Function and Procedure, Trigger, Partition Table

Presented by Sanam Maharjan

Functions

01 What ?

1. Logical database object or Schema object
2. Reusable block of code

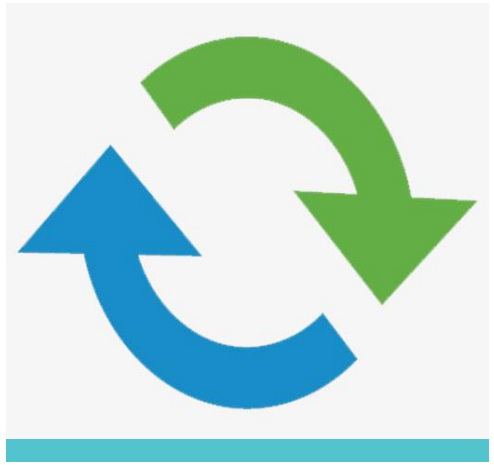
02 When ?

1. For repeated execution
2. To bind the logic of query
3. To compute a values

03 Where ?

Building applications

Built in Stored Functions



Conversion

Function

1. CAST (expression AS target_type);
2. column_name :: datatype
3. to_char()
4. to_number()

NVL

Function

1. COALESE (argument1, argument2,…);
2. NULLIF(argument_1,argument_2);

Character

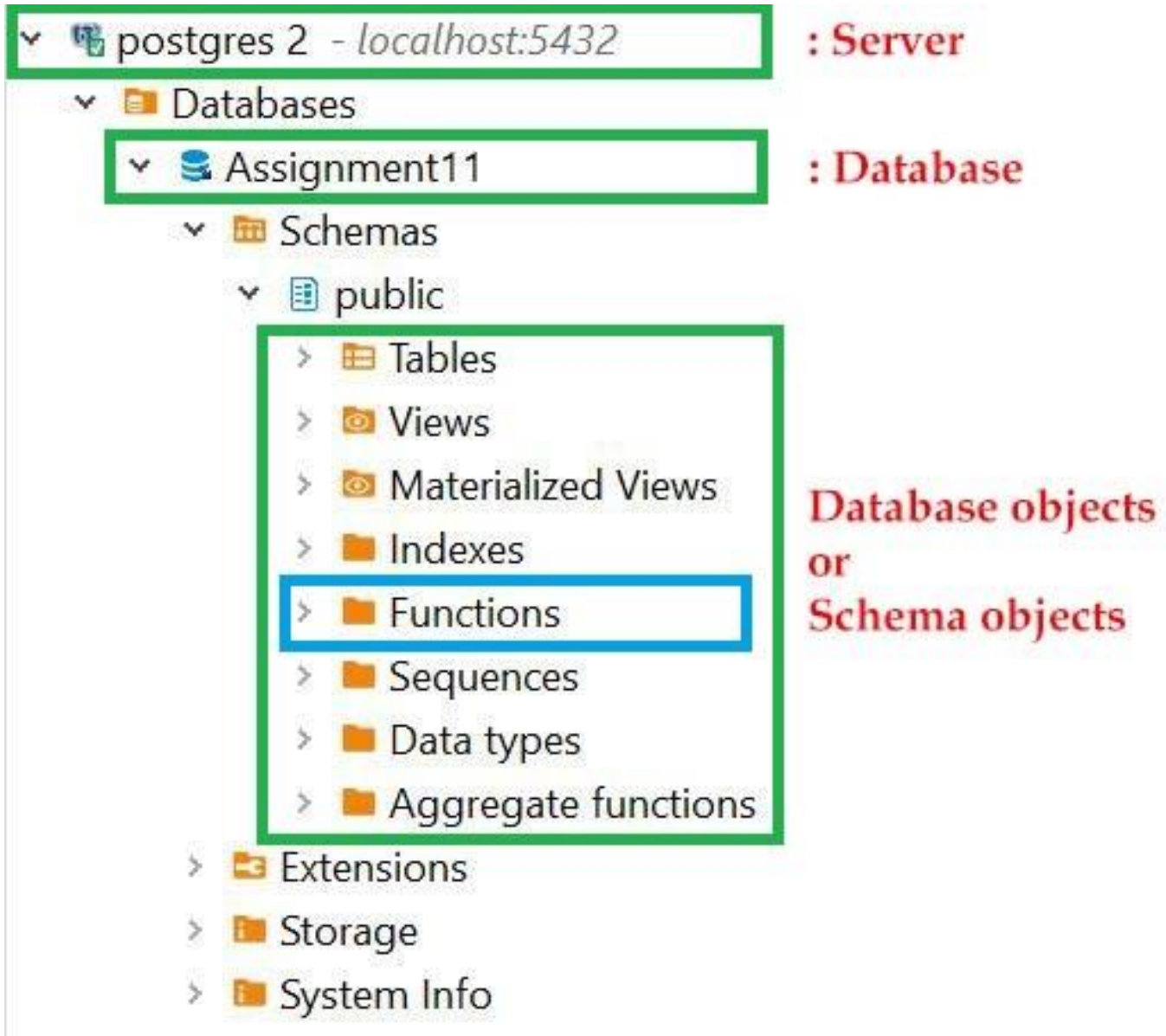
Function

1. Lower()
2. Concat()
3. Upper()
4. Trim()

Aggregate

Function

1. Min()
2. Max()
3. Count()
4. Sum()
5. Avg()



Basic Syntax

```
create [or replace] function function_name
            (param_list)
returns return_type
language plpgsql
as
$$
declare
            -- variable declaration
begin
            -- logic
end; $$
```

PL/pgSQL parameter modes

IN	**OUT**	**INOUT**
The default	Explicitly specified	Explicitly specified
Pass a value to function	Return a value from a function	Pass a value to a function and return an updated value.
In parameters act like constants	out parameters act like uninitialized variables	inout parameters act like an initialized variables
Cannot be assigned a value	Must assign a value	Should be assigned a value

```
create or replace function emp_name_by_id(id int)
returns varchar
language plpgsql
as $$

declare
e_name varchar;

begin
select concat(first_name,' ',last_name)::varchar
into e_name
from employees
where employee_id = id;

if not found then
raise 'Employee with id % not found',id;
end if;

return e_name;
end;
$$
```

IN id int (by default)

