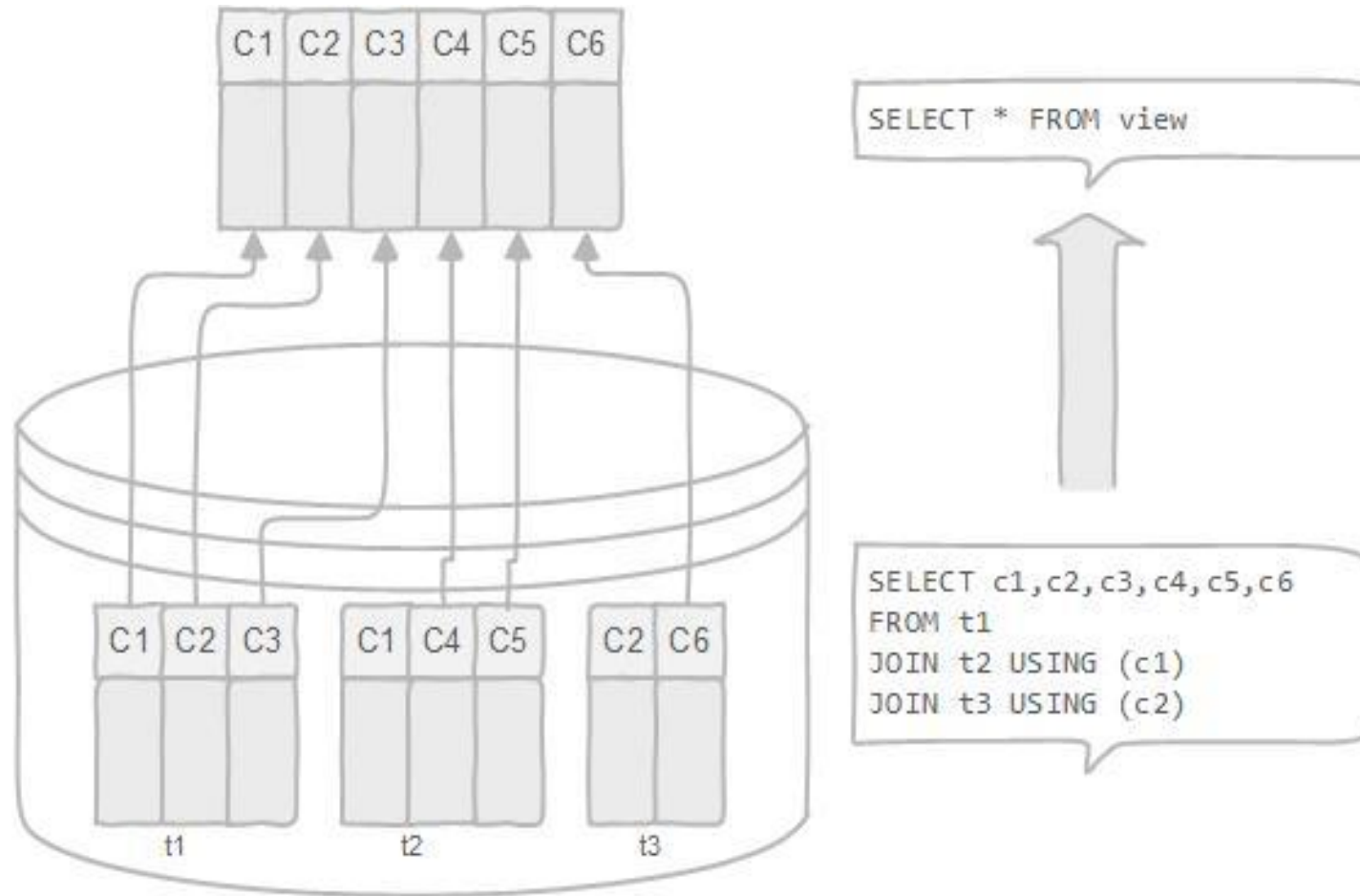


Views, Index, Sequence and Functions

Views

- A view is named query that provides another way to present data in the database tables.
- A view is defined based on one or more tables, which are known as base tables.
- When you create a view, you basically create a query and assign it a name.
- Note that a normal view does not store any data except the materialized view.
- In PostgreSQL, you can create a special view called a materialized view that stores data physically and refreshes the data periodically from the base tables.
- If data is changed in the underlying table, the same change is reflected in the view.
- A view can be built on top of a single or multiple tables.

Views



- A view can be very useful in some cases such as:
 - A view helps simplify the complexity of a query because you can query a view, which is based on a complex query, using a simple SELECT statement.
 - Like a table, you can grant permission to users through a view that contains specific data that the users are authorized to see.
 - A view provides a consistent layer even the columns of underlying table change.

Creating PostgreSQL Views

To create a view, we use `CREATE VIEW` statement.

