PSQL

The instance of the PostgreSQL is a cluster having databases.

Localhost: the instance of software which is running on this machine.

[postgres] is the default database of the PostgreSQL.

A database can also be referred as database schema.

Every database will have objects like:

Tables: a collection of data arranged as fields(cols) and records(rows)

Indexes: used to search data and increase performance of queries.

A table can have one or more indexes.

Views: read-only queries created on the table data.

Procedures and Functions: block of code that can be executed on a table.

Functions return single value.

Procedure returns no value or multiple value.

There are inbuit procedure and fucntions.

Triggers: A PostgreSQL trigger is **a function called automatically whenever an event such as an insert, update, or deletion occurs**.

Types of Database Software:

1. Relational: In RDBMS, tables are linked to each other based on keys(primary/foreign) keys.

Ecommerce/ library/ student 🡪 RDBMS

1. NoSQL: Used for unstructerd data like audio, video, long blogs, images, high volume of the data.

Social media apps which stores data in format of JSON ({}) / document.

1. Graph Database 🡪 Neo4j data in forms of nodes.
2. Network Database 🡪 based on owners of records containing sub owners arranged in branches.

String operators and functions

|| / CONCAT() 🡪 join to strings.

**Relations in DBMS**

1. Has one 🡪 one to many relationship.
2. Has many 🡪 many to one relationship.

e.g

1. Many to many

Movies – actors

Tasks – engg

Conference calls – employees

1. One to one relationships

**Keys**

Primary key 🡪 key which uniquely identifies the record.

Foreign key 🡪 primary key of another table is foreign key.

Primary vs Foreign Key

Primary Key:

Each row in every table has one primary key.

All values of primary key must be unique.

Will never change.

Foreign Key:

May or may not have foreign key.

Many rows in the same table may have same foreign key value.

Will change if relationship changes.

Serial in Posrtgres

PostgreSQL has a special kind of database object generator called SERIAL. It is **used to generate a sequence of integers which are often used as the Primary key of a table**.

Setting primary key syntax:

create table users ( id serial **primary key**, username varchar(50));

Setting foreign key syntax:

create table photos(id serial primary key, url varchar(50), **user\_id integer references users(id)**);

If we insert a record but the value specified in foreign key column does not exists in reference table then it raises the error.

\*\* In the foreign key column the value must be either value existing in reference table or NULL.

