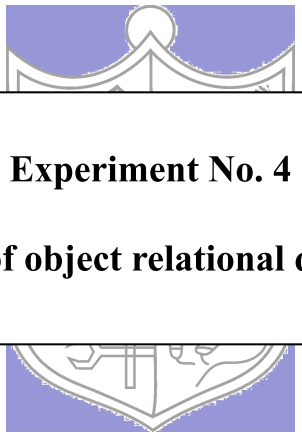


Experiment No. 4

Title: Execution of object relational queries

Batch: B-4 **Roll No.: 16010422234** **Name: Chandana Ramesh Galgali**

Experiment No.: 4

Title: Execution of object relational queries

Resources needed: PostgreSQL 9.3

Theory

Object types are user-defined types that make it possible to model real-world entities such as customers and purchase orders as objects in the database.

New object types can be created from any built-in database types and any previously created object types, object references, and collection types. Metadata for user-defined types is stored in a schema that is available to SQL, PL/SQL, Java, and other published interfaces.

Row Objects and Column Objects:
Objects that are stored in complete rows in object tables are called row objects. Objects that are stored as columns of a table in a larger row, or are attributes of other objects, are called column objects

Defining Types:

In PostgreSQL the syntax for creating simple type is as follows,

```
CREATE TYPE name AS ( attribute_name  data_type [, ... ] );
```

Example:

A definition of a point type consisting of two numbers in PostgreSQL is as follows,

```
create type PointType as(
      x int,
      y int
      );
```

An object type can be used like any other type in further declarations of object-types or table-types.

E.g. a new type with name LineType is created using PointType which was created earlier.

```
CREATE TYPE LineType AS(
      end1 PointType,
      end2 PointType
      );
```

Dropping Types :

To drop type for example LineType, command will be :

```
DROP TYPE Linetype;
```

Constructing Object Values:

Like C++, PostgreSQL provides built-in constructors for values of a declared type, and these constructors can be invoked using a parenthesized list of appropriate values.

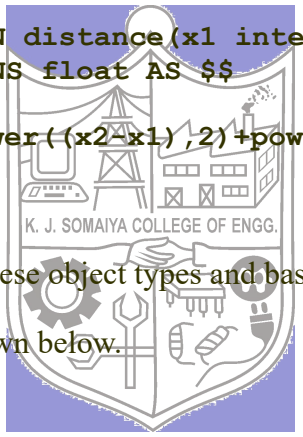
For example, here is how we would insert into Lines a line with ID 27 that ran from the origin to the point (3,4):

```
INSERT INTO Lines VALUES(27,((0,0),(3,4)),distance(0,0,3,4));
```

Declaring and Defining Methods:

A type declaration can also include methods that are defined on values of that type. The method is declared as shown in the example below.

```
CREATE OR REPLACE FUNCTION distance(x1 integer, y1 integer,x2
integer,y2 integer) RETURNS float AS $$
      BEGIN
            RETURN sqrt(power((x2-x1),2)+power((y2-y1),2));
      END;
$$ LANGUAGE plpgsql;
```



Then you can create tables using these object types and basic data types.

Creation on new table Lines is shown below.

```
CREATE TABLE Lines (
       lineID INT,
       line LineType,
        dist float
           );
```

Now after the table is created you can add a populate table by executing insert queries as explained above.

You can execute different queries on the Lines table. For example to display data from the Lines table, select a specific line from the Lines table etc.

Queries to Relations That Involve User-Defined Types:

Values of components of an object are accessed with the dot notation. We actually saw an example of this notation above, as we found the x-component of point end1 by referring to end1.x, and so on. In general, if *N* refers to some object *O* of type *T*, and one of the components (attribute or method) of type *T* is *A*, then N.A refers to this component of object *O*.

For example, the following query finds the x coordinates of both endpoints of the line.

```
SELECT lineID, ((L.line).end1).x,((L.line).end2).x FROM Lines L;
```

- Note that in order to access fields of an object, we have to start with an *alias* of a relation name. While lineID, being a top-level attribute of relation Lines, can be referred to normally, in order to get into the attribute line, we need to give relation Lines an alias (we chose L) and use it to start all paths to the desired subobjects.
- Dropping the ``L" or replacing it by ``Lines." doesn't work.
- Notice also the use of a method in a query. Since line is an attribute of type LineType, one can apply to it the methods of that type, using the dot notation shown.
- Here are some other queries about the relation lines.

```
SELECT (L.line).end2 FROM Lines L;
```

Prints the second end of each line, but as a value of type PointType, not as a pair of numbers.

