**Assignment on 07-08-2024**

**Assignment 1:** Write a SELECT query to retrieve all columns from a 'customers' table, and f f fi \_x000D\_# Confidential - RPS Data modify it to return only the customer name and email address for customers in a specific city.

Certainly! Here is an example of how to write the SELECT query to retrieve all columns from a customers table:

-- Retrieve all columns from the 'customers' table

SELECT \* FROM customers;

To modify this query to return only the customer name and email address for customers in a specific city (let's say the city is "New York"), you can use the following query:

-- Retrieve only the customer name and email address for customers in a specific city

SELECT CustomerName, Email

FROM customers

WHERE City = 'New York';

**Explanation**

* **SELECT CustomerName, Email**: This part specifies the columns you want to retrieve. In this case, only the CustomerName and Email columns.
* **FROM customers**: This specifies the table from which to retrieve the data.
* **WHERE City = 'New York'**: This condition filters the results to include only customers from the specified city, "New York".

You can replace 'New York' with any other city name to filter the results accordingly. If your actual column names differ (e.g., customer\_name instead of CustomerName), be sure to use the correct names as per your table schema.

4o

**Assignment 2:** Craft a query using an INNER JOIN to combine 'orders' and 'customers' tables for customers in a specified region, and a LEFT JOIN to display all customers including those without orders.

**Combining orders and customers Tables using INNER JOIN and Filtering by Region**

To combine the orders and customers tables for customers in a specified region using an INNER JOIN, you would write a query like this:

-- INNER JOIN to combine orders and customers for customers in a specified region

SELECT customers.CustomerID, customers.CustomerName, customers.Email, orders.OrderID, orders.OrderDate

FROM customers

INNER JOIN orders ON customers.CustomerID = orders.CustomerID

WHERE customers.Region = 'specified\_region';

In this query:

* **LEFT JOIN orders ON customers.CustomerID = orders.CustomerID**: This part joins the customers table with the orders table but includes all customers from the customers table, even if they do not have matching entries in the orders table. For customers without orders, the OrderID and OrderDate columns will be NULL.

**Combining both Queries**

If you want to display all customers in a specified region including those without orders, you can combine the conditions as follows:

-- LEFT JOIN to display all customers in a specified region including those without orders

SELECT customers.CustomerID, customers.CustomerName, customers.Email, orders.OrderID, orders.OrderDate

FROM customers

LEFT JOIN orders ON customers.CustomerID = orders.CustomerID

WHERE customers.Region = 'specified\_region';

This query:

* Uses a LEFT JOIN to ensure all customers from the specified region are included.
* Filters by the specified region using the WHERE clause.

**Example with a Specific Region**

Let's assume the specified region is 'North America'. Here is the combined query:

-- LEFT JOIN to display all customers in 'North America' including those without orders

SELECT customers.CustomerID, customers.CustomerName, customers.Email, orders.OrderID, orders.OrderDate

FROM customers

LEFT JOIN orders ON customers.CustomerID = orders.CustomerID

WHERE customers.Region = 'North America';

This query will list all customers in 'North America' and their respective orders, including those customers who have no orders.

**Assignment 3:** Utilize a subquery to find customers who have placed orders above the average order value, and write a UNION query to combine two SELECT statements with the same number of columns.

**Subquery to Find Customers Who Have Placed Orders Above the Average Order Value**

To find customers who have placed orders above the average order value, we can use a subquery to first calculate the average order value and then use it to filter the customers.

Assuming the orders table has a column OrderAmount, here is how you can achieve this:

-- Subquery to find customers who have placed orders above the average order value

SELECT customers.CustomerID, customers.CustomerName, customers.Email

FROM customers

WHERE customers.CustomerID IN (

SELECT orders.CustomerID

FROM orders

WHERE orders.OrderAmount > (

SELECT AVG(OrderAmount)

FROM orders

)

);

**UNION Query to Combine Two SELECT Statements with the Same Number of Columns**

A UNION query combines the results of two or more SELECT statements. Each SELECT statement within the UNION must have the same number of columns in the result sets with similar data types.

Assuming we have two queries that fetch customer data from different regions, here is how you can use UNION to combine them:

-- UNION query to combine two SELECT statements with the same number of columns

SELECT CustomerID, CustomerName, Email, 'Region1' AS Region

FROM customers

WHERE Region = 'Region1'

UNION

SELECT CustomerID, CustomerName, Email, 'Region2' AS Region

FROM customers

WHERE Region = 'Region2';

In this example:

* The first SELECT statement retrieves customer data from 'Region1'.
* The second SELECT statement retrieves customer data from 'Region2'.
* Both SELECT statements have the same columns: CustomerID, CustomerName, Email, and a constant value Region indicating the region.
* The UNION keyword combines the results of both queries into a single result set.

By default, UNION removes duplicate rows. If you want to include duplicates, you can use UNION ALL.

**Example Using UNION ALL**

-- UNION ALL query to include duplicates

SELECT CustomerID, CustomerName, Email, 'Region1' AS Region

FROM customers

WHERE Region = 'Region1'

UNION ALL

SELECT CustomerID, CustomerName, Email, 'Region2' AS Region

FROM customers

WHERE Region = 'Region2';

This query works similarly to the previous example but includes all rows, even if there are duplicates.