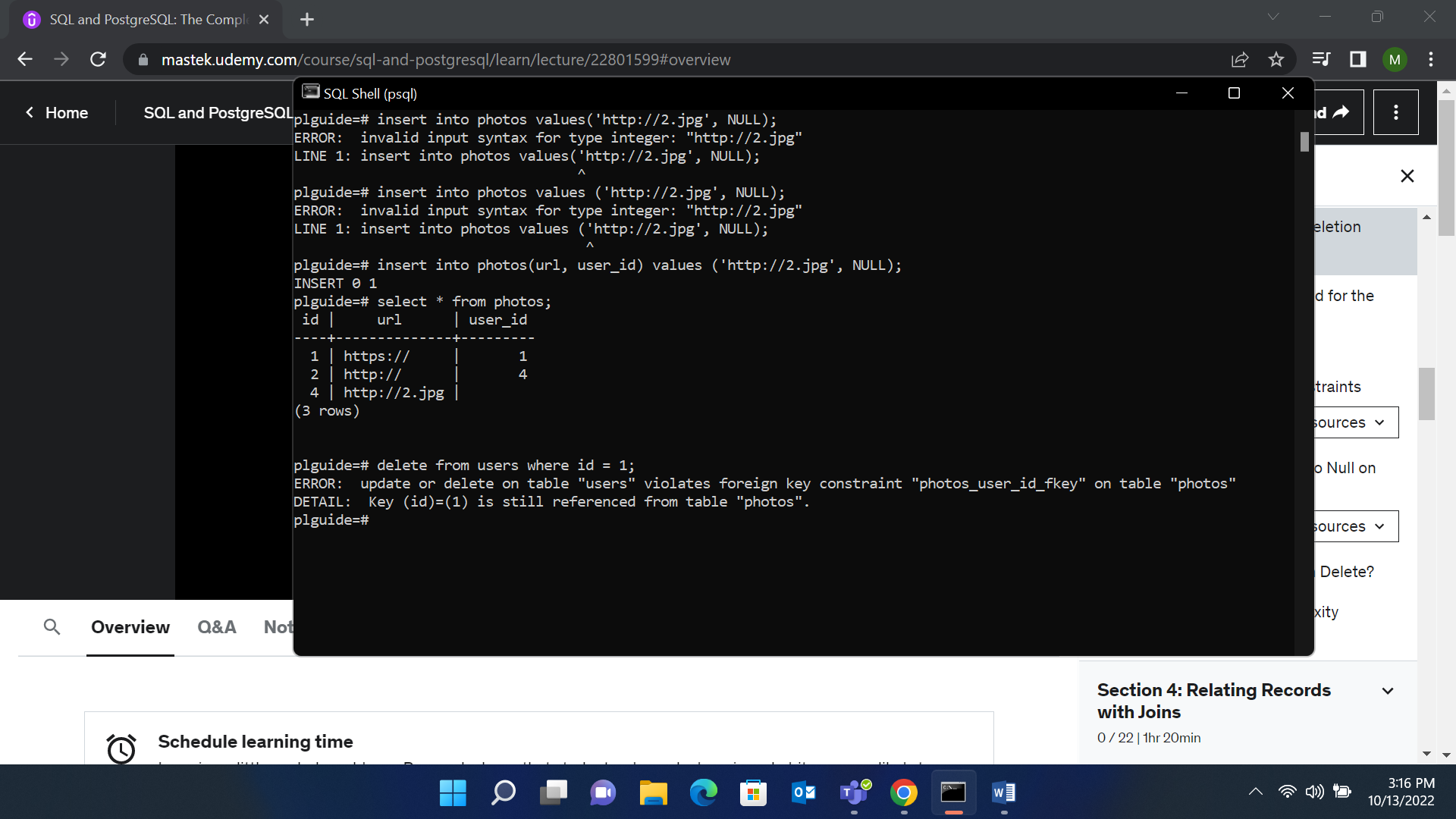
Two tables: Users and photos

Photos has foreign key referenced to the users.

Users(parent table)

Photos (child table) containing foreign key referenced to users.

**ON DELETE OPTIONS:**

1. On delete restrict: 
2. On delete cascade: It means if we delete a record in parent table then the record dependent on it in child table will also get deleted.

e.g

1. CREATE TABLE photos (
2. id SERIAL PRIMARY KEY,
3. url VARCHAR(200),
4. user\_id INTEGER REFERENCES users(id) **ON DELETE CASCADE**
5. );

1. On delete set null 🡪 Set the value of the foreign key column to NULL in the child table (photos).

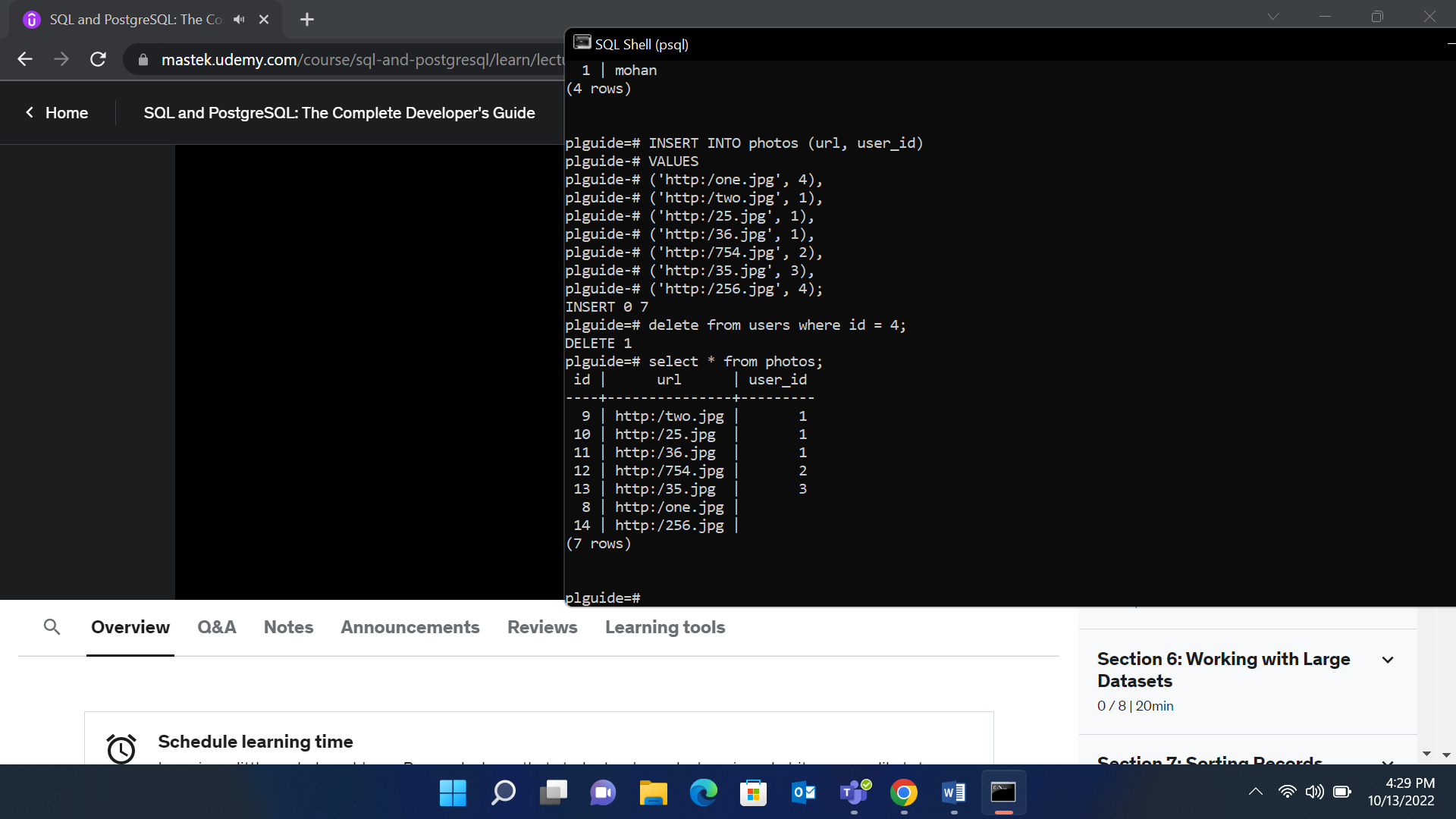
CREATE TABLE photos (

id SERIAL PRIMARY KEY,

url VARCHAR(200),

user\_id INTEGER REFERENCES users(id) **on delete set NULL**

);



1. On delete set default 🡪 It will set the default value for records in child table that are referenced to records deleted.

