create the OrgDb database
CREATE DATABASE OrgDb;
go
-- Use the OrgDb database
USE OrgDb;
go
-- Create the Department table
CREATE TABLE Department
(Did INT PRIMARY KEY IDENTITY(1, 1),
-- Identity column auto-increments
Dname NVARCHAR(50) NOT NULL,
Description NVARCHAR(255) NULL);
-- Create the Employee table
CREATE TABLE Employee
(Empid INT PRIMARY KEY IDENTITY(1, 1),
-- Identity column auto-increments
Name        NVARCHAR(50) NOT NULL,
Email       NVARCHAR(100) NOT NULL,
Gender      NVARCHAR(10) NULL,
Salary      DECIMAL(18, 2) NOT NULL,
Dateofbirth DATE NOT NULL,
Phone       NVARCHAR(15) NULL,
Did INT,
FOREIGN KEY (Did) REFERENCES Department(Did)
```

```
-- Foreign key constraint);
```

Explanation:

- **Create Database**: The CREATE DATABASE statement initializes a new database named OrgDb.
- **Use Database**: USE OrgDb; sets the context to the OrgDb database for subsequent operations.
- **Create Department Table**:
  - Did is an integer column set as a primary key with the IDENTITY property for automatic incrementation.
  - DName is a non-nullable text field for department names.
  - Description is optional and can hold additional information about the department.
- **Create Employee Table**:
  - EmpId is the primary key with automatic incrementation.
  - The other fields capture employee information such as name, email, gender, salary, date of birth, and phone.
  - DId is an integer that acts as a foreign key linking to the Did column in the Department table, establishing a relationship between employees and departments.

This script can be executed in SQL Server Management Studio (SSMS) by opening a new query window, pasting this script into it, and running it to create the database & tables as specified.

SQL Script for Populating Department and Employee Tables with Sample Data

Below is a script to insert 3 records into the Department table and 15 records into the Employee table without using a WHILE loop. The script ensures that the foreign key relationships between the tables are preserved.

```
--Insert records into the Department table
INSERT INTO Department (Dname, Description)
VALUES
('HR', 'Human Resources'),
('IT', 'Information Technology'),
('CS', 'Customer Support');

-- Insert records into the Employee table
INSERT INTO Employee
(Name, Email, Gender, Salary, Dateofbirth, Phone, Did)
VALUES
('John Doe','john.doe@ac.in','M',50000,'1985-03-02','555-0101',1),
('Jane Su','jane.su@ac.in','F',60000, '1990-07-15','555-0102',2),
('James D','james.d@ac.in','M',55000, '1988-09-21','555-0103',3),
('Emily Dav','emily.d@ac.in','F',65000, '1992-12-11','555-0104',1),
('Michel B','michel.b@ac.in','M',61000, '1983-04-17','555-0105',2),
('Lisa Web', 'lisa.w@ac.in', 'F', 48000, '1991-05-23', '555-0106', 3),
('David P', 'david.p@ac.in', 'M',53000, '1984-11-28', '555-0107',1),
('Susan G','susan.g@ac.in','F',59000,'1986-06-19','555-0108',2),
('Robert K','robert.k@ac.in','M',62000,'1989-01-31','555-0109',3),
```

```
('Nancy Y','nancy.y@ac.in','F',61500,'1993-09-12','555-0110',1),
('Charles','charles@ac.in','M',46500,'1978-07-27','555-0111',2),
('Mary Lok','mary.l@ac.in','F',47000,'1982-02-05','555-0112',3),
('Thomas J','thomas.j@ac.in','M',54000,'1987-04-14','555-0113',1),
('Hobes A','hobes.a@ac.in','F',34000,'1997-01-14','555-0114',3),
('Patric P','patric.p@ac.in','F',49500, '1994-12-30','555-0114',2);
```

Explanation:

- **Department Insertions**: Three department entries are added to represent typical organizational departments: HR, IT, and Finance.
- **Employee Insertions**: A total of 15 records are inserted into the Employee table:
  - Each employee record is constructed with unique values for Name, Email, Gender, Salary, DateOfBirth, and Phone.
  - DId is assigned to ensure a balance across departments. The insertion pattern (using integers 1, 2, or 3) maps each employee to a department.

This straightforward insertion respects the established database schema and the foreign key constraint set between the Employee's DId and the Department's Did. Adjust the values as necessary to suit your specific scenario or testing needs.

Chapter 5: Transact-SQL (T-SQL) Essentials: Understanding SQL and Query Writing

Introduction

In this chapter, we will explore various Structured Query Language (SQL) operations such as INSERT, DELETE, TRUNCATE, and UPDATE, and conduct a detailed examination of concurrency control techniques like Optimistic Concurrency Control and Last-In-Win. Through practical examples and dummy data, you will gain a comprehensive understanding of how these operations function and their practical applications in database management.

SQL Operations Explained

1. INSERT Query

The INSERT statement is used to add new rows to a table. Let's explore this through the **Department** and **Employee** tables.

Department Table:

```
INSERT INTO Department VALUES ('HR', 'Human Resource')
INSERT INTO Department VALUES
('PDG', 'Product Development Group');
INSERT INTO Department (DName, Description) VALUES
('HR1', 'Human Resource1');
INSERT INTO Department (Description, DName) VALUES
('Human Resource2', 'HR2');
```

Explanation: The first two queries insert department names and descriptions directly. The third and fourth queries specify the column names explicitly, allowing flexibility in column order.

Dummy Records for Department:

DName	Description
IT	Information Technology
FIN	Finance
MARK	Marketing
OPS	Operations
LEG	Legal

Employee Table:

```
INSERT INTO Employee VALUES
(1, 'Raju', 'raju@gmail.com', 'M', 9000, '03/24/2000', '9878765453',
 1001);
INSERT INTO Employee VALUES
(2, 'Prakash', 'prakash@gmail.com', DEFAULT, 90000, '03/24/20
00', '9878765453', 1001);
```

Explanation: The DEFAULT keyword is used to insert the default value for the gender column in the second record.

Dummy Records for Employee:

Emp Id	Name	Email	Gen	DOB	Phone	Dept Id
3	Anita	anita@act.in	F	06/15/1998	9123456789	1002
4	Mohan	mohan@gmail.com	M	12/12/1997	9234567890	1003
5	Rhea	rhea@act.in	F	11/25/1996	9345678901	1004
6	Suresh	suresh@act.in	M	05/14/1995	9456789012	1005
7	Leena	leena@act.in	F	09/01/1994	9567890123	1006

2. DELETE and TRUNCATE Queries

DELETE Query:

```
DELETE FROM Employee WHERE EmpId=9;
DELETE FROM Employee;
```

Explanation: The first DELETE query removes a specific employee by EmpId. The second query deletes all records from the Employee table.

TRUNCATE Query:

```
TRUNCATE TABLE Employee;
```

Explanation: TRUNCATE deletes all rows from a table but does not log individual row deletions, making it faster than DELETE without a WHERE clause.

TRUNCATE vs DELETE

TRUNCATE is more efficient for deleting all records due to less log space usage, whereas **DELETE** can be used with conditional statements to remove specific records.

3. UPDATE Query

The UPDATE statement modifies existing data within a table. Below is how it is applied to the Employee table:

```
UPDATE Employee
SET Name='Ravi', Email='ravi@gmail.com', Salary=87000
WHERE EmpId=2;
```

Explanation: This updates the Name, Email, and Salary for the employee with EmpId 2.

Concurrency Control

Concurrency control is crucial for managing simultaneous transactions in a database to ensure consistency.

1. Optimistic Concurrency Control

```
UPDATE Employee
SET Salary = 65000
WHERE EmpId = 2 AND Salary = 87000;
```

Explanation: This method checks the current value (Salary=87000) before updating, ensuring no other transaction has modified it.

2. Last-In-Win

```
UPDATE Employee
SET Salary = 73000
WHERE EmpId = 2;
```

Explanation: This method updates the Salary to 73000 regardless of any changes made by other transactions, thus the last transaction persists.

Conclusion

This chapter provided an in-depth look at SQL operations for manipulating database entries, outlining the differences between DELETE and TRUNCATE, and offering insights into concurrency control mechanisms. Understanding these concepts is fundamental for efficient database management and handling multi-user environments. By practicing with these operations and exploring scenarios like concurrency handling, database administrators can maintain data integrity and performance.

Chapter 6: Data Retrieval and Manipulation

Introduction

Data retrieval and manipulation are crucial processes in managing information within databases. They enable users to query, format, and analyze data efficiently using SQL (Structured Query Language). This chapter delves into various SQL SELECT statements, illustrating how to retrieve specific aspects of data from a hypothetical Employee table. Each section provides multiple queries, detailed explanations, and example records to facilitate understanding.

SQL Queries Explained

1. Selecting All Columns

```
SELECT * FROM Employee;
```

Explanation: This query retrieves all columns and rows from the Employee table, providing a comprehensive view of each employee's information for analysis.

Emp Id	Name	Email	Gender	Salary	DOB	Did
1	Raju	raju@gmail.com	M	9000	2000-03-24	101
2	Prakas	prakash@gmail.com	M	90000	2000-03-24	102
3	Anita	anita@gmail.com	F	50000	1998-06-15	101
4	Leena	leena@gmail.com	F	62000	1994-09-01	103

2. Selecting Specific Columns

```
SELECT EmpId, Name, Email FROM Employee;
```

Explanation: This query fetches only the EmpId, Name, and Email of all employees, allowing focused analysis on these specific attributes without unnecessary data.

EmpId	Name	Email
1	Raju	raju@gmail.com
2	Prakash	prakash@gmail.com
3	Anita	anita@gmail.com
4	Leena	leena@gmail.com
5	Mohan	mohan@gmail.com

3. Selecting with Aliasing

```
SELECT EmpId AS EmployeeId, Name AS 'Employee Name',
Email FROM Employee E;
```

Explanation: This query uses aliases to rename columns for clarity, making output easier to read and understand, especially when presenting to stakeholders.

EmployeeId	Employee Name	Email
1	Raju	raju@gmail.com
2	Prakash	prakash@gmail.com
3	Anita	anita@gmail.com
4	Leena	leena@gmail.com
5	Mohan	mohan@gmail.com

4. More Column Selection with Aliasing

```
SELECT E.EmpId, E.Name, E.Salary FROM Employee E;
```

Explanation: This query selects columns using aliases to represent the Employee table, providing a clear structure when analysing employee salaries alongside their identifiers.

EmpId	Name	Salary
1	Raju	9000
2	Prakash	90000
3	Anita	50000
4	Leena	62000
5	Mohan	55000

5. Selecting Gender Column

```
SELECT Gender FROM Employee;
```

Explanation: The query retrieves the Gender column, enabling gender-based analysis of employees, which could be useful for diversity metrics and reporting purposes.

Gender
M
M
F
F
M

6. Selecting Distinct Values

```
SELECT DISTINCT Gender FROM Employee;
```

Explanation: This query returns unique gender values from the Employee table, helpful for understanding gender distribution among employees without duplicates in results.

Gender
M

F

7. Selecting Distinct Did Values

```
SELECT DISTINCT Did FROM Employee;
```

Explanation: This query fetches unique Did values, showing the various departments employees belong to, which aids in departmental analysis and reporting.

Did
101
102
103

8. Selecting Distinct Did and Gender Values

```
SELECT DISTINCT Did, Gender FROM Employee;
```

Explanation: This query retrieves unique combinations of Did and Gender values, allowing assessment of gender representation across departments within the organization.

Did	Gender
101	M
101	F
102	M

103	F

9. Selecting EmpId, Name, and Salary

```
SELECT EmpId, Name, Salary FROM Employee;
```

Explanation: This query fetches EmpId, Name, and Salary columns, providing essential employee information useful for salary benchmarking and comparative analysis.

EmpId	Name	Salary
1	Raju	9000
2	Prakash	90000
3	Anita	50000
4	Leena	62000
5	Mohan	55000

10. Calculating HRA

```
SELECT EmpId, Name, Salary AS BS, Salary * 12 / 100 AS HRA
 FROM Employee;
```

Explanation: This query calculates HRA as 12% of the Salary, displaying both the basic salary and the calculated housing allowance for each employee.

EmpId	Name	BS	HRA
1	Raju	9000	1080
2	Prakash	90000	10800
3	Anita	50000	6000
4	Leena	62000	7440
5	Mohan	55000	6600

11. Total Salary Calculation

```
SELECT EmpId, Name, Salary AS BS, Salary * 12 / 100 AS HRA
, (Salary + (Salary * 12 / 100)) AS GS FROM Employee;
```

Explanation: This query calculates the Gross Salary (GS) by adding HRA to the Basic Salary (BS), providing a comprehensive view of employee compensation structures.

EmpId	Name	BS	HRA	GS
1	Raju	9000	1080	10080
2	Prakash	90000	10800	100800
3	Anita	50000	6000	56000
4	Leena	62000	7440	69440
5	Mohan	55000	6600	61600

12. Concatenating 'Mr.' with Name