Object Oriented features:
Inheritance:

```
CREATE TABLE point of PointType;

CREATE TABLE axis (
      z int
 ) inherits (point);

 INSERT INTO axis values(2,5,6);

 select * from axis;
```

Procedure / Approach /Algorithm / Activity Diagram:
Perform following tasks,

- Create a table using object type field
- Insert values in that table
- Retrieve values from the table
- Implement and use any function associated with the table created

---

Results: (Queries depicting the above said activity performed individually)

```
create type PointType as(
x int,
y int
);
CREATE TYPE LineType AS(
      end1 PointType,
      end2 PointType
);
CREATE OR REPLACE FUNCTION distance(x1 integer, y1 integer,x2
integer,y2 integer) RETURNS float AS $$
        BEGIN
                RETURN sqrt(power((x2-x1),2)+power((y2-y1),2));
        END;
$$ LANGUAGE plpgsql;
```

```
CREATE TABLE Lines (
      lineID INT,
      line LineType,
      dist float
);
INSERT INTO Lines VALUES(27,((0,0),(3,4)),distance(0,0,3,4));
SELECT lineID, ((L.line).end1).x,((L.line).end2).x FROM Lines L;
SELECT (L.line).end2 FROM Lines L;
CREATE TABLE point of PointType;
CREATE TABLE axis (
      z int
 ) inherits (point);
 INSERT INTO axis values(2,5,6);
 select * from axis;
```

Query Query History

```
1  create type PointType as(
2  x int,
3  y int
4  );
```

Data Output Messages Notifications

```
CREATE TYPE

Query returned successfully in 161 msec.
```

Query Query History

```
5  CREATE TYPE LineType AS(
6      end1 PointType,
7      end2 PointType
8  );
```

Data Output Messages Notifications

```
CREATE TYPE

Query returned successfully in 91 msec.
```

Query Query History

```
9  CREATE OR REPLACE FUNCTION distance(x1 integer, y1 integer,x2 integer,y2 integer) RETURNS float AS
10         BEGIN
11                 RETURN sqrt(power((x2-x1),2)+power((y2-y1),2));
12         END;
13 $$ LANGUAGE plpgsql;
```

Data Output Messages Notifications

```
CREATE FUNCTION

Query returned successfully in 144 msec.
```

Query Query History

```
14 CREATE TABLE Lines (
15     lineID INT,
16     line LineType,
17     dist float
18 );
19
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 85 msec.
```

Query Query History

```
19 INSERT INTO Lines VALUES(27,((0,0),(3,4)),distance(0,0,3,4));
```

Data Output Messages Notifications

```
INSERT 0 1

Query returned successfully in 80 msec.
```

Query Query History

```
20  SELECT lineID, ((L.line).end1).x,((L.line).end2).x FROM Lines L;
```

Data Output Messages Notifications

	lineid integer	x integer	x integer
1	27	0	3

Query Query History

```
21  SELECT (L.line).end2 FROM Lines L;
```

Data Output Messages Notifications

	end2 pointtype
1	(3,4)

Query Query History

```
22  CREATE TABLE point of PointType;
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 105 msec.
```

Query Query History

```
23  CREATE TABLE axis (
24      z int
25  ) inherits (point);
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 163 msec.
```

Query Query History

```
26  INSERT INTO axis values(2,5,6);
```

Data Output Messages Notifications

```
INSERT 0 1

Query returned successfully in 101 msec.
```