**Select 🡪** Used to retrieve records from the table.

**LIKE and ILIKE**

Like 🡪 It is used for the purpose of the pattern matching of the string data with the wildcard bcharacters(\*, \_)

Like is case sensitive.

ILike is case insensitive.

% 🡪 matches any sequence of characters (0 to many).

/ 🡪 matches any single characters (only one).

e.g

names begin with ‘A’ : A%

names end with ‘A’: %A

**NOT LIKE 🡪** use not like when you want records not matching the pattern.

**Comparison math operations**

= 🡪 check equality

<, >, != / <>, <=, >=, between

IN 🡪 check if value is in the list.

select name, area from cities where name in ('delhi', 'Shanghai');

NOT IN 🡪 check if values not in list.

**Commands**

\l 🡪 number of databases.

\dt show tables in database.

\c <<db\_name>> 🡪 connect to database/ switch to database.

Pg\_dump 🡪 it is used to create back file of the databases.

------------------------ day 4 -------------------------

**Aggregate functions**

AVG() 🡪 returns single floating point value which is average value of specified column with many decimal places.

\*\* Use ROUND to specify precision after the decimal place.

**OFFSET 🡪** it is used to skip the first (n-1) records in the table.

select first\_name from customer limit 10 offset 10;

**Fetch Keyword 🡪** It is equivalent to limit keyword.

Subquerry Example

select title from film where film\_id in (

select film\_id from film\_category where category\_id in (6, 11, 15)

);

Get Second Max

(select max(amount) from payment where amount< (select max(amount) from payment)))

**Group By Clause**

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

Columns associated by the aggregate functions must not be included in “group by”.

Syntax:

Select category\_col, agg(col) from table\_name group by category\_col;

Group by clause must appear after “from” or “where” statement.

select customer\_id,sum(amount) from payment group by customer\_id order by sum(amount) desc;

error:

select staff\_id, customer\_id, sum(amount) from payment

group by staff\_id

order by staff\_id, customer\_id;

**here the error is cutomer\_id must come in either group\_by clause or in must be aggregate function.**

Question: We have two staff members, with staff id 1 and 2. We want to give a bonus to those members which handled most payments.(most in sense of number of the payments processed.)

Ans:

select staff\_id, count(payment\_id) from payment group by staff\_id;

corporate HQ is conducting the result on relationship between replacement cost and a movie MPAA rating (e.g G, PG, R, etc …)

select staff\_id, count(payment\_id) from payment group by staff\_id;

what are the customer ids of the top 5 customers by total spend?

select customer\_id, sum(amount) from payment group by customer\_id order by sum(amount) desc limit 5;

**Having Clause**

**Having 🡪** It is used for specifying conditions in query with ‘aggregate functions’ as we cannot use aggregate functions with ‘where clause’.

e.g

select customer\_id, sum(amount) from payment

where customer\_id not in (184, 87, 477)

group by customer\_id

having sum(amount)>100;

Having challenge section 3

select customer\_id, count(payment\_id) from payment group by customer\_id having count(payment\_id)>= 40;

question: return customer id of the customers who have spend more than 100 dollars with staff id 2

answer:

select customer\_id, sum(amount) from payment where staff\_id = 2 group by customer\_id having sum(amount)>=100;

Section 4:

1. Return the customer IDs of customers who have spent at least $110 with the staff member who has an ID of 2.

select customer\_id, sum(amount) from payment where staff\_id = 2 group by customer\_id having sum(amount)>110;

 How many films begin with the letter J?

select count(title) from film where title like 'J%';

**JOINS**

AS operator 🡪 ‘AS’ operator is used to create an Alias(different name for the column) of the other column.

As operator gets executed at the end of the query means that alias wont work in ‘where’ clause.

Joins 🡪 to combine the multiple tables on the basis of common column.

/ Queries that can be executed to retrieve data from multiple tables.

Types:

1. Inner joins

Inner join returns the set of the record that matches in both tables.