1. Declare section

2. Execution section

3. Exceptional Handling section

OUT mode

```
create or replace function emp_name_by_id
        (id int,
        out e_name varchar)
language plpgsql
as $$
begin
select concat(first_name,' ',last_name)
        ::varchar
into e_name
from employees
where employee_id = id;

if not found then
raise 'Employee with id % not found',id;
end if;
end; $$
```

- Returned back as a part of the result.
- Very useful in functions that need to return multiple values.

```
--Returning multiple values
create or replace function get_sal_stat(
out max_salary int,
out min_salaray int,
out avg_salary int)
language plpgsql
as $$
begin
select max(salary), min(salary), avg(salary)
into max_salary, min_salaray, avg_salary
from employees;
end; $$
```

INOUT mode

```
create or replace function swap(
        inout x int,
        inout y int
        )
language plpgsql
as $$
begin
   select x,y into y,x;
end; $$;

select * from swap(10,20);
```

- The inout mode is the combination in and out modes.
- It means that the caller can pass an argument to a function.
- The function changes the argument and returns the updated value.

Calling a user-defined function

Consider we have function:

```
create or replace function get_emp_count(
        id_from int,
        id_to int)
returns int
language plpgsql
As $$
declare
   id_count integer;
begin
   select count(*)
   into id_count
   from employees
   where employee_id between
                        id_from and id_to;
   return id_count;
end;$$
```

Then we can call function in 3 ways.

1. Using positional notation:

```
select get_emp_count(100,200);
```

2. Using named notation:

```
select get_emp_count(id_from=>100,
                     id_to=>200);
```

3. Using the mixed notation:

```
select get_emp_count(100, id_to=>200);
```

Optimization

- ✓ We can see the cost of query with **Explain** Keyword provided by PSQL.
- ✓ Consider we have following query:

```
select e.employee_id,
       e.first_name,
       d.department_name,
       l.city,
       c.country_name,
       r.region_name
from employees e
join departments d on e.department_id = d.department_id
join locations l on d.location_id = l.location_id
join countries c on c.country_id = l.country_id
join regions r on r.region_id = c.region_id
where e.employee_id = 200;
```

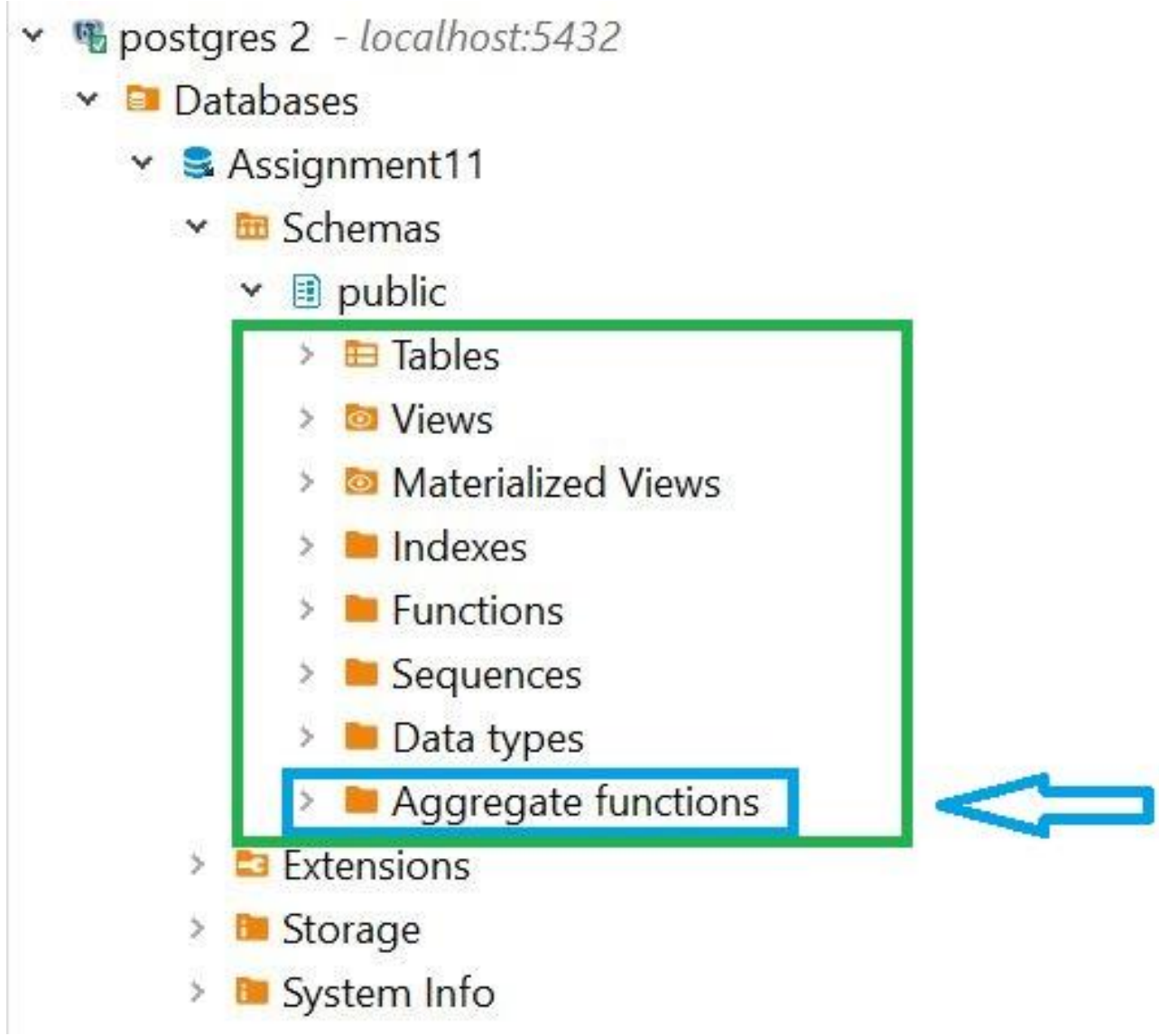
```
EXPLAIN
select e.employee_id,
       e.first_name,
       d.department_name,
       l.city,
       c.country_name,
       r.region_name
from employees e
join departments d
on e.department_id = d.department_id
join locations l
on d.location_id = l.location_id
join countries c
on c.country_id = l.country_id
join regions r
on r.region_id = c.region_id
where e.employee_id = 200;
```