Query Query History

```
27  select * from axis;
```

Data Output Messages Notifications

	x integer	y integer	z integer
1	2	5	6

Queries:

```
CREATE TYPE CustomerType AS (
    customerID INT,
    firstName VARCHAR(50),
    lastName VARCHAR(50),
    accountNumber VARCHAR(15)
);

CREATE TYPE TransactionType AS (
    transactionID INT,
    transactionDate TIMESTAMP,
    amount DECIMAL(10, 2),
    description VARCHAR(255)
);

CREATE OR REPLACE FUNCTION calculateBalance(p_accountNumber
VARCHAR(15)) RETURNS DECIMAL AS $$
    DECLARE
        totalAmount DECIMAL(10, 2);
    BEGIN
        SELECT COALESCE(SUM((t.transactionData).amount), 0)
        INTO totalAmount
        FROM Transactions t
        WHERE t.accountNumber = p_accountNumber;
        RETURN totalAmount;
    END;
$$ LANGUAGE plpgsql;

CREATE TABLE Customers (
    customerID SERIAL PRIMARY KEY,
    customerData CustomerType
);

CREATE TABLE Transactions (
    transactionID SERIAL PRIMARY KEY,
    transactionData TransactionType,
    accountNumber VARCHAR(15)
);

INSERT INTO Customers (customerData) VALUES
    ((1, 'John', 'Doe', 'ACC123')),
    ((2, 'Jane', 'Smith', 'ACC456'));

INSERT INTO Transactions (transactionData, accountNumber) VALUES
    ((1, '2024-01-23 10:30:00', 100.00, 'Deposit'), 'ACC123'),
    ((2, '2024-01-23 11:45:00', -50.00, 'Withdrawal'), 'ACC123'),
    ((3, '2024-01-23 12:15:00', 200.00, 'Deposit'), 'ACC456');

SELECT c.customerID, (c.customerData).firstName,
(c.customerData).lastName,
calculateBalance((c.customerData).accountNumber) AS balance
FROM Customers c;
```

Output(s):

Query Query History

```
1  CREATE TYPE CustomerType AS (
2      customerID INT,
3      firstName VARCHAR(50),
4      lastName VARCHAR(50),
5      accountNumber VARCHAR(15)
6  );
7
```

Data Output Messages Notifications

```
CREATE TYPE

Query returned successfully in 178 msec.
```

Query Query History

```
7  CREATE TYPE TransactionType AS (
8      transactionID INT,
9      transactionDate TIMESTAMP,
10     amount DECIMAL(10, 2),
11     description VARCHAR(255)
12 );
13
```

Data Output Messages Notifications

```
CREATE TYPE

Query returned successfully in 92 msec.
```

Query Query History

```
13  CREATE OR REPLACE FUNCTION calculateBalance(p_accountNumber VARCHAR(15)) RETURNS DECIMAL AS $$
14      DECLARE
15          totalAmount DECIMAL(10, 2);
16      BEGIN
17          SELECT COALESCE(SUM((t.transactionData).amount), 0)
18          INTO totalAmount
19          FROM Transactions t
20          WHERE t.accountNumber = p_accountNumber;
21          RETURN totalAmount;
22      END;
23  $$ LANGUAGE plpgsql;
24
```

Data Output Messages Notifications

```
CREATE FUNCTION

Query returned successfully in 48 msec.
```

Query Query History

```
23  CREATE TABLE Customers (
24      customerID SERIAL PRIMARY KEY,
25      customerData CustomerType
26  );
27
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 133 msec.
```

Query Query History

```
27  CREATE TABLE Transactions (
28      transactionID SERIAL PRIMARY KEY,
29      transactionData TransactionType,
30      accountNumber VARCHAR(15)
31  );
32
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 198 msec.
```

Query Query History

```
32  INSERT INTO Customers (customerData) VALUES
33      ((1, 'John', 'Doe', 'ACC123')),
34      ((2, 'Jane', 'Smith', 'ACC456'));
35
```

Data Output Messages Notifications

```
INSERT 0 2

Query returned successfully in 162 msec.
```

Query Query History

```
35  INSERT INTO Transactions (transactionData, accountNumber) VALUES
36      ((1, '2024-01-23 10:30:00', 100.00, 'Deposit'), 'ACC123'),
37      ((2, '2024-01-23 11:45:00', -50.00, 'Withdrawal'), 'ACC123'),
38      ((3, '2024-01-23 12:15:00', 200.00, 'Deposit'), 'ACC456');
39
```

Data Output Messages Notifications

```
INSERT 0 3

Query returned successfully in 160 msec.
```