Syntax:

Select \* from tableA inner join tableB on tableA.col\_match = tableB.col\_match;

1. Natural Join
2. Left Outer Join
3. Right Outer Join
4. Full Join
5. Self Join

10/17/22

-- display detail of customers did the payment customer\_id, customer\_name, payment\_amount, date using join --

select customer.customer\_id, concat(first\_name,' ',last\_name) as name, sum(amount)

from customer

inner join payment

on customer.customer\_id = payment.customer\_id

GROUP by customer.customer\_id

order by customer.customer\_id ASC;

select customer\_id, sum(amount) as total

from payment

GROUP by customer\_id

having sum(amount)<100

order by total;

select sum(amount), max(amount), min(amount) from payment;

select customer\_id, amount, payment\_date

from payment

order by customer\_id asc, payment\_date desc;

-- display film\_id, category id, title, release date, language id --

select film.film\_id, category\_id, title, release\_year, language\_id

from film\_category

inner join film

on film\_category.film\_id = film.film\_id;

display

select customer.customer\_id,

concat(customer.first\_name,' ',customer.last\_name) as customer\_name,

payment.amount, concat(staff.first\_name, ' ', staff.last\_name) as staff\_name

from customer

inner join payment

on customer.customer\_id = payment.customer\_id

inner join staff

on payment.staff\_id = staff.staff\_id;

-- display film and actors name --

select t1.title, t3.first\_name || ' ' || t3.last\_name as actor\_name

from film as t1

inner join film\_actor as t2

on t1.film\_id = t2.film\_id

inner join actor as t3

on t2.actor\_id = t3.actor\_id;

Can a unique column have single NULL or multiple NULL columns?

create table dept(

id int,

dname varchar(50) UNIQUE,

location varchar(50),

primary key(id)

);

//add the constraint

alter table dept

add constraint dname\_unique\_const UNIQUE(dname);

//drop the constraint

alter TABLE dept

drop constraint dname\_unique\_const;

//create table department and insert records

insert

into dept

(id, dname, location)

VALUES

(200, 'Admin', 'India'),

(300,'HR','India'),

(400, 'DCT', 'USA');

select \* from dept;

**-- left join --**

**-- display all the employees which are assigned and not assigned with department --**

select t1.id, t1.first\_name, t1.dept\_id, t2.dname

from employees as t1

left join dept as t2

on t1.dept\_id = t2.id;

To display all departments with employees and without employees

select t1.id, t1.first\_name || ' ' || t1.last\_name, t2.id, t2.dname from

employees as t1

right outer join dept as t2

on t1.dept\_id = t2.id;

Query - Display all employees with no department assigned yet as well as all depts with no employees

select t1.id, t1.first\_name || ' ' || t1.last\_name, t2.id, t2.dname

from employees as t1

full join dept as t2

on t1.dept\_id = t2.id;

Union 🡪 combine the result sets of the multiple queries in a single set.

**Advanced SQL**

Datatype related to time and date

TIME 🡪 store only time

DATE 🡪 store only date

TIMESTAMP 🡪 store date and time

TIMESTAMPZ 🡪 store date, time and timezone.

We can remove the historical information but cannot add.

Functions and operation related to time.

Timezone

Now

Timeofday

Current\_time

Current\_day

Select now()

Select timeofday();

Extract information using time based datatypes.

Extract()

Age()

To\_char()

Extract 🡪 It is used to extract/obtain the sub component of the date value like year, month, day, week, quarter.

Syntax:

Extract(year from date\_col)

Example:

select distinct(extract(year from payment\_date)) from payment;

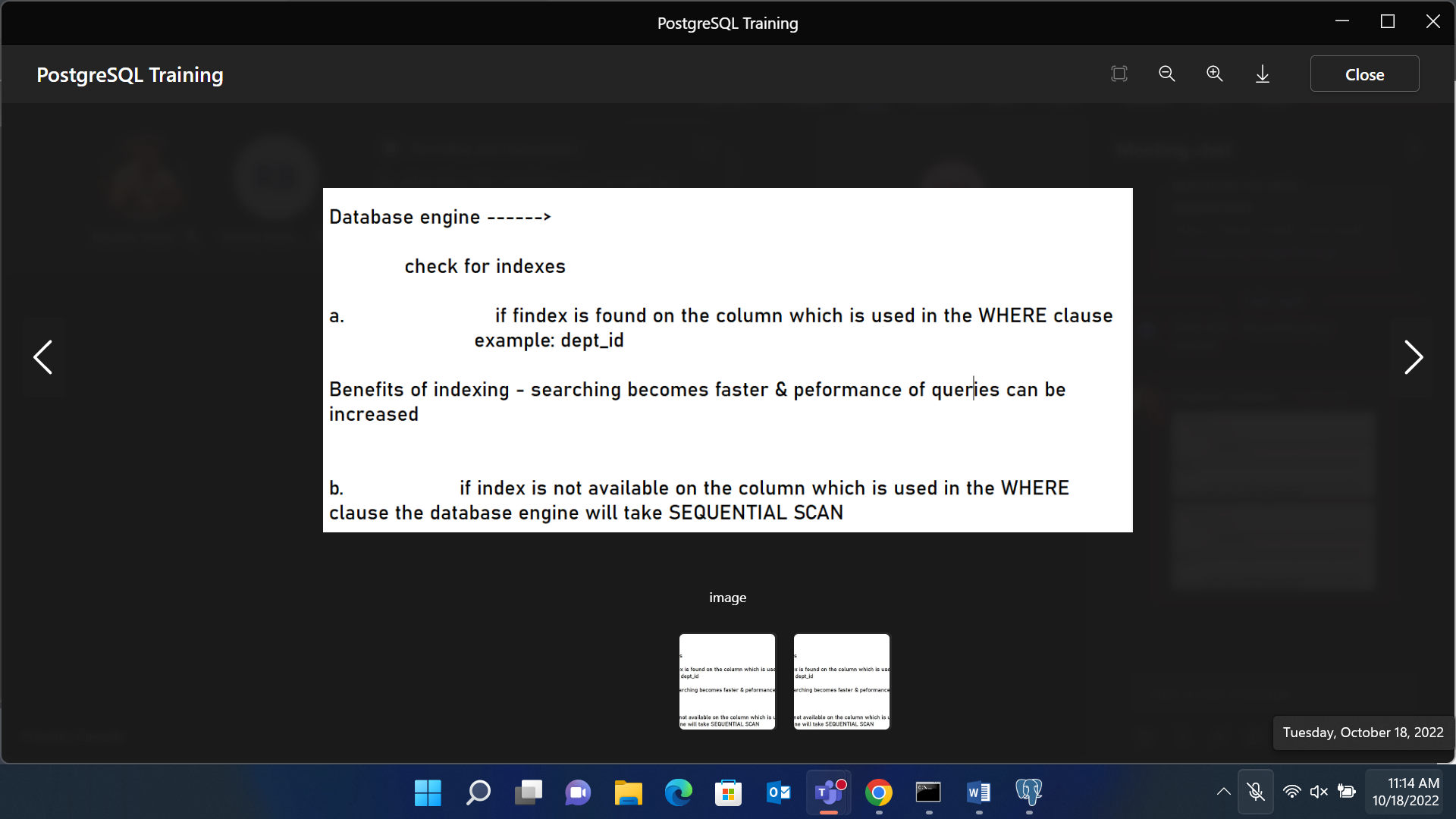
Age 🡪 calculate and returns the current age given a timestamp.

