COMPLETE NOTES ON SQL...

Index

1. Introduction to Databases and SQL

1.1. Understanding Databases

Databases are structured repositories for storing, managing, and retrieving data.

In this section, we'll explore the fundamental concepts of databases, including:

- Data Organisation: Databases help organize data into tables, rows, and columns, ensuring data integrity and easy retrieval.
- Data Management: Learn how databases handle data through various operations like insertion, modification, and deletion.
- Data Retrieval: Understand how to retrive information from databases efficiently.

1.2. What is SQL?

SQL (Structured Query Language) is the language used to interact with relational databases.

In this part, We'll cover:

- Origins of SQL: A brief history of SQL and its development.
- SQL's Role: How SQL is essential for querying and manipulating data in databases.
- SQL Variants: An introduction to various SQL dialects like MySQL, PostgreSQL, and Microsoft SQL server.

1.3. Relational Databases

Relational databases are a common type of database management system (DBMS) that use a structured, tabular format for data storage.

Topics covered include:

- Tables and rows: This part explains that relational databases are based on a tabular structure where data is organised into tables, each of which is made up of rows and columns. Tables are analogous to spreadsheets, rows represent individual records, and columns

contain specific data attributes.

- Relationships : Readers will learn about the concept of relationships in databases. These relationships are established through keys, which are used to connect data across different tables. Primary keys and foreign keys play a crucial role in maintaining data consistency and integrity.

- Advantages of Relational Databases : This section explores the reasons behind the popularity of relational databases, including data integrity, data consistency, and the ability to perform complex queries efficiently.

1.4. The History of SQL :

- Early Beginnings : SQL's origins can be tracked back to the early 1970s when IBM researchers, particularly Donald D. Chamberlin and Raymond F. Boyce, developed a language called SEQUEL (Structured English Query language) as a part of the system R project. The project aimed to create a prototype RDBMS.

- SEQUEL to SQL : Initially, the language was called SEQUEL, but due to trademark

issues, it was later renamed SQL, IBM's System R served as a pioneering RDBMS, and SQL was created to interact with it efficiently.

- Standardization: SQL's standardization process is a critical point of discussion. Organizations like ANSI (American National Standards Institute) and ISO (International Organization for standardization) have been responsible for creating and maintaining SQL standards. This ensures, SQL remains consistent and interoperable across various database systems.

- Key Milestones: The section may cover major milestones and versions of SQL, including SQL-86, SQL-92, SQL-1999, SQL-2003, and so on. These milestones represent the evolution and enhancement of SQL over the years.

1.5. Setting up SQL environment:

1. Installing a DBMS: A Database Management System (DBMS) is a software that manages databases, stores and retrieves data, and allows users to

interact with the data through SQL. Different DBMS options ~~are~~ are available, and readers may choose the one that best suits their needs.

Commonly used DBMS systems include:

- MySQL: An open-source DBMS known for it's performance and ease of use.
- PostgreSQL: Another open-source DBMS with advanced features, extensibility, and support for JSON data.
- SQLite: A self-contained, serverless, and file-based DBMS, suitable for small-scale projects and embedded systems.
- SQL Server: A microsoft product suitable for enterprise-level applications.

The process of installing a DBMS varies depending on the system chosen. Readers will find detailed instructions for downloading, installing, and configuring their preferred DBMS. This may include setting up username and passwords for database access.

2. IDEs and tools: Once the DBMS is installed, readers can access it through integrated Development Environments (IDEs) and tools designed for working with SQL. These tools provide a user-friendly interface for writing, testing, and managing SQL queries.

Some popular IDEs and tools include:

- SQL Server Management Studio (SSMS): A microsoft-developed tool for SQL server.
- DBeaver: An open-source, cross-platform database tool supporting multiple DBMS systems.

- phpMyAdmin: A web-based tool for managing MySQL databases.
- pgAdmin: A postgreSQL management tool with a user-friendly interface.

In this part of the section, readers are introduced to one or more of these tools based on their DBMS choice. They'll learn how to connect to their newly installed database and execute SQL commands within the tool's environment.

3. Creating your first Database: After the DBMS and IDE are setup, readers are guided through the process of creating their first database.

This handson exercise allows them to put their knowledge into practice.

Key steps include:

- Creating a database: Readers will learn how to create a new database within the DBMS. They might be prompted to choose a name for the database and specify any initial settings.

- Creating Tables: In addition to creating a database, readers will be guided in creating their first table within that database. They'll define the table's structure, specifying columns, data types and constraints.

- Executing SQL Commands: This part may include executing SQL commands to create the database and table within the IDE.

2. SQL Basics

2.1. SQL Syntax :

In this introductory section, we delve into the building blocks of SQL statements.

- Keywords : SQL has a set of reserved words known as Keywords. These include SELECT, FROM, WHERE, ORDER BY, and others. Understanding these is crucial as they convey specific meanings and functions within SQL statements.

- Clauses : SQL statements consist of various clauses that perform different functions. The SELECT statement, for instance, includes clauses like SELECT, FROM, WHERE and ORDER BY, Each clause has a specific purpose in defining what the query should do.

- Statements : SQL statements are complete instructions that interact with database. The most basic is the SELECT statement, which is used to retrieve data. Other statements include INSERT, UPDATE, and DELETE for modifying data.

A solid understanding of SQL syntax sets the foundation for constructing meaningful and accurate queries.

2.2. SELECT Statement :

Here, the focus is on the **SELECT** statement, which is central to retrieving data from a database.

- **SELECT Clause:** This part of the statement determines which columns to include in the result set. It can be specific column names or a wildcard (*) to select all the columns.
- **FROM Clause:** Specifies the table or tables from where to retrieve data. Multiple tables can be included for more complex queries.
- **Wildcard Characters:** The use of an asterisk (*) allows for the selection of all columns in a table. This is particularly useful when exploring the structure of a table.
- **Alias:** Columns or tables can be temporarily renamed using aliases. This can improve

the readability of the query.

By grasping the SELECT statement, we gain the ability to extract specific data from one or more tables.

2.3. Filtering Data with WHERE:

The WHERE Clause enhances the SELECT statement by allowing the filtering of data based on specific conditions.

- Comparison Operators: We learn how to use operators like =, <, >, <=, >=, and <> to compare values. This is essential for specifying conditions in the WHERE Clause.
- Logical Operators: AND, OR, and NOT are logical operators used to combine conditions, offering flexibility in constructing complex queries.
- NULL values: Dealing with NULL values is covered, allowing us to handle situations where certain columns have no data.

- Multiple Conditions: Constructing queries with multiple conditions provides the ability to retrieve precisely the data needed.

By mastering WHERE Clause, we can efficiently narrow down the data we retrieve, ensuring relevance to our specific requirements.

2.4. Sorting Data with ORDER BY:

The ORDER BY Clause comes into play when organizing the results of a query.

- Ascending and Descending Order: The ORDER BY Clause can arrange results in ascending (ASC) or descending (DESC) order. This is crucial for presenting data in a meaningful way.
- Sorting by Multiple Columns: When sorting involves multiple columns, we learn how to prioritize the order in which columns are considered.
- Numeric and Alphanumeric Sorting: Different data types are sorted differently. Understan

ding this ensures that the sorted results align with expectations.

By mastering the ORDER BY clause, we gain control over the presentation of our query results.

2.5. Filtering and Sorting Data exercises:

This section provides hands-on exercises to reinforce the concepts of filtering and sorting data using SQL queries.

Each exercise aims to apply the knowledge gained in the preceding sections.

Exercise 1: Filter Data:

Objective: Craft a SQL Query to retrieve specific data based on given conditions.

Example:

Suppose we have a "Products" table with columns "ProductID," "ProductName", "Category" and "Price". The task is to retrieve all products in the "Electronics" category with a price greater than $500.

SQL Query:

```
SELECT ProductID, ProductName, Category, Price
FROM Products
WHERE Category = 'Electronics' AND Price > 500;
```

Explanation:

The Query uses the WHERE Clause to filter the products based on two conditions - Category equals 'Electronics' and Price greater than 500. The result will include only those products that meet both criteria.

Exercise 2: Sort Data:

Objective: Construct a SQL Query to sort data in a specified order.

Example:

Consider a "Employees" table with columns "EmployeeID", "First Name", "LastName", and "Salary". The goal is to retrieve all employees and sort them in descending order based on their salary.

SQL Query:

```
SELECT EmployeeID, FirstName, LastName,
       Salary
FROM Employees
ORDER BY Salary DESC
```

Explanation:

This query uses the ORDER BY Clause to sort the results based on the "Salary" column in descending order (DESC). The result set will display employees starting from the highest salary.

