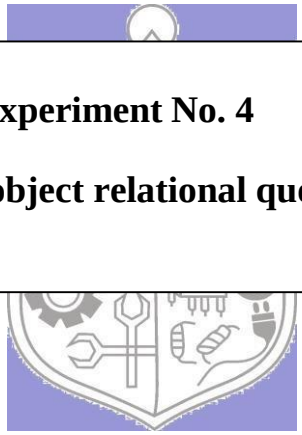


Experiment No. 4

Title: Execution of object relational queries

Batch: A3 **Roll No.: 16010422013** **Experiment No.:4**

Title: Execution of object relational queries

Resources needed: PostgreSQL 9.3

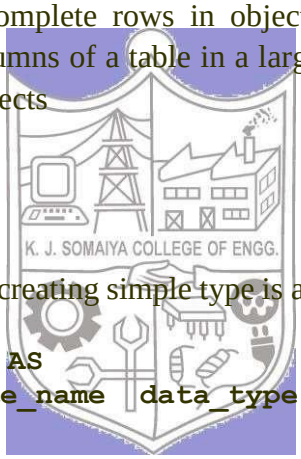
Theory

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 is 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 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 datatypes.

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 populate table by executing insert queries as explained above.

You can execute different queries on Lines table. For example to display data of Lines table, select specific line from 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 co-ordinates of both endpoints of 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
);
```

Data Output Explain Messages Notifications

```
CREATE TYPE

Query returned successfully in 110 msec.
```

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

Data Output Explain Messages Notifications

```
CREATE TYPE

Query returned successfully in 83 msec.
```

```
CREATE OR REPLACE FUNCTION distance(x1 integer, y1 integer,x2integer,y2 integer)
RETURNS float AS $$
  BEGIN
  RETURN sqrt(power((x2-x1),2)+power((y2-y1),2));END;
$$ LANGUAGE plpgsql;
```

Data Output Explain Messages Notifications

```
CREATE FUNCTION

Query returned successfully in 114 msec.
```

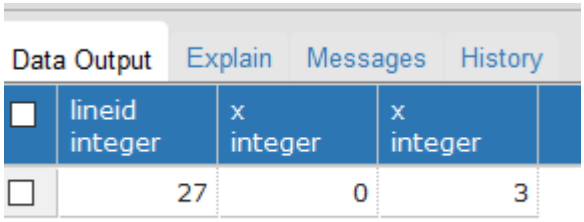
```
CREATE TABLE Lines (
lineID INT, line
LineType,dist float
);
```

Data Output Explain Messages Notifications

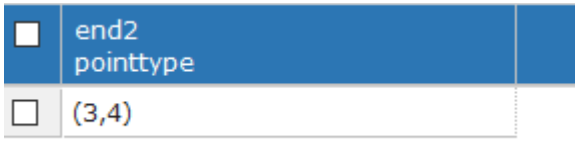
```
CREATE TABLE

Query returned successfully in 157 msec.
```

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

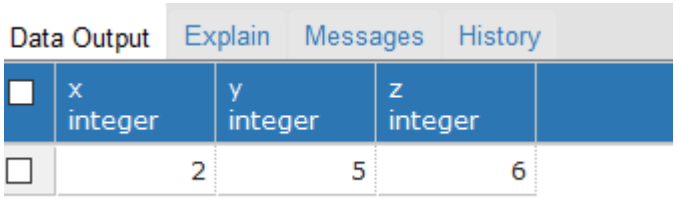
SELECT lineID, ((L.line).end1).x,((L.line).end2).xFROM Lines L;




```
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;



create type MarksType as(marks int,percentage float
);

