

# Batch: B-1 Roll No.: 16010422234 Name: Chandana Ramesh Galgali

# 

# Experiment No: 06

**Aim:** To create nested queries for the given database



**Resources needed:** PostgreSQL PgAdmin3



# not in:

This connective tests for absence of the set membership.

For example to select details of the books written by authors other than r.p.jain and d.perry use

select book\_id, book\_name,price from book where author not in(„r.p.jain‟, „d. perry‟,‟godse‟);

# all:

this keyword is basically used in set comparison query. It is used in association with relational operators.

“> all” corresponds to the phrase „greater than all‟.

For example to display details of the book that have prices greater than all the books published in 2000.

Select book\_id, book\_name, price from book where price >all (select price from book where pub\_year=‟2000‟);

# any or some:

These keywords are used with relational operators in the where clause of set comparison query. “=some” is identical to in and “<>some” is identical to not in.

“>any “ is nothing but „greater than at least one‟.

# exists and not exists:

exists is the test for non empty set. It is represented by an expression of the form ‘exists (select ……. From …….) ‘. Such expression evaluates to true only if the result evaluating the subquery represented by the (select ……. From ……) is non empty.

for example to select names of the books for which order is placed use

select book\_name from book where exists( select \* from order where book\_id=order.book\_id);

**Procedure / Approach /Algorithm / Activity Diagram:**

Refer to the different syntax given in the theory section and formulate queries consisting of nested sub queries, in , not in, as, group by, having etc clauses and different set operations for your database.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Results: (Program printout with output / Document printout as per the format)**

Program:

SELECT \* FROM Flights WHERE Flight\_ID IN (1, 3, 5);

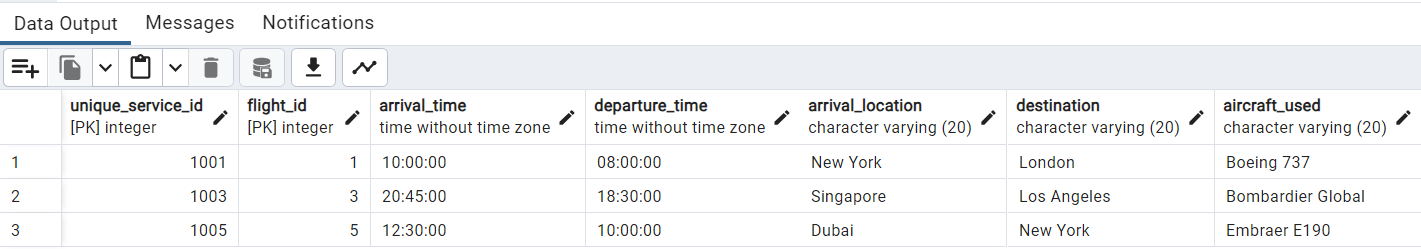
SELECT \* FROM Flights WHERE Flight\_ID NOT IN (1, 3, 5);

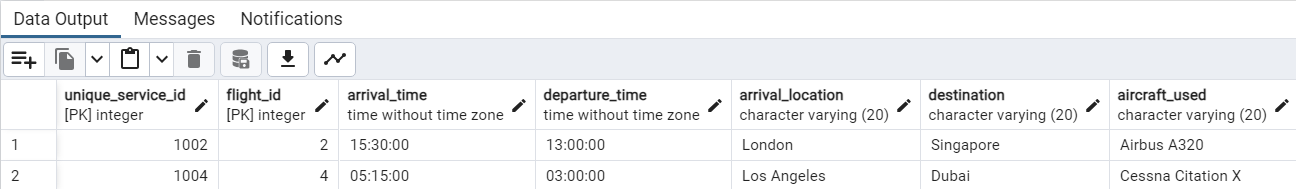
SELECT \* FROM Passenger WHERE EXISTS (SELECT \* FROM Tickets WHERE Passenger.Passenger\_ID = Tickets.Passenger\_ID);