Exercise #3: Combine Filtering and Sorting:

Objective: Create a SQL Query that incorporates both filtering and sorting techniques.

Example:

In the 'Orders' table with columns "OrderID", "CustomerID", "OrderDate", and "TotalAmount", find all orders placed by the customer with ID 'C001' and sort them by the order date in ascending

order.

SQL Query:

```
SELECT OrderID, CustomerID, OrderDate,
       TotalAmount
FROM Orders
WHERE CustomerID = 'C001'
ORDERBY OrderDate ASC;
```

Explanation:

This query combines the WHERE Clause to filter orders by the customer with ID 'C001' and the ORDER BY Clause to sort the filter results based on the "OrderDate" column is ascending order.

These exercises are designed to provide practical application of SQL concepts learned in the chapter.

By working through these examples, users gain hands-on experience in constructing queries that filter and sort data, preparing them for more complex SQL tasks in future Chapters.

By the end of Chapter 2, we should feel confident in constructing basic SQL Queries, filtering data based on conditions, and presenting results in a sorted manner.

The combination of theory and practical exercises sets a strong foundation for our SQL Journey.

3. Data Manipulation

3.1. INSERT, UPDATE and DELETE Statements:

This section introduces three fundamental SQL statements for modifying data in a database.

- **INSERT Statement**: Used to add new records to a table. The syntax typically involves specifying the target table and providing values for each column or using a subquery.

Example:

```
INSERT INTO Employees (FirstName, LastName,
                       Salary);
VALUES ('John', 'Doe', 6000);
```

- **UPDATE Statement**: Enables the modification of existing records in a table. It involves specifying the target table, setting new values for the desired columns, and applying conditions to identify the records to be updated.

Example:

```
UPDATE Products
SET Price = Price * 1.1
WHERE Category = 'Electronics';
```

- **DELETE Statement**: Remove records from a table based on specified conditions. It involves specifying the target table and using a WHERE clause to identify the records to be deleted.

Example:

```
DELETE FROM Customers
WHERE LastPurchaseDate < '2023-01-01';
```

3.2. Modifying Data:

This section explores more advanced techniques for modifying data, including updating data based on values from another table, using subqueries in the WHERE clause for complex conditions, and handling data in multiple tables.

- **Updating Data with Subqueries:**

  - Purpose: Allows updating data in a table based on values from another table or a subquery.

  - Example:

```
UPDATE Products
SET Category = 'Electronics'
WHERE ProductID IN (SELECT ProductID FROM
                    TempProducts);
```

- Deleting Data with Subqueries
  - Purpose : Deletes records from a table based on conditions involving a sub-query.
  - Example :

```
DELETE FROM Customers.
WHERE CustomerID NOT IN (Select DISTINCT
      CustomerID FROM Orders);
```

3.3. Transaction Management

This part introduces the concept of transactions, which ensures the consistency and integrity of the database when multiple statements are executed together.

Transactions typically follow the ACID properties (Atomicity, Consistency, Isolation, Durability).

- Atomicity : A transaction is atomic, meaning

it's treated as a single, indivisible unit. It either completes entirely or has no effect at all. If any part of the transaction fails, the entire transaction is rolled back, ensuring that the database remains in a constant / consistent state.

- **Consistency**: Transactions bring the database from one consistent state to another. The database must satisfy integrity constraints before and after the transaction.

- **Isolation**: Each transaction should be isolated from other transactions, meaning the changes made by one transaction should not be visible to other transactions until it's commited. This prevents interference between concurrent transactions.

- **Durability**: Once a tracisaction is commited, its changes should be permanent and survive system failures. The changes are stored securely in the database.

BEGIN TRANSACTION:

- The 'BEGIN TRANSACTION' statement marks the start of a new transaction. All

SQL statements executed after this statements are considered part of the same transaction until it's either committed or rolled back.

COMMIT:

- The 'COMMIT' statement finalizes the transaction. It takes all the changes within the transaction permanent. If there are no issues or errors, executing 'COMMIT' ensues that the changes are saved to the database.

ROLLBACK:

- The 'ROLLBACK' statement is used to undo the changes made during the transaction. If any part of the transaction encounters an error or if there's a need to cancel the transaction for any reason, executing 'ROLLBACK' will revert the database to its state before the transaction started.

Example:

```
BEGIN TRANSACTION
UPDATE Accounts SET Balance = Balance - 100
   WHERE AccountID = 123;
INSERT INTO Transactions(AccountID, Amount,
   TransactionType) VALUES (123, 100, 'Withdrawl');
COMMIT;
```

In this example, we're conducting a transaction involving two statements. The first statement reduces the balance in the "Accounts" table for account number 123 by 100, simulating a withdrawl. The second statement records this withdrawl as a transaction in the "Transaction" table.

3.4. Data Manipulation Exercises:

This section is designed to provide hands-on practice for readers, allowing them to apply their knowledge of SQL data manipulation techniques. The exercise cover four main aspect.

~~Int~~ Inserting Data:

Objective: These exercises focus on inserting new records into tables with specified values.

Example:

Let's say you have a "Students" table with columns "StudentID", "FirstName", "LastName", and "GPA". An exercise might involve inserting a new student record into this table.

```
INSERT INTO Students (StudentID, FirstName,
                      LastName, GPA)
VALUES ('S001', 'Alice', 'Johnson', 3.8);
```

This exercise tests the reader's ability to correctly structure an INSERT statement to add data to a table.

Updating Data:

Objective: These challenges require updating existing records based on certain conditions.

Example:

Suppose you have an "Employees" table with columns "EmployeeID", "FirstName", "LastName", and "Salary". An exercise might ask readers to update the salary of all employees in the "Sales" department to reflect a 10% raise.

```
UPDATE Employees
SET Salary = Salary * 1.1
WHERE Department = 'Sales';
```

This exercise tests the reader's ability to use the UPDATE statement effectively to modify existing data.

Deleting Data:

Objective: Exercises guide users to delete records based on specific criteria.

Example:

Imagine you have a "Orders" table with columns "OrderId", "CustomerID", "OrderDate" and "TotalAmount". An exercise might involve deleting all orders that ~~that~~ were placed before a certain date.

```
DELETE FROM Orders
WHERE OrderDate < '2023-01-01';
```

This exercise tests the reader's ability to use the DELETE Statement to remove data from a table based on specified conditions.

Transaction Management:

Objective: Practical exercises help readers create and manage transactions to ensure data consistency.

Example:

An exercise might require readers to create

a transaction that transfers $500 from one bank account to another. It would involve both an update statement to decrease the balance in one account and an INSERT statement to record the transaction. The transaction should be wrapped in a BEGIN TRANSACTION and COMMIT statements to ensure data consistency.

```
BEGIN TRANSACTION

UPDATE Accounts SET Balance = Balance - 500
    WHERE AccountID = '123';
INSERT INTO Transactions (AccountID, Amount,
    TransactionType) VALUES ('123', 500,
        'Transfer Out');
UPDATE Accounts SET Balance = Balance + 500
    WHERE AccountID = '456';
INSERT INTO Transactions (AccountID, Amount,
    TransactionType) VALUES ('456', 500,
        'Transfer In');

COMMIT;
```

This exercise tests the reader's ability to create and manage a transaction effectively.

By the end of Chapter 3, readers should be proficient in using INSERT, UPDATE and DELETE statements, applying advanced

modifications technique, understanding transaction management and confidently completing data manipulation exercises.

This knowledge is crucial for anyone working with databases and SQL.

4. Retrieving Data with SQL.

4.1. Retrieving Data from a Simple Table:

This section focuses on the fundamental SQL operation of retrieving data from a single table.

- SELECT Statement: The SELECT Statement is used to specify which columns you want to retrive and from which table.
- FROM Clause: The FROM clause specifies the table from which you want to retrieve data.
- Example:

```
SELECT FirstName , LastName
FROM Employees ;
```

This query retrieves a list of unique product categories from the "Products" table, removing any duplicates.

4.2. Distinct Keyword

The ~~DIN~~ DISTINCT Keyword is used in the select statement to ensure that only unique

values are retrieved from a column. It eliminates duplicate values.

- Example :

```
SELECT DISTINCT Category
FROM Products ;
```

This query retrieves a list of unique product categories from the "Products" table, removing any duplicates.

4.3. Limiting Results with LIMIT :

The LIMIT Clause is used to restrict the number of rows retrieved from a result set. It is particularly useful when you want to limit the number of rows displayed or to sample a portion of a data.