**Assignment 4:** Compose SQL statements to BEGIN a transaction, INSERT a new record into the 'orders' table, COMMIT the transaction, then UPDATE the 'products' table, and ROLLBACK the transaction.

**SQL Statements for Transactions**

Here is a sequence of SQL statements that demonstrate how to use transactions to manage operations on the orders and products tables.

**Step 1: Begin a Transaction and Insert a New Record into the orders Table**

-- Begin the transaction

BEGIN TRANSACTION;

-- Insert a new record into the orders table

INSERT INTO orders (OrderID, CustomerID, OrderDate, OrderAmount)

VALUES (1, 123, '2024-08-05', 250.00);

-- Commit the transaction

COMMIT;

Step 2: Begin a New Transaction, Update the products Table, and Rollback the Transaction

-- Begin the transaction

BEGIN TRANSACTION;

-- Update the products table

UPDATE products

SET StockQuantity = StockQuantity - 10

WHERE ProductID = 456;

-- Rollback the transaction

ROLLBACK;

**SQL Statements for Transactions**

Here is a sequence of SQL statements that demonstrate how to use transactions to manage operations on the orders and products tables.

**Step 1: Begin a Transaction and Insert a New Record into the orders Table**

-- Begin the transaction

BEGIN TRANSACTION;

-- Insert a new record into the orders table

INSERT INTO orders (OrderID, CustomerID, OrderDate, OrderAmount)

VALUES (1, 123, '2024-08-05', 250.00);

-- Commit the transaction

COMMIT;

**Step 2: Begin a New Transaction, Update the products Table, and Rollback the Transaction**

-- Begin the transaction

BEGIN TRANSACTION;

-- Update the products table

UPDATE products

SET StockQuantity = StockQuantity - 10

WHERE ProductID = 456;

-- Rollback the transaction

ROLLBACK;

**Explanation**

1. **BEGIN TRANSACTION**:
   * This starts a new transaction. All subsequent SQL statements will be part of this transaction until a COMMIT or ROLLBACK is issued.
2. **INSERT INTO orders**:
   * This statement inserts a new record into the orders table with the specified OrderID, CustomerID, OrderDate, and OrderAmount.
3. **COMMIT**:
   * This saves all changes made in the current transaction to the database. The transaction is successfully completed, and all changes are now permanent.
4. **BEGIN TRANSACTION**:
   * This starts another new transaction.
5. **UPDATE products**:
   * This statement updates the StockQuantity of a product in the products table where the ProductID is 456.
6. **ROLLBACK**:
   * This undoes all changes made in the current transaction. None of the updates made after the BEGIN TRANSACTION statement will be saved to the database.

**Example Scenario**

1. **Insert a New Order**:
   * An order is placed, and the details are inserted into the orders table. This action is committed, making it a permanent change in the database.
2. **Update Product Stock**:
   * An attempt is made to update the stock quantity of a product. However, for some reason (e.g., a condition not being met or an error occurring), the changes need to be rolled back. The ROLLBACK statement ensures that the stock quantity update is not saved, and the database remains unchanged from the start of this second transaction.

**Assignment 5:** Begin a transaction, perform a series of INSERTs into 'orders', setting a SAVEPOINT after each, rollback to the second SAVEPOINT, and COMMIT the overall transaction.

**Transaction with SAVEPOINT, ROLLBACK, and COMMIT**

Here is a sequence of SQL statements that demonstrate how to use transactions with SAVEPOINTs to manage operations on the orders table. We'll insert multiple records, set SAVEPOINTs, rollback to a specific SAVEPOINT, and then commit the overall transaction.

**Step-by-Step Example**

-- Begin the transaction

BEGIN TRANSACTION;

-- Perform the first INSERT and set a SAVEPOINT

INSERT INTO orders (OrderID, CustomerID, OrderDate, OrderAmount)

VALUES (1, 123, '2024-08-05', 250.00);

SAVEPOINT sp1;

-- Perform the second INSERT and set a SAVEPOINT

INSERT INTO orders (OrderID, CustomerID, OrderDate, OrderAmount)

VALUES (2, 124, '2024-08-06', 150.00);

SAVEPOINT sp2;

-- Perform the third INSERT and set a SAVEPOINT

INSERT INTO orders (OrderID, CustomerID, OrderDate, OrderAmount)

VALUES (3, 125, '2024-08-07', 300.00);

SAVEPOINT sp3;

-- Rollback to the second SAVEPOINT

ROLLBACK TO SAVEPOINT sp2;

-- Perform another INSERT after the rollback

INSERT INTO orders (OrderID, CustomerID, OrderDate, OrderAmount)

VALUES (4, 126, '2024-08-08', 200.00);

-- Commit the overall transaction

COMMIT;