```
SELECT EmpId, ('Mr.' + Name) AS EmployeeName, Gender F
ROM Employee;
```

Explanation: This query appends "Mr." to each employee's name based on gender, producing a friendly welcome format, and enhancing employee representation in results.

EmpId	EmployeeName	Gender
1	Mr.Raju	M
2	Mr.Prakash	M
3	Ms.Anita	F
4	Ms.Leena	F
5	Mr.Mohan	M

13. Gender-Based Name Title

```
SELECT EmpId, CASE WHEN Gender='M' THEN 'Mr.' + Nam
e WHEN Gender='F' THEN 'Ms.' + Name END AS EmployeeN
ame, Gender FROM Employee;
```

Explanation: This query introduces conditional logic, appending either "Mr." or "Ms." to the employee's name based on their gender, enhancing personalization of user interaction.

EmpId	EmployeeName	Gender
1	Mr.Raju	M
2	Mr.Prakash	M
3	Ms.Anita	F
4	Ms.Leena	F
5	Mr.Mohan	M

14. Salary Grading

```
SELECT EmpId, Name, Salary, CASE WHEN Salary >= 10000 AND Salary <= 50000 THEN 'B' WHEN Salary >= 50001 AND S alary <= 100000 THEN 'A' END AS Grade FROM Employee;
```

Explanation: This query categorizes employees into grades based on salary ranges, providing insights for management decisions surrounding compensation reviews and raises.

EmpId	Name	Salary	Grade
1	Raju	9000	NULL
2	Prakash	90000	A
3	Anita	50000	B
4	Leena	62000	A
5	Mohan	55000	A

15. Alternate Salary Grading

```
SELECT EmpId, Name, Salary, CASE WHEN Salary >= 10000 AND Salary <= 50000 THEN 'B' ELSE 'A' END AS Grade FRO M Employee;
```

Explanation: This query assigns grades based on a simpler criterion – any salary above a specified amount receives an 'A', while all else gets a 'B'.

EmpId	Name	Salary	Grade
1	Raju	9000	B
2	Prakash	90000	A
3	Anita	50000	A
4	Leena	62000	A
5	Mohan	55000	A

16. Sorting by Name

```
SELECT * FROM Employee ORDER BY Name;
```

Explanation: This query fetches all data while sorting the result set alphabetically by the employee names, facilitating easier navigation through employee details.

EmpId	Name	Email	Gender	Salary	DOB	Did
3	Anita	anita@gmail.com	F	50000	1998-06-15	101
4	Leena	leena@gmail.com	F	62000	1994-09-01	103
5	Mohan	mohan@gmail.com	M	55000	1997-12-12	102
1	Raju	raju@gmail.com	M	9000	2000-03-24	101
2	Prakash	prakash@gmail.com	M	90000	2000-03-24	102

17. Sorting by Salary Descending

```
SELECT * FROM Employee ORDER BY Salary DESC;
```

Explanation: The query retrieves data sorted by salary in descending order, providing an overview of the highest earners in the organization for review.

EmpId	Name	Email	Gender	Salary	DOB	Did
2	Prakash	prakash@gmail.com	M	9000	2000-03-24	102
4	Leena	leena@gmail.com	F	6200	1994-09-01	103
5	Mohan	mohan@gmail.com	M	5500	1997-12-12	102
3	Anita	anita@gmail.com	F	4500	1998-06-15	101
1	Raju	raju@gmail.com	M	4000	2000-03-24	101

18. Sorting with Multiple Columns

```
SELECT Did, Gender, EmpId, Name FROM Employee ORDER
 BY Did ASC;
```

Explanation: This query retrieves Did and Gender, sorted by Did to group employees by department, facilitating easy departmental comparisons in the analysis.

Did	Gender	EmpId	Name
101	M	1	Raju
101	F	3	Anita
102	M	2	Prakash
102	M	5	Mohan
103	F	4	Leena

19. Top Limit of Records

```
SELECT TOP (5) * FROM Employee;
```

Explanation: This query fetches the first 5 records from the Employee table, useful for displaying limited employee information without overwhelming viewers.

EmpId	Name	Email	Gender	Salary	DOB	Did
1	Raju	raju@gmail.com	M	9000	2000-03-24	101
2	Prakash	prakash@gmail.com	M	8000	2000-03-24	102
3	Anita	anita@gmail.com	F	7500	1998-06-15	101
4	Leena	leena@gmail.com	F	6200	1994-09-01	103
5	Mohan	mohan@gmail.com	M	5500	1997-12-12	102

20. Generating a New Unique Identifier

```
SELECT NEWID();
```

Explanation: This function generates a unique identifier, useful for ensuring unique records across mobile applications, user sessions, or transactional databases.

NewId
6C8AA129-9F05-4B72-B4C7-8A67B0514FAF
A08A2E3F-B961-4B41-820F-7E6C6F734A5E

21. Selecting Top Records

```
SELECT TOP (3) EmpId, Name, Gender, Salary, DOB
FROM Employee;
```

Explanation: This query retrieves the top 3 records, providing a quick glance at employee data without overwhelming detail, useful for sampling datasets.

EmpId	Name	Gender	Salary	DOB
1	Raju	M	99000	2000-03-24
2	Prakash	M	90000	2000-03-24
3	Anita	F	50000	1998-06-15

22. Top Records Sorted by EmpId

```
SELECT TOP (3) EmpId, Name, Gender, Salary, DOB

FROM Employee ORDER BY EmpId DESC;
```

Explanation: This query retrieves the 3 most recent employee records based on EmpId, useful for understanding the latest hires in the organization.

EmpId	Name	Gender	Salary	DOB
5	Mohan	M	55000	1997-12-12
4	Leena	F	62000	1994-09-01
3	Anita	F	50000	1998-06-15

23. Top Record with Ties

```
SELECT TOP (1) WITH TIES * FROM Employee ORDER BY Salary DESC;
```

Explanation: This query retrieves the top record(s) while accounting for ties based on Salary, providing insights into the highest earning employees.

EmpId	Name	Email	Gender	Salary	DOB	Did
2	Prakash	prakash@gmail.com	M	9000	2000-03-24	102
4	Leena	leena@gmail.com	F	9000	1994-09-01	103

24. Selecting with Unique Identifier

```
SELECT *, NEWID() AS Uid FROM Employee;
```

Explanation: This query adds a new unique identifier to each employee record, enabling traceability and management of individual records within applications.

Emp Id	Name	Email	Gender	Uid
1	Raju	raju@gmail.com	M	3B3C3FF8-FD56-43DA-B271-365AFA196D17
2	Prakash	prakash@gmail.com	M	48825B1F-6A93-4988-BA77-E5BE4F5D2BC4
3	Anita	anita@gmail.com	F	E8EAD8F7-DF29-4496-8A9E-BA521F4C74D6
4	Leena	leena@gmail.com	F	20EC664C-FD45-41BC-9CFC-60B7BF20298D
5	Mohan	mohan@gmail.com	M	6D9A420A-045A-475E-B3DA-993FA47C5FFC

25. Sorting with a New Unique Identifier

```
SELECT *, NEWID() AS Uid FROM Employee ORDER BY Uid;
```

Explanation: This query retrieves all employee information while generating a unique identifier for each employee, sorted by this unique identifier for best practices in database handling and processing.

Emp Id	Name	Email	Gender	Uid
1	Raju	raju@gmail.com	M	3B3C3FF8-FD56-43DA-B271-365AFA196D17
2	Prakash	prakash@gmail.com	M	48825B1F-6A93-4988-BA77-E5BE4F5D2BC4
3	Anita	anita@gmail.com	F	E8EAD8F7-DF29-4496-8A9E-BA521F4C74D6
4	Leena	leena@gmail.com	F	20EC664C-FD45-41BC-9CFC-60B7BF20298D
5	Mohan	mohan@gmail.com	M	6D9A420A-045A-475E-B3DA-993FA47C5FFC

26. Top Records with Unique Identifier

```
SELECT TOP (5) *, NEWID() AS Uid FROM Employee
ORDER BY Uid;
```

Explanation: Retrieves the top 5 employee records alongside generating and sorting by a new unique identifier, enhancing tracking and data integrity.

Emp Id	Name	Email	Gender	Uid
1	Raju	raju@gmail.com	M	3B3C3FF8-FD56-43DA-B271-365AFA196D17
2	Prakash	prakash@gmail.com	M	48825B1F-6A93-4988-BA77-E5BE4F5D2BC4
3	Anita	anita@gmail.com	F	E8EAD8F7-DF29-4496-8A9E-BA521F4C74D6
4	Leena	leena@gmail.com	F	20EC664C-FD45-41BC-9CFC-60B7BF20298D
5	Mohan	mohan@gmail.com	M	6D9A420A-045A-475E-B3DA-993FA47C5FFC

Conclusion

This chapter provided an in-depth look at SQL operations for manipulating database entries, outlining the differences between DELETE and TRUNCATE, and offering insights into concurrency control mechanisms. Understanding these concepts is fundamental for efficient database management and handling multi-user environments. By practicing with these operations and exploring scenarios like concurrency handling, database administrators can maintain data integrity and performance.

Chapter 7: SQL Functions and Data Processing

Introduction

Structured Query Language (SQL) is a powerful language used for managing and manipulating relational databases. This chapter focuses on essential SQL functions that facilitate data processing, retrieval, and computation. Understanding SQL functions is critical for database management and provides the ability to interact with data effectively. This chapter will cover various SQL queries to showcase how to filter, manipulate, and aggregate data using SQL, along with explanations and sample records to demonstrate each query's output.

SQL Queries and Explanations

1. Selecting All Columns from Employee Table

```
SELECT * FROM Employee;
```

This command retrieves all columns for every employee in the Employee table.

EmpId	Name	Gender	Salary	Did
1	John Smith	M	60000	1001
2	Jane Doe	F	45000	1002
3	Alan Brown	M	52000	1003
4	Lisa White	F	48000	1002
5	Mark Lee	M	72000	1001

2. Selecting All Columns Where EmpId Equals 5

```
SELECT * FROM Employee WHERE EmpId = 5;
```

This query retrieves all columns for the employee with an EmpId of 5, isolating specific data.

EmpId	Name	Gender	Salary	Did
5	Mark Lee	M	72000	1001

3. Selecting All Columns Where Gender is 'M'

```
SELECT * FROM Employee WHERE Gender = 'M';
```

This statement filters the results to include only male employees from the Employee table.

EmpId	Name	Gender	Salary	Did
1	John Smith	M	60000	1001
3	Alan Brown	M	52000	1003
5	Mark Lee	M	72000	1001

4. Selecting Female Employees with Salary Less Than 50000

```
SELECT * FROM Employee WHERE Gender = 'F' AND Salary< 50000
```

This query targets female employees earning less than 50,000, providing insights into salary distributions.

EmpId	Name	Gender	Salary	Did
4	Lisa White	F	48000	1002

5. Selecting Employees Where Did Equals 1002

```
SELECT * FROM Employee WHERE Did = 1002;
```

This command retrieves all employee records assigned to department ID 1002.

EmpId	Name	Gender	Salary	Did
2	Jane Doe	F	45000	1002
4	Lisa White	F	48000	1002

6. Selecting Employees Where EmpId is Either 7 or 10

```
SELECT * FROM Employee WHERE EmpId=7 OR EmpId=10;
```

This query fetches employees whose EmpId is either 7 or 10.

EmpId	Name	Gender	Salary	Did
7	Sarah Black	F	55000	1003
10	Tom Green	M	69000	1001

7. Selecting Employees Where EmpId is 7, 10, or 24

```
SELECT * FROM Employee WHERE EmpId=7 OR EmpId=10
OR EmpId=24;
```

This query allows for expanded results based on multiple specified EmpIds.

EmpId	Name	Gender	Salary	Did
7	Sarah Black	F	55000	1003
10	Tom Green	M	69000	1001
24	Angela White	F	60000	1002

8. Selecting Employees Where EmpId is in the List (7, 10, 24, 56)

```
SELECT * FROM Employee WHERE EmpId IN (7, 10, 24, 56);
```

This statement retrieves records for employees with EmpId matching those in a specific list.

EmpId	Name	Gender	Salary	Did
7	Sarah Black	F	55000	1003
10	Tom Green	M	69000	1001
24	Angela White	F	60000	1002

9. Selecting Employees Where EmpId Equals 7

```
SELECT * FROM Employee WHERE EmpId=7;
```

This command fetches the specific employee record for EmpId 7.

EmpId	Name	Gender	Salary	Did
7	Sarah Black	F	55000	1003

10. Selecting Employees Where EmpId is Not Equals 7