- Example :

```
SELECT ProductName
FROM Products
LIMIT 10 ;
```

This query retrieves the names of the first 10 products from the "Products" table.

4.4. Retrieving Data from Multiple Tables (JOINS):

In the world of relational databases, data is often distributed across multiple tables. JOIN operations are powerful SQL techniques that enable you to combine and retrieve data from multiple tables in a single query.

This section covers the types of JOINs and their functionalities:

INNER JOIN:

- Purpose: Retrieves rows that have matching values in both tables.

- Syntax:

```
SELECT Columns
From Table1
INNER JOIN table2 ON table1.column =
        table2.column;
```

• Example:

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

This query retrieves the order IDs and customer names for orders where the customer IDs match in both the "Orders" and "Customers" table.

LEFT JOIN (or LEFT OUTER JOIN):

• Purpose: Retrieves all rows from the left table and the matched rows from the right table. Non-matching rows from the left table have NULL, values in the right table's columns.

• Syntax:

```
SELECT columns
FROM table1
LEFT JOIN table2 ON table1.column =
    table2.column
```

- Example:

```
SELECT Employees.FirstName, Orders.OrderID
FROM Employees
LEFT JOIN Orders ON Employees.EmployeeID =
        Orders.EmployeeID;
```

The query retrieves the first names of employees and their associated order IDs. It includes all employees, even those who haven't placed any orders. For employees without orders, the OrderID column will contain NULL values.

RIGHT JOIN (or RIGHT OUTER JOIN)

- Purpose: Retrieves all rows from the right table and the matched rows from the left table. Non-matching rows from the right table have NULL values in the left table's columns.

- Syntax:

```
SELECT columns
FROM table1
RIGHT JOIN table2 ON table1.column =
        table2.column;
```

- Example:

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

This query retrieves the order IDs and customer names for customers who have placed orders. It includes all orders, even those without associated customers. For orders without customers, the customerName column will contain NULL values.

FULL JOIN (or FULL OUTER JOIN):

- Purpose: Retrieves all rows from both tables, combining the results of LEFT and RIGHT JOINs. This includes all records from both tables, and NULL values in the columns where there are no matches.

- Syntax:

```
SELECT columns
FROM table1
FULL JOIN table2 ON table1.column = table2.column
```

- Example:

```
SELECT Employee.FirstName, Orders.OrderID
FROM Employee
FULL JOIN Orders ON Employees.EmployeeID =
          Orders.EmployeeID;
```

This query retrieves the first names of employees and their associated orderIDs. It includes all employees and all orders, and NULL values appear where there are no matches.

JOIN operations are fundamental for working with relational databases, as they allow you to extract and combine data from multiple related tables.

4.5. Retrieving Data Exercises:

This section offers practical exercises to allow readers to apply and reinforce their understanding of retrieving data from databases.

- Retrieving Data from a Single table:

Objective: These exercises focus on constructing SELECT statements to retrieve specific data from a single table.

Example:

Suppose you have a "Products" table with columns "ProductID", "ProductName", "Category" and "Price". An exercise might ask you to retrieve the names and prices of products in the "Electronics" category.

```
SELECT ProductName, Price
FROM Products
WHERE Category = 'Electronics';
```

This exercise tests your ability to craft a SELECT statement to retrieve specific data based on conditions.

- DISTINCT Keyword:

Objective: Challenges in this category require using the DISTINCT Keyword to retrieve unique values from a column.

Example:

Imagine you have a "Customers" table with a "country" column, and you are asked to retrieve a list of unique countries where customers are located.

```
SELECT DISTINCT Country
FROM Customers;
```

This exercise tests your understanding of the DISTINCT Keyword to eliminate duplicate values in your result set.

- Limiting Results with LIMIT:

Objective: These tasks involve using the LIMIT clause to restrict the number of rows in the result set.

Example:

In the "Orders" table with columns "OrderID", "CustomerID", and "OrderDate", you might be asked to retrieve the first 5 orders placed.

```
SELECT OrderID, CustomerID, OrderDate
FROM Orders
LIMIT 5;
```

This exercise tests your ability to limit the number of rows retrieved from a result set, which can be useful for data sampling or displaying only a subset of the data.

- Retrieving Data from Multiple Table (JOINS)

Objective: Exercises guide readers through performing various types of JOINs to retrieve data from multiple tables.

Examples:

You could have scenerio where you're asked to retrieve a list of employees and their assigned projects. This would involve using an INNER JOIN between the "Employees" and "Projects" tables based on the EmployeeID.

```
SELECT Employees.EmployeeID , Employees.
    FirstName , Projects.ProjectName
FROM Employees
INNER JOIN Projects ON Employees.
        EmployeeID = Projects.EmployeeID;
```

This exercise tests your ability to construct JOIN statements to retrieve data from related tables, combining information from multiple sources.

[redacted]

5. Data Filtering and Sorting

5.1. The WHERE Clause

The WHERE clause is a fundamental component of SQL queries. It's used to filter and select specific rows from a table based on specific conditions.

- Purpose: The WHERE Clause allows you to define conditions that must be met for a row to be included in the result set.
- Syntax:

```
SELECT Columns
FROM table
WHERE condition;
```

- Example:

```
SELECT ProductName , Price
FROM Products
WHERE Category = 'Electronics' AND Price>500;
```

This query retrieves product names and prices for items in the "Electronics" category with a price higher than $500.

5.2. Logical Operators:

Logical operators are used in combination with the WHERE Clause to create more complex conditions.

- AND : Requires both conditions to be true for a row to be selected.
- OR : Requires atleast one condition to be true for a row to be selected.
- NOT : Negates a condition, selecting rows that do not meet the specified condition.

Example (using AND):

```
SELECT ProductName, Price
FROM Products
WHERE Category = 'Electronics' AND
                 Price > 500;
```

This query selects products in the "Electronics" category with a price greater than $500.

Example (using OR):

```
SELECT ProductName , Price
FROM Products
WHERE Category = 'Electronics' OR
                 Category = 'Computers';
```

The query selects products in the "Electronics" or "Computers" category.

Example (using NOT):

```
SELECT ProductName , Category
FROM Products
WHERE NOT Category = 'Electronics';
```

The query retrieves product names and their categories for items that are not in the "Electronics" category. It includes products from all other categories.

5.3. Sorting Data with ORDER BY:

The ORDER BY clause is used to sort the result set in a specified order.

- Purpose: ORDER BY arranges the rows in ascending (ASC) or descending (DESC) order based on one or more columns.
- Syntax:

```
SELECT   columns
FROM   table
WHERE   condition
ORDER BY   column1 [ASC][DESC] ,
           column2 [ASC] DESC] , ... ;
```

Example:

```
SELECT ProductName , Price
FROM Products
WHERE Category = 'Electronics'
ORDER BY Price DESC ;
```

This query retrieves products in the "Electronics" category and sorts ~~name in~~ them in descending order by price.

5.4. Advanced Filtering and Sorting Exercises:

- Advanced Filtering:

Exercise: Retrieve products with a price greater than $500 in the "Electronics" category and from the "2023" model year.

Example:

```
SELECT ProductName, Price, Category, ModelYear
FROM Products
WHERE Category = 'Electronics' AND
                 Price > 500 AND
                 ModelYear = 2023;
```

This exercise challenges readers to construct a query with multiple conditions using logical operators to filter data based on complex criteria.

- Advanced Sorting:

Exercise: Sort products first category in ascending order and then within each category, sort by price in descending order.

Example:

```
SELECT ProductName, Price, Category
FROM Products
ORDER BY Category ASC, Price DESC;
```

This challenge encourages readers to use the ORDER BY clause to sort data in multiple dimensions, providing a nuanced ordering of the result set.

- Combining Filtering and Sorting:

Task: Retrieve the top 5 highest-priced products in the "Electronics" category and sort them by price in descending order.

Example:

```
SELECT ProductName , Price , Category
FROM Products
WHERE Category = 'Electronics'
ORDER BY Price DESC
LIMIT 5;
```

This task gives readers through combining both filtering and sorting techniques to retrieve a specific subset of data, demonstrating how these operations work in tandem.

6. Aggregating Data

6.1. Aggregate Functions (COUNT, SUM, AVG, MAX, MIN)

Aggregate functions in SQL perform a calculation on a set of values and return a single value.

These functions are often used to summarize data and provide insights into the dataset.