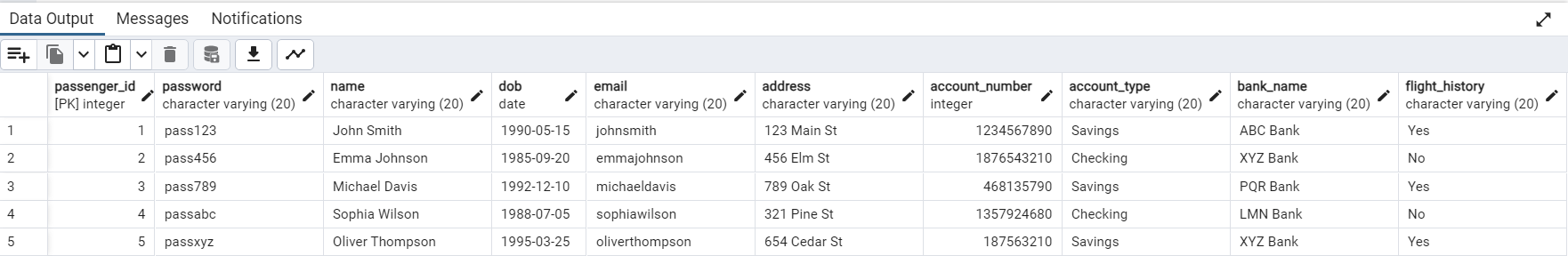
SELECT \* FROM Passenger WHERE NOT EXISTS (SELECT \* FROM Tickets WHERE Passenger.Passenger\_ID = Tickets.Passenger\_ID);

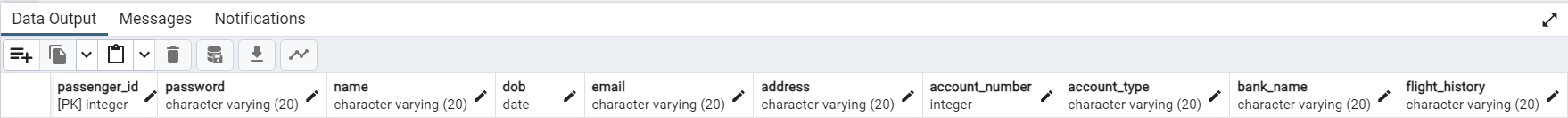
SELECT \* FROM Flights WHERE Arrival\_Time > ANY (SELECT Departure\_Time FROM Flights WHERE Destination = 'New York');

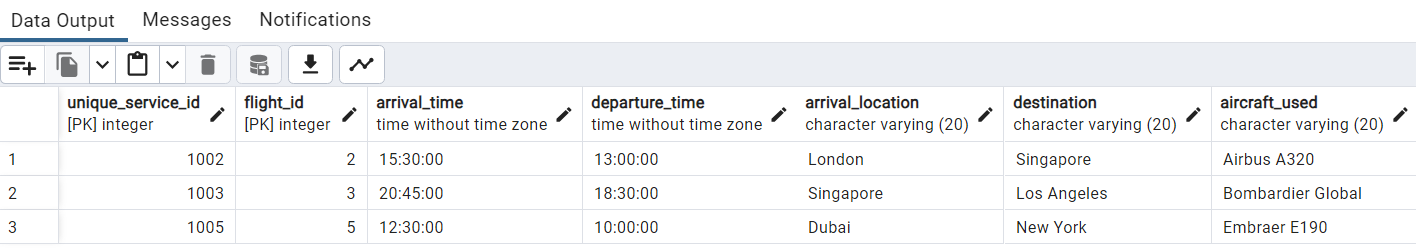
Output:

****

****

****

****

****

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

**Questions:**

**1. Explain what are the disadvantages using view on update function.**

**Ans:** Using views for update functions in a database has some disadvantages. Here are a few of them:

1. Complexity: Updating data through a view can introduce complexity, especially when the view is based on multiple tables or involves complex joins. It can be challenging to ensure that the update operation is correctly applied to the underlying tables and that the view remains consistent with the updated data.

2. Performance: Updating data through a view can have an impact on performance. Since views are virtual tables derived from underlying tables, updating a view may require updating the underlying tables as well. This can result in additional overhead and potentially slower update operations compared to directly updating the underlying tables.

3. Limited functionality: Views have certain limitations when it comes to update operations. Not all views are updatable, especially if they involve complex queries, aggregations, or joins. Additionally, some views may have restrictions on which columns can be updated or may not allow updates at all. This can limit the flexibility and functionality of update operations through views.

4. Data integrity: Updating data through a view can introduce challenges in maintaining data integrity. If the view is based on complex logic or involves multiple tables, ensuring that the update operation maintains referential integrity, constraints, and triggers can be more difficult. This can lead to potential data inconsistencies or integrity violations.