Age(data\_col)

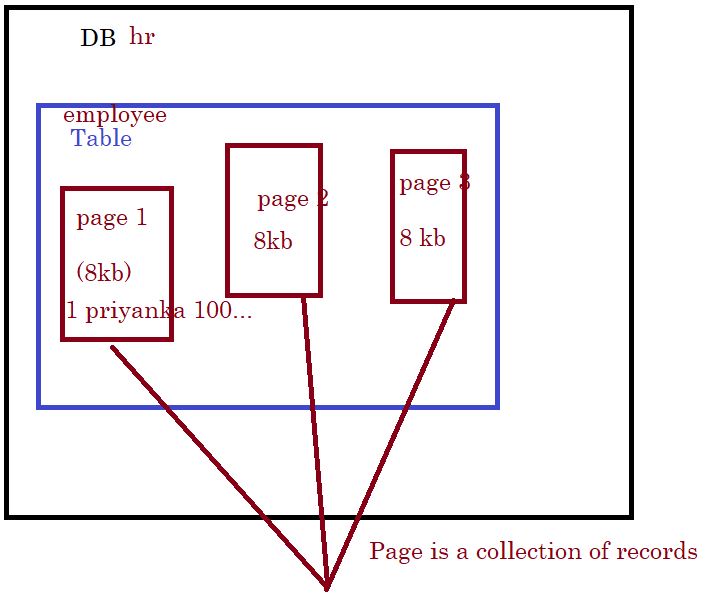
10/ 18/ 22

Indexes

Why indexes are important?



Indexes are created by default if the column has constraints unique or primary key.



explain verbose select \* from address where phone = '991562402283';

explain (format json) select \* from address where phone = '991562402283';

QUERY PLAN

drop index if exists idx\_address\_phone;

**pg\_indexes view is used to show index details.**

select

tablename, indexname, indexdef

from pg\_indexes

where schemaname = 'public'

order by

tablename, indexname;

**PSQL**

Sequence 🡪 It is user-defined schema-bound object that yields sequence of the integers based on the specified condition.

e.g

Create sequence:

**Syntax:**

CREATE SEQUENCE [ IF NOT EXISTS ] sequence\_name

[ AS { SMALLINT | INT | BIGINT } ]

[ INCREMENT [ BY ] increment ]

[ MINVALUE minvalue | NO MINVALUE ]

[ MAXVALUE maxvalue | NO MAXVALUE ]

[ START [ WITH ] start ]

[ CACHE cache ]

[ [ NO ] CYCLE ]

[ OWNED BY { table\_name.column\_name | NONE } ]

The CYCLE allows you to restart the value if the limit is reached. The next number will be the minimum value for the ascending sequence and maximum value for the descending sequence. If you use NO CYCLE, when the limit is reached, attempting to get the next value will result in an error. The NO CYCLE is the default if you don’t explicitly specify CYCLE or NO CYCLE.

e.g

CREATE SEQUENCE mysequence

INCREMENT 5

START 10;

select nextval('inc\_5');

it will give o/p 10 and as we implement result will be incremented by 5.