- COUNT: Counts the number of rows in a result set.

```
SELECT COUNT (*) FROM ORDERS;
```

- SUM: Calculates the sum of values in a numeric column.

```
SELECT SUM (Price) FROM Products;
```

- AVG: Computes the average of values in a numeric column.

```
SELECT AVG (Quantity) FROM OrderDetails;
```

- MAX: Retrieves the maximum value in a column.

```
SELECT MAX(Price) FROM Products;
```

- MIN : Retrieves the minimum value in a column.

```
SELECT MIN(Price) FROM Products;
```

These aggregate functions are powerful tools for summarizing and extracting meaningful information from large datasets.

6.2. GROUP BY Clause:

The GROUP BY Clause is used in ~~conjunct~~ conjunction with aggregate functions to group rows that have the same values in specified columns into summary rows.

- Example:

```
SELECT Category, AVG(Price)
FROM Products
GROUP BY Category;
```

This query groups products by category and calculates the average price for each category. The GROUP BY Clause divides the result set into subsets based on the specified column (in this case, "Category")

6.3. HAVING Clause :

The HAVING Clause filters the results of a GROUP BY Clause based on a specified condition.

- Example :

```
SELECT Category , AVG (Price)
FROM Products
GROUP BY Category
HAVING AVG (Price) > 500 ;
```

This query groups products by category, calculates the average price for each category and then filters the result to include only those categories with an average price greater than 500.

6.4. Aggregating Data Exercises :

- Example Using Aggregate Functions :

Exercise : Retrieve the total number of products and the average price of those products.

Example :

[redacted]

```
SELECT COUNT(*) AS TotalProducts,
       AVG(Price) AS AveragePrice
FROM Products;
```

In this exercise, the 'COUNT(*)' aggregate function is used to get the total number of products, and the 'AVG(Price)' function calculates the average price of all products.

- GROUP BY and Aggregations:

Exercise: For each category, find the total number of products, the highest price, and the lowest price.

Example:

```
SELECT Category, COUNT(*) AS TotalProducts
     , MAX(Price) AS HighestPrice,
       MIN(Price) AS LowestPrice
FROM Products
GROUP BY Category;
```

This exercise involves grouping products by category using the GROUP BY clause. The aggregate functions 'COUNT(*)', 'MAX(Price)', and 'MIN(Price)' are then applied to get the total number of products, the highest price, and the

lowest price for each category.

- Filtering Aggregate Date:

Exercise: Find the categories where the average price is greater than $500.

Example:

```
SELECT Category, AVG(Price) AS AveragePrice
FROM Products
GROUP BY Category
HAVING AVG(Price) > 500;
```

This task uses the HAVING Clause to filter the result set based on a condition. It retrieves the categories and their average prices but only includes those categories where the average price is greater than $500.

7. Subqueries

7.1. Subqueries in WHERE Clause:

A subquery in the WHERE clause is a query nested within another query's WHERE Clause. The result of the subquery is used to filter the rows returned by the outer query.

- Example:

```
SELECT ProductName
FROM Products
WHERE CategoryId IN (SELECT CategoryId
    FROM Categories WHERE CategoryName
        = 'Electronics');
```

In this example, the subquery retrieves the categoriesId for the category 'Electronics', and the outer query retrieves the product names for products in that category.

7.2. Subqueries in SELECT Clause:

A subquery in the SELECT clause is used to retrieve values that will be used in the main query's result set.

- Example:

```
SELECT ProductName , (SELECT AVG(Price)
        FROM Products) AS AvgPrice
FROM Products;
```

Here, the subquery calculates the average price of all products, and the outer query retrieves each product's name along with the overall average price.

7.3. Correlated Subqueries :

A correlated subquery is a subquery that refers to values from the other query. It is executed once for each row processed by the outer query.

- Example:

```
SELECT EmployeeName
FROM Employees e
WHERE Salary > (SELECT AVG (Salary)
    FROM Employees WHERE DepartmentID
        = e.DepartmentID);
```

In this examples, the correlated subquery compares each and every employee's salary

to the average salary within their department.

7.4. Subqueries Exercises:

- **Subqueries in WHERE Clause:**

Exercise: Retrieve the names of customers who have placed orders for products in the 'Electronics' category.

Example:

```
SELECT CustomerName
FROM Customers
WHERE CustomerID IN (SELECT CustomerID
    FROM Orders WHERE ProductID IN
        (SELECT ProductID FROM Products
        WHERE Category = 'Electronics'));
```

In this example, the subquery in the WHERE clause retrieves the customerIDs of customers who have placed orders for products in the 'Electronics' category. The outer query then retrieves the names of these customers.

- **Subqueries in SELECT clause:**

Exercise: Retrieve the product names along with the differences between each product's price and the average price ~~and~~ of all products.

Example:

```
SELECT ProductName, Price, Price - (SELECT
    AVG(Price) FROM Products) AS PriceDiff-
    erence
FROM Products
```

Here, the subquery in the SELECT clause calculates the average price of all products, and the outer query retrieves each product's name, price, and the difference between the product's price and the overall average price.

- Correlated Subqueries:

Exercise: Retrieve the names of employees whose salary is higher than the average salary in their department.

Example:

```
SELECT EmployeeName
FROM Employees e
WHERE Salary > (SELECT AVG(Salary)
```

```
FROM Employees WHERE DepartmentID =
    e.DepartmentID);
```

In this example, the correlated subquery compares each employee's salary to the average salary within their department, and the outer query retrieves the names of employees who meet this condition.

8. Data Modification and Transactions

8.1. Transactions in SQL:

A transaction in SQL is a sequence of one or more SQL statements that are executed as a single unit of work. Either all the statements within a transaction are executed successfully, or none of them are.

Example:

```
BEGIN TRANSACTION;

UPDATE Accounts
SET Balance = Balance - 100.
WHERE AccountID = 123;

INSERT INTO Transactions (AccountID, Amount,
    TransactionType)
VALUES (123, 100, 'Withdrawl');

COMMIT;
```

In this example, the statements between 'BEGIN TRANSACTION' and 'COMMIT' are part of a transaction fails, you can use a 'ROLLBACK' statement to undo the changes made so far.

8.2. COMMIT and ROLLBACK:

- COMMIT: Finalizes the transaction, making all changes permanent.
- ROLLBACK: Undoes the changes made during the transaction, reverting the database to its state before the transaction started.
- Example (ROLLBACK):

```
BEGIN TRANSACTION;

UPDATE Accounts
SET Balance = Balance - 100
WHERE AccountID = 123;

IF <condition>
BEGIN
   ROLLBACK;
END
ELSE
BEGIN
   COMMIT;
END;
```

In this example, the transaction is rolled back if a certain condition is met, ensuring that no changes are made

if an error occurs during the transaction.

8.3. SAVEPOINT :

A SAVEPOINT is a point within a transaction to which you can later roll back. It allows you to create partial rollbacks within a transaction.

- Example :

```
BEGIN TRANSACTION;

UPDATE Accounts
SET BALANCE = Balance - 100
WHERE AccountID = 123;

SAVEPOINT WithdrawlSavepoint;

INSERT INTO Transactions (AccountID, Amou-
    nt, TransactionType)
VALUES (123, 100, 'WithDrawl');

IF <condition>
BEGIN
    ROLLBACK TO WithdrawlSavepoint;
END
ELSE
BEGIN
```

```
    COMMIT;
END;
```

In this example, a SAVEPOINT named 'WithdrawlSavepoint' is created before the INSERT statement. If an error occurs later in the transaction, the ROLLBACK TO statement can be used to undo changes up to the specified savepoint.

8.4. Transaction Exercises:

- Simple Transactions:

Exercise: Create a transaction that deducts $50 from the balance of account with AccountID 456 and inserts a corresponding transaction record.

Example:

```
BEGIN TRANSACTION;

UPDATE Accounts
SET Balance = Balance - 50
WHERE AccountID = 456;

INSERT INTO Transactions (AccountID, Amount,
```

```
TransactionType)
Values (456, 50, 'Withdrawl');

COMMIT;
```

In this exercise, a simple transaction is created to deduct $50 from the balance of the account with AccountID 456 and insert a withdrawl transaction record.

- Exe Deposit Transaction:

Exercise: Create a transaction that add $100 to the balance of account with accountID 789 and inserts a corresponding transaction record.

Example:

```
BEGIN TRANSACTION;

UPDATE Accounts
SET Balance = Balance + 100;
WHERE AccountID = 789;

INSERT INTO Transactions (AccountID, Amount,
        TransactionType)
VALUES (789, 100, 'Deposit');

COMMIT;
```

- Transfer Transaction:

Exercise: Create a transaction that transfers $200 from AccountID 456 to AccountID 789. Ensure that both the deduction from the source account and the addition to the destination account are part of the same transaction.