```
SELECT * FROM Employee WHERE EmpId<>7;
```

This query retrieves records for employees that do not have an EmpId of 7.

EmpId	Name	Gender	Salary	Did
1	John Smith	M	60000	1001
2	Jane Doe	F	45000	1002
3	Alan Brown	M	52000	1003
4	Lisa White	F	48000	1002
5	Mark Lee	M	72000	1001

11. Selecting Employees Where EmpId is Not Equals 7

```
SELECT * FROM Employee WHERE EmpId != 7;
```

This query retrieves records for employees that do not have an EmpId of 7.

EmpId	Name	Gender	Salary	Did
1	John Smith	M	60000	1001
2	Jane Doe	F	45000	1002
3	Alan Brown	M	52000	1003
4	Lisa White	F	48000	1002

12. Selecting Employees Where EmpId is Not in the List (7, 10, 24, 56)

```
SELECT * FROM Employee WHERE EmpId NOT IN (7,10,24,56);
```

Filtering out employees based on specified IDs provides insights into those who are not part of specific groups.

EmpId	Name	Gender	Salary	Did
1	John Smith	M	60000	1001
2	Jane Doe	F	45000	1002
3	Alan Brown	M	52000	1003
4	Lisa White	F	48000	1002
5	Mark Lee	M	72000	1001

13. Selecting Employees Where EmpId is Between 10 and 20

```
SELECT * FROM Employee WHERE EmpId>=10 AND EmpId<=20;
```

This command targets employees within a specific range of EmpIds, aiding in demographic assessments.

EmpId	Name	Gender	Salary	Did
10	Tom Green	M	69000	1001
11	Anne Blue	F	53000	1003
12	Brian Red	M	55000	1001

14. Selecting Employees Where EmpId is Less than 10 or Greater than 20

```
SELECT * FROM Employee WHERE EmpId<10 OR EmpId>20;
```

This query identifies employees outside the range of 10 to 20, useful for isolating data points.

EmpId	Name	Gender	Salary	Did
1	John Smith	M	60000	1001
2	Jane Doe	F	45000	1002
3	Alan Brown	M	52000	1003
4	Lisa White	F	48000	1002
5	Mark Lee	M	72000	1001

15. Employees Where Did Equals 1002 (again)

```
SELECT * FROM Employee WHERE Did=1002;
```

Repetitive queries allow for confirmatory analysis of employees in a specific department.

EmpId	Name	Gender	Salary	Did
2	Jane Doe	F	45000	1002
4	Lisa White	F	48000	1002

16. Employees Where Did is NULL

```
SELECT * FROM Employee WHERE Did IS NULL;
```

This query reveals employees who are not assigned to any department, useful for resource assessments.

EmpId	Name	Gender	Salary	Did
6	Chris Grey	M	33000	NULL
8	Jennifer Hill	F	47000	NULL

17. Employees Where Did is NOT NULL

```
SELECT * FROM Employee WHERE Did IS NOT NULL;
```

This command retrieves all employees assigned to departments, ensuring focus on eligible resources.

EmpId	Name	Gender	Salary	Did
1	John Smith	M	60000	1001
2	Jane Doe	F	45000	1002
3	Alan Brown	M	52000	1003
4	Lisa White	F	48000	1002
5	Mark Lee	M	72000	1001

18. Youngest Female Employee of Department 1003

```
SELECT * FROM Employee WHERE Gender='F' AND Did=1003;
```

This query aims to find the youngest female employee in department 1003.

EmpId	Name	Gender	Salary	Did
6	Nancy Green	F	57000	1003

19. Top Youngest Female Employee of Department 1003, Ordered by Date of Birth

```
SELECT TOP (1) WITH TIES * FROM Employee WHERE
   Gender='F' AND Did=1003 ORDER BY DOB DESC;
```

This command selects the youngest female employee specifically in department 1003.

EmpId	Name	Gender	Salary	Did
6	Nancy Green	F	57000	1003

20. Employees Where Name Starts with 'A'

```
SELECT * FROM Employee WHERE Name LIKE 'A%';
```

The usage of the LIKE operator fetches names starting with 'A', allowing for specific categorization.

EmpId	Name	Gender	Salary	Did
3	Alan Brown	M	52000	1003
11	Anne Blue	F	53000	1003

21. Employees Where Name Ends with 'o'

```
SELECT * FROM Employee WHERE Name LIKE '%o';
```

This query filters employees with names ending in 'o,' helping to develop targeted searches.

EmpId	Name	Gender	Salary	Did
5	Marco Polo	M	64000	1002
12	Alfredo Pinto	M	70000	1001

22. Employees Where Name Contains 'an'

```
SELECT * FROM Employee WHERE Name LIKE '%an%';
```

Utilizing a substring filter, this command identifies employees with 'an' in their names.

EmpId	Name	Gender	Salary	Did
3	Alan Brown	M	52000	1003
11	Anne Blue	F	53000	1003

23. Selecting Employees Where Name is 'Michael'

```
SELECT * FROM Employee WHERE Name='Michael';
```

This query retrieves specific records for employees named Michael, focusing on individual identification.

EmpId	Name	Gender	Salary	Did
9	Michael Scott	M	55000	1001

24. Employees Where Soundex of Name Matches 'Michal'

```
SELECT * FROM Employee WHERE Soundex(Name)=Soundex ('Michal');
```

This command matches names producing the same Soundex code as 'Michal,' enhancing phonetic searches.

EmpId	Name	Gender	Salary	Did
9	Michael Scott	M	55000	1001

25. Employees Where Soundex of Name Matches 'Mychal'

```
SELECT * FROM Employee WHERE Soundex(Name)=Soundex ('Mychal');
```

Similar to previous examples, it filters names matching the Soundex code for Mychal.

EmpId	Name	Gender	Salary	Did
9	Michael Scott	M	55000	1001

26. Employees Where Soundex of Name Matches 'ryan'

```
SELECT * FROM Employee WHERE Soundex(Name)=Soundex('ryan');
```

This query extracts records for names that phonologically resemble 'Ryan' according to Soundex matching.

EmpId	Name	Gender	Salary	Did
10	Ryan Jackson	M	60000	1002

27. Employees Where Name Starts with a letter in Range 'e' to 'i'

```
SELECT * FROM Employee WHERE Name LIKE '[e-i]%' ORDER BY Name;
```

This statement returns employees whose names begin with the letter's 'E' through 'I,' ordered by name.

EmpId	Name	Gender	Salary	Did
5	Erin Kelly	F	57000	1002
6	Isaac Clarke	M	58000	1003

28. Using String Functions: Len(), Lower(), Ltrim(), Substring()

```
SELECT LEN('ASP.Net');
```

This function calculates the length of the string "ASP.Net".

Length
7

29. Select Employees with Character Length of Their Names

```
SELECT EmpId, Name, LEN(Name) AS 'NoOfCharacters' FRO
M Employee;
```

This query retrieves each employee's ID and name along with the length of their names.

EmpId	Name	NoOfCharacters
1	John Smith	10
2	Jane Doe	8
3	Alan Brown	10
4	Lisa White	10
5	Mark Lee	8

30. Retrieve Employees Ordered by Name Length

```
SELECT EmpId, Name, LEN(Name) AS NoOfCharacters FRO
M Employee ORDER BY NoOfCharacters DESC;
```

This query orders employees by the length of their names, longest to shortest.

EmpId	Name	NoOfCharacters
1	John Smith	10
3	Alan Brown	10
4	Lisa White	10
2	Jane Doe	8
5	Mark Lee	8

31. Convert Names to Lowercase

```
SELECT EmpId, LOWER(Name) FROM Employee;
```

This command queries employees and formats their names in lowercase.

EmpId	Name
1	john smith
2	jane doe
3	alan brown
4	lisa white
5	mark lee

32. Convert Names to Uppercase

```
SELECT EmpId, UPPER(Name) FROM Employee;
```

This filters the employee names, converting them to uppercase for standardization.

EmpId	Name
1	JOHN SMITH
2	JANE DOE
3	ALAN BROWN
4	LISA WHITE

33. Trimming Strings

```
SELECT LTRIM('        Manzoor');
```

This function removes leading spaces from the string.

Trimmed Name
Manzoor

34. Employees Where Name Equals Trimmed Value

```
SELECT * FROM Employee WHERE Name=TRIM(' Isaac ');
```

This query retrieves records, ensuring that leading and trailing spaces don't affect string matching.

EmpId	Name	Gender	Salary	Did
7	Isaac R.	M	47000	1003

35. Employees Where Name is LTrimmed

```
SELECT * FROM Employee WHERE Name=LTRIM(' Isaac');
```

Here, employees matched have names that have been left-trimmed, ensuring cleaned inputs.

EmpId	Name	Gender	Salary	Did
7	Isaac R.	M	47000	1003

36. Substring Function Example

```
SELECT SUBSTRING ('Manzoor', 1, 3);
```

This function extracts the first three characters from the string 'Manzoor'.

Substring
Man

37. Reverse and Substring Manipulation

```
SELECT REVERSE(SUBSTRING(REVERSE('Manzoor'), 1, 4));
```

This command retrieves a substring and reverses the selected string.

Result
Ooz

38. Employee Password Generation

```
SELECT EmpId, Name, Contact, SUBSTRING(Name, 1, 4) + R
EVERSE(SUBSTRING(REVERSE(Contact), 1, 4)) AS Password
FROM Employee;
```

This query creatively generates employee passwords based on their name and contact details.

EmpId	Name	Password
1	John Smith	Johnth
2	Jane Doe	Jane64
3	Alan Brown	Alan02

39. String Split Function Usage

```
SELECT * FROM STRING_SPLIT ('C,C++,C#', ',');
```

This command separates the string by commas, returning distinct language names.

Value
C
C++
C#

40. String Split with Dynamic Contact

```
DECLARE @contact AS VARCHAR (50);
SELECT @contact=Contact FROM Employee WHERE EmpId=56;
SELECT * FROM STRING_SPLIT (@contact, '-');
```

This query dynamically splits a contact string for an employee, which is useful for parsing complex data.

Value
1234567890
john.doe@

41. Basic Arithmetic Example

```
SELECT 1+1;
```

This command performs a basic mathematical operation.

Result
2

42. String and Integer Addition

```
SELECT 1+'5';
```

This example highlights how SQL treats string and numeric types when performing operations.

Result
6

43. Concatenation of Strings

```
SELECT '1'+'5';
```

This command concatenates two string values together.

Result
15

44. Concatenation of Characters

```
SELECT 'A'+'B';
```

Similar to the previous example, this concatenates two character strings.

Result
AB

45. Casting Data Types

```
SELECT CAST(124 AS VARCHAR(10))+'678i';
```

This statement demonstrates converting an integer into a string before concatenating.

Result
124678i

46. Generating Employee Codes

```
SELECT EmpId, Name, Salary, (CAST(EmpId AS VARCHAR(10))
+'-'+SUBSTRING(Name,1,3)) AS EmpCode FROM Employee;
```

This query generates a unique employee code using an EmpId and the first three characters of their name.

EmpId	Name	Salary	EmpCode
1	John Smith	60000	1-Joh
2	Jane Doe	45000	2-Jan
3	Alan Brown	52000	3-Ala
4	Lisa White	48000	4-Lis
5	Mark Lee	72000	5-Mar

47. Arithmetic with Floats

```
SELECT 5.0/2;
```

This command illustrates division resulting in a float output.

Result
2.5

48. Cast Integer to Float Division

```
SELECT CAST (5 AS FLOAT)/2;
```

This example shows how casting before an operation influences the output of a float type.

Result
2.5

49. Employees Where Did is NULL

```
SELECT * FROM Employee WHERE Did IS NULL;
```

Querying to retrieve all records where department ID is null.

EmpId	Name	Gender	Salary	Did
6	Chris Grey	M	33000	NULL
8	Jennifer Hill	F	47000	NULL

50. Specific EMPID Retrieval and NULL Checks

```
SELECT EmpId, Name, Salary, Did FROM Employee WHERE Did IS NULL;
```

This command provides a focused view on ID, name, and salary of employees with null Did values.

EmpId	Name	Salary	Did
6	Chris Grey	33000	NULL
8	Jennifer Hill	47000	NULL

51. Using ISNULL Function

```
SELECT EmpId, Name, Salary, ISNULL(Did, 0) AS Did FROM
 Employee WHERE Did IS NULL;
```

This command shows how to replace null values in the Did column with 0.

EmpId	Name	Salary	Did
6	Chris Grey	33000	0
8	Jennifer Hill	47000	0

52. Advanced NULL Handling