	QUERY PLAN
1	Nested Loop (cost=3.76..6.88 rows=1 width=332)
2	-> Nested Loop (cost=3.63..6.06 rows=1 width=268)
3	-> Nested Loop (cost=3.49..5.34 rows=1 width=178)
4	-> Hash Join (cost=3.35..4.70 rows=1 width=92)
5	Hash Cond: (d.department_id = e.department_id)
6	-> Seq Scan on departments d (cost=0.00..1.27 rows=27 width=86
7	-> Hash (cost=3.34..3.34 rows=1 width=14)
8	-> Seq Scan on employees e (cost=0.00..3.34 rows=1 width=14
9	Filter: (employee_id = 200)
10	-> Index Scan using locations_pkey on locations l (cost=0.14..0.60 rows
11	Index Cond: (location_id = d.location_id)
12	-> Index Scan using countries_pkey on countries c (cost=0.14..0.68 rows=1
13	Index Cond: (country_id = l.country_id)
14	-> Index Scan using regions_pkey on regions r (cost=0.13..0.63 rows=1 width=
15	Index Cond: (region_id = c.region_id)

```
create or replace function hr_select(id int)
returns table (employee_id int, first_name varchar,
               department_name varchar, city varchar,
               country_name varchar,
               region_name varchar)
language plpgsql
as $$
declare
        emp_id int;
begin
emp_id = id;
return query
select e.employee_id, e.first_name, d.department_name,
l.city,c.country_name,r.region_name
from employees e
join departments d on e.department_id = d.department_id
join locations l on d.location_id = l.location_id
join countries c on c.country_id = l.country_id
join regions r on r.region_id = c.region_id
where e.employee_id = emp_id;
end; $$
```

```
explain
select * from hr_select(200);
```

	QUERY PLAN
1	Function Scan on hr_select (cost=0.25..10.25 rows=1000 width=164)



Aggregate Function

- For creating our own custom aggregates
- To perform conditional counts

Step 1: Creating a Function

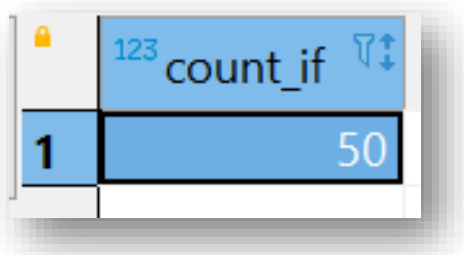
```
CREATE OR REPLACE FUNCTION countif_add(
        current_count int,
        expression bool)
RETURNS int
AS
$BODY$
    SELECT CASE expression
    WHEN true THEN
       current_count + 1
    ELSE
       current_count
END;
$BODY$

LANGUAGE SQL IMMUTABLE;
```

Step 2: Creating a Aggregate

```
CREATE AGGREGATE count_if (boolean)
(
  sfunc = countif_add,
  stype = int,
  initcond = 0
);
```

Step 3: Quering with aggregate

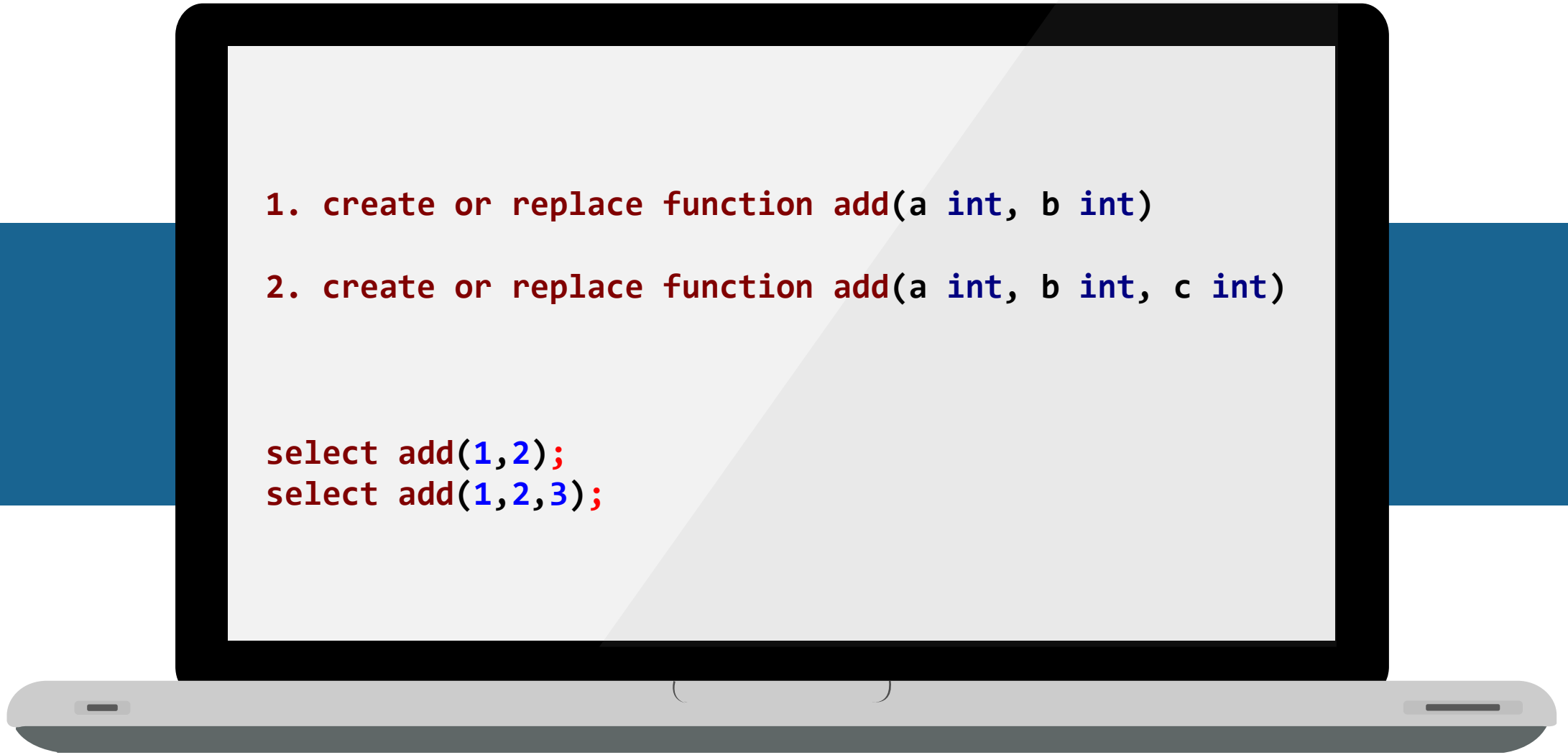
```
SELECT count_if(employee_id  < 150)
FROM employees;
```