The simplest syntax of the `CREATE VIEW` statement is as follows:

```
CREATE VIEW view_name AS query;
```

First, you specify the name of the view after the `CREATE VIEW` clause, then you put a query after the `AS` keyword.

A query can be a simple `SELECT` statement or a complex `SELECT` statement with joins

View- Example

Consider the following tables:

Emp(EMPLOYEE_ID , FIRST_NAME, LAST_NAME,EMAIL, PHONE_NUMBER ,HIRE_DATE , JOB_ID ,
SALARY, COMMISSION_PCT,MANAGER_ID ,DEPARTMENT_ID)

Locations(LOCATION_ID, STREET_ADDRESS,POSTAL_CODE,CITY,STATE_PROVINCE,COUNTRY_ID)

Departments(DEPARTMENT_ID,DEPARTMENT_NAME,MANAGER_ID,LOCATION_ID)

PostgreSQL CREATE VIEW with WHERE

```
CREATE VIEW emp_view AS SELECT employee_id,  
first_name,last_name, hire_date FROM employees  
WHERE department_id = 200;
```

- PostgreSQL CREATE VIEW with AND and OR

```
CREATE VIEW my_view AS SELECT *  
FROM locations  
WHERE (country_id='US' AND city='Seattle')  
OR (country_id=JP' AND city='Tokyo');
```

- PostgreSQL CREATE VIEW with GROUP BY

```
CREATE VIEW my_view AS SELECT department_id, count(*)  
FROM employees GROUP BY department_id;
```

- PostgreSQL CREATE VIEW with ORDER BY

```
CREATE VIEW my_view AS SELECT department_id, count(*) FROM employees GROUP BY  
department_id ORDER BY department_id;
```

- PostgreSQL CREATE VIEW with BETWEEN and IN

```
CREATE VIEW my_view AS SELECT * FROM employees WHERE first_name  
BETWEEN 'A' AND 'H' AND salary IN(4000,7000,9000,10000,12000);
```


- PostgreSQL CREATE VIEW with LIKE
 - CREATE VIEW my_view AS SELECT * FROM employees WHERE first_name NOT LIKE 'T%' AND last_name NOT LIKE 'T%';
- PostgreSQL CREATE VIEW using subqueries
 - CREATE VIEW my_view AS SELECT employee_id,first_name,last_name FROM employees WHERE department_id IN(
SELECT department_id FROM departments WHERE location_id IN
(1500,1600,1700));

- PostgreSQL CREATE VIEW with JOIN

```
CREATE VIEW my_view AS SELECT  
a.employee_id,a.first_name,a.last_name, b.department_name,  
b.location_id FROM employees a,departments b WHERE  
a.department_id=b.department_id;
```

- PostgreSQL CREATE VIEW with UNION

```
CREATE VIEW my_view AS SELECT * FROM employees WHERE  
manager_id=100 UNION SELECT * FROM employees WHERE  
first_name BETWEEN 'P' AND 'W' UNION SELECT * FROM employees  
WHERE salary IN(7000,9000,10000,12000);
```

Dropping Views

```
DROP VIEW [IF EXISTS] view_name [, view_name] ... [ CASCADE | RESTRICT]
```

Parameters:

Operator	Description
IF EXISTS	Do not throw an error if the view does not exist. A notice is issued in this case.
name	The name (optionally schema-qualified) of the view to remove.
CASCADE	Automatically drop objects that depend on the view (such as other views).
RESTRICT	Refuse to drop the view if any objects depend on it. This is the default

INDEXES

- CREATE INDEX constructs an index on the specified column(s) of the specified table.
- Indexes are primarily used to enhance database performance.
- The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses.
- Multiple fields can be specified if the index method supports multicolumn indexes.

Index Types

- Single-Column Indexes

- A single-column index is one that is created based on only one table column. The basic syntax is as follows :
- `CREATE INDEX index_name ON table_name (column_name);`

- Multicolumn Indexes

- A multicolumn index is defined on more than one column of a table.
- The basic syntax is as follows :
- `CREATE INDEX index_name ON table_name (column1_name, column2_name);`

The DROP INDEX Command

- An index can be dropped using PostgreSQL **DROP** command.
- The syntax is as follows –
 - DROP INDEX [CONCURRENTLY][IF EXISTS] index_name
[CASCADE | RESTRICT];

When Should Indexes be Avoided?

- Although indexes are intended to enhance a database's performance, there are times when they should be avoided.
- Use of index should be avoided in following cases:–
 - Indexes should not be used on small tables.
 - Tables that have frequent, large batch update or insert operations.
 - Indexes should not be used on columns that contain a high number of NULL values.
 - Use Columns that are frequently manipulated should not be indexed.