```
SELECT EmpId, Name, Salary, ISNULL(CAST(Did AS VARC
HAR(20)), 'No Department') AS Did FROM Employee;
```

This query provides a clearer picture of employees without an assigned department.

EmpId	Name	Salary	Did
6	Chris Grey	33000	No Department
8	Jennifer Hill	47000	No Department

53. Using CEILING() Function

```
SELECT CEILING(122.1);
```

This returns the smallest integer greater than or equal to 122.1.

Result
123

54. CEILING() on Negative Numbers

```
SELECT CEILING(-3.2);
```

Returning the ceiling of a negative floating-point value.

Result
-3

55. FLOOR() Function Example

```
SELECT FLOOR(122.1);
```

This retrieves the largest integer less than or equal to 122.1.

Result
122

56. FLOOR() on Negative Numbers

```
SELECT FLOOR(-3.2);
```

Applying FLOOR to a negative value demonstrates smaller bounds.

Result
-4

57. ROUND() Function Example

```
SELECT ROUND(32.4464276, 4);
```

This command rounds a value to the specified number of decimal places.

Result
32.4464

58. Rounding Division Example

```
SELECT ROUND(10.0/3, 3);
```

This command shows how to round the output of a division operation.

Result
3.333

59. Generating Random Numbers

```
SELECT FLOOR(RAND()*1000);
```

This generates a random number between 0 and 999, demonstrating randomization.

Result
563

60. Counting Employees

```
SELECT COUNT(*) AS NOE FROM Employee;
```

This function counts the total number of employees present in the dataset.

NOE
10

61. Count Employees with Did Values

```
SELECT COUNT(Did) AS NOE FROM Employee;
```

Counting employees assigned to a department or having a Did value.

NOE
8

62. Counting Employees with Salary Greater than 50000

```
SELECT COUNT(*) AS NOE FROM Employee WHERE Salary >50000;
```

This command assesses the number of employees with salaries over the specified threshold.

NOE
5

63. Summing Salaries

```
SELECT SUM(Salary) AS SOS FROM Employee;
```

This function calculates the total salary expenses of all employees.

SOS
542000

64. Average Salary Calculation

```
SELECT AVG(Salary) AS Avg FROM Employee;
```

The average salary of employees is useful for salary benchmarking.

Avg
54200

65. Maximum Salary Determination

```
SELECT MAX(Salary) FROM Employee;
```

This function finds the highest salary among employees.

Result
72000

66. Minimum Salary Determination

```
SELECT MIN(Salary) FROM Employee;
```

This command reveals the lowest salary recorded in the Employee database.

Result
33000

67. Current Date and Time Retrieval

```
SELECT GETDATE();
```

This command fetches the current date and time, beneficial for logging operations.

Current Date and Time
09/06/2024 10:45:30 PM

68. Extracting Year from Current Date

```
SELECT YEAR(GETDATE());
```

This function retrieves the current year from the date.

Result
2024

69. Static Year Extraction

```
SELECT YEAR('05/23/2002');
```

Retrieving the year from a specific date string.

Result
2002

70. Extracting Month from Date

```
SELECT MONTH('05/23/2002');
```

This retrieves the month number from a specified date.

Result
5

71. Extracting Day from Date

```
SELECT DAY('05/23/2002');
```

This command extracts the day of the month from a specified date.

Result
23

72. Employees Born in the Current Month

```
SELECT *, MONTH(DOB) FROM Employee WHERE MONTH
(DOB)=MONTH(GETDATE());
```

Finding employees whose birthdays occur in the current month.

EmpId	Name	DOB
2	Jane Doe	09/10/1990
5	Mark Lee	09/14/1985

73. Counting Employees Born This Month

```
SELECT COUNT(*) FROM Employee WHERE MONTH(DOB)
=MONTH(GETDATE());
```

The number of employees celebrating birthdays in the current month.

Result
2

74. Salary Calculation Based on Birth Month

```
SELECT COUNT(*)*1250 FROM Employee WHERE MONTH(
DOB)=MONTH(GETDATE());
```

Calculating bonuses for employees whose birthdays fall within this month.

Result
2500

75. Count for Specific Month Birthdays

```
SELECT COUNT(), COUNT()*1250 FROM Employee WHERE MONTH(DOB)=3;
```

Counting birthday occurrences in March, along with a projected cost.

Count	Total Cost
3	3750

76. Employees Born in 2000

```
SELECT * FROM Employee WHERE YEAR(DOB)=2000;
```

Querying employees based on their year of birth.

EmpId	Name	DOB
6	Chris Grey	01/05/2000
7	Sarah Black	08/19/2000

77. Employees Ordered by Year of Birth

```
SELECT *, YEAR(DOB) AS YOB FROM Employee ORDER BY YOB;
```

Retrieving employee records sorted by year of birth.

EmpId	Name	DOB	YOB
6	Chris Grey	01/05/2000	2000
5	Mark Lee	09/14/1985	1985
3	Alan Brown	05/20/1978	1978

78. Years Difference Calculation

```
SELECT DATEDIFF(YY,'11/06/1984',GETDATE());
```

Calculating differences in years from a static date to the current date.

Result
39

79. Month Difference Calculation

```
SELECT DATEDIFF(MM,'11/06/1984',GETDATE());
```

Calculating the total months between two dates.

Result
474

80. Days Difference Calculation

```
SELECT DATEDIFF(DD,'11/06/1984',GETDATE());
```

This shows the total day count between two specified dates.

Result
14310

81. Hour Difference Calculation

```
SELECT DATEDIFF(HOUR,'11/06/1984',GETDATE());
```

This calculates the number of hours between two timestamp values.

Result
858600

82. Seconds Difference Calculation

```
SELECT DATEDIFF(SECOND,'11/06/1984',GETDATE());
```

This extracts the difference in total seconds from a fixed point in time.

Result
51516012

83. Age Calculation of Employees

```
SELECT *, DATEDIFF(YY, DOB, GETDATE()) AS Age FROM Employee;
```

Calculating the age of employees based on their date of birth.

EmpId	Name	DOB	Age
1	John Smith	06/12/1990	34
2	Jane Doe	08/15/1992	32

84. Adding Days to Current Date

```
SELECT DATEADD(DD, 20, GETDATE());
```

This command adds a specified number of days to the current date.

Result
09/26/2024

85. Adding Days to a Given Date

```
SELECT DATEADD(DD, 90, '07/10/2024');
```

Calculating a future date based on a 90-day addition.

Result
10/08/2024

86. Date of Retirement Calculation

```
SELECT *, DATEADD (YY, 50, DOB) AS DOR FROM Employee;
```

Calculating retirement dates based on employee DOB.

EmpId	Name	DOB	DOR
1	John Smith	06/12/1990	06/12/2040
2	Jane Doe	08/15/1992	08/15/2042

87. Date Conversion to a Specific Format

```
SELECT CONVERT (VARCHAR (50), GETDATE(), 107);
```

Changing the date format provides better readability.

Result
Sep 6, 2024

Conclusion

In this chapter, we explored a wide range of SQL functions and queries that are essential for effective data processing in relational databases. Each SQL query demonstrated how to retrieve, manipulate, and analyze data across various dimensions such as employee demographics, salary distributions, and department assignments. Understanding these functions will allow you to efficiently interact with databases and draw meaningful insights from your data. As you've seen, SQL is not just a language for querying databases, but a powerful tool for data analysis and decision-making. With practice and exploration, You will gain proficiency in harnessing SQL's capabilities to meet your data processing needs.

Chapter 8: Advanced Querying Techniques

Introduction

Take your querying skills to the next level. Master different types of JOINs to combine data from multiple tables, write complex multi-table queries and implement subqueries and Common Table Expressions. These advanced techniques will allow you to handle complex data retrieval scenarios. Learn the art and science of database design. You will understand key design principles, analyze and implement complex database structures, and design efficient schemas for real-world scenarios. This module will help you create databases that are performant, scalable, and maintainable.

SQL Queries and Explanations

1. Count the Number of Male Employees

```
SELECT COUNT(*) FROM Employee WHERE Gender='M';
```

Explanation: This query tallies the total number of employees designated as male. It provides a clear insight into the gender distribution in the workforce.

Count
50

2. Count the Number of Female Employees

```
SELECT COUNT(*) FROM Employee WHERE Gender='F';
```

Explanation: This query counts the total number of female employees, contributing to understanding gender representation within the company.

Count
45

3. Count of Employees by Gender (Error Example)

```
SELECT Gender, COUNT(*) AS NOE FROM Employee;
```

Explanation: This query aims to count employees by gender but fails due to the missing GROUP BY clause, violating SQL's rules for aggregate functions.

Error Msg
Msg 8120: Column 'Employee.Gender' is invalid...

4. Corrected Count of Employees by Gender

```
SELECT Gender,COUNT(*)AS NOE FROM Employee GROUP
 BY Gender;
```

Explanation: This revised query successfully counts employees classified by gender, enabling a view of workforce gender dynamics.

Gender	NOE
M	50
F	45

5. Count of Employees by Department

```
SELECT Did,COUNT(*)AS NOE FROM Employee GROUP BY Did;
```

Explanation: This query counts employees per department, helping evaluate departmental workforce size and resource allocation.

Did	NOE
1	25
2	20
3	10

6. Count by Department and Gender

```
SELECT Did, Gender, COUNT(*) AS NOE FROM Employee
GROUP BY Did, Gender ORDER BY Did;
```

Explanation: This query gives a breakdown of employee counts by both department and gender, offering insights into departmental gender balance.

Did	Gender	NOE
1	M	15
1	F	10
2	M	20
2	F	10

7. Count and Sum Salaries by Department and Gender

```
SELECT Did, Gender, COUNT(*) AS NOE, SUM (Salary) AS SOS
FROM Employee GROUP BY Did, Gender ORDER BY Did;
```

Explanation: This query not only counts employees by department and gender but also sums their salaries, revealing financial insights related to gender distribution.

Did	Gender	NOE	SOS
1	M	15	450000
1	F	10	300000
2	M	20	600000
2	F	10	250000

8. Count Female Employees and Sum Salaries

```
SELECT Did, Gender, COUNT(*) AS NOE, SUM(Salary) AS S
OS FROM Employee WHERE Gender='F' GROUP BY Did, Ge
nder ORDER BY Did;
```

Explanation: This query focuses on female employees, counting them and summing their salaries across departments, which is valuable for gender pay analysis.

Did	Gender	NOE	SOS
1	F	10	300000
2	F	10	250000

9. Filtering with HAVING Clause

```
SELECT Did, Gender, COUNT(*) AS NOE, SUM(Salary) AS
SOS FROM Employee WHERE Gender='F' GROUP BY Did,
Gender HAVING COUNT(*) < 10 ORDER BY Did;
```

Explanation: This query employs the HAVING clause to filter departments with less than 10 female employees, providing a focused look at underrepresented groups.

Did	Gender	NOE	SOS
-	-	-	-

10. Average Salary of Male Employees by Department

```
SELECT Did, Gender, AVG(Salary) AS AOS FROM Employee
WHERE Gender='M' AND Did IS NOT NULL GROUP BY Did
, Gender ORDER BY AOS;
```

Explanation: This query calculates the average salary of male employees across departments, highlighting salary disparities.

Did	Gender	AOS
1	M	30000
2	M	35000

11. Inner Join to Display Employee and Department Information

```
SELECT E.EmpId, E.Name, E.Did, D.DName FROM Employee
 E INNER JOIN Department D ON E.Did=D.Did;
```

Explanation: This inner join connects employee and department data, displaying relevant employee details along with their associated department names.

EmpId	Name	Did	DName
1	Alice	1	HR
2	Bob	2	IT

12. Left Outer Join

```
SELECT E.EmpId, E.Name, E.Did, D.DName FROM Employee
 E LEFT OUTER JOIN Department D ON E.Did=D.Did ORDE
R BY Did;
```

Explanation: This left outer join retrieves all employees, including those without a department assignment, ensuring no employee data is lost.

EmpId	Name	Did	DName
1	Alice	1	HR
2	Bob	2	IT
3	Charlie	NULL	NULL

13. Right Outer Join

```
SELECT E.EmpId, E.Name,D.Did, D.DName FROM Employee
 E RIGHT OUTER JOIN Department D ON E.Did=D.Did ORD
ER BY E.Did;
```

Explanation: This right outer join displays all departments, including those without employees, illustrating how organizational structure might affect staffing.

EmpId	Name	Did	DName
NULL	NULL	3	Sales

14. Full Outer Join

```
SELECT E.EmpId, E.Name,D.Did, D.DName FROM Employee
 E FULL OUTER JOIN Department D ON E.Did=D.Did ORDE
R BY E.Did;
```

Explanation: This full outer join fetches all employees and all departments, providing a complete view of the organizational framework and staffing.

EmpId	Name	Did	DName
1	Alice	1	HR
2	Bob	2	IT
NULL	NULL	3	Sales

15. Count of Employees by Department Name Using Full Outer Join