Lastval() 🡪 most recent nextval();

e.g

select lastval ();

create sequence test\_cycle

increment 50

MINVALUE 100

MAXVALUE 5000

start 100;

sequence starts from 23 and increment 7.

create sequence test\_sequence\_4

increment 7

start 23;

Attach sequence to a field

Create sequence sequence\_name OWNED BY table\_name.col\_name;

alter table employees

alter column employeeid set default nextval('employees\_employeesid\_seq');

create sequence and bind to order 🡪 orderid

alter table orders

alter column orderid set default nextval('seq\_order\_orderid');

**Alter the sequence**

1. ALTER SEQUENCE sequence\_name RESTART number;

insert into orders

(customerid, employeeid, requireddate, shippeddate)

VALUES

('VINET', '1', '1900-05-03', '1990-03-05');

1. ALTER SEQUENCE sequence\_name RENAME new\_name;

**Drop the sequence**

Drop sequence sequence\_name;

**SERIALS**

Increment the value automatically.

1. bigserial 🡪 increment big int
2. serial 🡪 increment int.
3. smallserial 🡪 increment small int.

create table\_name(

exid serial;)

is equivalent to

create sequence sequence\_name;

create table table\_name(

exid int set default nextval(‘sequence\_name’);

);

Alter sequence\_name OWNED BY table\_name.col\_name;

**# Views**

The views are the virtual tables that are based on the result set of the query.

Complex queries can be stored inside the view.

We can view the number of views using informaiton\_schema.views;

Syntax:

select \* from information\_schema.views;

Syntax:

Create view view\_name AS SELECT statement;

Modify Views

Add column in the views.

Change name of the view

ALTER VIEW view\_name RENAME TO new\_view\_name;

Creating updatable views.

create view protein\_products AS

select \* from products

where categoryid in (4,6,8);

select \* from protein\_products;

insert into protein\_products

(productid, productname, supplierid, categoryid, discontinued)

values(78, 'Kobe', 12, 8, 0);

update protein\_products

set unitprice = 55

where productid = 78;

select \* from protein\_products;

delete from protein\_products where productid = 78;

Insertion or updating in the views table will also reflect the changes in the base table (real table).

**WITH CHECK**

‘With check option’ prevents user from inserting/ updating the wrong record by specifying the condition.

Two types:

1. with local check option

create or replace view protein\_products AS

select \* from products

where categoryid in (4,6,8)

with local check option;

now we cannot insert any other records other than categoryid 4, 6, 8

e.g this will not get inserted as categoryid is 5

insert into protein\_products (productid, productname, supplierid,

categoryid, discontinued)

values (79, 'tasty tea', 12, 5, 0);

1. with cascade check option

**Conditional Expressions**

1. CASE WHEN

select companyname, country,

case

when country in ('Austria', 'Germany', 'Poland') THEN 'Europe'

when country in ('Mexico', 'USA', 'Canada') then 'North America'

when country in ('Brazil', 'Venezuala', 'Argentina') then 'South America'

else 'unknown'

end as continent

from customers;

**COALESCE**

It can have number of the arguments but it returns the first non-null value.

It is often used to replace the null value with some other value.

Coalesce(feild1, feild2, …)

e.g

select companyname, COALESCE(homepage, 'call to find')

from suppliers;

**NULLIF**

NULLIF returns null if two arguments are same or else it returns argument1.

Syntax: nullif(argument1, argument2)

select companyname, phone,

coalesce(nullif(homepage, ''), 'need to call')

from suppliers;

e.g 2

return fax number and if not available then phone numbers.

select coalesce(nullif(fax, phone), 'no info') from customers;

10/20/22

PSQL 🡪 provides programming language capabilities to the sql.

Block of code will reduce network overhead.

Procedure 🡪 may or may not return the result

Function 🡪 will return the value.

Triggers 🡪 executed when DML commands are executed on table or views.

Structure of PL/pgSQL in postgresql

[<<label>>]

Declare

Variable declarations …..; //declare to begin is different scope

Begin

….statements; //begin to end is the different scope

End <<label>>;

Valid datatypes that can be used are

Integer

Smallint

bigint

char

Varchar

Numeric(precision, scale)

e.g 1

do $$

DECLARE

x integer := 10;

begin

SELECT count(x) into x from employees;

raise notice 'value of x: %',x;

end $$;

e.g 2

do $$

DECLARE

x integer = 0;

y integer;

begin

raise notice 'Value y: %',y;

y = 400;

raise notice 'value y: % ', y;

end $$;

e.g 3

Here to get the type of the column and assign it to the variable use syntax

Variable\_name table\_name.column\_name%type := 0;

do $$

<<employee\_details>>

DECLARE

employeeFirstName employees.first\_name%type;

employee\_id employees.id%type = 78;

begin

select first\_name into employeeFirstName

from employees

where employee\_id = 78;

raise notice 'employee 78 name is %', employeeFirstName;

end employee\_details

Outer block

x-> variable

----------------

Inner block

x->variable (local to inner block)

-------------

End inner block

End outer block

To access the variable of the outer block use: outer\_block\_name.variable\_name

E.g

do $$

<<outer\_block>> //outer block name

DECLARE

x integer = 10;

begin

raise notice 'Value of outer x is %', x;

declare

x integer = 40;

begin

raise notice 'Value of x inner block is %', x;

x = outer\_block.x+10;

raise notice 'value of x inner block after addition %', x;

end;

end outer\_block $$;

do $$

declare

x employees%rowtype;

begin

SELECT \* INTO x FROM employees

WHERE id = 1;