Sequences

- A sequence in PostgreSQL is a user-defined schema-bound object that generates a sequence of integers based on a specified specification.
- To create a sequence in PostgreSQL, you use the CREATE SEQUENCE statement

```
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 } ]
```


- **sequence_name**

Specify the name of the sequence after the CREATE SEQUENCE clause. The IF NOT EXISTS conditionally creates a new sequence only if it does not exist.

- **[AS { SMALLINT | INT | BIGINT }]**

- Specify the data type of the sequence. The valid data type is SMALLINT, INT, and BIGINT. The default data type is BIGINT if you skip it.

- The data type of the sequence which determines the sequence's minimum and maximum values

- **INCREMENT [BY] increment]**

- The increment specifies which value to be added to the current sequence value to create new value.

- A positive number will make an ascending sequence while a negative number will form a descending sequence.

- The default increment value is 1.

[MINVALUE minvalue | NO MINVALUE]

[MAXVALUE maxvalue | NO MAXVALUE]

- Define the minimum value and maximum value of the sequence. If you use NO MINVALUE and NO MAXVALUE, the sequence will use the default value.
- For an ascending sequence, the default maximum value is the maximum value of the data type of the sequence and the default minimum value is 1.
- In case of a descending sequence, the default maximum value is -1 and the default minimum value is the minimum value of the data type of the sequence.

- [START [WITH] start]
 - The START clause specifies the starting value of the sequence.
 - The default starting value is minvalue for ascending sequences and maxvalue for descending ones.
- cache
 - The CACHE determines how many sequence numbers are preallocated and stored in memory for faster access. One value can be generated at a time.
 - By default, the sequence generates one value at a time i.e., no cache.

- CYCLE | NO CYCLE
 - 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.
- OWNED BY table_name.column_name
 - The OWNED BY clause allows you to associate the table column with the sequence so that when you drop the column or table, PostgreSQL will automatically drop the associated sequence.

PostgreSQL CREATE SEQUENCE examples

- **Creating an ascending sequence example**
- create a new ascending sequence starting from 100 with an increment of 5:
 - CREATE SEQUENCE mysequence
INCREMENT 5
START 100;
- To get the next value from the sequence to you use the nextval() function:
 - SELECT nextval('mysequence')

	nextval bigint
1	100

- If you execute the statement again, you will get the next value from the sequence:
 - `SELECT nextval('mysequence');`

	nextval bigint
1	105

2) Creating a descending sequence example

- The following statement creates a descending sequence from 3 to 1 with the cycle option:

```
CREATE SEQUENCE three  
INCREMENT -1  
MINVALUE 1  
MAXVALUE 3  
START 3  
CYCLE;
```

- When you execute the following statement multiple times, you will see the number starting from 3, 2, 1 and back to 3, 2, 1 and so on:
 - `SELECT nextval('three');`

3) Creating a sequence associated with a table column.

Consider the table order_details:

```
CREATE TABLE order_details(  
    order_id SERIAL,  
    item_id INT NOT NULL,  
    product_id INT NOT NULL,  
    price DEC(10,2) NOT NULL,  
    PRIMARY KEY(order_id, item_id)  
);
```

- create a new sequence associated with the item_id column of the order_details table:

```
CREATE SEQUENCE order_item_id  
START 10  
INCREMENT 10  
MINVALUE 10  
OWNED BY order_details.item_id;
```


- Now we can insert rows into order_details table as follows:

```
INSERT INTO order_details(order_id, item_id, item_text, price)
```

```
VALUES
```

```
(100, nextval('order_item_id'),'DVD Player',100),
```

```
(100, nextval('order_item_id'),'Android TV',550),
```

```
(100, nextval('order_item_id'),'Speaker',250);
```

- Now if we query data from the order_details table:
 - SELECT order_id, item_id, item_text, price FROM order_details;

-

	order_id integer	item_id integer	item_text character varying (100)	price numeric (10,2)
1	100	10	DVD Player	100.00
2	100	20	Android TV	550.00
3	100	30	Speaker	250.00

- Deleting sequences

- If a sequence is associated with a table column, it will be automatically dropped once the table column or entire table is dropped. Otherwise, you need to remove the sequence manually using the DROP SEQUENCE statement:

```
DROP SEQUENCE [ IF EXISTS ] sequence_name [, ...]
```

```
[ CASCADE | RESTRICT ];
```

PL/pgSQL

- This section shows you step by step on how to use the PL/pgSQL to develop PostgreSQL user-defined functions and stored procedures.
- PL/pgSQL procedural language adds many procedural elements, e.g., control structures, loops, and complex computations, to extend standard SQL.
- It allows you to develop complex functions and stored procedures in PostgreSQL that may not be possible using plain SQL.