```
SELECT D.DName, COUNT(E.EmpId) AS NOE FROM Emplo
yee E FULL OUTER JOIN Department D ON E.Did=D.Did GR
OUP BY D.DName ORDER BY D.DName;
```

Explanation: This query aggregates the number of employees per department, even accounting for departments lacking employees, hence allowing for overall analysis of staffing.

DName	NOE
HR	25
IT	20
Sales	0

16. Cartesian Product Due to Missing Join Condition

```
SELECT * FROM Employee, Department ORDER BY EmpId;
```

Explanation: Without a specified join condition, this query creates a Cartesian product, a combination of every employee with every department, leading to overly large datasets.

EmpId	Name	Did	...	DName
1	Alice	1	...	HR
2	Bob	2	...	IT
3	Charlie	NULL	...	NULL

17. Count Employees by Department Name with Inner Join

```
SELECT D.DName, COUNT(E.EmpId) AS NOE FROM
Employee E JOIN Department D ON E.Did=D.Did GROUP BY
 D.DName ORDER BY D.DName;
```

Explanation: This inner join query counts employees based on their respective department names, allowing for clear insights into departmental staffing levels.

DName	NOE
HR	25
IT	20

Conclusion

In this chapter, we delved into the various SQL functions and techniques essential for effective data processing and analysis in relational databases, particularly focusing on employee information management. We examined different aggregate functions, such as counting and summing, to derive meaningful insights about employee demographics and departmental structures.

Through practical examples, we highlighted the importance of accurate SQL syntax, including the use of GROUP BY, HAVING, and different types of joins (inner, left, right, full outer) to present comprehensive data analyses. The demonstrated queries illustrate how to retrieve essential insights, such as gender distribution, department sizes, and salary averages, all of which are crucial for informed decision-making in human resources and organizational management.

Moreover, our exploration of common pitfalls—such as the necessity of including appropriate grouping and avoiding Cartesian products—emphasizes the importance of understanding SQL's operational principles. Mastering these SQL queries equips professionals with the tools necessary for efficient data manipulation, enabling organizations to leverage their data fully for strategic planning and performance improvement. Overall, the ability to extract and analyze data with SQL is an invaluable skill in today's data-driven world.

Chapter 9. The 3 Key Rules of Database Design Techniques - Writing Join Queries on 3 or More Tables

Database design is a crucial aspect of developing a structured system for storing & retrieving data. Understanding how to establish relationships between different entities (or objects) is key. Here we explain the three rules for structuring relationships in a database, along with detailed examples for each.

1. One Table for Each Object

This rule emphasizes that each distinct entity should have its own table in the database. Each table will contain attributes relevant to that entity.

Example 1:

Customer Table:

- **Attributes:** CustomerID (Primary Key), Name, Email, Phone

Example 2:

Orders Table:

- **Attributes:** OrderID (Primary Key), CustomerID (Foreign Key), OrderDate, TotalAmount

2. 1:M Relationships

In a one-to-many (1:M) relationship, one record in the first table can relate to multiple records in the second table. The primary key of the "one" (master) table becomes a foreign key in the "many" (child) table.

Example 1:

Department Table:

- **Attributes:** DepartmentID (Primary Key), DepartmentName

Employees Table:

- **Attributes:** EmployeeID (Primary Key), EmployeeName, DepartmentID (Foreign Key links to DepartmentID)

Example 2:

Authors Table:

- **Attributes:** AuthorID (Primary Key), AuthorName

Books Table:

- **Attributes:** BookID (Primary Key), Title, AuthorID (Foreign Key links to AuthorID)

3. M:M Relationships

In a many-to-many (M:M) relationship, records in one table can relate to multiple records in another table and vice versa. To manage this, you create a junction table (or transaction table) that contains foreign keys referencing the primary keys from both master tables.

Example 1:

Students Table:

- **Attributes:** StudentID (Primary Key), StudentName

Subjects Table:

- **Attributes:** SubjectID (Primary Key), SubjectName

Enrollments Table (Junction Table):

- **Attributes:** EnrollmentID (Primary Key), StudentID (Foreign Key), SubjectID (Foreign Key)

Example 2:
Products Table:

- **Attributes:** ProductID (Primary Key), ProductName

Categories Table:

- **Attributes:** CategoryID (Primary Key), CategoryName

ProductCategories Table (Junction Table):

- **Attributes:** ProductCategoryID (Primary Key), ProductID (Foreign Key), CategoryID (Foreign Key)

Summary

In summary, database design follows structured rules to maintain integrity and relationships among entities. Each distinct entity is represented in its own table. In a 1:M relationship, the primary key of the master table is used as a foreign key in the child table. For M:M relationships, a junction table is created to contain foreign keys that link the two master tables. This approach is critical in ensuring data accuracy and ease of retrieval in a relational database.

Chapter 10: Introduction

In the realm of database design, understanding how to efficiently retrieve and manipulate data is paramount. This chapter focuses on join queries that combine data from multiple tables, showcasing their significance through practical examples. We will explore how to perform joins using the Student, Course, and StudentCourse tables, demonstrating effective techniques to capture insights from relational databases.

1. Retrieving All Records from the Student Table

```
Select * from Student
```

This query retrieves all records from the Student table, providing a complete overview of student information.

Sid	SName
1	John Doe
2	Jane Smith
3	Alice Lee
4	Bob Brown
5	Carol White

2. Retrieving All Records from the Course Table

```
Select * from Course
```

This query fetches every entry in the Course table, highlighting all available courses.

Cid	CName
101	Mathematics
102	Science
103	History
104	Literature
105	Computer Science

3. Retrieving All Records from the StudentCourse Table

```
Select * from StudentCourse
```

This query retrieves all data from the StudentCourse table, which holds links between students and courses.

Sid	Cid	Exam	Marks
1	101	Midterm	85
2	102	Final	90
3	103	Midterm	75
4	101	Final	NULL
5	104	Midterm	95

4. Retrieving Specific Details of Students

```
Select S.Sid, S.SName, SC.Exam, C.CName, SC.Marks
from StudentCourse SC
join Student S on S.Sid = SC.Sid
join Course C on C.Cid = SC.Cid
```

This query selects specific details like student IDs, names, exam types, course names, and marks through joining three tables.

Sid	SName	Exam	CName	Marks
1	John Doe	Midterm	Mathematics	85
2	Jane Smith	Final	Science	90
3	Alice Lee	Midterm	History	75
4	Bob Brown	Final	Mathematics	NULL
5	Carol White	Midterm	Literature	95

5. Retrieving Average Marks of Students per Exam

```
Select S.Sid, S.SName, SC.Exam, Avg(SC.Marks) as Avg
from StudentCourse SC
join Student S on S.Sid = SC.Sid
join Course C on C.Cid = SC.Cid
Group by S.Sid, S.SName, SC.Exam
```

This querjy calculates average marks for each student in their respective exams, showcasing performance trends.

Sid	SName	Exam	Avg
1	John Doe	Midterm	85
2	Jane Smith	Final	90
3	Alice Lee	Midterm	75
4	Bob Brown	Final	NULL
5	Carol White	Midterm	95

6. Retrieving the Student with the Highest Average Marks

```
Select Top(1) S.Sid, S.SName, SC.Exam,Avg(SC.Marks) as Avg
from StudentCourse SC
join Student S on S.Sid = SC.Sid
join Course C on C.Cid = SC.Cid
Group by S.Sid, S.SName, SC.Exam
order by Avg Desc
```

This query identifies the student who has achieved the highest average marks in any exam, showcasing academic excellence.

Sid	SName	Exam	Avg
5	Carol White	Midterm	95

7. Retrieving Student Details with Null Marks

```
Select S.Sid, S.SName, SC.Exam, C.CName, SC.Marks
from StudentCourse SC
join Student S on S.Sid = SC.Sid
join Course C on C.Cid = SC.Cid
Where SC.Marks is null
```

This query filters for students who have missing marks, helping to identify potential issues in tracking academic performance.

Sid	SName	Exam	CName	Marks
4	Bob Brown	Final	Mathematics	NULL

8. Retrieving Student Details with Marks as 'Absent'

```
Select S.Sid, S.SName, SC.Exam, C.CName,
isnull(cast(SC.Marks as varchar(50)), 'Absent') as Marks
from StudentCourse SC
join Student S on S.Sid = SC.Sid
join Course C on C.Cid = SC.Cid
```

This query replaces null marks with 'Absent', allowing for clearer communication of student status regarding exams.

Sid	SName	Exam	CName	Marks
1	John Doe	Midterm	Mathematics	85
2	Jane Smith	Final	Science	90
3	Alice Lee	Midterm	History	75
4	Bob Brown	Final	Mathematics	Absent
5	Carol White	Midterm	Literature	95

Conclusion

In this chapter, we've explored the essential techniques for writing join queries involving three or more tables. By understanding how to retrieve, manipulate, and analyze data through these techniques, database designers and users can gain deeper insights into their data relationships and effectively evaluate student performance. Mastery of such queries not only enhances database functionality but also supports informed decision-making based on accurate data representations.

Chapter 11 - Simple Sub Queries (Scalar, Multi-Valued) - Nested Sub Queries - Correlated Sub Queries - CTE

Introduction

In this chapter, we delve into the powerful world of SQL subqueries. Subqueries are incredibly useful for breaking complex problems into manageable parts. They allow you to perform multiple levels of filtering, aggregation, and data retrieval without excessive joins or nested queries. We will cover simple subqueries, including scalar and multi-valued subqueries, as well as nested and correlated subqueries, followed by an exploration of Common Table Expressions (CTEs). Each section will include query explanations and sample outputs, showcasing their practical applications.

Simple Sub Queries

1.Selecting all employees with the maximum salary

```
SELECT * FROM Employee
WHERE Salary = (SELECT Max(Salary) FROM Employee);
```

Explanation: This scalar subquery fetches all employee records whose salary matches the highest salary in the Employee table.

Eid	Name	Salary	Did
1	John	95000	1000
2	Alice	95000	1001
3	Maria	95000	1002

2.Selecting students with specific IDs (1 and 3)

```
SELECT * FROM Student WHERE Sid IN (1, 3)
```

Explanation: This query retrieves student records with the specified student IDs of 1 and 3 from the Student table.

Sid	SName	Age	Gender
1	Emma	20	F
3	Liam	21	M

3.Selecting distinct student IDs from StudentCourse where marks are NULL

```
SELECT DISTINCT Sid FROM StudentCourse WHERE Marks IS NULL
```

Explanation: This retrieves unique student IDs from the StudentCourse table where students did not receive any marks.

Sid
2
4
5

4.Students whose IDs are in the set of student IDs with NULL marks

```
SELECT * FROM Student WHERE Sid IN (SELECT DISTINCT Sid FROM StudentCourse WHERE Marks IS NULL)
```

Explanation: This multi-valued subquery selects student records whose IDs correspond to those with NULL marks in the StudentCourse table.

Sid	SName	Age	Gender
2	Sophia	22	F
4	Oliver	23	M
5	Ava	20	F

5.Distinct department IDs where salary equals the maximum salary

```
SELECT DISTINCT Did FROM Employee WHERE Salary =
(SELECT Max(Salary) FROM Employee)
```

Explanation: This subquery identifies unique department IDs for departments with at least one employee earning the highest salary.

Did
1000
1001
1002
1003

6.Departments with IDs in the set (1001, 1003)

```
SELECT * FROM Department WHERE Did IN (1001, 1003)
```

Explanation: This query fetches department records that match the specified department IDs, 1001 and 1003.

Did	DName
1001	Marketing
1003	Finance

Nesting Queries

1.Departments with IDs from the employees with maximum salary

```
SELECT * FROM Department WHERE Did IN (SELECT DISTI
NCT Did FROM Employee WHERE Salary = (SELECT Max(Sa
lary) FROM Employee))
```

Explanation: This nested subquery fetches departments that have employees earning the highest salaries.

Did	DName
1000	HR
1001	Marketing
1002	Engineering

Correlated Subqueries

1.Employee with the highest salary from department 1000

```
SELECT * FROM Employee WHERE Salary = (SELECT Max(Salary) FROM Employee WHERE Did=1000) AND Did=1000
```

Explanation: This correlated subquery selects the employee with the highest salary specifically in department 1000.

Eid	Name	Salary	Did
1	John	95000	1000

2.Employee with the highest salary from department 1001

```
SELECT * FROM Employee WHERE Salary = (SELECT Max(Salary) FROM Employee WHERE Did=1001) AND Did=1001
```

Explanation: Similarly, this fetches the highest-paid employee within department 1001.

Eid	Name	Salary	Did
2	Alice	90000	1001

3.Employee with the highest salary from department 1002