-- now we get complete row in the row variable

-- print the row

RAISE NOTICE 'The employee first name % and last name are % ' , x.first\_name, x.last\_name;

end; $$

**Data types In PostgreSQL**

1. Timestamp (both date and time)

Specify precision of seconds value

1. Timestamp(3)

1/8/2017 12:22:22.231

1. Time(4)

12:22:22.1234

The time can be from 0 to 6

1. We can allow or not allow the timezones. By default it is no time zone

Timestamp with time zone

Timestamp without time zone

Time with time zone

Time without time zone

Inputting Dates

PostgreSQL accepting date format

1. ‘2017-01-10’
2. ‘January 8, 2018’
3. ‘20180108’

Setting date style

DateStyle – ISO, Postgres, SQL or German followed by DMY, MDY, or YMD

e.g

Set DateStyle = ‘ISO, MDY’

Set/see the time zone for the session

Show time zone;

**Inputting Time**

Types:

1. 04:05:06 – ISO 8601
2. 04:05 AM
3. 04:05 PM (internally stored as 16:05)
4. 04:05:06.789

**Inputting Timestamp**

Concatenating date and time followed by optional timezone

E,g 2018-01-08 04:05:06

**Time Zones**

Format of writing time zones.

1. Full Zone Name: America/Los Angeles

**Conditional Control Statement**

Syntax:

If <<condition>>

Then

Statement

Endif;

Example

do $$

DECLARE

count integer = 1;

begin

if count>10

THEN

raise notice 'big';

end if;

if count<10

then

raise notice 'less than then';

end if;

end $$;

If … else … example

do $$

DECLARE

count integer = 1;

begin

if count>10

THEN

raise notice 'greater than 10';

ELSE

raise notice 'less than 10';

end if;

end $$;

example

film with film id 100 exist or not

do $$

DECLARE

row film%rowtype;

begin

select \* into row from film where film\_id = 100;

if row is not NULL

then

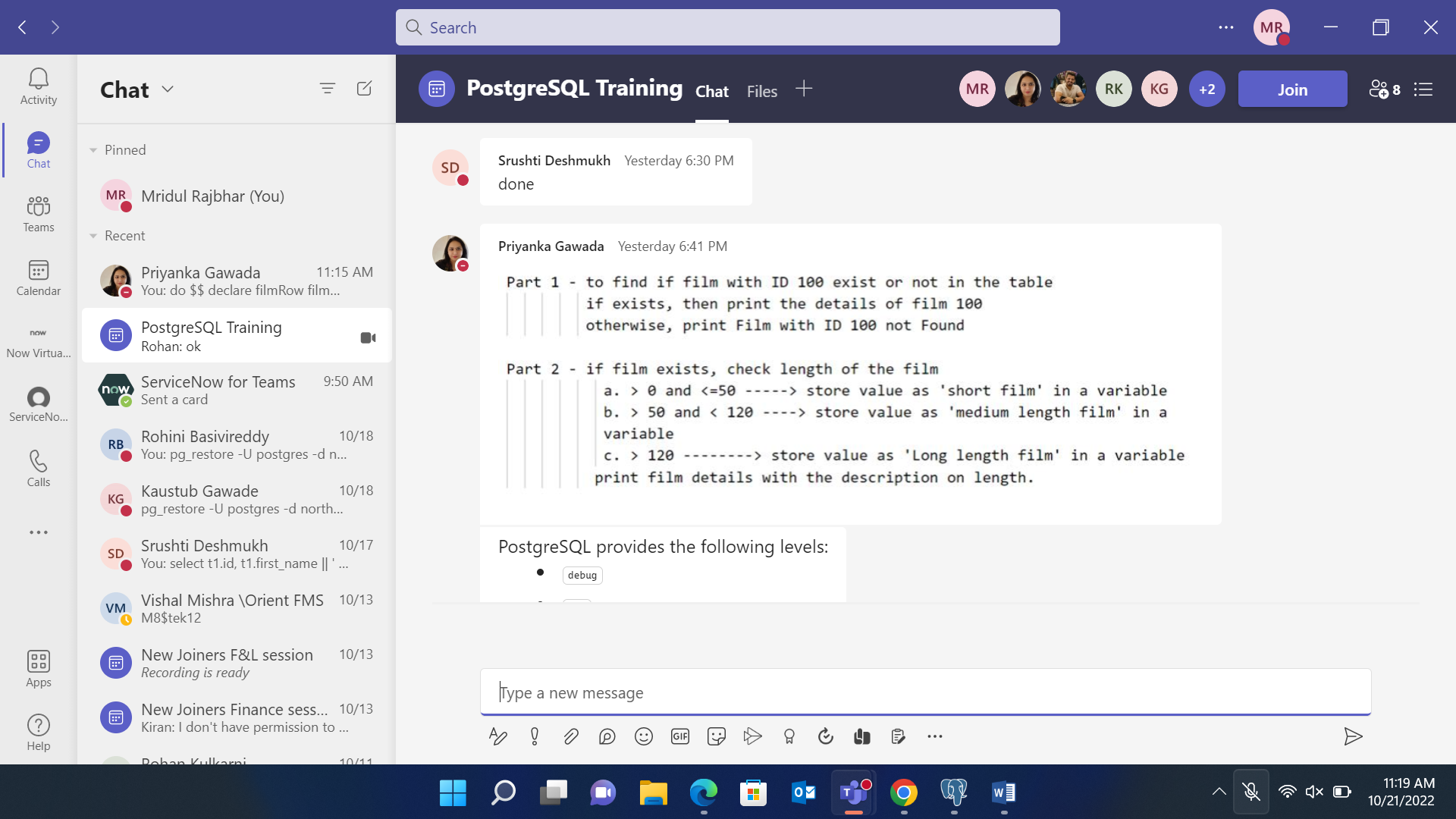
raise notice 'details are %', row;