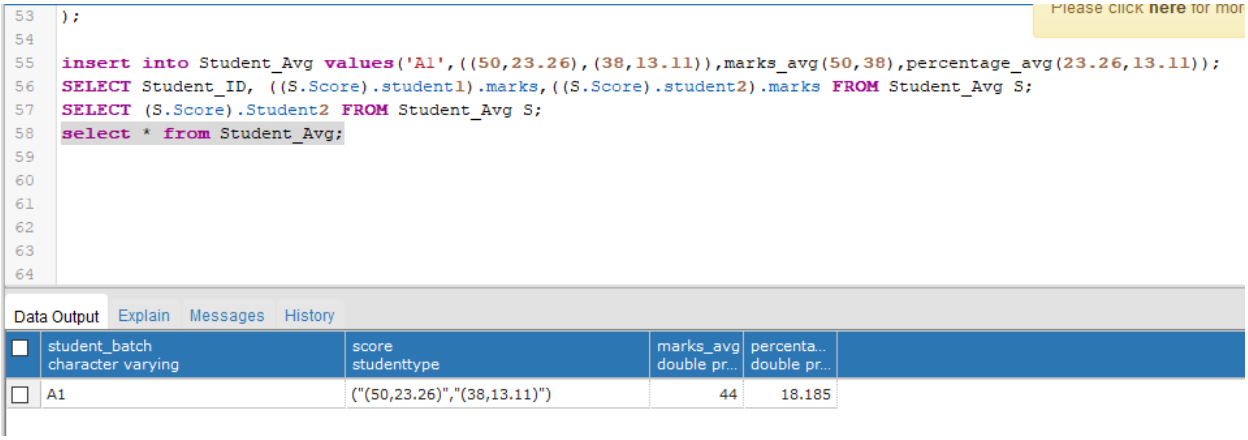
CREATE TYPE StudentType AS(
student1 MarksType,student2 MarksType
);

CREATE OR REPLACE FUNCTION marks_avg(marks1 integer, marks2 integer)RETURNS float AS $$
BEGIN
RETURN (marks1+marks2)/2;

END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION percentage_avg(percentage1 float,percentage2 float)
RETURNS float AS $$
BEGIN

```
RETURN (percentage1+percentage2)/2;

END;
$$ LANGUAGE plpgsql;

CREATE TABLE Student_Avg (
Student_Batch varchar, Score StudentType, marks_avg float,percentage_avg float
);

insert into Student_Avg values('A1',((50,23.26),(38,13.11)),marks_avg(50,38),percentage_avg(23.26,13.11));

select * from Student_Avg;
```

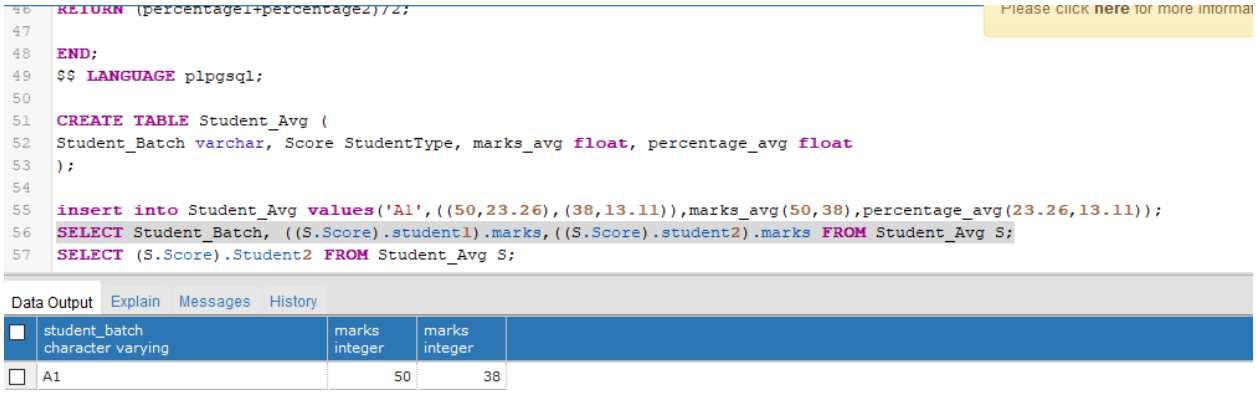


```
);

insert into Student_Avg values('A1',((50,23.26),(38,13.11)),marks_avg(50,38),percentage_avg(23.26,13.11));
SELECT Student_ID, ((S.Score).student1).marks,((S.Score).student2).marks FROM Student_Avg S;
SELECT (S.Score).Student2 FROM Student_Avg S;
select * from Student_Avg;
```

Data Output | Explain | Messages | History

student_batch character varying	score studenttype	marks_avg double pr...	percenta... double pr...
A1	("(50,23.26)","(38,13.11)")	44	18.185

```
SELECT Student_Batch,
((S.Score).student1).marks,((S.Score).student2).marks FROM Student_Avg
S;
```



```
RETURN (percentage1+percentage2)/2;

END;
$$ LANGUAGE plpgsql;

CREATE TABLE Student_Avg (
Student_Batch varchar, Score StudentType, marks_avg float, percentage_avg float
);

insert into Student_Avg values('A1',((50,23.26),(38,13.11)),marks_avg(50,38),percentage_avg(23.26,13.11));
SELECT Student_Batch, ((S.Score).student1).marks,((S.Score).student2).marks FROM Student_Avg S;
SELECT (S.Score).Student2 FROM Student_Avg S;
```