Example:

```
BEGIN TRANSACTION;

UPDATE Accounts
SET Balance = Balance - 200;
WHERE AccountID = 456;

UPDATE Accounts
SET Balance = Balance + 200
WHERE AccountID = 789;

COMMIT;
```

These exercises provide simple scenarios where transactions are used to modify data in a controlled and constant manner. They involve updating account balances and inserting corresponding transaction records ~~without~~ within a ~~sing~~ single transaction block.

9. Working with Dates and Times

9.1. Date and Time Data Types:

In SQL, various date and time data types are used to store temporal information.

Common types include:

- DATE: Represents a date without a time component (e.g. '2023-11-10').
- TIME: Represents a time without a date component (e.g., '14:30:00').
- DATETIME or TIMESTAMP: Represents both date and time together (eg., '2023-11-10 14:30:00')
- INTERVAL: Represents a duration of time.

Example:

```
CREATE TABLE Events (
    EventID INT PRIMARY KEY;
    EventName Varchar (50);
    EventDate DATE,
    StartTime TIME,
    EventDateTime DATETIME,
```

```
    Duration INTERVAL
);
```

9.2. Date and Time Functions:

SQL provides a variety of functions to manipulate and extract information from date and time values.

Some common functions include:

- CURRENT_DATE: Returns the current date.
- CURRENT_TIME: Returns the current time.
- CURRENT_TIMESTAMP: Returns the current date and time.
- EXTRACT: Extracts components (year, month, day, e.t.c.) from a date or time.
- DATE_PART: Similar to EXTRACT, it extracts specific components.
- DATE_ADD and DATE_SUB: Add or subtract intervals from dates.

Example:

```
SELECT
    CURRENT_DATE AS Today,
    CURRENT_TIME AS CurrentTime,
    CURRENT_TIMESTAMP AS CurrentDateTime,
    EXTRACT (YEAR FROM EventDate) AS EventYear,
    DATE_PART ('month', EventDate) AS EventMonth,
    DATE_ADD (EventDate, INTERVAL '1' DAY) AS
                NextDay,
    DATE_SUB (EventDateTime, INTERVAL '1' HOUR)
        AS PreviousHour
FROM Events
  WHERE EventID = 1;
```

9.3. Date and Time Exercises:

- Manipulating Date and Time Data:

~~Example~~ Exercise: Increase the EventDate of all events in the Events table by one week.

Example:

```
UPDATE Events
SET EventDate = ~~Data~~ DATE_ADD (EventDate,
            INTERVAL '1' WEEK);
```

In this exercise, the 'DATE_ADD' function is used to add one week to the existing EventDate values in the Events table.

- Extracting Components:

Exercise : Extract the year and month from the EventDate - Time column in the Events table.

Example:

```
SELECT
    EventName,
    Extract (YEAR FROM EventDateTime) AS
            EventYear,
    DATE_PART ('month', EventDateTime) AS
            EventMonth
FROM Events;
```

In this challenge, the 'EXTRACT' and 'DATE-PART' functions are used to extract the year and month from the Event Date Time column in the Events Table.

- Calculating Durations:

Exercise: Calculate the duration (in hours) between the StartTime and EndTime of events in the Eventstable.

Example:

[redacted]

```
SELECT
    EventName,
    StartTime,
    EndTime,
    DATE-PART ('hour', EndTime - StartTime)
        AS DurationHours
FROM Events;
```

In this task, the ''DATE-PART' function is used to calculate the duration (in hours) between the StartTime and EndTime columns in the Events table.

10. String Manipulation

10.1. String Functions:

String functions in SQL allow you to manipulate and perform operations on string data.

Some common string functions include:

- CONCAT or || : Concatenates two or more strings.
- LENGTH or LEN : Returns the length of a string.
- UPPER or UCASE : Converts a string to uppercase.
- LOWER or LCASE : Converts a string to lowercase.
- SUBSTRING or SUBSTR : Extracts a substring from a string.
- TRIM : Removes leading and trailing spaces from a string.
- CHARINDEX or POSITION : Returns the position of a substring with a string.

- REPLACE : Replaces occurences of a substring with another substring.

Example:

```
SELECT
    CONCAT ('Hello', ' ', 'World') AS Greeting,
    LENGTH ('SQL') AS LengthOfSQL,
    UPPER ('hello') AS UppercaseHello,
    LOWER ('WORLD') AS LowercaseWorld,
    SUBSTRING ('Database', 1, 4) AS SubstrResult,
    TRIM ('  spaces  ') AS TrimmedString,
    CHARINDEX ('is', 'SQL is Powerful') AS
                PositionOfIs,
    REPLACE ('I love SQL', 'SQL', 'databases')
            AS ReplacedString;
```

10.2. Concatenation:

Concatenation in SQL involves combining two or more strings into a single string. This is often done using the 'CONCAT' function or the '||' operator.

Example:

```
SELECT CONCAT ('Hello', ' ', 'World') AS Greeting;
SELECT 'Hello' || ' ' || 'World' AS Greeting;
```

In these examples, the strings 'Hello', ', and 'World' are concatenated to form the greeting 'Hello World!'

10.3. String Manipulation Exercises:

- **Combining Strings:**

Example Exercise: Combine the first name and last name of employees in the Employees table into a single string.

Example:

```
SELECT
    EmployeeId,
    CONCAT(FirstName, ' ', LastName) AS
        FullName.
FROM Employees;
```

In this exercise, the 'CONCAT' function is used to combine the first name and last name columns of employees into a single string called FullName.

- **Changing Case:**

Exercise: Convert all product names in the Products table to uppercase.

Example:

```
SELECT
    ProductID,
    Upper (ProductName) AS UppercaseProductName
FROM Products;
```

In this exercise, the 'UPPER' function is used to convert all product names to uppercase in the Products table.

- Extracting Substrings:

Exercise: Extract the first three characters from the category names in the Categories table.

Example:

```
SELECT
    CategoryID,
    SUBSTRING (CategoryName, 1, 3) AS
            FirstThreeChars
FROM Categories;
```

In this example, the 'SUBSTRING' function is used to extract the first three characters from the CategoryName Column in the Categories table.

- Replacing Text :

Exercise : Replace the word "Limited" with "Corp" in the CompanyName column of the Suppliers table.

Example :

```
SELECT
    SupplierID,
    REPLACE (CompanyName, 'Limited.',
        'Corp') AS UpdatedCompanyName
FROM Suppliers;
```

In this exercise, the 'REPLACE' function is used to replace occurences of the word "Limited" with "Corp" in the CompanyName column of the Suppliers table.

11. Views and Indexes

11.1. Creating and using views:

In SQL, a view is a virtual table that is based on the result of a SELECT query. It doesn't store the data itself but provides a way to represent the result of a query as if it were a table. Views can be used to simplify complex queries, encapsulate business logic, and provide a security layer by restricting access to certain columns.

Example:

```
CREATE VIEW EmployeeDetails AS
SELECT EmployeeID, FirstName, LastName,
    Department
FROM Employees
WHERE Department = 'IT';

SELECT * FROM EmployeeDetails;
```

In this example, a view named Employee-Details is created to represent employees in the IT department. The view is then used in a query to retrieve information about IT department employees.

11.2. Indexes and Performance:

Indexes in a database are structures that provide a faster way to retrieve data. They work similar to the index in a book, allowing the database engine to quickly locate and access the rows that match a certain condition.

Indexes are particularly beneficial for large tables where searching for data without an index could result in a significant performance hit.

Example:

```
CREATE INDEX idx-ProductName ON
        Products (ProductName);

SELECT * FROM Products WHERE
    ProductName = 'Widget';
```

In this example, an index named idx-ProductName is created on the ProductName column of the Products table. This index improves the performance of queries that involves searching for a specific product name.

11.3. Views and Index Exercises:

• Creating Views:

Exercise: Create a view named "ActiveCustomers" that includes customers who have made a purchase in the last 30 days.

Example:

```
CREATE VIEW ActiveCustomers AS
SELECT CustomerID, FirstName, LastName,
       LastPurchaseDate
FROM Customers
WHERE LastPurchaseDate >= CURRENT-DATE
                -INTERVAL '30' DAY;
```

In this exercise, a view named "ActiveCustomers" is created to represent customers who have made a purchase in the last 30 days.

• Using Views:

Exercise: Use the "ActiveCustomers" view to retrieve the details of customers who made a purchase in the last 30 days.

Example:

```
SELECT * FROM ActiveCustomers;
```

In this exercise, the "ActiveCustomers" view is used in a query to retrieve details of customers who made a ~~purc~~ purchase in the last 30 days.