else

raise notice 'film not found';

end if;

end $$;



do $$

declare

filmRow film%rowtype;

filmID film.film\_id%type = 20000;

filmLength film.length%type;pl\_

filmType varchar(30);

begin

select \* into filmRow from film where film.film\_id = filmID;

filmLength = filmRow.length;

if filmLength is NULL

then

raise exception 'film with id % not found', filmID;

elsif filmLength >0 and filmLength<=50

THEN

filmType = 'short film';

elsif filmLength > 50 and filmLength < 120

THEN

filmType = 'medium film';

ELSif filmLength > 120

THEN

filmType = 'long film';

end if;

raise notice '% and length is % ', filmType, filmLength;

end; $$

Procedures in PSQL

Syntax:

Create or replace procedure procedure\_name(parameter 1, parameter 2, …..)

Language pl\_pgsql

as $$

begin

end $$

Example

create or REPLACE procedure payment\_amount\_segment(paymentID payment.payment\_id%type)

language plpgsql

as $$

declare

paymentSegment varchar(20);

paymentRow payment%rowtype;

paymentAmount payment.amount%type;

BEGIN

select \* into paymentRow from payment where payment\_id = paymentID;

if paymentRow is not null then

paymentAmount = paymentRow.amount;

case paymentAmount

when 0.99

THEN

paymentSegment = 'Mass payment';

when 2.99

THEN

paymentSegment = 'Mainstream payment';

when 4.99

THEN

paymentSegment = 'High payment';

ELSE

paymentSegment = 'Specific payment';

end case;

raise notice 'PaymentID: % Payment amount: % and Payment Segment: %', paymentID, paymentAmount, paymentSegment;

else

raise exception 'Payment with paymentid % not found', paymentID;

end if;

end; $$

**In postgres switch case there is no BREAK statement.**

Question: create procedure and display ‘hello world’ 5 times.

create or replace PROCEDURE display\_hello(count integer)

language plpgsql

as $$

DECLARE

tempInt integer=0;

begin

loop

raise info 'hello world';

tempInt= tempInt+1;

exit when tempInt = count;

end loop;

end; $$

For Loop Syntax

For <<counter\_variable\_name>> in [reverse] [start]..[end] [by stepping]

Loop

statements

End loop;

Reverse For Loop

create or replace procedure reverse\_hello\_world\_for(count integer)

LANGUAGE plpgsql

as $$

DECLARE

begin

for counter in reverse count..1 by 1

loop

raise notice 'hello world: %', counter;

end loop;

end; $$

Function

Syntax:

Create function function\_name(parameters, ….)

Langage plpgsql

RETURNS datatype

AS $$

DECLARE

BEGIN

RETURN

END

Example:

create FUNCTION func\_sum\_of\_numbers(num1 integer, num2 integer)

returns integer

language plpgsql

AS $$

declare

sum integer = 0;

BEGIN

raise notice 'Sum is ';

sum = num1+num2;

return sum;

end; $$

select \* from func\_sum\_of\_numbers(5, 5); // To execute the function

**Interval Date Type**

**SQL Standard Formats**

‘4 32:12:10’ 🡪 4days 32 hours 12 minutes 10 seconds

‘200-10’ 🡪 200 years 10 months

‘1-2’ 🡪 1year and 2 months

insert into test\_time (span) values ('20-5 32:11:33');

// 20 years 5 months 32 hours 11 minutes 33 seconds

‘5 DECADES 3 YEARS 5 MONTHS 5 DAYS’

**ISO 8061 format**

Add ‘P’ at the start.

‘P53Y6M3D’ 🡪 53 years, 6 months, 3 days

**Date Arithmetic**

Add

select TIMESTAMP '2018-09-28 13:22:44' - interval '80 minutes 10 seconds';

subtract

To get days (days – days)

select date '2016-12-30' - date '2009-04-07';

subtract Time or timestamps

select timestamp '2016-12-30 20:11:33' - time '11:44:22';

//2016-12-30 08:27:11

Multiplication

select 5 \* interval '7 hours 5 minutes';

//PT35H25M 🡪 35:25:00

Division

select interval '5 hours 2 minutes'/3;

//1:40:40

**Age Funciton**

age between two times mentioned

select age(timestamp '2025-10-03', timestamp '1990-10-05');

// +34-11 +29 +0:00:00 🡪 34 years 11 months 29 days

Age between time mentioned till current time