--Part One: Query Problems Using the “alanparadise/cm” repo

----====================================================----

--1

-- List the names of the cities in alphabetical order where Classic Models has offices. (7)

SELECT DISTINCT city

FROM "alanparadise/cm"."offices"

order by 1;

--2 List the EmployeeNumber, LastName, FirstName, Extension for all employees working out of

the Paris office. (5)

select EmployeeNumber, LastName, FirstName, Extension

from "alanparadise/cm"."employees"

where officecode = 4;

--3

--List the ProductCode, ProductName, ProductVendor, QuantityInStock and ProductLine for all

products with a QuantityInStock between 200 and 1200. (11)

select ProductCode, ProductName, ProductVendor, QuantityInStock, ProductLine

from "alanparadise/cm"."products"

where QuantityInStock between 200 and 1200;

--4

-- (Use a SUBQUERY) List the ProductCode, ProductName, ProductVendor, BuyPrice and MSRP for

the least expensive (lowest MSRP) product sold by ClassicModels. (“MSRP” is the

Manufacturer’s Suggested Retail Price.) (1)

select ProductCode, ProductName, ProductVendor, BuyPrice, MSRP from

"alanparadise/cm"."products" order by 5 asc limit (select 1);

--5

-- What is the ProductName and Profit of the product that has the highest profit (profit =

MSRP minus BuyPrice). (1)

select ProductName, (MSRP - BuyPrice) as "profit"

from "alanparadise/cm"."products"

order by 2 desc limit 1;

--6

-- List the country and the number of customers from that country for all countries having

just two customers. List the countries sorted in ascending alphabetical order. Title the

column heading for the count of customers as “Customers”. (7)

select country, count(customernumber) as “Customers”

from "alanparadise/cm"."Customers"

group by 1

having count(customernumber) = 2

order by 1 asc;

--7

-- List the ProductCode, ProductName, and number of orders for the products with exactly 25

orders. Title the column heading for the count of orders as “OrderCount”. (12)

select o.ProductCode, p.ProductName, count(orderlinenumber) as “OrderCount”

from "alanparadise/cm"."orderdetails" as o

join "alanparadise/cm"."products" as p on o.ProductCode = p.ProductCode

group by 1, 2

having count(orderlinenumber) = 25;

--8

-- List the EmployeeNumber, Firstname + Lastname (concatenated into one column in the answer

set, separated by a blank and referred to as ‘name’) for all the employees reporting to Diane

Murphy or Gerard Bondur. (8)

select EmployeeNumber, Firstname || ' ' || Lastname as ‘name’

from "alanparadise/cm"."employees";

--9

-- List the EmployeeNumber, LastName, FirstName of the president of the company (the one

employee with no boss.) (1)

select EmployeeNumber, Lastname, Firstname from "alanparadise/cm"."employees" where reportsto

is null;

--10

-- List the ProductName for all products in the “Classic Cars” product line from the

1950’s. (6)

select ProductName from "alanparadise/cm"."products"

where ProductName like('%195%') and productline = 'Classic Cars';

--10

-- List the ProductName for all products in the “Classic Cars” product line from the

1950’s. (6)

select ProductName

from "alanparadise/cm"."products"

where ProductName like('%195%') and productline = 'Classic Cars';

--11

-- List the month name and the total number of orders for the month in 2004 in which

ClassicModels customers placed the most orders. (1)

--Working to find date

SELECT orderdate AS ordermonth, count(ordernumber)

FROM "alanparadise/cm"."orders"

WHERE orderdate like('%2004%')

GROUP BY ordermonth ORDER BY 2 desc limit 1;

--Used that month

SELECT 'November' AS ordermonth, count(ordernumber)

FROM "alanparadise/cm"."orders"

WHERE orderdate like('%2004%')

GROUP BY ordermonth ORDER BY 2 desc limit 1;

-- 12

-- List the firstname, lastname of employees who are Sales Reps who have no assigned

customers. (2)

select distinct e.firstname, e.lastname

from "alanparadise/cm"."Customers" as c,

"alanparadise/cm"."employees" as e

where c.salesrepemployeenumber is null

and e.jobtitle = 'Sales Rep';