- Creating Indexes:

Exercise: Create an index named "idx_ProductCategory" on the "Category" column of the "Products" table.

Example:

```
CREATE INDEX idx_ProductCategory ON
    Products (ProductCategory);
```

In this task, an index named "idx_ProductCategory" is created on the "ProductCategory" column of the "Products" table.

- Query Performance:

Example: Measure the performance improvement of a query by creating an index on the "EmployeeID" column in the "Orders" table.

Example :

```
CREATE INDEX idx_EmployeeID ON Orders
  (EmployeeID);

SELECT * FROM Orders WHERE EmployeeID
      = 101 ;
```

In this exercise, an index named "idx_EmployeeID" is created on the "EmployeeID" column of the "Orders" table, and the query performance is measured when using this index.

12. Stored Procedures and Functions:

12.1. Creating and Calling Stored Procedures:

Stored Procedures are precompiled sets of one or more SQL statements that are stored in the database and can be executed multiple times. They help in encapsulating business logic, improving performance, and enhancing security by controlling data access.

Creating a stored Procedure:

```
CREATE PROCEDURE GetEmployeeID
                (IN empID INT)
BEGIN
SELECT * FROM Employees WHERE
          EmployeeID = empID
END;
```

Calling a stored procedure:

```
CALL GetEmployeeID (123);
```

In this example, a stored procedure named "GetEmployeeID" is created to retrieve employee details based on the EmployeeID.

The procedure is then called by providing the EmployeeID as an argument.

12.2. Creating and Calling user-Defined Functions:

User-defined functions (UDFs) are resuable code blocks that perform specific tasks and return a value. They enhance code modularity and readability.

Creating a User-Defined Function:

```
CREATE FUNCTION CalculateTax (price
    DECIMAL (10,2)) RETURNS DECIMAL (10,2)
BEGIN
  DECLARE tax DECIMAL (10,2);
  SET TAX tax = price * 0.1 ;
END
```

Calling a User-Defined Function:

```
SELECT ProductName , Price , CalculateTax
    (Price) AS TaxAmount FROM Products.
```

Here, a function named "CalculateTax" is created to calculate the tax amount based on the product price. It's then invoked

within a SELECT query to calculate tax for each product.

12.3. Stored Procedures and Function Exercises:

- **Creating Stored Procedures:**

Exercise: Create a stored procedure named "GetProductsByCategory" to retrieve products based on a given category.

Example:

```
CREATE PROCEDURE GetProductsByCategory
    (IN CategoryID INT)
BEGIN
  SELECT * FROM Products WHERE CategoryID =
    CategoryID;
END;
```

In this exercise, a stored procedure "GetProductsByCategory" is ~~related~~ created to fetch products based on a specified category ID.

- **Calling stored Procedure:**

Exercise: Call the "GetProductsByCategory"

procedure to retrieve products in the 'Electronics' category.

Example:

```
CALL GetProductsByCategory(1);
```

Here, the "GetProductsByCategory" procedure is invoked with the category ID 1 (assuming 'Electronics' category) to retrieve products falling under that category.

- Creating Functions:

Exercise: Create a function named "CalculateDiscount" to calculate the discounted price based on the original price and discount percentage.

Example:

```
CREATE FUNCTION CalculateDiscount (OriginalPrice
    DECIMAL(10,2), discountPercentage DECIMAL
    (5,2)) RETURNS DECIMAL(10,2)
BEGIN
  DECLARE discount DECIMAL(10,2);
  SET discount = originalPrice * (discountPer-
      centage / 100);
  RETURN originalPrice - discount;
END;
```

In this exercise, the "CalculateDiscount" function is created to compute the discounted price based on the original price and discount percentage provided.

- Using Functions:

Exercise: Utilize the "CalculateDiscount" function to retrieve discounted prices for products in a certain category.

Example:

```
SELECT ProductName, Price, CalculateDiscount
       (Price, 10) AS DiscountedPrice
FROM Products
WHERE CategoryID = 2;
```

In this exercise, the "CalculateDiscount" function is used within a query to calculate discounted prices for products falling under a specific category (assuming category ID 2 correspond to 'Clothing').

13. Advanced Query Techniques

13.1. Common Table Expressions (CTEs):

CTEs are temporary result sets that exist only within the scope of a single SQL statement, They enhance readability and maintainability by allowing the decomposition of complex queries into simpler, more managable parts.

Example:

```
WITH HighValueProducts AS (
    SELECT ProductName, Price
    FROM Products
    WHERE Price > 100
)
SELECT * FROM HighValueProducts;
```

In this example, a CTE named "HighValueProducts" is created to retrieve productis with a price greater than 100. The main query then utilises this CTE to select and display the desired information.

13.2. Window Functions:

Window functions perform calculations across

a set of rows related to the current row. They provide a way to perform aggregate functions without grouping the result set.

Example:

```
SELECT ProductName, Price
    ROW-NUMBER() OVER (ORDER BY Price
        ASC) AS Rank
FROM Products
```

Here, the 'ROW-NUMBER()' window function is used to assign a ranking to each product based on the ascending order of their prices.

13.3. Pivoting Data:

Pivoting involves rotating rows into columns to achieve a different representation of data.

Example:

```
SELECT
    CustomerID,
    SUM (CASE WHEN ProductID = 1 THEN
        Quantity ELSE 0 END) AS Product1Qty,
```

```
SUM ( CASE WHEN ProductID = 2 THEN Quantity
        ELSE 0 END) AS Product2Qty.
FROM Orders
GROUP BY CustomerID ;
```

This query uses 'CASE' statements to pivot data from rows (where each row represents an order) into columns showing the quantities of different products ordered by each customer.

13.4. Advanced Query Techniques Exercises:

- CTE Utilization:

Exercise : Use a CTE to find the total sales amount for each customer.

Example:

```
WITH CustomerSales AS(
    SELECT CustomerID , SUM (Amount ) AS
        TotalSales
    FROM Orders
    GROUP BY CustomerID
)
SELECT * FROM CustomerSales ;
```

In this exercise, a CTE named "CustomerSales" is created to calculate the total sales amount for each customer by aggregating the orders amount from the Orders table.

- Window Function Applications:

Exercise: Use a window function to rank products based on their sales quantity.

Example:

```
SELECT ProductID, Quantity,
    RANK() OVER (ORDER BY Quantity DESC)
        AS SalesRank
FROM OrderDetails;
```

Here, the 'RANK()' window function is employed to assign a ranking to products based on their sales quantity recorded in the OrderDetails table.

- Pivoting Data:

Exercise: Pivot the order quantities of different products for each customer.

Example:

```
SELECT CustomerID,
  SUM (CASE WHEN ProductID = 1 THEN Quanti-
    ty ELSE 0 END) AS Product1Qty,
  SUM (CASE WHEN ProductID = 2 THEN Quantity
    ELSE 0 END) AS Product2Qty.
FROM Orders
GROUP BY CustomerID;
```

This query uses 'CASE' statements to pivot data from rows into columns showing the quantities of different products ordered by each customer from the Orders table.

14. Security and Permissions

14.1. User Authentication and Authorization:

User Aunthication involves verifying the identity of users attempting to access a system. This typically involves username / password validation or other secure ~~aunthe~~ authentication mechanisms.

User Authorization determines the level of access or permissions granted to authenticated users. It defines what actions users can perform within the database, such as read, write or execute specific operations.

14.2. GRANT and REVOKE Statements:

- GRANT : This SQL command is used to grant specific privileges (eg. SELECT, INSERT, UPDATE, DELETE) to users or roles.

Example :

```
GRANT SELECT ON Customers TO
marketing_team;
```

- REVOKE : This command is used to ~~grant~~

revoke previously granted permissions from users or roles.

Example:

```
REVOKE INSERT ON Orders FROM sales-
    role;
```

These statements are crucial for controlling access and managing permissions at a granular level within the database.

14.3. Security Best practices:

- Enforcing strong password policies.
- Regularly updating and patching the database system to address security vulnerabilities.
- Implementing encryption for sensitive data.
- Applying the principle of least privilege, granting users only the permissions they need.

14.4. Security and Permissions Exercises:

- Access Control:

Task: Setting up user access to specific tables or views.

Example:

Suppose we have a database "Sales DB" with tables "Customers" and "Orders". We want to grant access to the "Orders" table for a user named "analyst-user":

```
GRANT SELECT ON Orders TO analyst-user;
```

The query grants the "SELECT" permission on the "Orders" table to the user "analyst-user", allowing them to retrieve data from this table.