Function Stability Categories

1. VOLATILE
2. STABLE
3. IMMUTABLE

Function Overloading

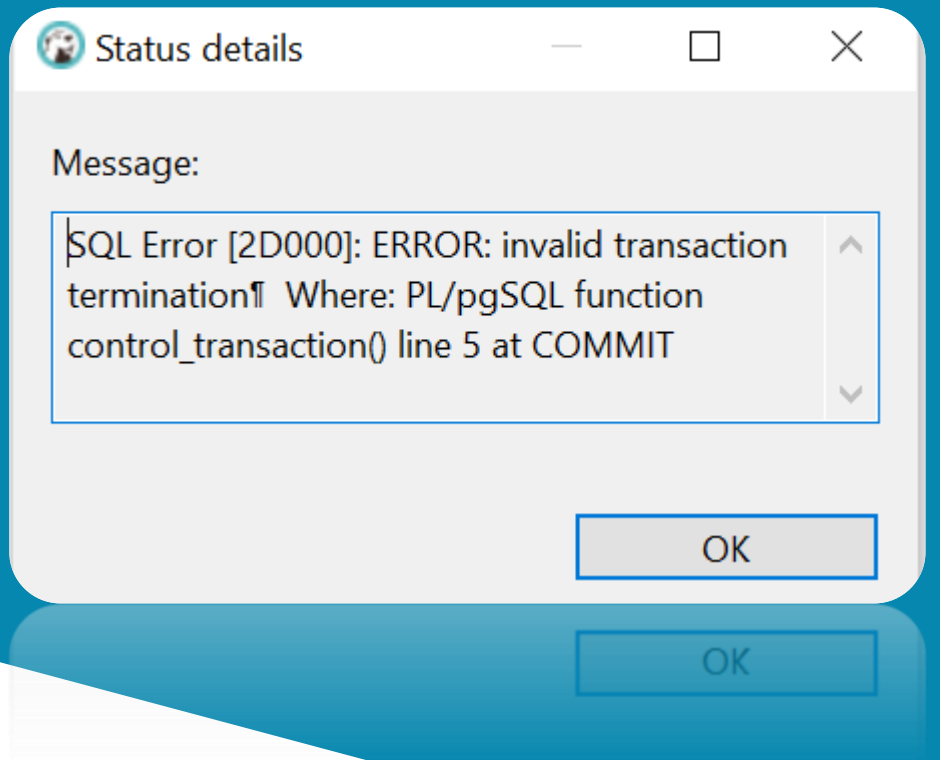


Limitation of Function

- We cannot start a transaction, and commit or rollback
  i.e we cannot execute transaction

Transaction in Function

```
CREATE OR REPLACE function control_transaction()
returns boolean
LANGUAGE plpgsql
AS $$
BEGIN
    CREATE TABLE test1 (id int);
    INSERT INTO test1 VALUES (1);
    COMMIT;
    CREATE TABLE test2 (id int);
    INSERT INTO test2 VALUES (1);
    ROLLBACK;
    return true;
END $$;

select control_transaction();
```



Procedure

01 What ?

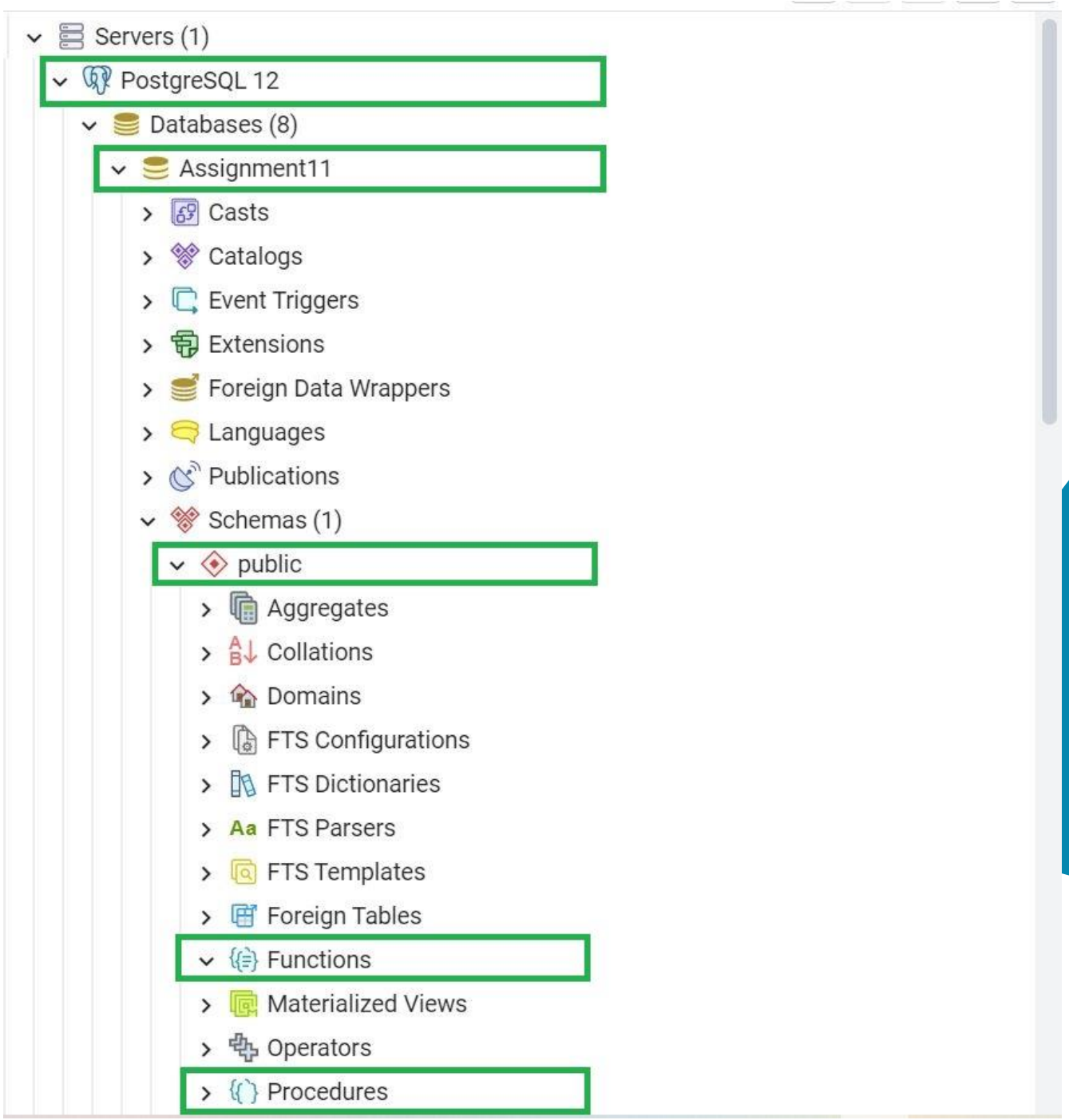
Logical database object or Schema object