Data Output | Explain | Messages | History

student_batch character varying	marks integer	marks integer
A1	50	38

```
SELECT (S.Score).Student2 FROM Student_Avg S;
```

```
47
48  END;
49  $$ LANGUAGE plpgsql;
50
51  CREATE TABLE Student_Avg (
52  Student_Batch varchar, Score StudentType, marks_avg float, percentage_avg float
53  );
54
55  insert into Student_Avg values('A1',((50,23.26),(38,13.11)),marks_avg(50,38),perc
56  SELECT Student_Batch, ((S.Score).student1).marks,((S.Score).student2).marks FROM Stu
57  SELECT (S.Score).Student2 FROM Student_Avg S;
```

Data Output | Explain | Messages | History

	student2 markstype
☐	(38,13.11)

```
29
30  SELECT Student_Batch,
31  ((S.Score).student1).marks,((S.Score).student2).marks FROM Student_Avg S;
32  SELECT (S.Score).Student2 FROM Student_Avg S;
33
34  select * from Student_Avg;
35
36  insert into Student_Avg values ('A2',((52,19),(30,15.5)),marks_avg(43,28),percentage_avg(15.9,40.9));
37
```

Data Output | Explain | Messages | History

```
INSERT 0 1

Query returned successfully in 559 msec.
```

```
31  ((S.Score).student1).marks,((S.Score).student2).marks FROM Student_Avg S;
32  SELECT (S.Score).Student2 FROM Student_Avg S;
33
34  select * from Student_Avg;
35
36  insert into Student_Avg values ('A2',((52,19),(30,15.5)),marks_avg(43,28),perce
37
38
39
```

Data Output | Explain | Messages | History

	student_batch character varying	score studenttype	marks_avg double pr...	percenta... double pr...
☐	A1	("(50,23.26)","(38,13.11)")	44	18.155
☐	A2	("(52,19)","(30,15.5)")	35	28.4

Questions:

1. **What is the difference between object relational and object oriented databases?**
   Ans: An object-oriented database is one that stores data in the form of objects, similar to how object-oriented programming works. An object-relational database, on the other hand, is based on both the relational and object-oriented database models.

2. **Give comparison of any two database systems providing object relational database features.**
   Ans: An object-relational database (ORD), also known as an object-relational database management system (ORDBMS), is a type of database management system (DBMS) that is similar to a relational database but has an object-oriented database model, which means that objects, classes, and inheritance are directly supported in database schemas and queries.

Comparison of PostgreSQL vs MySQL database systems:-

PostgreSQL's architecture is more advanced and versatile than that of MySQL. It supports more complex features such as transactions, stored procedures, and user-defined types. MySQL, on the other hand, is well-known for being simple and easy to use.

Scalability: While both PostgreSQL and MySQL can manage massive volumes of data, PostgreSQL is better known for its ability to grow to suit the needs of large organisations.

In PostgreSQL, a table can inherit from zero or more other tables, and a query can refer to all rows of a table or all of its children. MySQL doesn't support table inheritance. It employs foreign key constraints rather than inheritance.

Open-source: While both PostgreSQL and MySQL are open-source database systems, PostgreSQL is known for being more community-driven and accepting of developer contributions.

Performance: In terms of performance, both systems offer advantages and disadvantages. MySQL is known for its rapid speed with read-intensive workloads, but PostgreSQL is known for its capacity to handle complicated queries and transactions.

ACID Compliance: ACID (Atomicity, Consistency, Isolation, and Durability) is a set of qualities that guarantee data integrity in database transactions. PostgreSQL is fully ACID.

compliant, while MySQL provides weaker guarantees in some cases.

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

Ans: The ALTER command, which is often used to modify standard tables/relations in SQL Server, does not play the same function when it comes to user defined table types. The simplest way is to establish a new table type, modify all objects that reference the old type, and then delete the old type.
This can be accomplished by creating a new type that references the old one.
Modifying an existing User Defined Data Type is banned because it may jeopardise the validity of current dependant data.

Outcomes:

CO2: 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)

We successfully built and implemented an advanced database system based on an object relational database.

Signature of faculty in-charge with date

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

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