- Permission Mamgement:

Task: Granting and revoking permissions for certain operations.

Example:

Let's assume we initially granted the "insert" permission to the "sales-team" role for the "orders" table. Now, we want to revoke this permission.

```
REVOKE INSERT ON Orders FROM
    sales_team;
```

This revokes the "INSERT" permission from the "sales_team" role, restricting their ability to insert records into the "Orders" table.

- Security Configuration:

Task: Implementing security best practices within a database environment.

Example:

Implementing encryption for sensitive data, such as credit card numbers in the "PaymentInfo" table.

```
ALTER TABLE PaymentInfo
  MODIFY COLUMN CreditCardNumber
    VARCHAR(255) ENCRYPTED WITH
    (COLUMN_ENCRYPTION_KEY = Column-
      EncryptionKey, ENCRYPTION_TYPE =
    Deterministic, ALGORITHM = 'AEAD_AES_256_
      CBC_HMAC_SHA_256');
```

This example demonstrates implementing encryption for the "CreditCardNumber" column in the "PaymentInfo" table to enhance security.

15. Database Design and Normalization

15.1. Basics of Database Design:

This section covers fundamental principles and considerations when designing a database.

It includes:

- Identifying entities and their relationships.
- Defining tables, columns, and relationships among them.
- Ensures data integrity through proper constraints and keys. (e.g. primary keys, foreign keys).

15.2. Normalisation:

Normalisation is a systematic approach to organizing data in a database to minimize redundancy and dependency. It involves dividing larger tables into smaller, related tables to reduce data duplication and improve efficiency. It typically follows different normal forms (e.g. 1NF, 2NF, 3NF).

Example:

Consider a table that stores both customers details and their orders. Normalization would involve splitting this into separate tables for customers and orders to eliminate redundancy.

15.3. Denormalization:

Denormalization is the process of intentionally introducing redundancy into a database design to improve query performance by reducing the need for joins or complex operations. It's often used in read-heavy or analytical systems.

Example:

In a highly accessed reporting database, combining multiple normalised tables into a single-table might be considered for quicker data retrieval, despite introducing some redundancy.

15.4. Database Design and Normalisation Exercises:

- Database Design tasks:

Example : Identifying entities and defining tables.

```
CREATE TABLE Products (
        ProductId INT PRIMARY KEY,
        ProductName VARCHAR (50),
        PRICE DECIMAL (10,2),
        CategoryID INT
);

CREATE TABLE Categories (
        CategoryID INT PRIMARY KEY,
        CategoryName VARCHAR(50)
);

ALTER TABLE Products
ADD CONSTRAINT FK_CategoryID FOREIGN
    KEY (CategoryID) REFERENCES Categories
      (CategoryID);
```

- Normalisation Exercises :

Example : Splitting a denormalised table into normalized tables.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    EmployeeName VARCHAR(50),
    Department VARCHAR (50),
```

```
    Salary DECIMAL (10, 2)
);

CREATE TABLE Departments (
        DepartmentID PRIMARY KEY,
        DepartmentName VARCHAR (50)
);

CREATE TABLE NormalizedEmployees (
        EmployeeID INT PRIMARY KEY,
        EmployeeName VARCHAR (50),
        DepartmentID INT,
        Salary DECIMAL (10, 2)
);

ALTER TABLE NormalizedEmployees
ADD CONSTRAINT FK_departmentID FOREIGN
    KEY (DepartmentID) REFERENCES
            Departments (DepartmentID);
```

- Denormalization Exercises:

Example: Creating a view for simplified reporting.

```
CREATE VIEW EmployeeDetails AS
    SELECT Employees.EmployeeID, Employees.
        EmployeeName, Departments.Department-
```

```
Name
FROM Employees
JOIN Departments ON Employees.Depart-
mentID = Departments.DepartmentID
```

These ~~cons~~ concise examples showcase creating tables, normalizing data to eliminate redundancy, and denormalizing for better query performance or reporting purposes within a database environment.

16. Advanced Topics in SQL

16.1. Working with XML and JSON Data:

- XML Handling : SQL provides functions and methods to parse, extract, and manipulate XML data stored within the database.

Example:

```
SELECT xmlcolumn.value('(*/root/element)[1]',
       'nvarchar(max)') AS ExtractedValue
FROM xmldata;
```

- JSON Support : SQL allows processing JSON data, offering functions to extract, query, and modify JSON objects.

Example:

```
SELECT jsoncolumn ->> '$.key' AS Value
FROM jsontable;
```

16.2. Geospatial Data and GIS:

SQL offers capabilities to handle geospatial data like points, lines, polygons, e.t.c. and perform spatial operations like

distance calculations, intersections e.t.c.

Example:

```
SELECT ST_DISTANCE(PointA, PointB) AS
    Distance
FROM Locations;
```

16.3. Full-Text Search:

Full-Text search enables efficient searching of text within large text-based columns by indexing words and phrases.

Example:

```
SELECT *
FROM Articles
WHERE CONTAINS(Content, 'search keyword');
```

16.4. Advanced Topics Exercises:

- XML and JSON manipulation:

Task: Querying, extracting, or modifying XML/JSON data.

Example : XML manipulation

```
SELECT xmlcolumn.value ('(//book/title)[1]',
       'nvarchar(max)') AS BookTitle
FROM books_table;
```

Example : JSON manipulation

```
SELECT jsoncolumn ->> '$.key' AS value
FROM jsontable;
```

These examples demonstrate querying XML and JSON data stored in respective columns within tables.

- Geospatial Data - Handling :

Task : Performing spatial operations and queries on geospatial data.

Example : Geospatial Query

```
SELECT LocationName
FROM Locations
WHERE ST_DISTANCE (PointA, PointB) < 1000;
```

This example showcases querying to find locations within a certain distance, utilizing geospatial functions like 'ST_DISTANCE'.

- FULL-Text Search Practice:

Exercise: Constructing queries for efficient text-based searches in SQL.

Example: Full-text search Query.

Let's consider a table 'ProductDescriptions' containing descriptions of various products. We aim to search for products containing specific keywords within their descriptions.

```
SELECT ProductName, Description
FROM ProductDescriptions
WHERE CONTAINS(Description, 'smartphone
                OR camera');
```

This query retrieves the product names and descriptions for items that match the specified keywords using the 'CONTAINS' function for a more refined and targetted search.

17. SQL in NoSQL Databases

17.1. SQL in NoSQL Databases:

- Use of SQL and NoSQL Systems: This section illustrates how some NoSQL databases support SQL-like querying or interfaces to facilitate querying data using SQL-like syntax.

- Examples:

  - In MongoDB, libraries like MongoDB Atlas Data lake or connectors like MongoDB connector for BI allow SQL queries to access and analyze MongoDB data.

  - SQLite, while traditionally an SQL database, provides an interface and support for NoSQL-style JSON data storage and retrieval.

17.2. Bridging the Gap between SQL and NoSQL:

- Compatibility and Translation: This part addresses the challenges and solutions involved in translation of SQL queries to NoSQL databases or providing interfaces that allow the usage of SQL-

like queries in NoSQL systems.

- Usage Scenarios :

  - Libraries or connectors that act as bridges, allowing SQL-based applications to interact with NoSQL databases.

  - Techniques to convert SQL statements into NoSQL-compatible queries for data retrieval.

17.3. SQL in NoSQL database Exercises:

Exercise 1 : Writing SQL-like queries in MongoDB Atlas Data lake.

MongoDB Atlas Data lake allows querying data residing in cloud object storage using SQL-based expressions, particularly when handling JSON documents.

Example : SQL-like Query in MongoDB Atlas Data lake.

Suppose we have JSON documents representing product information stored in cloud storage.

[redacted]

```
SELECT product_id , product_name , price
FROM Products
WHERE price > 100
```

This SQL-like query accesses and retrieves product information from the 'products' collection stored in the MongoDB Atlas Data lake, filtering products with prices greater than 100.

Exercise 2 : Using SQLite's support for NoSQL JSON storage.

SQLite offers support for NoSQL-style JSON storage and querying of JSON documents within an SQL environment.

Example : SQL query on JSON data in SQLite.

Suppose we have a SQLite table 'employees' with a JSON column 'employee_data':