```
SELECT * FROM Employee WHERE Salary = (SELECT Max(Salary) FROM Employee WHERE Did=1002) AND Did=1002
```

Explanation: This query retrieves the highest salary earners from department 1002.

Eid	Name	Salary	Did
3	Maria	85000	1002

4.Employee with the highest salary from department 1003

```
SELECT * FROM Employee OE WHERE OE.Salary = (SELECT
Max (IE.Salary) FROM Employee IE WHERE IE.Did = 1003)
 AND OE.Did = 1003;
```

Explanation: This returns the highest-paid employee specifically from department 1003.

Eid	Name	Salary	Did
4	James	80000	1003

5.Employee with the highest salary in their respective department

```
SELECT * FROM Employee OE WHERE OE.Salary = (SELECT
Max(IE.Salary) FROM Employee IE WHERE IE.Did=OE.Did)
```

Explanation: Each department's highest salary employee is selected using this correlated subquery.

Eid	Name	Salary	Did
1	John	95000	1000
2	Alice	90000	1001
3	Maria	85000	1002
4	James	80000	1003

Exclusions and Ordering

1.Select the top 1 employee excluding the one with the highest salary

```
SELECT TOP (1) * FROM Employee WHERE Salary NOT IN (SEL
ECT Max (Salary) FROM Employee) ORDER BY Salary DESC
```

Explanation: This retrieves the highest salary employee but excludes the one earning the maximum salary.

Eid	Name	Salary	Did
2	Alice	90000	1001

2.Select the top 1 employee excluding the top 8 highest salaries

```
SELECT TOP (1) * FROM Employee WHERE Salary NOT IN
(SELECT DISTINCT TOP (8) WITH TIES Salary FROM
Employee ORDER BY Salary DESC) ORDER BY Salary DESC
```

Explanation: The query selects the next highest salary after excluding the top 8 highest salaries from any department.

Eid	Name	Salary	Did
3	Maria	85000	1002

3.nth Highest Salary

```
DECLARE @n AS INT

SET @n = 2

SELECT TOP (1) * FROM Employee WHERE Salary NOT IN
(SELECT DISTINCT TOP(@n-1) WITH TIES Salary FROM
Employee ORDER BY Salary DESC) ORDER BY Salary DESC
```

Explanation: The query retrieves the nth highest salary by excluding the upper n-1 salaries.

Eid	Name	Salary	Did
3	Maria	85000	1002

CTE (Common Table Expressions)

1.Inner join to select employee details along with their department names

```
SELECT Employee.Eid, Employee.Name, Employee.Gender,
Department.DName FROM Employee INNER JOIN
Department ON Employee.DId = Department.Did
```

Explanation: This query combines employee details with department names through an inner join, yielding cohesive results.

Eid	Name	Gender	DName
1	John	M	HR
2	Alice	F	Marketing
3	Maria	F	Engineering
4	James	M	Finance

2.CTE Definition for combining student information with their course and marks

```
WITH StudentsResult (Sid, SName, CName, Marks) AS (SELE
CT Student.Sid, Student.SName, Course.CName, StudentCour
se.Marks FROM Student INNER JOIN StudentCourse ON Stu
dent.Sid = StudentCourse.Sid INNER JOIN Course ON Studen
tCourse.Cid = Course.Cid)
```

Explanation: This CTE merges student data with corresponding course and marks information for easier access and analysis.

Selecting student IDs, names, and their average marks

```
SELECT Sid, SName, AVG (Marks) AS Avg FROM StudentsRe
sult GROUP BY Sid, SName
```

Explanation: This final query calculates the average marks for each student, showcasing their overall performance with aggregate functions in the CTE context.

Sid	SName	Avg
1	Emma	85.0
2	Sophia	75.0
3	Liam	90.0
4	Oliver	70.0
5	Ava	92.0

Conclusion

In this chapter, we explored various forms of subqueries – scalar, multi-valued, nested, and correlated – emphasizing their application in complex data retrieval. Additionally, we introduced Common Table Expressions (CTEs) as a clean approach to organizing SQL queries for improved readability and functionality. By mastering these constructs, you can enhance your SQL skills, making data manipulation and reporting significantly more efficient. Understanding these queries and their outputs empowers you to handle real-world data challenges with confidence.

Chapter 12 - Views - DML On Views - Stored Procedures - With Input and Output Parameters - If Else and Else If Ladder

Introduction

In this chapter, we delve into the intricacies of database manipulation using SQL, particularly focusing on views and stored procedures. Views serve as virtual tables that allow users to simplify complex queries, while stored procedures enable the encapsulation of business logic on the server side. By understanding how to create, modify, and manipulate views alongside utilizing stored procedures with both input and output parameters, readers can enhance their data management capabilities while ensuring efficient and effective database interactions.

This chapter will provide practical examples and detailed explanations of queries using dummy data, offering insights into each operation's nuances.

1.Creating a View: vw_StudentsDetailsMVC

```
CREATE VIEW vw_StudentsDetailsMVC AS

SELECT Student.Sid, Student.SName, Course.Cid,

Course.CName, StudentCourse.Marks FROM Student

INNER JOIN StudentCourse ON Student.Sid =StudentCourse.Sid


INNER JOIN Course ON StudentCourse.Cid = Course.Cid;
```

Explanation: This command establishes a view that consolidates student information with their respective courses and marks. It accomplishes this through inner joins, ensuring only records with matches in all tables are included. The result set might include:

Sid	SName	Cid	CName	Marks
1	Alice	101	Introduction to CS	85
2	Bob	102	Data Structures	90
1	Alice	103	Operating Systems	78
3	Charlie	101	Introduction to CS	76
2	Bob	103	Operating Systems	92

2.Dropping a View

```
DROP VIEW vw_StudentsDetailsMVC;
```

Explanation: This command removes the specified view from the database. It is crucial for database management to maintain an updated schema, and views that are no longer needed should be discarded to avoid confusion.

3.Selecting Average Marks by Student

```
SELECT Sid, SName, AVG(Marks) AS avg FROM vw_Student
sDetailsMVC GROUP BY Sid, SName;
```

Explanation: This query calculates the average marks per student by grouping results based on student IDs and names. A potential result set is:

Sid	SName	avg
1	Alice	81.5
2	Bob	91.0
3	Charlie	76.0

4.Selecting Average Marks by Course

```
SELECT CName, AVG(Marks) AS avg FROM
vw_StudentsDetailsMVC GROUP BY CName;
```

Explanation: This query presents the average marks for each course, showcasing overall student performance within specific subjects:

CName	Avg
Introduction to CS	80.5
Data Structures	90.0
Operating Systems	85.0

5.Creating Another View: vw_EmployeeDetailsMVC

```
CREATE VIEW vw_EmployeeDetailsMVC AS
SELECT E.Eid, E.Name, E.Gender FROM Employee E;
```

Explanation: This view pulls specific columns from the Employee table, focusing on essential details like employee ID, name, and gender. The structure aids in easily accessing employee data without cluttering the main tables.

6.Selecting All Columns from Employee View

```
SELECT * FROM vw_EmployeeDetailsMVC;
```

Explanation: This command retrieves all employee records from the created view, simplifying data retrieval and ensuring clarity in information presented.

7.Inserting New Records into Views

```
INSERT INTO vw_EmployeeDetailsMVC VALUES('Huzaifa','M')
;
```

Explanation: This command attempts to insert a new employee record into the view. However, it's worth noting that many views are not updateable due to their complexity or the underlying SELECT statement restrictions.

8.Deleting Records from Views

```
DELETE FROM vw_EmployeeDetailsMVC WHERE Eid = 105;
```

Explanation: This statement attempts to remove an employee by their ID. The ability to delete from views is often contingent on how they were defined.

9.Updating Records in Views

```
UPDATE vw_EmployeeDetailsMVC SET Name = 'Huzaifa'
WHERE Eid = 103;
```

Explanation: This command modifies an existing employee record's name based on their ID. Update operations on views can succeed as long as they keep to the updatable constraints set by the base tables.

10.Viewing Student Details Again

```
SELECT * FROM vw_StudentsDetailsMVC;
```

Explanation: This query retrieves all student data from the view, presenting a comprehensive summary of student performance across all courses as previously defined in **vw_StudentsDetailsMVC**.

11.Counting Courses by Name

```
SELECT CName, COUNT (*) AS NOS FROM

vw_StudentsDetailsMVC
```

```
WHERE CName = 'Operating Systems' GROUP BY CName;
```

Explanation: This query counts the number of students taking the "Operating Systems" course. The result provides insight into the course's popularity:

CName	NOS
Operating Systems	2

12.Creating a Stored Procedure for Employee Details

```
CREATE PROC sp_EmployeeDetailsMVC AS
SELECT Employee.Eid, Employee.Name, Employee.Email,
Employee.Gender, Employee.Salary, Employee.DOB,
Department.DName, Department.Description,
Department.IsActive FROM Employee
INNER JOIN Department ON Employee.DId=Department.Did
```

Explanation: This stored procedure encapsulates the logic to retrieve comprehensive employee details linked to their departments. It becomes a reusable query facilitating easy access to critical information.

13.Executing the Stored Procedure

```
EXEC sp_EmployeeDetailsMVC;
```

Explanation: Executing the procedure runs the predefined query, returning all employee details along with department information in a structured format.

14.Filtering Employee Details by Department ID

```
CREATE PROC sp_EmployeeDetailsByDidMVC
(@Did AS INT) AS
SELECT Employee.Eid, Employee.Name, Employee.Email,
Employee.Gender, Employee.Salary, Employee.DOB,
Department.DName, Department.Description,
Department.IsActive FROM Employee
INNER JOIN Department ON Employee.DId=Department.Did
WHERE Employee.Did = @Did;
```

Explanation: Here, an input parameter (**@Did**) enables filtering by department ID, thus providing tailored employee data based on departmental associations. Executing this procedure would yield results specific to the given department.

15.Handling Null Parameters in Procedures

```
ALTER PROC sp_EmployeeDetailsByDidMVC (@Did AS INT)
AS
IF (@Did IS NULL)
BEGIN
```

```
SELECT Emp.Eid, Emp.Name, Emp.Email, Emp.Gender,
Emp.Salary, Emp.DOB, Dept.DName, Dept.Description,
Dept.IsActive FROM Employee Emp

INNER JOIN Department Dept ON Emp.Did = Dept.Did
END

ELSE
BEGIN
SELECT Emp.Eid, Emp.Name, Emp.Email, Emp.Gender,
Emp.Salary, Emp.DOB, Dept.DName, Dept.Description,
Dept.IsActive FROM Employee Emp

INNER JOIN Department Dept ON Emp.Did = Dept.Did
WHERE Emp.Did = @Did;
END;
```

Explanation: This logical structure allows flexibility in handling requests, where if a null parameter is passed, it retrieves all employee records, otherwise, it filters to the specified department.

16.Running the Modified Procedure

```
EXEC sp_EmployeeDetailsByDidMVC NULL;
```

Explanation: Executing with NULL yields all employee records, answering overarching queries regarding employee demographics without restrictions.

17.Extending Procedure Functionality

```
ALTER PROC sp_EmployeeDetailsByDidMVC
(@Did AS INT, @Gender AS VARCHAR (10))
AS
IF (@Did IS NULL AND @Gender IS NULL)
```

```
BEGIN
SELECT Emp.Eid, Emp.Name, Emp.Email, Emp.Gender,
Emp.Salary, Emp.DOB, Dept.DName, Dept.Description,
Dept.IsActive FROM Employee Emp
INNER JOIN Department ON Emp.DId = Dept.Did;
END
ELSE IF (@Did IS NOT NULL AND @Gender IS NULL)
BEGIN
 ...
END
-- Additional conditions here...
```

Explanation: This modification allows filtering by multiple criteria—department and gender, enriching the procedure's utility in managing diverse queries.

18.Final Executions and Outputs

```
EXEC sp_EmployeeDetailsByDidMVC NULL, 'M';
```

Explanation: This retrieves all male employees, offering insight into workforce demographics concerning gender distribution among employees.

Conclusion

This chapter explored the fundamental operations of views and stored procedures in SQL. We demonstrated how to create, manipulate, and query views effectively while encapsulating complex logic within stored procedures. By leveraging input and output parameters, you can craft flexible and robust data retrieval mechanisms, ultimately optimizing database interactions. Moving forward, mastering these concepts will enhance your proficiency in SQL and empower you to design more effective data-driven applications.

Chapter 13 - Functions - Try ... Catch - Transactions

Introduction

In today's programming landscape, error handling and transaction management are vital for building robust applications. This chapter delves into advanced SQL techniques including functions, error handling with Try...Catch constructs, and transaction management. We will explore various SQL queries step-by-step, illustrating how to implement functions for calculations, manage errors gracefully, and ensure data integrity through transactions. Let's unlock the power of SQL with these essential practices that facilitate seamless database interactions and maintain consistency in business logic.

1.Current System Date and Time