5. Security concerns: Using views for update functions can raise security concerns. If the view exposes sensitive or restricted data, allowing updates through the view can potentially lead to unauthorized modifications. Care must be taken to ensure that appropriate access controls and permissions are in place to mitigate these risks.

In summary, while views can provide convenience and abstraction in querying data, using them for update functions can introduce complexity, performance issues, limited functionality, data integrity challenges, and security concerns. It is important to carefully consider these disadvantages and evaluate the trade-offs before deciding to use views for update operations.

**2. Can we use where clause with group by clause? Justify your answer.**

**Ans:** Yes, we can use the WHERE clause with the GROUP BY clause in SQL queries. The WHERE clause is used to filter rows based on specific conditions, while the GROUP BY clause is used to group rows based on one or more columns.

When used together, the WHERE clause allows us to filter the rows before they are grouped by the specified columns in the GROUP BY clause. This can be useful when we want to apply conditions to the individual rows before they are grouped.

For example, let's consider a table called "Sales" with columns like "Product", "Category", and "Quantity". If we want to calculate the total quantity sold for a specific category, we can use the GROUP BY clause to group the rows by the "Category" column and the SUM function to calculate the total quantity. We can also use the WHERE clause to filter the rows based on a specific condition, such as only considering sales for a particular year.

Here's an example query:

SELECT Category, SUM(Quantity) AS TotalQuantity

FROM Sales

WHERE YEAR(SaleDate) = 2023

GROUP BY Category;

In this query, the WHERE clause filters the rows based on the condition "YEAR(SaleDate) = 2023", ensuring that only sales from the year 2023 are considered. The GROUP BY clause groups the filtered rows by the "Category" column, and the SUM function calculates the total quantity sold for each category.

Therefore, using the WHERE clause with the GROUP BY clause allows us to apply conditions to the individual rows before they are grouped, enabling more precise and targeted aggregations in our queries.

**3. Can we use having and group by clause without Aggregate functions? Justify your answer.**

**Ans:** No, we cannot use the HAVING and GROUP BY clauses without aggregate functions in SQL queries. The HAVING clause is used to filter the result set based on conditions applied to the aggregated values, while the GROUP BY clause is used to group rows based on one or more columns.

The purpose of the GROUP BY clause is to perform calculations on groups of data, and the aggregate functions are used to calculate values for each group. Without aggregate functions, there would be no meaningful calculations to perform on the grouped data.

The HAVING clause operates on the aggregated values, allowing us to specify conditions that must be met by the groups in order to be included in the output. Without aggregate functions, there would be no aggregated values to filter and no meaningful conditions to apply.

For example, let's consider a table called "Sales" with columns like "Product", "Category", and "Quantity". If we want to find categories with a total quantity sold greater than 100, we can use the GROUP BY clause to group the rows by the "Category" column and the SUM function to calculate the total quantity. Then, we can use the HAVING clause to filter the groups based on the condition "SUM(Quantity) > 100".

Here's an example query:

SELECT Category, SUM(Quantity) AS TotalQuantity

FROM Sales

GROUP BY Category

HAVING SUM(Quantity) > 100;

In this query, the GROUP BY clause groups the rows by the "Category" column, and the SUM function calculates the total quantity sold for each category. The HAVING clause filters the groups based on the condition "SUM(Quantity) > 100", ensuring that only categories with a total quantity sold greater than 100 are included in the output.

Therefore, the HAVING and GROUP BY clauses are designed to work together with aggregate functions to perform calculations on groups of data and filter the result set based on aggregated values. Without aggregate functions, these clauses would not have any meaningful purpose or functionality.

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

**Outcomes: Apply data models to real world scenarios.**

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

**Conclusion: (Conclusion to be based on the objectives and outcomes achieved)**

In conclusion, the experiment successfully demonstrated the creation and usage of nested queries for the given database. We gained a deeper understanding of how to retrieve data from multiple tables and apply conditions based on the results of subqueries. The knowledge and skills acquired through this experiment can be applied to real-world scenarios where complex data retrieval and filtering are required.

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

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of faculty in-charge with date**

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

**References:**

**Books/ Journals/ Websites:**

1. Korth, Slberchatz,Sudarshan, :”Database System Concepts”, 6th Edition, McGraw –

Hill

1. Elmasri and Navathe, “ Fundamentals of Database Systems”, 5thEdition, PEARSON

Education