

**Experiment No. 4**

**Title: Execution of object relational queries**

**Batch: B2 Roll No.:1601421119 Experiment No.:4 Title:** Execution of object relational queries

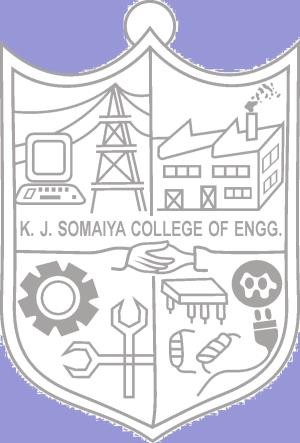
**Resources needed:** PostgreSQL 9.3

# 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,**

1. **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 :

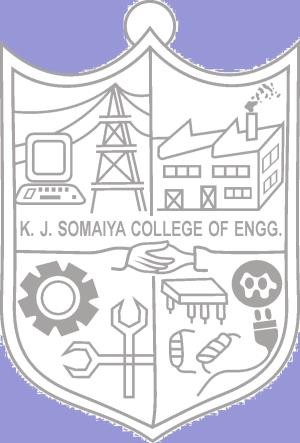
To drop type for example LineType, command will be :

**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.

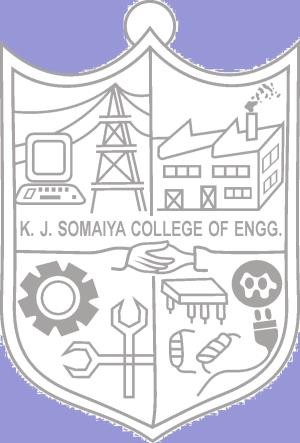
You can execute different queries on Lines table. For example to display data of Lines table, select specific line from Lines table etc.

# 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 (**

1. **int**

**) inherits (point);**

**INSERT INTO axis values(2,5,6); select \* from axis;**

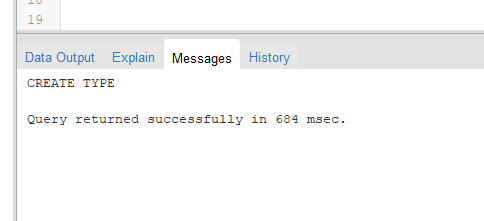
# Procedure / Approach /Algorithm / Activity Diagram:

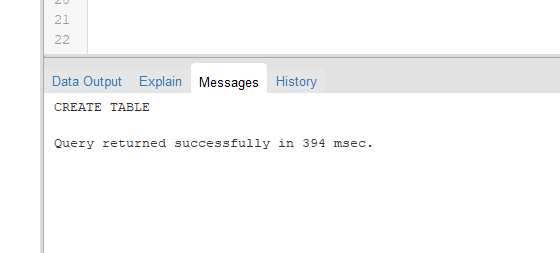
Perform following tasks,

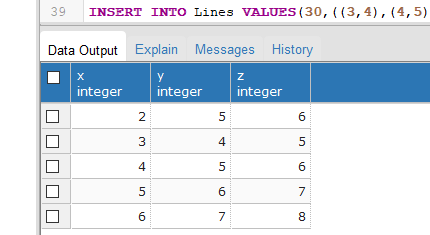
* Create a table using object type field
* Insert values in that table
* Retrieve values from the table
* Implement and use any function associated with the table created

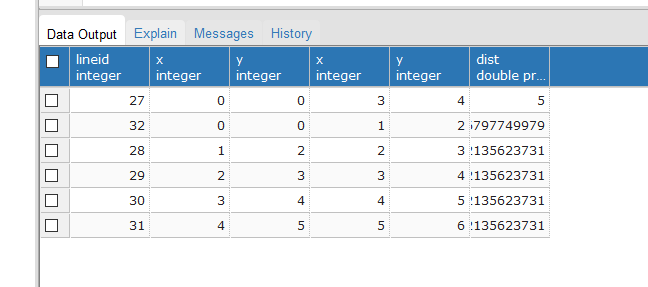
# Results: (Queries depicting the above said activity performed individually)