```
Select GetDate()
```

The `GetDate()` function retrieves the current system date and time from the SQL server. It's useful for logging transactions or any operations that require a timestamp. When executed, it produces an output such as `2024-09-06 10:25:30.123`, displaying the date and exact time.

Example Result Set:

Date and Time
2024-09-06 10:25:30.123
2024-09-06 10:25:30.456

2024-09-06 10:25:30.789
2024-09-06 10:25:31.000
2024-09-06 10:25:31.321

2.Calculating Gross Salary

```
Select Eid, Name, Email, Salary as BS,
(Salary + Salary*(11.0/100) + Salary*(9.0/100)) as GS
From Employee
```

In this query, we are calculating the Gross Salary (GS) for employees by adding the Basic Salary (BS), a Housing Rent Allowance (HRA) of 11%, and a Dearness Allowance (DA) of 9%. This is crucial for understanding compensation packages and ensuring fair employee remuneration.

Example Result Set:

Eid	Name	Email	BS	GS
1	John Doe	john@example.com	50000	60500
2	Jane Smith	jane@example.com	60000	72600
3	Mike Brown	mike@example.com	45000	52800
4	Lisa White	lisa@example.com	70000	84700
5	Tony Stark	tony@example.com	80000	94800

3.Creating a Function for Gross Salary

```
Create Function GrossSalary (@Salary as float)
returns float
as
begin
   declare @GS as float
   set @GS = (@Salary + @Salary*(11.0/100) + @Salary*(9.0/100))
   return @GS
end
```

Here, we created a scalar function named GrossSalary, which accepts a salary value and returns the calculated gross salary. The formula encapsulated in the function makes it reusable, ensuring consistent calculations across various queries and reports.

4.Calling the GrossSalary Function

```
Select dbo.GrossSalary (786543)
```

This invocation calls the GrossSalary function with the specific salary value of 786,543. It helps in validating the function's implementation and understanding how it calculates gross salary.

Example Result Set:

Gross Salary
966309.87

5.Using GrossSalary for All Employees

```
Select Eid, Name, Email, Salary as BS, dbo.GrossSalary(Salary)
 as GS From Employee
```

This query demonstrates how to leverage the GrossSalary function to calculate gross salaries for all employees efficiently. By using the function in the SELECT statement, we maintain accuracy, and minimize redundancy in computations.

Example Result Set:

Eid	Name	Email	BS	GS
1	John Doe	john@example.com	50000	60500
2	Jane Smith	jane@example.com	60000	72600
3	Mike Brown	mike@example.com	45000	52800
4	Lisa White	lisa@example.com	70000	84700
5	Tony Stark	tony@example.com	80000	94800

6.Function to Determine Grades

```
Alter Function GetGradeMVC(@AvgMarks as float)
returns varchar(10)
as
begin
   declare @Grade as varchar(10)
```

```
    Set @Grade = case
        when @AvgMarks >= 35 and @AvgMarks < 50 then 'C'
        when @AvgMarks >= 50 and @AvgMarks < 60 then 'B'
        when @AvgMarks >= 60 and @AvgMarks < 75 then 'A'
        when @AvgMarks >= 75 then 'A++'
    end
    return @Grade
end
```

This function, GetGradeMVC, accepts average marks as an argument and returns a grade based on predefined thresholds. It utilizes a CASE statement to evaluate the average marks and assign the corresponding grade. This is essential for educational systems to maintain a standardized grading process.

7.Calling GetGradeMVC Function

```
Select [dbo].[GetGradeMVC](78)
```

In calling the GetGradeMVC function with an average of 78, we can effectively determine its corresponding grade. This single-use example demonstrates the function's utility.

Example Result Set:

Grade
A++

8.Average Marks and Grade Assignment

```
Select Sid, SName, Avg (Marks) as Avg,
dbo.GetGradeMVC (Avg(Marks)) as Grade
from vw_StudentsDetailsMVC group by Sid, SName
```

This query retrieves students' IDs and names while computing their average marks from a view called vw_StudentsDetailsMVC. It then uses the GetGradeMVC function to assign grades based on these averages. Such evaluations help institutions streamline performance assessment processes.

Example Result Set:

Sid	SName	Avg	Grade
1	Alice	82.5	A++
2	Bob	65.0	A
3	Charlie	40.0	C
4	David	55.0	B
5	Emily	90.0	A++

9.Using CTE for Student Results

```
With StudentResult (Grade, Sid) as
(
   Select dbo.GetGradeMVC (Avg(Marks)) as Grade, Sid
   from vw_StudentsDetailsMVC
   Group by Sid
)
Select Grade, Count (Sid) as NOS from StudentResult
Group by Grade
```

This CTE computes grades for students and counts how many belong to each grade classification. The use of CTE simplifies query logic, improves readability, and enhances maintainability, making it easier to analyze educational outcomes.

Example Result Set:

Grade	NOS
A++	2
A	1
B	1
C	1

10.Handling Pass/Fail Status

```
With StudentResult(Grade, Sid) as
(
   SELECT Case
      When Avg(StudentCourse.Marks) < 75 then 'Fail'
      When Avg(StudentCourse.Marks) >= 75 then 'Pass'
   end as Grade, Student.Sid
FROM StudentCourse
INNER JOIN Student ON StudentCourse.Sid = Student.Sid
INNER JOIN Course ON StudentCourse.Cid = Course.Cid
Group by Student.Sid
)
Select Grade, Count(Sid) as NOS from StudentResult Group b
y Grade
```

This CTE assesses students' average marks while categorizing them as "Pass" or "Fail." It plays a crucial role in the educational landscape by allowing institutions to pinpoint areas needing improvement or recognizing outstanding performance.

Example Result Set:

Grade	NOS
Pass	3
Fail	2

11.Average Marks and Pass/Fail Grade

```
SELECT Student.Sid, Student.SName, Avg(StudentCourse.Marks) As Avg,
Case
When Avg(StudentCourse.Marks) < 75 then 'Fail'
When Avg(StudentCourse.Marks) >= 75 then 'Pass'
end as Grade FROM StudentCourse
INNER JOIN Student ON StudentCourse.Sid = Student.Sid
INNER JOIN Course ON StudentCourse.Cid = Course.Cid
Group by Student.Sid, Student.SName
```

This query aggregates average marks for students and assigns them a pass/fail grade based on performance criteria. It facilitates the identification of students who may need additional support to succeed academically.

Example Result Set:

Sid	SName	Avg	Grade
1	Alice	80	Pass
2	Bob	73	Pass
3	Charlie	65	Fail
4	David	84	Pass
5	Emily	49	Fail

12.Summarizing Pass/Fail Results Across Students

```
SELECT
Case
When Avg(StudentCourse.Marks) < 75 then 'Fail'
When Avg(StudentCourse.Marks) >= 75 then 'Pass'
end as Grade, count (Student.Sid) as NOE
FROM StudentCourse
INNER JOIN Student ON StudentCourse.Sid = Student.Sid
INNER JOIN Course ON StudentCourse.Cid = Course.Cid
Group by Case
When Avg(StudentCourse.Marks) < 75 then 'Fail'
When Avg(StudentCourse.Marks) >= 75 then 'Pass'
end
```

This summarizing query counts the number of students passing or failing across the year. It collates assessment data, allowing educators to observe academic performance trends and make informed decisions for future curriculum enhancements.

Example Result Set:

Grade	NOE
Pass	4
Fail	1

13.Function to Get Employees by Department

```
Create Function GetEmployees (@Did as int)
returns Table
as
return
(Select * from Employee Where Did = @Did)
```

We create a table-valued function named GetEmployees that retrieves all employees from a specified department. This approach allows for easy access to department-specific employee data, enhancing modularity in database interactions.

14.Retrieve All Employees by Department ID

```
Select * from dbo.GetEmployees(1000)
```

With the GetEmployees function, we can efficiently fetch records of employees belonging to a specific department. This showcases the effectiveness of using functions for focused queries.

Example Result Set:

Eid	Name	Email	Salary
1	John Doe	john@example.com	50000
2	Jane Smith	jane@example.com	60000
3	Mike Brown	mike@example.com	45000
4	Lisa White	lisa@example.com	70000
5	Tony Stark	tony@example.com	80000

15.Displaying All Customers

```
Select * from Customer
```

This simple query retrieves and displays all customer records from the Customer table. Understanding customer data is crucial for businesses to tailor their services effectively.

Example Result Set:

CustomerID	CustomerName	Balance
1	Adam	25000
2	Eve	30000
3	John	15000
4	Sarah	40000
5	Tom	50000

16.Inserting Customer with Error Handling

```
Begin Try
insert into Customer values (7, 'AdamAdam(long name)', 780)
End Try
Begin Catch
print 'Admin is working on it'
End Catch
```

This block attempts to insert a new customer into the Customer table, handling potential errors with Try...Catch. Such error management is vital for maintaining user experience and application stability.

Example Output:
Output: 'Admin is working on it' if an error occurs.

17.Transferring Funds Between Customers

```
Begin Try
Select * from Customer
Update Customer Set Bal = Bal - 5000 where CName = 'Lilly'
Update Customer Set Bal = Bal + 5000 where CName = 'Tom'
Select * from Customer
End Try
Begin Catch
print 'Admin is working on it'
End Catch
```

This section attempts to transfer $5,000 from one customer (Lilly) to another (Tom) using try/catch for error handling. It highlights the importance of transactional integrity and immediate feedback in cases of errors.

18.Another Attempt to Transfer Funds

```
Begin Try
Select * from Customer
Update Customer Set Bal = Bal - '5000' where CName = 'Lilly'
Update Customer Set Bal = Bal + '5000' where CName = 'Tom'
Select * from Customer
End Try
Begin Catch
print 'Admin is working on it'
End Catch
```

While this block tries to perform the same fund transfer as before, it redundantly refers to '5000' as a string, risking an error. Best practices assert that currency values should be handled as appropriate data types to avoid runtime exceptions.

19.Transaction Management for Fund Transfer

```
Begin Transaction MyTrans
Begin Try
Select * from Customer
```

```
Update Customer Set Bal = Bal - '5000' where CName = 'Lilly'

Update Customer Set Bal = Bal + '5000' where CName = 'Tom'

Select * from Customer

Commit Transaction MyTrans

End Try
Begin Catch

print 'Admin is working on it'

Rollback Transaction MyTrans
End Catch
```

The final section emphasizes transactions. By encapsulating the fund transfer in a transaction, any errors during execution will cause a rollback, reverting all changes. It ensures that the data remains consistent and reliable, making it an essential practice in any transactional system.

Conclusion

In this chapter, we covered critical SQL concepts such as functions, error handling with Try...Catch, and transaction management. By implementing functions, not only did we enhance our ability to process data efficiently, but we also set a precedent for error handling that promotes a user-friendly experience. Moreover, through transactions, we learned how to protect our data's integrity during failures in execution. These techniques empower developers to create reliable systems that maintain data consistency and uphold business logic. Mastering these practices is essential for anyone looking to excel in database management and application development.

Chapter 14 - Realtime Scenarios For Transaction - Scope_Identity() - Triggers Introduction

Introduction

In the realm of database management systems, ensuring data integrity and consistency is paramount, particularly when handling transactions that involve multiple operations. This chapter delves into the practical applications of the Scope_Identity() function in SQL transactions, as well as the implementation of triggers to enhance functionality within databases. By understanding these components through real-world scenarios, developers and database administrators can leverage these features to maintain robust and reliable data operations in any application.

1.Selecting Department Records

```
SELECT * FROM Department ORDER BY 1 DESC;
```

The above SQL command retrieves all records from the Department table and orders them by the first column in descending order. When executed, it might return a result set like this:

DepartmentID	DepartmentName	Description
5	HR	Human Resources
4	Marketing	Marketing Division

3	IT	Information Technology
2	Sales	Sales and Marketing
1	Finance	Financial Department

2.Selecting Employee Records

```
SELECT * FROM Employee ORDER BY 1 DESC;
```

This SQL command fetches all records from the Employee table and organizes them by the first column in descending order. The typical output might look like this:

EmpID	Name	Email	Gender	Salary	DeptID
3	Jane Doe	jane@example.com	F	8000	2
2	John Smith	john@example.com	M	7000	1
1	Ravi	ravi@gmail.com	M	9000	3

3.Creating a Stored Procedure for Employee Creation