-- 13

-- List the customername of customers from Switzerland with no orders. (2)

select customername

from "alanparadise/cm"."Customers" as c

left join "alanparadise/cm"."orders" o on c.customernumber = o.customernumber

where o.customernumber is null

and c.country = 'Switzerland';

--14

-- List the customername and total quantity of products ordered for customers who have

ordered more than 1650 products across all their orders. (8)

select distinct c.customername, sum(d.quantityOrdered) as totalquantity

from "alanparadise/cm"."Customers" as c

left join "alanparadise/cm"."orders" as o

on c.customerNumber = o.customerNumber

left join "alanparadise/cm"."orderdetails" as d

on o.ordernumber = d.ordernumber

group by c.customername

having sum(d.quantityOrdered) > 1650;

--Completed Part One--

--==================================================--

To create the database based on the provided queries, you need to execute the following SQL statements:

CREATE DATABASE alanparadise;

-- Switch to the newly created database

\c alanparadise;

-- Create the necessary tables

CREATE TABLE offices (

officeCode INT PRIMARY KEY,

city VARCHAR(50) NOT NULL,

phone VARCHAR(50) NOT NULL,

addressLine1 VARCHAR(50) NOT NULL,

addressLine2 VARCHAR(50),

state VARCHAR(50),

country VARCHAR(50) NOT NULL,

postalCode VARCHAR(15) NOT NULL,

territory VARCHAR(10) NOT NULL

);

CREATE TABLE employees (

employeeNumber INT PRIMARY KEY,

lastName VARCHAR(50) NOT NULL,

firstName VARCHAR(50) NOT NULL,

extension VARCHAR(10) NOT NULL,

email VARCHAR(100) NOT NULL,

officeCode INT NOT NULL,

reportsTo INT,

jobTitle VARCHAR(50) NOT NULL,

FOREIGN KEY (officeCode) REFERENCES offices (officeCode),

FOREIGN KEY (reportsTo) REFERENCES employees (employeeNumber)

);

CREATE TABLE products (

productCode VARCHAR(15) PRIMARY KEY,

productName VARCHAR(70) NOT NULL,

productLine VARCHAR(50) NOT NULL,

productScale VARCHAR(10) NOT NULL,

productVendor VARCHAR(50) NOT NULL,

productDescription TEXT NOT NULL,

quantityInStock INT NOT NULL,

buyPrice DECIMAL(10,2) NOT NULL,

MSRP DECIMAL(10,2) NOT NULL

);

CREATE TABLE customers (

customerNumber INT PRIMARY KEY,

customerName VARCHAR(50) NOT NULL,

contactLastName VARCHAR(50) NOT NULL,

contactFirstName VARCHAR(50) NOT NULL,

phone VARCHAR(50) NOT NULL,

addressLine1 VARCHAR(50) NOT NULL,

addressLine2 VARCHAR(50),

city VARCHAR(50) NOT NULL,

state VARCHAR(50),

postalCode VARCHAR(15),

country VARCHAR(50) NOT NULL,

salesRepEmployeeNumber INT,

creditLimit DECIMAL(10,2) NOT NULL,

FOREIGN KEY (salesRepEmployeeNumber) REFERENCES employees (employeeNumber)

);

CREATE TABLE orderdetails (

orderNumber INT,

productCode VARCHAR(15),

orderLineNumber INT NOT NULL,

quantityOrdered INT NOT NULL,

priceEach DECIMAL(10,2) NOT NULL,

orderLineNumber INT NOT NULL,

PRIMARY KEY (orderNumber, productCode, orderLineNumber),

FOREIGN KEY (orderNumber) REFERENCES orders (orderNumber),

FOREIGN KEY (productCode) REFERENCES products (productCode)

);

CREATE TABLE orders (

orderNumber INT PRIMARY KEY,

orderDate DATE NOT NULL,

requiredDate DATE NOT NULL,

shippedDate DATE,

status VARCHAR(15) NOT NULL,

comments TEXT,

customerNumber INT,

FOREIGN KEY (customerNumber) REFERENCES customers (customerNumber)

);

-- Execute the provided queries

-- Part One: Query Problems Using the "alanparadise/cm" repo

-- 1

SELECT DISTINCT city

FROM offices

ORDER BY city;

-- 2

SELECT EmployeeNumber, lastName, firstName, extension

FROM employees

WHERE officeCode = 4;