```
SELECT employee_data ->> '$.name',
       employee_data ->> '$.department'
FROM employees
WHERE employee_data ->> '$.age' > '30'
```

This SQL queries demonstrates extracting specific fields from JSON documents stored in the 'employee_data' column within the 'employees' table in SQLite, filtering employees based on their age.

These exercises simulate the usage of SQL-like querying with NoSQL environments such as MongoDB Atlas Data Lake and SQLite, where SQL queries interact with and retrieve data from NoSQL-stored JSON documents.

18. Case Studies and Best Practices

18.1. Real-world Case Studies:

This section involves real-world examples showcasing how SQL was effectively employed to solve complex problems or improve operations in various domains like finance, healthcare, e-commerce e.t.c.

Example:

- Case study demonstrating how SQL was used to optimize inventory management in an e-commerce platform, reducing redundant stock and enhancing order processing.

18.2. Best Practices in SQL:

- Optimization Techniques: This part focuses on SQL coding best practices, emphasizing efficient querying, index usage, schema design, and other strategies to enhance SQL performance.

Examples:

1. Proper use of indexes:
Employing indexes judiciously based on query patterns and data access frequencies to

expedite data retrieval and query execution.

2. * Avoiding SELECT:
Avoiding the use of SELECT * in queries to reduce unnecessary data transfer and improve performance. Explicitly specifying required columns enhances efficiency.

3. Normalization for Reducing Redundancy:
Organizing database tables through normalization to eliminate redundant data and improve data integrity. Normalization minimizes data redundancy, reducing storage requirements and ensuring consistency.

18.3. Performance Tuning:

Improving Efficiency: Discusses techniques and methodologies aimed at enhancing the performance of SQL databases through query optimization, indexing, database design improvements, and resource utilization.

Example:

- Analyzing query execution plans to identify bottlenecks.

- Using proper indexing strategies based on query patterns.
- Optimizing database configurations for memory allocation and caching.

18.4. Case Studies and Best Practices Examples:

- **Query Optimization task:**

Exercise: Optimizing Query Performance

Problem Statement:
Given a large database table with millions of of records, participants are provided with slow-performing query and are tasked with optimizing it for better performance.

```
SELECT *
FROM SalesData
  WHERE Date BETWEEN '2023-01-01'
                 AND '2023-12-31'
  AND ProductID = 100
ORDER BY SaleAmount DESC;
```

Optimized Query:

```
SELECT SaleDate , SaleAmount
FROM SalesData
WHERE Date >= '2023-01-01'
      AND DATE <= '2023-12-31'
   AND ProductID = 100
ORDER BY SaleAmount DESC ;
```

- ## Schema Design Exercise:

Exercise : Normalizing Database Tables

Problem Statement:
Participants are given a denormalized database schema with repeating groups and are asked to normalize it to eliminate redundancy and improve data integrity.

Original Schema:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    CustomerName VARCHAR(100),
    ProductID INT,
    ProductName VARCHAR(100),
    OrderDate DATE,
```

```
    Quantity INT,
    ...
);
```

Normalized Schema:

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    CustomerName VARCHAR(100),
    ...
);

CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100),
    ...
);

CREATE TABLE ORDERS (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    ProductID INT,
    OrderDate DATE,
    Quantity INT,
    FOREIGN KEY (CustomerID) REFERENCES
        Customers (CustomerID),
    FOREIGN KEY (ProductID) REFERENCES
        Products (ProductID),
    ...
);
```

- Index usage Scenerio:

Exercise : Identifying Indexing Strategies

Problem Statement:
Participants are presented with various query patterns and are asked to identify appropriate columns for indexing to improve query performance.

Query Example:

```
SELECT ProductName , Price
FROM Products
WHERE Category = 'Electronics'
ORDER BY Price ;
```

Recommended Index:

```
CREATE INDEX idx_category_Price ON
        Products ( Category , Price );
```

19. Future Trends in SQL

19.1. Emerging Trends

- Overview: This section explores novel paradigms in SQL technology, such as SQL over Data Lakes and SQL-on-Hadoop, which aim to integrate SQL querying capabilities with newer data storage systems and frameworks.

- Purpose: These emerging trends aim to bridge the gap between traditional SQL databases and modern big data technologies.

For instance, SQL over Data Lakes facilitates querying data stored in distributed and unstructured environments like data lakes using familiar SQL syntax.

19.2. AI and SQL:

- Overview: This part talks about how cool two technologies, Artificial intelligence (AI) and SQL are coming together.
It explores how we can use AI in SQL databases to make them even better at analyzing data and helping us to make decisions.

Purpose: AI integration in SQL allows for more intelligent query optimization, predictive analytics, and automated decision support systems within database management systems. This combination enables more efficient and data-driven decision-making process.

In simpler words, it's like giving our database a brain boost with AI, so they become super helpful in understanding and using data.

19.3. The future of SQL:

Overview: Envisions the future trajectory of SQL technology, highlighting potential advancements, innovations and shifts in the SQL database / landscape.

Purpose: Discusses how SQL is expected to evolve further to accomodate the increasing complexity and scale of data. It might cover topics like enhanced querying capabilities, support for new data types, improvements in parallel processing, and advancements in user interfaces for database interaction in detail.

In simple words, it's like imagining a super-smart and upgraded version of the language we use to talk to databases.

This upgrade helps it handle bigger and trickier data while making it easier for people to use.

20. Conclusion and Further Learning

20.1. Recap of key Concepts:

This section revisits and summarizes the main ideas and important points covered in the entire book on SQL.

Here's a detailed breakdown of what it might involve:

1. Basic SQL concepts: This section would summarize foundational SQL concepts like querying databases, retrieving data, modifying data (inserting, updating, deleting) and understanding how databases are organized.

2. Data manipulation techniques: It might cover key commands or statements for manipulating data, such as using SELECT statements, filtering data with WHERE clauses, and managing transactions (ensuring the consistency of database changes.)

3. Data Retrieval and Aggregation: Highlighting methods for retrieving specific data, using aggregate functions (like SUM, AVG) and grouping data for analysis

using the GROUP BY clause.

4. Advanced SQL Techniques: Recap of advanced topics like subqueries, joins (retrieving data from multiple tables), working with dates, string manipulation, view, indexes, stored procedures / functions, and optimizing queries.

5. Database Design and normalisation: Summarizing the basics of database design, normalisation, and best practices for structuring databases to minimize redundancy and ensure efficient data storage.

6. Security and Permissions: Reviewing concepts related to user authentication, authorization, implementing security measures (like GRANT and REVOKE statements), and ensuring data protection.

7. Future Trends and Further learning: Briefly touching upon emerging trends in SQL technology, the integration of AI, and suggesting additional resources for further learning.

20.2. Further learning Resources:

[redacted]

Here's a detained breakdown:

1. Books and Publications: Recommending additional books or publications that cover SQL in more depth, exploring advanced topics, specific database systems, or specialised areas like optimization,

2. Online Courses and Tutorials: Suggesting online learning platforms or websites that offer comprehensive SQL courses or tutorials. These resources often provide interactive learning experiences, exercises, and a real-world projects to enhance practical skills.

3. Educational Websites and Forums: Directing readers to educational websites, forums or communities dedicated to SQL. These platforms might offer articles, tutorials, forums for discussions, and Q & A sessions when users can engage with experts and peers.

4. Certification Programs: Recommending recognised certification programs for SQL. These certifications can validate expertise and proficiency in SQL and database management enhancing ~~carreer~~ career opportunities.

5. Documentation and Official Resources: Pointing towards official documentation and resources provided by database management system vendors. For instance, Oracle, MySQL, or Microsoft SQL Server offer extensive documentation with guides, tutorials and examples.

6. Blogs and Industry Publications: Highlighting blogs, articles, or publications in the field of SQL and database management. These sources might offer insights into current trends, best practices, and real-world applications.

7. Video Tutorials and Webinars: Suggesting video-based learning resources or webinars that visually demonstrates SQL concepts, query demonstrations, and database management techniques.

20.3. Preparing for SQL Certification:

This part might provide guidance on how to get ready for official certifications in SQL. It could include tips, study guides, or recommended practices to help you prepare and pass certification exams related to SQL.

- What is SQL certification?

It's like getting a special badge or certificate that shows you really know a lot about SQL. It's a way to prove to employers or others that you're really good at working with databases.

- How to Prepare?

→ Study SQL
→ Practice, Practice, Practice
→ Review Materials.
→ Take Practice Exams
→ Explore official Resources.

- Why get certified?

→ Better Opportunities: Having an SQL certification can open doors to better job opportunities in database-related fields.

→ Proving Skills: It shows that you're skilled and knowledgeable in SQL, making you stand out to employeers.

→ Career growth: It can lead to promotions or better-paying jobs as companies often prefer certified professionals.

Preparing for an SQL certification is like studying hard for an important test.

It shows that you're serious about your skills in SQL and can help you move forward in your career.