02 When ?

1. To execute transactions
2. To performs an action

03 Where ?

Building applications

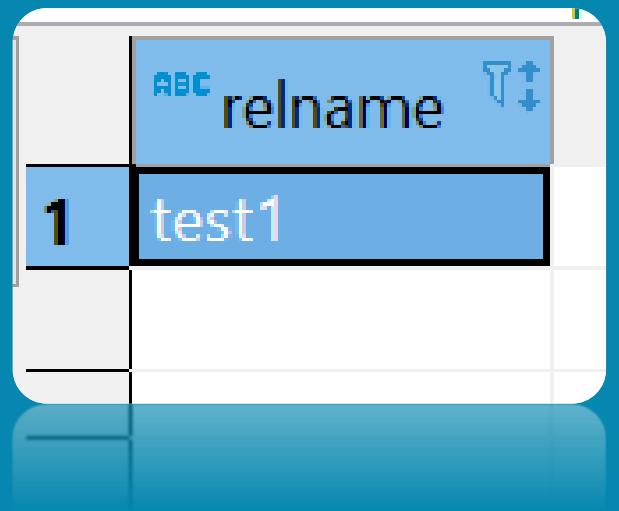


Basic Syntax

```
create [or replace] procedure procedure_name
            (parameter_list)
language plpgsql
as $$
declare
            -- variable declaration
begin
            -- stored procedure body
end; $$
```

Transaction in Procedure

```
CREATE OR REPLACE PROCEDURE control_transaction()
LANGUAGE plpgsql
AS $$
BEGIN
    CREATE TABLE test1 (id int);
    INSERT INTO test1 VALUES (1);
    COMMIT;
    CREATE TABLE test2 (id int);
    INSERT INTO test2 VALUES (1);
    ROLLBACK;
END $$;

call control_transaction();

select relname from pg_class where relname like
'%test%';
```



Benifits

Function and Procedure

1. Improved performance
2. Easy maintenance
3. Improved data security and integrity
4. Improved code clarity

Procedure Vs Function

1. Execute as a PL/SQL statement
2. Do not contain RETURN clause in header
3. Can contain a RETURN statement

1. Invoke as part of an expression
2. Must contain a RETURN clause in header
3. Must contain at least one RETURN statement

Trigger

01 What ?

Named database object that is associated with a table and gets activated when a particular event occur

02 When ?

1. Audit maintain
2. Forcing check constraint

03 Where ?

While doing any action in database

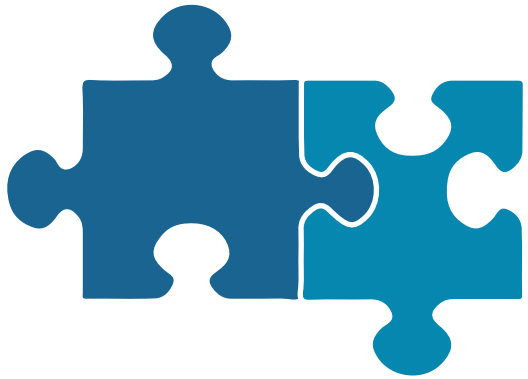
Types of triggers used on tables and views

When	Event	Row-level	Statement-level
BEFORE	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
AFTER	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
INSTEAD OF	INSERT/UPDATE/DELETE	Views	—
	TRUNCATE	—	—