**Explanation**

1. **BEGIN TRANSACTION**:
   * Starts a new transaction. All subsequent operations are part of this transaction until a COMMIT or ROLLBACK is issued.
2. **First INSERT and SAVEPOINT**:
   * Inserts a record into the orders table.
   * Sets a SAVEPOINT named sp1. This SAVEPOINT marks the point in the transaction to which you can roll back.
3. **Second INSERT and SAVEPOINT**:
   * Inserts another record into the orders table.
   * Sets a SAVEPOINT named sp2.
4. **Third INSERT and SAVEPOINT**:
   * Inserts a third record into the orders table.
   * Sets a SAVEPOINT named sp3.
5. **ROLLBACK TO SAVEPOINT sp2**:
   * Rolls back the transaction to the state at SAVEPOINT sp2. This undoes the third INSERT operation but retains the changes made before sp2.
6. **Fourth INSERT**:
   * Inserts another record into the orders table after rolling back to SAVEPOINT sp2.
7. **COMMIT**:
   * Commits the overall transaction, making all changes from the beginning of the transaction up to the current point permanent. This includes the first, second, and fourth INSERT operations, but not the third since it was rolled back.

Using SAVEPOINTs allows fine-grained control over transactions, enabling partial rollbacks within a larger transaction. This can be very useful for complex operations where some changes may need to be undone without affecting the entire transaction.

**Assignment 6:** Draft a brief report on the use of transaction logs for data recovery and create a hypothetical scenario where a transaction log is instrumental in data recovery after an unexpected shutdown.

**Report on the Use of Transaction Logs for Data Recovery**

**Introduction**

Transaction logs are essential components of modern database management systems (DBMS). They keep a record of all transactions and database modifications to ensure data integrity and provide mechanisms for data recovery in case of system failures, unexpected shutdowns, or other disasters. This report outlines the importance of transaction logs in data recovery and presents a hypothetical scenario demonstrating their practical application.

**Purpose of Transaction Logs**

Transaction logs serve several critical functions in a DBMS:

1. **Data Integrity**: They ensure that all transactions adhere to the ACID (Atomicity, Consistency, Isolation, Durability) properties.
2. **Recovery Mechanism**: In case of a system crash or failure, transaction logs enable the recovery of lost or corrupted data.
3. **Audit Trail**: They provide a history of all transactions, which is useful for auditing and tracking changes.

**Structure of Transaction Logs**

A transaction log typically contains the following information:

* **Transaction ID**: A unique identifier for each transaction.
* **Timestamp**: The time when the transaction was logged.
* **Operation Type**: The type of database operation (INSERT, UPDATE, DELETE).
* **Affected Data**: Details of the data that was modified.
* **Before and After Images**: The state of the data before and after the operation.
* **Commit or Rollback Information**: Status indicating whether the transaction was committed or rolled back.

**Data Recovery Process Using Transaction Logs**

1. **Crash Recovery**: Upon restarting after a crash, the DBMS uses the transaction log to identify incomplete transactions. It can roll back these transactions to maintain data consistency.
2. **Point-in-Time Recovery**: Administrators can use transaction logs to restore the database to a specific point in time, which is crucial in scenarios involving accidental data corruption or deletion.
3. **Redo and Undo Operations**: The transaction log allows the DBMS to redo completed transactions and undo incomplete or erroneous transactions during recovery.

**Hypothetical Scenario: Data Recovery After an Unexpected Shutdown**

**Scenario Description**

A retail company uses a database to manage its sales transactions. One day, the database server experiences an unexpected shutdown due to a power failure. At the time of the shutdown, several sales transactions were being processed. When the system is restarted, it is discovered that some transactions were incomplete, and the database state is inconsistent.

**Recovery Using Transaction Logs**

1. **System Restart and Log Analysis**: Upon restarting the database server, the DBMS immediately begins analyzing the transaction log. It identifies transactions that were in progress at the time of the shutdown.
2. **Identifying Incomplete Transactions**: The log reveals three transactions that were incomplete. The DBMS finds entries indicating the operations that were performed for each transaction.
3. **Undo Operations**: The DBMS uses the transaction log to undo changes made by the incomplete transactions. This involves reversing any partial updates to restore the data to its consistent state before the transactions began.
4. **Redo Operations**: For transactions that were completed but not yet written to the database due to the shutdown, the DBMS re-applies these transactions using the transaction log to ensure all committed changes are reflected in the database.
5. **Point-in-Time Recovery (Optional)**: If the database administrators want to restore the database to a state just before the shutdown, they can use the transaction log to perform a point-in-time recovery. This involves rolling back all transactions that occurred after a specific timestamp, ensuring that the database reflects the state before the power failure.

**Outcome**

Thanks to the transaction log, the retail company’s database is successfully recovered to a consistent state. All completed transactions are retained, incomplete transactions are rolled back, and the database is ready for normal operations with minimal data loss.

**Conclusion**

Transaction logs are invaluable tools for maintaining data integrity and facilitating data recovery in modern database systems. They provide a detailed record of all transactions, enabling efficient recovery from system failures, ensuring data consistency, and supporting auditing and troubleshooting efforts. The hypothetical scenario demonstrates how transaction logs can be instrumental in recovering from unexpected shutdowns, highlighting their critical role in robust database management.