```
ALTER PROC Create Employee (
   @DName AS VARCHAR (50),
   @Description AS VARCHAR (500),
   @EmpId AS INT,
   @Name AS VARCHAR (50),
   @Email AS VARCHAR (50),
   @Gender AS VARCHAR (10),
   @Salary AS FLOAT,
   @DOB AS DATE,
   @Contact AS VARCHAR (50))
AS
BEGIN
BEGIN TRAN CEmp;
BEGIN TRY
INSERT INTO Department VALUES (@DName, @Description)
INSERT INTO Employee VALUES (@EmpId, @Name, @Email,
@Gender, @Salary, @DOB, @Contact, SCOPE_IDENTITY());

COMMIT TRAN CEmp;
END TRY
BEGIN CATCH
PRINT 'Insertion Failed';
ROLLBACK TRAN CEmp;
END CATCH;
END;
```

This stored procedure CreateEmployee aims to streamline the employee creation process by taking parameters for both the employee and the department. When executed, the procedure first initiates a transaction to ensure that both inserts (to the Department and Employee tables) occur together or not at all, enhancing integrity. If the department insertion succeeds, it utilizes SCOPE_IDENTITY() to remember the last inserted department ID for the new employee record. Should any error arise during the process, it will print an error message and rollback the transaction, ensuring no partial data remains in the database.

Example Execution - Adding an Employee to IT Department

```
EXEC CreateEmployee 'IT', 'Info Tech', 666, 'Ravi', 'ravi@gmail.com', 'M', 9000, '1984-04-23', '9898767654';
```

Suppose we execute this stored procedure to add an employee named Ravi to the IT department. If successful, the Department will now have an entry for 'IT', and the Employee table will include Ravi's details along with the appropriate department ID. This demonstrates efficient employee onboarding, ensuring all relevant data aligns correctly.

4.Transaction for Adding Employees & Their Salaries

```
BEGIN TRAN SEmp;
BEGIN TRY
INSERT INTO Emp VALUES ('Jack', 7000, NULL);

INSERT INTO EmpSalary VALUES

(SCOPE_IDENTITY(), 8000, GETDATE(), NULL);

COMMIT TRAN SEmp;
END TRY
BEGIN CATCH
PRINT 'Insertion Failed';

ROLLBACK TRAN SEmp;
END CATCH;
```

This transaction illustrates inserting into the Emp table and subsequently the EmpSalary table. The first command creates a new employee record for 'Jack', while the SCOPE_IDENTITY() function ensures that the salary record links to the correct employee without hardcoding IDs. If either insertion fails, the whole transaction will rollback, preventing partial records from disrupting data integrity.

5.Resulting Employee and Salary Records

When adding Jack, the resultant tables might look like this:

Emp Table

EmployeeID	Name	Salary
4	Jack	7000

EmpSalary Table

SalaryID	EmployeeID	Salary	Date
1	4	8000	2024-09-06

In this example, Jack's details, including his salary, are recorded successfully thanks to the SCOPE_IDENTITY().

6.Creating a Trigger for Customer Insertion

```
CREATE TRIGGER AfterInsertingCustomer ON Customer
AFTER INSERT
AS
BEGIN
PRINT 'You have inserted the record!';
END;
```

This trigger is assigned to the Customer table to execute after any new record insertion. Whenever a record is added to the Customer table, the message indicates a successful insertion. It helps in maintaining awareness about data changes and can be useful for logging or auditing purposes.

7.Executing an Insert to the Customer Table

```
INSERT INTO Customer VALUES (7, 'John', 7900);
```

Upon executing this insert command, the trigger will automatically print 'You have inserted the record!', showcasing the dynamic response capabilities of triggers to data operations.

Conclusion

In conclusion, the chapter has provided insights into the practical applications of transactions, the SCOPE_IDENTITY() function, and triggers within SQL Server. By leveraging these features, developers and database administrators can ensure robust data integrity and performance within their applications. Understanding how to efficiently manage employee records, department alignments, and timely feedback via triggers reinforces the importance of transactional integrity in modern database systems. This knowledge empowers professionals to implement best practices in database management, leading to more reliable and maintainable systems.

Chapter 15 - Triggers - Inserted and Deleted Temp Tables - Self Join - Union Concept - DB BackUp - DB Restore - DBScript - DB Design Tasks

Introduction

In the realm of database management, understanding the inner workings of triggers, joins, and union operations is crucial for maintaining data integrity and enriching data interaction. This chapter delves into various SQL features that enhance how we work with relational databases, particularly focusing on the utilization of triggers, temporary tables, self-joins, and union queries. These concepts are exemplified through different scenarios involving employee and customer data, showcasing not just the queries themselves, but also the practical implications of each.

1.Selecting all columns from the Emp table (Emp details)

```
SELECT * FROM EMPLOYEE;
```

The SQL command above retrieves all columns from the **Emp** table, which contains complete information about the employees. For instance, the result set may look like this:

Eid	Name	Salary	ManagerId
1	John	60000	NULL
2	Alice	75000	1
3	Bob	45000	1

4	Kitty	56000	NULL
5	OLA	56000	NULL

This query is foundational in database management, offering a comprehensive view of current employee data.

2. Selecting all columns from the EmpSalary table (Employee salary details)

```
SELECT * FROM EMPSALARY;
```

This command fetches all records from the **EmpSalary** table that holds the salary history of the employees. For example, the output might include:

Eid	Salary	Date	Comment
1	60000	2024-09-06	NULL
2	75000	2024-09-06	NULL
3	45000	2024-09-06	NULL
4	56000	2024-09-06	NULL
5	56000	2024-09-06	NULL

This data enables tracking salary changes over time, enhancing fiscal oversight in corporate settings.

3. Creating a trigger that activates after a new row is inserted into the Emp table

Create Trigger SalHistory on Emp After Insert as Select * from inserted

This SQL command establishes a trigger named **SalHistory** that executes after a new employee record is inserted into the **Emp** table. The **inserted** pseudo-table contains the new rows. For example, if a new employee named 'Kitty' is added, the trigger would output:

Eid	Name	Salary	ManagerId
4	Kitty	56000	NULL

The trigger not only registers the new entry but is pivotal in ensuring that secondary actions such as logging or notification follow.

4. Example system procedure to describe the structure of the Emp table

SP_Help Emp
This command invokes a system stored procedure to present detailed information about the structure of the **Emp** table, such as column names, data types, and constraints. The output may include:

ColumnName	Type	Length	Nullable
Eid	int	4	NO
Name	varchar	50	NO
Salary	float	8	NO

ManagerId	int	4	YES

This smart overview helps database administrators and developers understand data schema for effective querying.

5. Inserting a new employee record into the Emp table with a name 'Kitty' and salary 56000

insert into Emp values ('Kitty',56000,NULL)

This command adds a record for a new employee, Kitty, with a salary of 56000. After this operation, querying the **Emp** table would yield:

Eid	Name	Salary	ManagerId
4	Kitty	56000	NULL

This addition demonstrates how new employees can be seamlessly added to the system, impacting subsequent financial records.

6. Altering the existing trigger 'SalHistory' to include additional functionality

Alter Trigger SalHistory on Emp After Insert as declare @Eid as int declare @Salary as float Select @Eid=Eid,@Salary=Salary from inserted insert into EmpSalary values (@Eid,@Salary,GetDate(),NULL)

This command modifies the **SalHistory** trigger to insert the new employee's salary into the **EmpSalary** table whenever a new record is added to **Emp**. For instance, after inserting 'Kitty', the **EmpSalary** table would update to look like:

Eid	Salary	Date	Comment
4	56000	2024-09-06	NULL

This action ensures that all salary entries remain current, automatically tracking employee salaries.

7. Inserting another employee record into the Emp table with name 'OLA' and salary 56000

```
INSERT INTO Employee Values (‘OLA’, 56000, NULL);
```

Adding another employee, OLA, increases the employee count in the **Emp** table. If ‘OLA’ shares the same salary, the updated table would include:

Eid	Name	Salary	ManagerId
5	OLA	56000	NULL

Such operations highlight the dynamic nature of the table, as employees can be added without manual array adjustments.

8. Selecting Employee ID, Name, Salary from Emp table and left joining it with Emp table to get manager information

Select E.Eid, E.Name, E.Salary, M.Name as Manager, M.Salary as ManagerSalary from Emp E left outer join Emp M on E.ManagerId = M.Eid

This query retrieves each employee's details alongside their manager's name and salary by employing a left outer join on the **Emp** table. An example result could be:

Eid	Name	Salary	Manager	ManagerSalary
1	John	60000	NULL	NULL
2	Alice	75000	John	60000
3	Bob	45000	John	60000
4	Kitty	56000	NULL	NULL
5	OLA	56000	NULL	NULL

In this case, employees without managers appear clearly, which is vital for understanding organizational hierarchy.

9. Selecting all columns from CustomerA

```
SELECT * FROM CustomerA;
```

This command fetches all records from the **CustomerA** table. Its potential output could be:

CustId	Name	Contact
1	Alice	123-456-7890
2	Bob	234-567-8901
3	Charlie	345-678-9012
4	David	456-789-0123
5	Ella	567-890-1234

The ability to extract all customer data enhances customer service and database analytics.

10. Selecting all columns from CustomerB

```
SELECT * FROM CustomerB;
```

Similar to CustomerA, this command acquires all data from the **CustomerB** table. The result may showcase:

CustId	Name	Contact
1	Alice	098-765-4321
2	Frank	876-543-2109
3	Charlie	345-678-9012
4	Grace	678-901-2345
5	Bob	234-567-8901

Consolidating different customer data aids in comparative analysis between customer groups.

11. Combining results from CustomerA and CustomerB using UNION ALL

Select Name, Contact from CustomerA Union All Select Name, Contact from CustomerB order by Name

This statement merges data from both **CustomerA** and **CustomerB**, retrieving all records without eliminating duplicates. The output might look like this:

Name	Contact
Alice	123-456-7890
Alice	098-765-4321
Bob	234-567-8901
Bob	234-567-8901
Charlie	345-678-9012
Charlie	345-678-9012
David	456-789-0123
Ella	567-890-1234
Frank	876-543-2109
Grace	678-901-2345

Utilizing **UNION ALL** allows comprehensive data analysis without excluding any entries.

12. Combining results from CustomerA & CustomerB using UNION

Select Name, Contact from CustomerA Union Select Name, Contact from CustomerB order by Name

This command merges customer data while removing duplicates, providing a unique result set. A potential result might be:

Name	Contact
Alice	123-456-7890
Bob	234-567-8901
Charlie	345-678-9012
David	456-789-0123
Ella	567-890-1234
Frank	876-543-2109
Grace	678-901-2345

The **UNION** operation ensures that duplicate customer entries from both tables do not skew analyses.

13. Finding common records (intersection) between CustomerA and CustomerB

Select Name, Contact from CustomerA intersect Select Name, Contact from CustomerB order by Name

This query identifies common customers between **CustomerA** and **CustomerB** by examining overlapping records. A possible output could be:

Name	Contact
Charlie	345-678-9012
Bob	234-567-8901

By leveraging **INTERSECT**, organizations can pinpoint shared customers, facilitating integrated marketing strategies.

14. Finding records in CustomerA that are not in CustomerB using EXCEPT

Select Name, Contact from CustomerA except Select Name, Contact from CustomerB order by Name

The above SQL command retrieves customers in **CustomerA** who are absent from **CustomerB**. The resulting set might appear as follows:

Name	Contact
Alice	123-456-7890
David	456-789-0123
Ella	567-890-1234

Utilizing **EXCEPT** allows businesses to identify exclusive customers, aiding targeted outreach efforts.

15. Finding common employee names between Employee and Emp tables

Select Name from Employee intersect Select Name from Emp

This command discovers overlapping employee names between the **Employee** and **Emp** tables, which may yield:

Name
Alice
Bob

This intersection can provide valuable employee insights for cross-analysis.

Conclusion

In this chapter, we explored pivotal SQL features such as triggers, joins, union operations, and specialized queries that facilitate effective data analysis and management. By understanding and implementing these techniques, database users can significantly improve their efficiency and accuracy in handling relational data. With adaptable trigger management, robust data querying, and insightful union queries, the mastery of these concepts prepares individuals for more complex database tasks and optimizes overall data workflows.

MS SQL Server Simplified

Unlock the Power of SQL Server with Ease!

In MS SQL Server Simplified, Manzoor Ahmed Mohammed, a Microsoft Certified Trainer and renowned software development educator, demystifies the complexities of SQL Server. Whether you're a beginner looking to enter the world of database management or a seasoned IT professional seeking to enhance your T-SQL skills, this book is your go-to guide.

With step-by-step instructions, real-world examples, and hands-on exercises, you'll learn how to install, manage, and query databases efficiently using Microsoft SQL Server. From foundational concepts like RDBMS to advanced querying techniques, this comprehensive resource equips you with the knowledge and tools needed to succeed in database administration and development.

Start your journey today and master SQL Server, the backbone of modern data management!