States of row and table

OLD

existing row, table before event



NEW

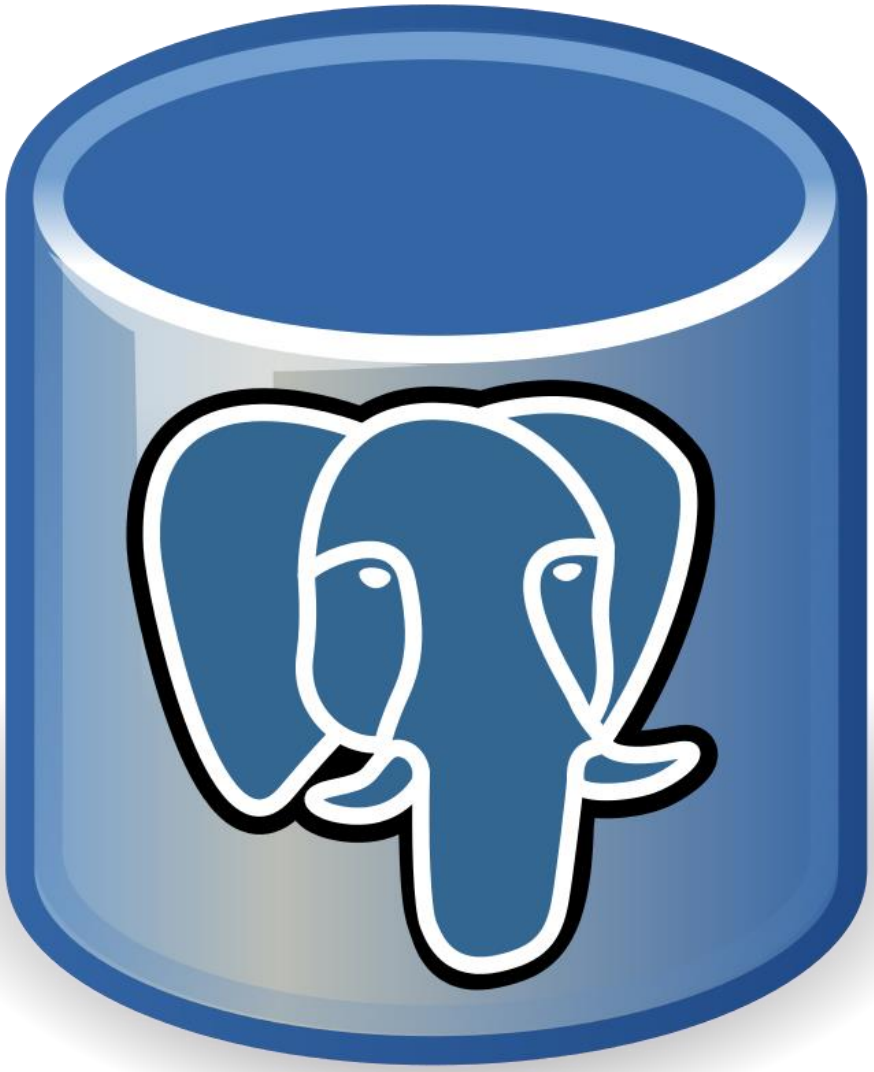
new row, table after event

Step 1: Function Syntax to create trigger

```
CREATE FUNCTION trigger_function()
    RETURNS TRIGGER
    LANGUAGE PLPGSQL
AS $$
BEGIN
    -- trigger logic
END;
$$
```

Step 2: Creating Trigger

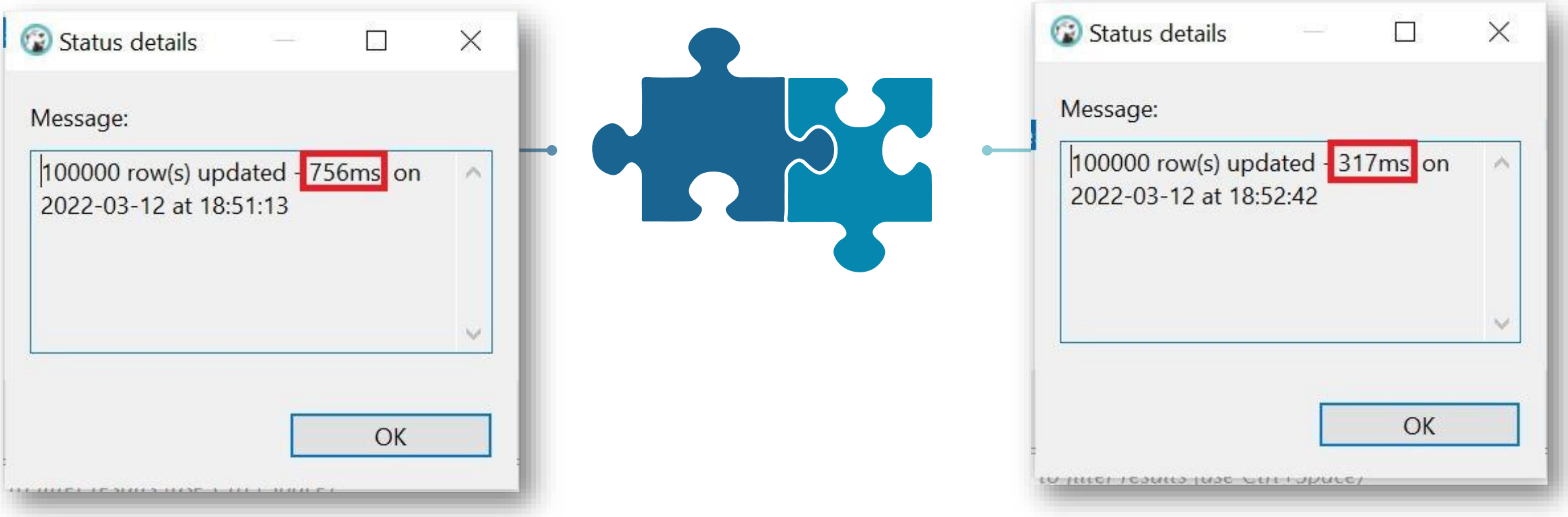
```
CREATE TRIGGER trigger_name
    {BEFORE | AFTER} { event }
    ON table_name
    [FOR [EACH] { ROW | STATEMENT }]
        EXECUTE PROCEDURE trigger_function
```



Row level Vs Statement Level

QUARY:

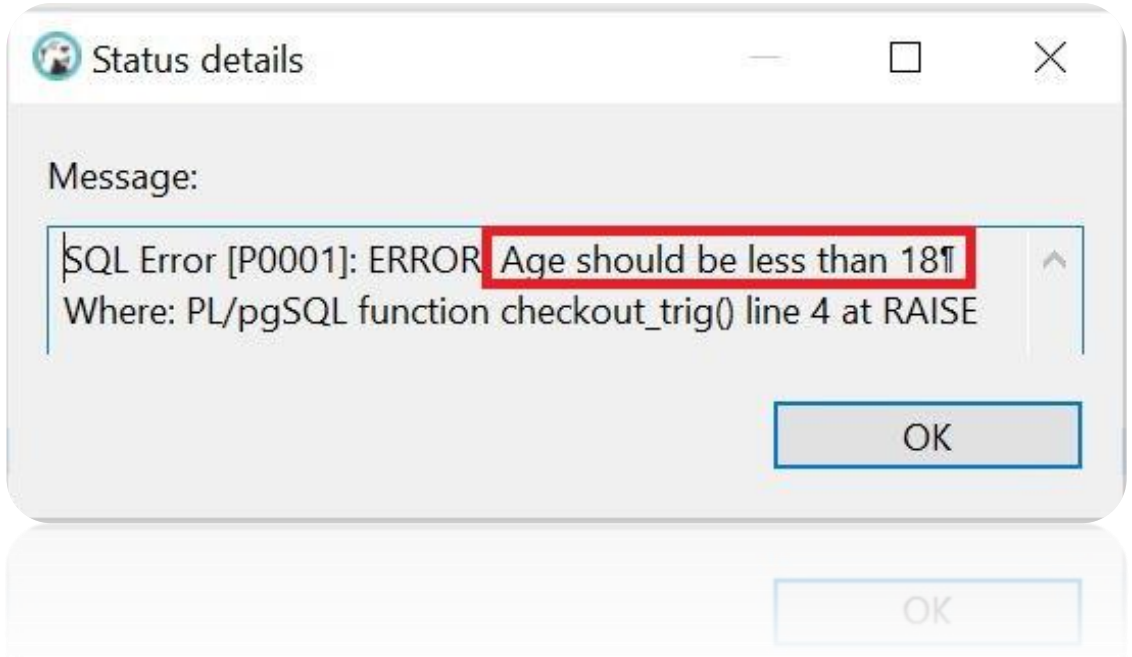
```
insert into emp select 'Sanam',generate_series(1, 100000);
```