-- 3

SELECT productCode, productName, productVendor, quantityInStock, productLine

FROM products

WHERE quantityInStock BETWEEN 200 AND 1200;

-- 4

SELECT productCode, productName, productVendor, buyPrice, MSRP

FROM products

ORDER BY MSRP

LIMIT 1;

-- 5

SELECT productName, (MSRP - buyPrice) AS profit

FROM products

ORDER BY profit DESC

LIMIT 1;

-- 6

SELECT country, COUNT(customerNumber) AS Customers

FROM customers

GROUP BY country

HAVING COUNT(customerNumber) = 2

ORDER BY country;

-- 7

SELECT o.productCode, p.productName, COUNT(orderLineNumber) AS OrderCount

FROM orderdetails AS o

JOIN products AS p ON o.productCode = p.productCode

GROUP BY o.productCode, p.productName

HAVING COUNT(orderLineNumber) = 25;

-- 8

SELECT e.employeeNumber, e.firstName || ' ' || e.lastName AS name

FROM employees AS e

WHERE e.reportsTo IN (

SELECT employeeNumber

FROM employees

WHERE lastName = 'Murphy' OR lastName = 'Bondur'

);

-- 9

SELECT employeeNumber, lastName, firstName

FROM employees

WHERE reportsTo IS NULL;

-- 10

SELECT productName

FROM products

WHERE productName LIKE '%195%'

AND productLine = 'Classic Cars';

-- 11

SELECT 'November' AS ordermonth, COUNT(orderNumber)

FROM orders

WHERE orderDate LIKE '%2004%'

GROUP BY ordermonth

ORDER BY COUNT(orderNumber) DESC

LIMIT 1;

-- 12

SELECT e.firstName, e.lastName

FROM employees AS e

WHERE e.jobTitle = 'Sales Rep'

AND e.employeeNumber NOT IN (

SELECT DISTINCT c.salesRepEmployeeNumber

FROM customers AS c

WHERE c.salesRepEmployeeNumber IS NOT NULL

);

-- 13

SELECT c.customerName

FROM customers AS c

LEFT JOIN orders AS o ON c.customerNumber = o.customerNumber

WHERE c.country = 'Switzerland'

AND o.customerNumber IS NULL;

-- 14

SELECT c.customerName, SUM(d.quantityOrdered) AS totalquantity

FROM customers AS c

LEFT JOIN orders AS o ON c.customerNumber = o.customerNumber

LEFT JOIN orderdetails AS d ON o.orderNumber = d.orderNumber

GROUP BY c.customerName

HAVING SUM(d.quantityOrdered) > 1650;

Store Procedure to insert dummy data in each table.

To create a stored procedure that inserts dummy data into each table, you can use the following SQL code as an example:

CREATE OR REPLACE PROCEDURE insert\_dummy\_data AS

BEGIN

-- Insert dummy data into the offices table

INSERT INTO offices (officeCode, city, phone, addressLine1, addressLine2, state, country, postalCode, territory)

VALUES

(1, 'San Francisco', '123456789', '123 Main St', 'Suite 101', 'CA', 'USA', '94101', 'NA'),

(2, 'New York', '987654321', '456 Broadway', NULL, 'NY', 'USA', '10001', 'NA'),

(3, 'London', '555555555', '789 Oxford St', NULL, NULL, 'UK', 'SW1A 1AA', 'EMEA');

-- Insert dummy data into the employees table

INSERT INTO employees (employeeNumber, lastName, firstName, extension, email, officeCode, reportsTo, jobTitle)

VALUES

(1, 'Doe', 'John', '123', 'john.doe@example.com', 1, NULL, 'Manager'),

(2, 'Smith', 'Jane', '456', 'jane.smith@example.com', 1, 1, 'Sales Rep'),

(3, 'Johnson', 'Robert', '789', 'robert.johnson@example.com', 2, 1, 'Sales Rep');

-- Insert dummy data into the products table

INSERT INTO products (productCode, productName, productLine, productScale, productVendor, productDescription, quantityInStock, buyPrice, MSRP)

VALUES

('P1', 'Product 1', 'Category 1', '1:10', 'Vendor 1', 'Description 1', 100, 10.99, 19.99),