Query Query History

```
40  SELECT c.customerID, (c.customerData).firstName, (c.customerData).lastName, calculateBalance((c.customerData).accountNumber) AS balance
41  FROM Customers c;
```

Data Output Messages Notifications

	customerid [PK] integer	firstname character varying (50)	lastname character varying (50)	balance numeric
1	1	John	Doe	50.00
2	2	Jane	Smith	200.00

Questions:

1. **What is the difference between object relational and object oriented databases?**

Ans: Object Relational Databases (ORD):

- Structure: ORD combines features of relational databases with object-oriented database capabilities.
- Data Modeling: Tables are used for data storage, but they can contain complex data types and support relationships similar to object-oriented models.
- Flexibility: Offers more flexibility than traditional relational databases, allowing users to define complex data types and structures.
- SQL: Supports SQL for querying and manipulating data, often with extensions to handle complex data types.

Example: PostgreSQL with its support for user-defined types and object-oriented features.

Object Oriented Databases (OODB):

- Structure: OODBs store data in the form of objects, similar to object-oriented programming languages.
- Data Modeling: Objects encapsulate data and behavior, allowing for more natural representation of real-world entities.
- Flexibility: Provides high flexibility in handling complex relationships and supports inheritance and polymorphism.
- Query Language: Uses Object Query Language (OQL) or similar languages for querying, which is more closely aligned with object-oriented concepts.

Example: db4o, ObjectDB.

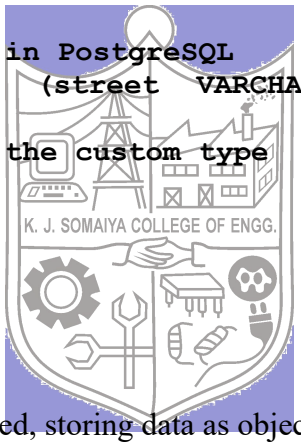
2. **Give comparison of any two database systems providing object relational database features.**

Ans: PostgreSQL (Object Relational Database):

- Model: Combines relational and object-oriented models.
- Data Types: Supports user-defined types, allowing users to create custom data types.
- Query Language: Uses SQL for querying and manipulation.

Example:

```
-- Creating a custom type in PostgreSQL
CREATE TYPE address AS (street VARCHAR, city VARCHAR, state VARCHAR);
-- Creating a table using the custom type
CREATE TABLE person (
    id SERIAL PRIMARY KEY,
    name VARCHAR,
    home_address address
);
```



db4o (Object Oriented Database):

- Model: Purely object-oriented, storing data as objects.
- Data Types: Represents data using objects with attributes and methods.
- Query Language: Utilizes Object Query Language (OQL) for querying.

Example:

```
// Storing an object in db4o
ObjectContainer db = Db4o.openFile("database.db4o");
Person person = new Person("John Doe", new Address("123 Main St", "City", "State"));
db.store(person);
db.close();
```

3. **Explore how the user defined types can be modified with queries.**

Ans: Modifying user-defined types typically involves using specific SQL statements to alter the type or updating the instances of the type. For example, in PostgreSQL:

```
-- Adding a new attribute to the user-defined type
ALTER TYPE address ADD ATTRIBUTE country VARCHAR;
-- Modifying an instance of the user-defined type
UPDATE person SET home_address.country = 'USA' WHERE id = 1;
```

Outcomes: Design advanced database systems using Object relational, Spatial and NOSQL databases and its implementation.

Conclusion: (Conclusion to be based on the objectives and outcomes achieved)
The execution of object-relational queries is a valuable skill in the realm of database management. The experiment successfully achieved its objectives by imparting knowledge on the principles of object-relational databases, facilitating a comparative analysis, and providing practical insights into the execution of queries. This newfound understanding equips individuals with the tools necessary to navigate and manipulate data effectively in object-relational database environments.

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

Signature of faculty in-charge with date

References:

1. Elmasri and Navathe, “Fundamentals of Database Systems”, Pearson Education
2. Raghu Ramakrishnan and Johannes Gehrke, “Database Management Systems” 3rd Edition, McGraw Hill,2002
3. Korth, Silberchatz, Sudarshan, “Database System Concepts” McGraw Hill