Adding check constraint using Trigger

```
CREATE FUNCTION checkout_trig()
RETURNS trigger
LANGUAGE plpgsql as $$
BEGIN
   IF new.age > 18 THEN
      RAISE EXCEPTION 'Age should be less
than 18';
   END IF;
   RETURN new;
END;$$;

create trigger trg_chec
before insert or update on age_check
for each row
execute procedure checkout_trig();
```

```
insert into age_check values (1,20);
```



Managing Trigger

1. Modifying TRIGGER

```
alter trigger trigger_name on table_name rename to new_trigger_name;
```

2. Diabling and enabling TRIGGER

```
alter table table_name disable trigger trigger_name | all;
alter table table_name enable trigger trigger_name | all;
```

3. Removing TRIGGER

```
drop trigger trigger_name on table_name;
```

Partition

01 What ?

1. Process of dividing large table into multiple smaller parts
2. Physical Database Schema Object

02 Why ?

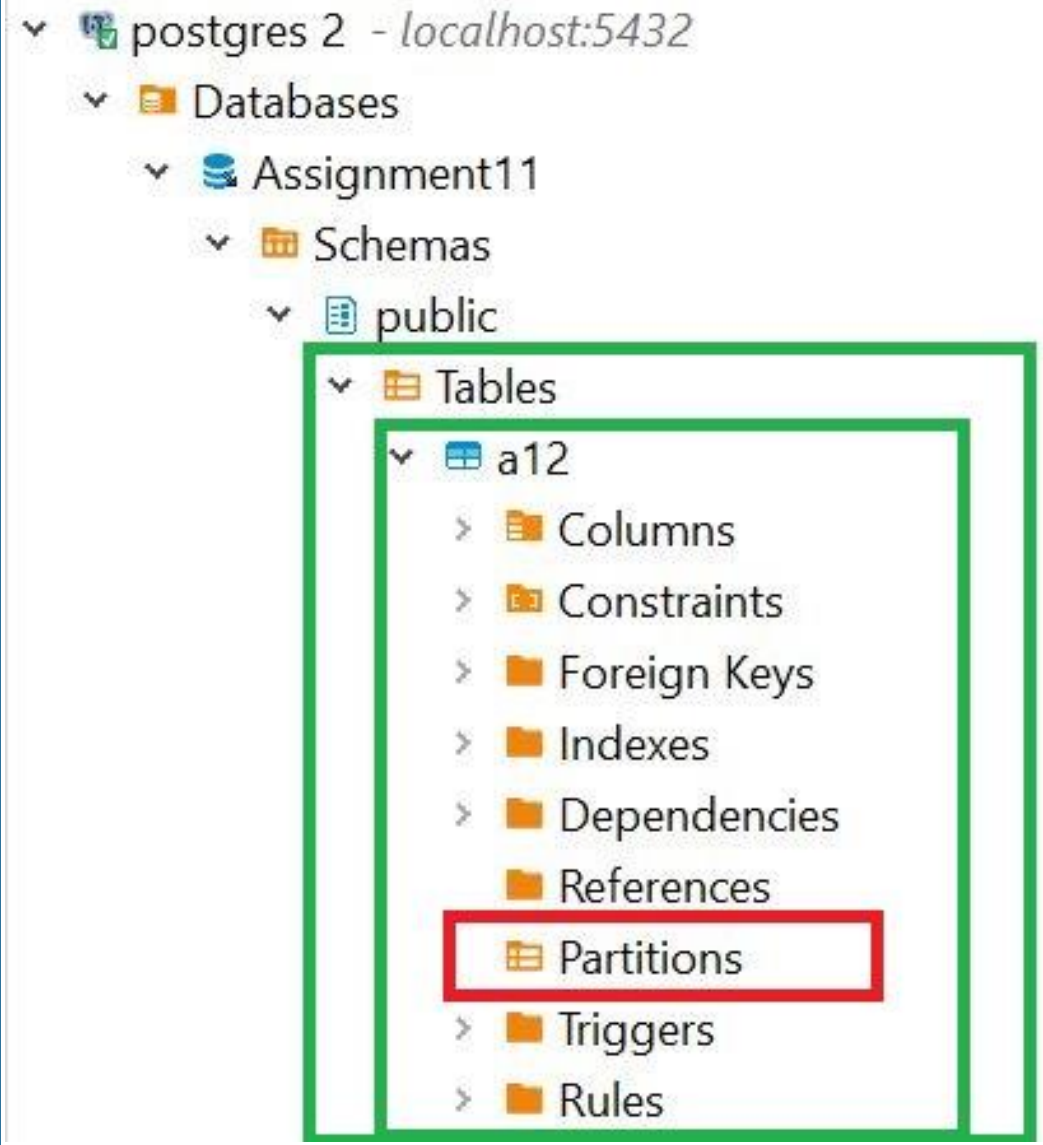
1. For Tuning database
2. Optimization of database