('P2', 'Product 2', 'Category 1', '1:12', 'Vendor 2', 'Description 2', 200, 20.99, 29.99),

('P3', 'Product 3', 'Category 2', '1:18', 'Vendor 3', 'Description 3', 300, 30.99, 39.99);

-- Insert dummy data into the customers table

INSERT INTO customers (customerNumber, customerName, contactLastName, contactFirstName, phone, addressLine1, addressLine2, city, state, postalCode, country, salesRepEmployeeNumber, creditLimit)

VALUES

(1, 'Customer 1', 'Smith', 'John', '123456789', '123 Main St', 'Suite 101', 'San Francisco', 'CA', '94101', 'USA', 2, 10000.00),

(2, 'Customer 2', 'Johnson', 'Jane', '987654321', '456 Broadway', NULL, 'New York', 'NY', '10001', 'USA', 3, 20000.00);

-- Insert dummy data into the orderdetails table

INSERT INTO orderdetails (orderNumber, productCode, orderLineNumber, quantityOrdered, priceEach)

VALUES

(1, 'P1', 1, 5, 10.99),

(1, 'P2', 2, 10, 20.99),

(2, 'P1', 1, 3, 10.99),

(2, 'P3', 2, 7, 30.99);

-- Insert dummy data into the orders table

INSERT INTO orders (orderNumber, orderDate, requiredDate, shippedDate, status, comments, customerNumber)

VALUES

(1, '2023-07-01', '2023-07-05', '2023-07-02', 'Shipped', 'Order 1 comment', 1),

(2, '2023-07-02', '2023-07-06', NULL, 'In Process', 'Order 2 comment', 2);

COMMIT;

END;

This stored procedure, named insert\_dummy\_data, includes sample data insertion statements for each table.

CALL insert\_dummy\_data;

Executing this statement will insert the dummy data into the respective tables.

Take backup for the database at 11AM and 4PM.

To take backups of a PostgreSQL database at specific times, such as 11 AM and 4 PM, you can use the pg\_dump utility along with a cron job to schedule the backups. Here's an example of how you can accomplish this:

Create a script file that contains the pg\_dump command to take the backup. Let's assume you create a file named backup\_script.sh:

bash

#!/bin/bash

# Set the current date and time

current\_date=$(date +%Y-%m-%d\_%H-%M-%S)

# Set the backup directory path

backup\_dir="/path/to/backup/directory"

# Set the database name

database\_name="your\_database\_name"

# Set the output file name

output\_file="$backup\_dir/$database\_name-$current\_date.sql"

# Run the pg\_dump command to take the backup

pg\_dump -U your\_username -d $database\_name > $output\_file

Make sure to replace your\_username and your\_database\_name with the appropriate values. Adjust the backup\_dir variable to the desired backup directory path.

Make the script file executable by running the following command:

bash

chmod +x backup\_script.sh

Schedule the backup using cron jobs. Open the crontab for editing by running the following command:

bash

crontab -e

Add the following lines to the crontab file to schedule the backups at 11 AM and 4 PM:

bash

# Backup at 11 AM

0 11 \* \* \* /path/to/backup\_script.sh

# Backup at 4 PM

0 16 \* \* \* /path/to/backup\_script.sh

Save the crontab file.

With this configuration, the pg\_dump command will be executed at 11 AM and 4 PM every day, creating backups in the specified backup directory with the timestamp appended to the file name.

Note: Make sure to replace /path/to/backup\_script.sh with the actual path to the script file you created in step 1.

Remember to adjust the permissions and paths according to your setup, and test the backup script to ensure it runs correctly before relying on it for critical data backups.

Create View, CTE, functions based on the database.

To create a view, CTE (Common Table Expression), and functions based on the database, you need to define the specific logic and requirements for each case. Below, I'll provide you with examples of creating a view, a CTE, and a function based on the existing tables provided in the previous queries.

Creating a View:

A view is a virtual table that is based on the result of an SQL query. It allows you to save a query as a view and use it like a regular table. Here's an example of creating a view:

CREATE OR REPLACE VIEW order\_details\_view AS

SELECT o.orderNumber, o.orderDate, p.productCode, p.productName, od.quantityOrdered

FROM orders o

JOIN orderdetails od ON o.orderNumber = od.orderNumber

JOIN products p ON od.productCode = p.productCode;

The order\_details\_view view combines information from the orders, orderdetails, and products tables to display the order number, order date, product code, product name, and quantity ordered.