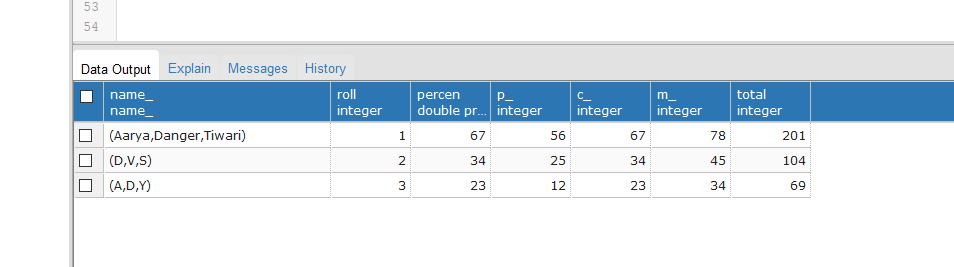
**USE COURIER NEW FONT WITH SIZE =11 FOR QUERY STATEMENTS**

****

****

****

****

****

**CODE:-**

**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**

**);**

**CREATE TABLE point of PointType;**

**CREATE TABLE axis**

**(**

**Z int**

**) inherits (point);**

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

**INSERT INTO axis values(3,4,5);**

**INSERT INTO axis values(4,5,6);**

**INSERT INTO axis values(5,6,7);**

**INSERT INTO axis values(6,7,8);**

**select \* from axis;**

**INSERT INTO Lines VALUES(32,((0,0),(1,2)),distance(0,0,1,2));**

**INSERT INTO Lines VALUES(28,((1,2),(2,3)),distance(1,2,2,3));**

**INSERT INTO Lines VALUES(29,((2,3),(3,4)),distance(2,3,3,4));**

**INSERT INTO Lines VALUES(30,((3,4),(4,5)),distance(3,4,4,5));**

**INSERT INTO Lines VALUES(31,((4,5),(5,6)),distance(4,5,5,6));**

**SELECT lineID, ((L.line).end1).x,((L.line).end1).y,((L.line).end2).x,((L.line).end2).y,L.dist FROM Lines L;**

**SELF SOLVED DATABASE CODE:-**

**create type name\_ as(**

**fname text,**

**mname text,**

**lname text**

**);**

**create table marks(**

**p\_ int,**

**c\_ int,**

**m\_ int**

**);**

**drop type marks;**

**create table student(**

**Naming name\_,**

**roll int,**

**percen float**

**);**

**Select \* from student**

**drop table markings;**

**CREATE OR REPLACE FUNCTION percentages(p\_ integer, c\_ integer,m\_ integer) RETURNS float AS $$**

**BEGIN**

**RETURN (p\_+c\_+m\_)/3;**

**END;**

**$$ LANGUAGE plpgsql;**

**CREATE OR REPLACE FUNCTION total(p\_ integer, c\_ integer,m\_ integer) RETURNS float AS $$**

**BEGIN**

**RETURN (p\_+c\_+m\_);**

**END;**

**$$ LANGUAGE plpgsql;**

**SELECT (L.info).name,(L.info).roll,(L.marks).p\_,(L.marks).m\_,L.percen FROM student L;**

**INSERT INTO student VALUES(('Aarya',1),(23,34,45),percentages(23,34,45));**

**INSERT INTO student VALUES(('Dhairya',2),(34,45,56),percentages(34,45,56));**

**INSERT INTO student VALUES(('Ameya',3),(45,56,67),percentages(45,56,67));**

**INSERT INTO student VALUES(('Awhad',4),(56,67,78),percentages(56,67,78));**

**select \* from student;**

**CREATE TABLE markings(**

**total int**

**) inherits (student,marks)**

**INSERT INTO personal\_info values('Aarya',1,6);**

**Select \* from student**

**Select \* from markings**

**Select \* from personal\_info**

**INSERT INTO markings VALUES(('Aarya','Danger','Tiwari'),1,percentages(56,67,78),56,67,78,total(56,67,78));**

**INSERT INTO markings VALUES(('D','V','S'),2,percentages(25,34,45),25,34,45,total(25,34,45));**

**INSERT INTO markings VALUES(('A','D','Y'),3,percentages(12,23,34),12,23,34,total(12,23,34));**

# Questions:

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

**Ans -** The main difference between object oriented database and object relational database is that object oriented database is a database that represents data in the form of objects like in object-oriented programming while object-relational database is a database that is based on the relational model and object-oriented database model.

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

**Ans :-**

This is a comparison of object–relational database management systems (ORDBMSs). Each system has at least some features of an object–relational database; they vary widely in their completeness and the approaches taken.

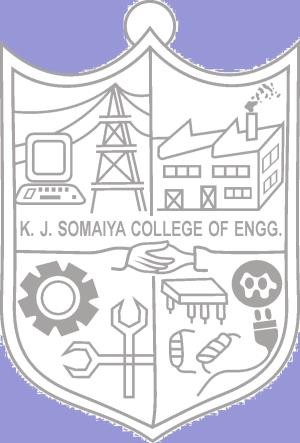
1. **Explore how the user defined types can be modified with queries.**

**Ans:-**

This is kind of a hack, but does seem to work. Below are the steps and an example of modifying a table type. One note is the sp\_refreshsqlmodule will fail if the change you made to the table type is a breaking change to that object, typically a procedure.

1. Use sp\_rename to rename the table type, I typically just add z to the beginning of the name.
2. Create a new table type with the original name and any modification you need to make to the table type.
3. Step through each dependency and run sp\_refreshsqlmodule on it.
4. Drop the renamed table type.

**Outcomes:-**   
CO2 Design advanced database systems using Object Relational, Spatial and NOSQL Databases and its implementation.



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

**Conclusion:**

**We can conclude that we have learnt about object-relational models and databases.Along with the Execution of object relational queries.**

**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