03 Where ?

Bulk loads like QR code data

Database Optimization Technique

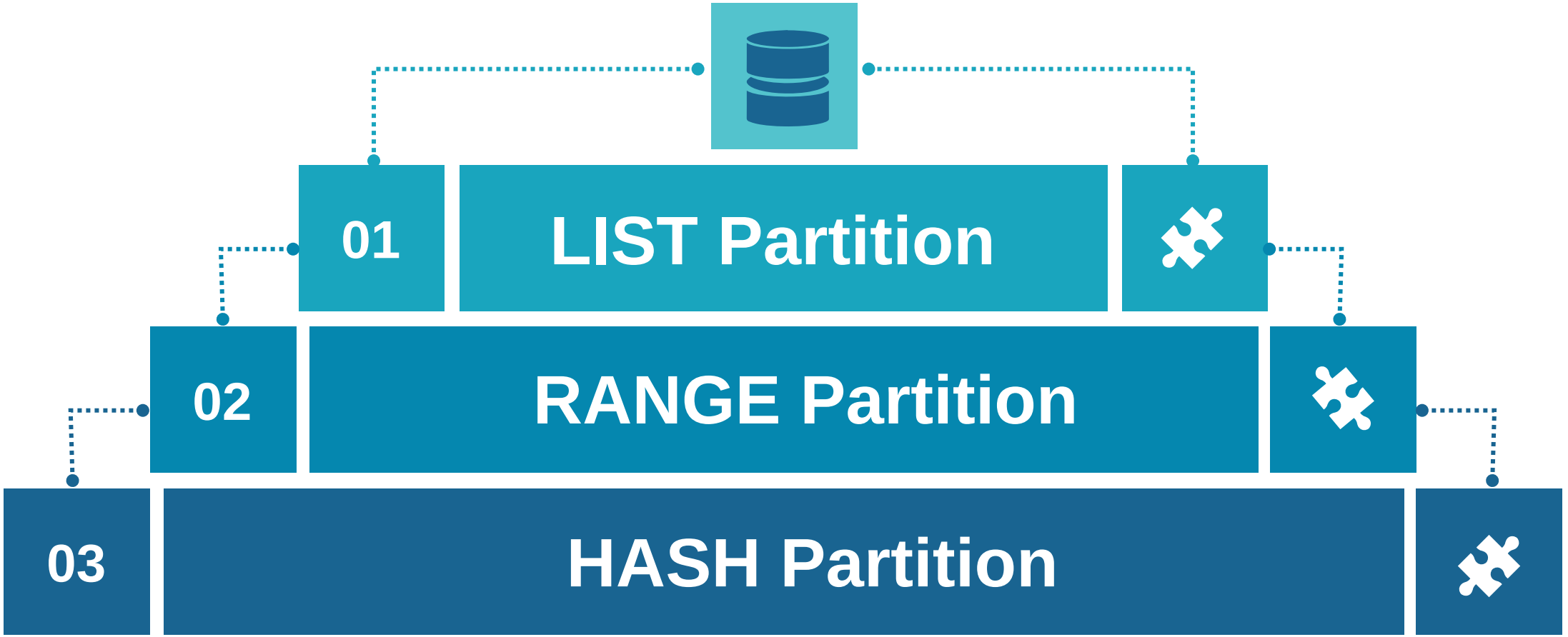
Why Partition ???



Syntax:

```
CREATE TABLE main_table_name (
column_1 data type,
column_n data type,
)
PARTITION BY LIST (column_n) ;
--PARTITION BY RANGE (column_n);
--PARTITION BY HASH (column_n);

-- LIST
CREATE TABLE part_one PARTITION OF parent_table
FOR VALUES IN ('A');
-- Range
create table part_one partition of parent_table
FOR VALUES FROM (minvalue) to (2500000);
-- Hash
CREATE TABLE part_one PARTITION OF parent_table
FOR VALUES WITH (modulus 3, remainder 0);
```

Types of Partition



LIST Partition

1. When to use List Partition
   - ✓ Group Wise like (department wise, area wise, gender wise)

2. Syntax:

```
CREATE TABLE main_table_name (
column_1 data type,
column_n data type,
)
PARTITION BY LIST (column_n);

CREATE TABLE part_one PARTITION OF parent_table
FOR VALUES IN ('A');
```

LIST Partition example

Creating partition table for gender wise:

```
create table partition_test (id int, name varchar, gender varchar)
partition by list(gender);

CREATE TABLE person_male PARTITION OF partition_test FOR VALUES IN ('Male');
CREATE TABLE person_female PARTITION OF partition_test FOR VALUES IN ('Female');
CREATE TABLE person_other PARTITION OF partition_test default;

insert into partition_test values(1,'Sanam','Male'), (2,'Kt','Female'),
                                 (3,'thxaina','Others');

SELECT tableoid::regclass,* FROM partition_test;
```

RANGE Partition

1. When to use RANGE Partition
   - ✓ Numerical
   - ✓ Monthly or Quartey Range (Call history)

1. Syntax:

```
CREATE TABLE main_table_name (
column_1 data type,
column_n data type,
)
PARTITION BY RANGE (column_n);

create table part_one partition of parent_table
FOR VALUES FROM (minvalue) to (2500000);
```

RANGE Partition example

Creating partition table for range:

```
create table partition_test (id int, name varchar) partition by range(id);

create table p1 partition of partition_test for values from (minvalue) to (2500000);
create table p2 partition of partition_test for values from (2500000) to (5000000);
create table p3 partition of partition_test for values from (5000000) to (maxvalue);

insert into partition_test
select x,'sanam' from generate_series(1, 10000000) as x;

select * from partition_test;

SELECT tableoid::regclass,* FROM partition_test;
```

HASH Partition

1. What is HASH Partition
   - ✓ Created by using modulus and remainder for each partition, where rows are inserted by generating a hash value using these modulus and remainders

2. Syntax:

```
CREATE TABLE main_table_name (
column_1 data type,
column_n data type,
)
PARTITION BY HASH (column_n);

CREATE TABLE part_one PARTITION OF parent_table FOR VALUES
WITH (modulus 3, remainder 0);
```

HASH Partition example

Creating partition table Using HASH:

```
CREATE TABLE customers (id int, status TEXT, arr NUMERIC) PARTITION BY HASH(id);

CREATE TABLE cust_part1 PARTITION OF customers FOR VALUES WITH (modulus 3,
remainder 0);
CREATE TABLE cust_part2 PARTITION OF customers FOR VALUES WITH (modulus 3,
remainder 1);
CREATE TABLE cust_part3 PARTITION OF customers FOR VALUES WITH (modulus 3,
remainder 2);

INSERT INTO customers VALUES (1,'ACTIVE',100), (2,'RECURRING',20),
(3,'EXPIRED',38), (4,'REACTIVATED',144);

SELECT tableoid::regclass,* FROM customers;
```

Limitations of Partition

01 NULL

Range partition does not allow NULL values

02 Primary Key

Doesn’t take primary key in LIST and RANGE

03 Subpartition

PostgreSQL does not create a system-defined subpartition when not given it explicitly, so if a subpartition is present at least one partition should be present to hold values.

04 Before Row Trigger

Partition does not support BEFORE ROW triggers on partitioned tables. If necessary, they must be defined on individual partitions, not the partitioned table.


THANK YOU