Creating a CTE:

A CTE (Common Table Expression) is a named temporary result set that can be referenced within a single SQL statement. It is useful for complex queries or when you need to reuse a subquery. Here's an example of creating a CTE:

WITH high\_profit\_products AS (

SELECT productName, (MSRP - buyPrice) AS profit

FROM products

WHERE (MSRP - buyPrice) > 50

)

SELECT productName, profit

FROM high\_profit\_products;

In this example, the CTE named high\_profit\_products calculates the profit for each product where the profit is greater than 50. The final query selects the product name and profit from the CTE.

Creating a Function:

A function is a stored procedure that returns a value. It allows you to encapsulate complex logic and reuse it in different parts of your database. Here's an example of creating a function:

CREATE OR REPLACE FUNCTION calculate\_order\_total(order\_id INT)

RETURNS DECIMAL AS

$$

DECLARE

total DECIMAL;

BEGIN

SELECT SUM(priceEach \* quantityOrdered)

INTO total

FROM orderdetails

WHERE orderNumber = order\_id;

RETURN total;

END;

$$

LANGUAGE plpgsql;

The calculate\_order\_total function takes an order ID as input and calculates the total order amount by multiplying the price of each item (priceEach) with the quantity ordered (quantityOrdered). The function returns the total as a decimal value.

Use Transaction

To incorporate transactions into the previous examples, you can wrap the SQL statements within a transaction block. This ensures that the operations within the transaction are treated as a single unit of work, and either all the changes are committed or none of them are. Here's an updated version of creating a view, CTE, function, and including transactions:

Creating a View with Transaction:

BEGIN;

CREATE OR REPLACE VIEW order\_details\_view AS

SELECT o.orderNumber, o.orderDate, p.productCode, p.productName, od.quantityOrdered

FROM orders o

JOIN orderdetails od ON o.orderNumber = od.orderNumber

JOIN products p ON od.productCode = p.productCode;

COMMIT;

The BEGIN statement starts the transaction, and the COMMIT statement commits the changes to the database after creating the view. If any error occurs during the execution of the statements, you can use the ROLLBACK statement to undo the changes and abort the transaction.

Creating a CTE with Transaction:

BEGIN;

WITH high\_profit\_products AS (

SELECT productName, (MSRP - buyPrice) AS profit

FROM products

WHERE (MSRP - buyPrice) > 50

)

SELECT productName, profit

FROM high\_profit\_products;

COMMIT;

The BEGIN statement starts the transaction, and the COMMIT statement commits the changes after executing the CTE. If an error occurs, you can use the ROLLBACK statement to undo the changes and abort the transaction.

Creating a Function with Transaction:

BEGIN;

CREATE OR REPLACE FUNCTION calculate\_order\_total(order\_id INT)

RETURNS DECIMAL AS

$$

DECLARE

total DECIMAL;

BEGIN

SELECT SUM(priceEach \* quantityOrdered)

INTO total

FROM orderdetails

WHERE orderNumber = order\_id;

RETURN total;

END;

$$

LANGUAGE plpgsql;

COMMIT;

The BEGIN statement starts the transaction, and the COMMIT statement commits the changes after creating the function. If any error occurs, you can use the ROLLBACK statement to undo the changes and abort the transaction.

By using transactions, you can ensure that all the changes related to creating the view, CTE, or function are performed as a single atomic operation. If an error occurs at any point, the transaction can be rolled back to maintain data integrity.

Use CASE, Union

Here's an example that incorporates the use of CASE statements, UNION, and transactions:

BEGIN;

-- Create a view using a CASE statement

CREATE OR REPLACE VIEW order\_status\_view AS

SELECT orderNumber,

CASE

WHEN status = 'Shipped' THEN 'Completed'

WHEN status = 'In Process' THEN 'Pending'

ELSE 'Unknown'

END AS orderStatus

FROM orders;

-- Use UNION to combine two result sets

SELECT 'Classic Cars' AS productLine

UNION

SELECT 'Motorcycles' AS productLine;

COMMIT;

In this example:

The order\_status\_view view is created using a CASE statement. It maps the status column values from the orders table to a more meaningful orderStatus value: 'Completed' for 'Shipped', 'Pending' for 'In Process', and 'Unknown' for any other status.

UNION is used to combine two result sets. In this case, it combines two SELECT statements that retrieve product line values for 'Classic Cars' and 'Motorcycles'. The UNION operator eliminates duplicate values and merges the results into a single result set.

The entire script is wrapped in a transaction block using BEGIN and COMMIT statements to ensure atomicity. If any error occurs during the execution, the ROLLBACK statement can be used to undo the changes and abort the transaction.

DDL and DML queries.

Here are examples of DDL (Data Definition Language) and DML (Data Manipulation Language) queries:

DDL Queries:

Create a new table:

CREATE TABLE employees (

id SERIAL PRIMARY KEY,

name VARCHAR(50) NOT NULL,

age INT,

salary DECIMAL(10, 2)

);

Alter a table to add a new column:

ALTER TABLE employees

ADD COLUMN department VARCHAR(50);

Create a unique index on a table:

CREATE UNIQUE INDEX idx\_employee\_email

ON employees (email);

Create a foreign key constraint:

ALTER TABLE employees

ADD CONSTRAINT fk\_employee\_department

FOREIGN KEY (department\_id) REFERENCES departments(id);

DML Queries:

Insert a new record into a table:

INSERT INTO employees (name, age, salary)

VALUES ('John Doe', 30, 5000.00);

Update records in a table:

UPDATE employees

SET salary = 5500.00

WHERE department = 'Sales';

Delete records from a table:

DELETE FROM employees

WHERE age > 40;

Retrieve data using SELECT statement:

SELECT name, age, salary

FROM employees

WHERE department = 'HR'

ORDER BY salary DESC;

These are just a few examples of DDL and DML queries. DDL is used to define the database structure, such as creating or altering tables, indexes, and constraints. DML is used to manipulate the data within the tables, including inserting, updating, deleting, and retrieving data.

-- Switch to the newly created database

\c alanparadise;

To modify the provided snippet to be used in a query script, you need to replace the "\c alanparadise;" statement, which is a psql meta-command used in an interactive session, with the equivalent SQL statement to connect to the "alanparadise" database.

The SQL statement to establish a connection to the database would be:

-- Connect to the alanparadise database

\connect alanparadise;

However, since you are working with a query script, you don't need the connection statement. You can assume that the script is already connected to the "alanparadise" database.

Therefore, you can simply remove the line "\c alanparadise;" from the script. The modified script would look like this:

--Part One: Query Problems Using the "alanparadise/cm" repo

----====================================================----

--1

-- List the names of the cities in alphabetical order where Classic Models has offices. (7)

SELECT DISTINCT city

FROM "alanparadise/cm"."offices"

ORDER BY city;

--2 List the EmployeeNumber, LastName, FirstName, Extension for all employees working out of the Paris office. (5)

SELECT EmployeeNumber, LastName, FirstName, Extension

FROM "alanparadise/cm"."employees"

WHERE officecode = 4;

--3

--List the ProductCode, ProductName, ProductVendor, QuantityInStock and ProductLine for all

--products with a QuantityInStock between 200 and 1200. (11)

SELECT ProductCode, ProductName, ProductVendor, QuantityInStock, ProductLine

FROM "alanparadise/cm"."products"

WHERE QuantityInStock BETWEEN 200 AND 1200;

-- Rest of the queries...

By removing the connection statement, the script can be executed directly without any modifications.

ERROR: syntax error at or near "\"

LINE 5: \connect alanparadise;

^

SQL state: 42601

Character: 112

Apologies for the confusion. The "\connect" command is specific to the psql command-line tool and cannot be used directly within a SQL script.

If you are executing the script using the psql command-line tool, you can pass the database connection information as command-line arguments. Here's an example:

bash

psql -h <hostname> -p <port> -U <username> -d alanparadise -f script.sql

Replace <hostname>, <port>, and <username> with the appropriate values for your database connection. The -d option specifies the database to connect to, and the -f option specifies the SQL script file to execute.

If you are executing the script using a different tool or method, make sure you are already connected to the "alanparadise" database before executing the script.