BLOCK STRUCTURE

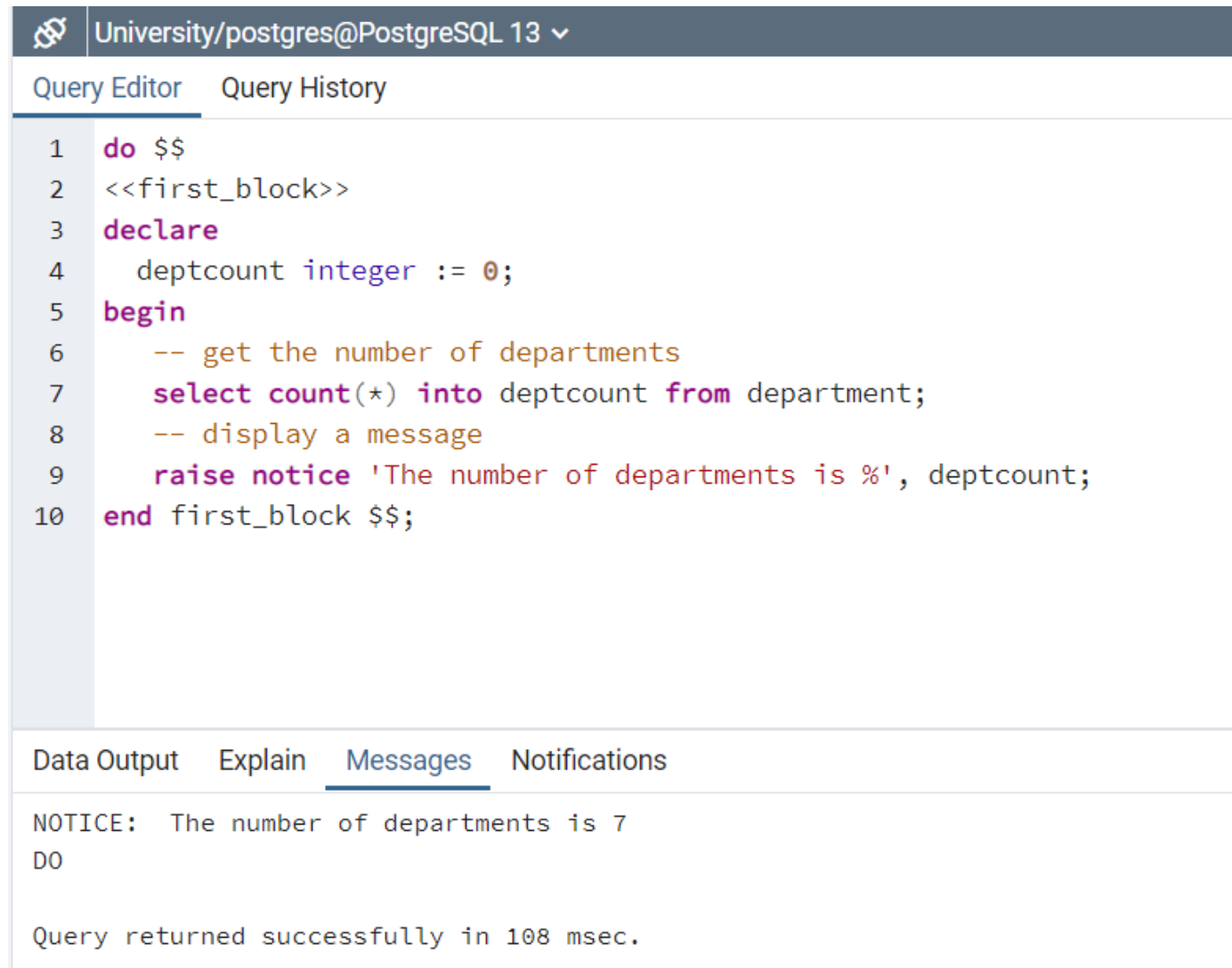
- PL/pgSQL is a block-structured language, therefore, a PL/pgSQL function or stored procedure is organized into blocks.
- The following illustrates the syntax of a complete block in PL/pgSQL:

```
[ <<label>> ]  
[ declare  
    declarations ]  
begin  
    statements;  
    ...  
end [ label ];
```

- Each block has two sections: declaration and body. The declaration section is optional while the body section is required. A block is ended with a semicolon (;) after the **END** keyword.
- A block may have an optional label located at the beginning and at the end. You use the block label when you want to specify it in the **EXIT** statement of the block body or when you want to qualify the names of **variables** declared in the block.
- The declaration section is where you **declare all variables** used within the body section. Each statement in the declaration section is terminated with a semicolon (;).
- The body section is where you place the code. Each statement in the body section is also terminated with a semicolon (;).

PL/pgSQL block structure example

Display the total number of departments



```
University/postgres@PostgreSQL 13 ▾  
Query Editor Query History  
1 do $$  
2 <<first_block>>  
3 declare  
4   deptcount integer := 0;  
5 begin  
6   -- get the number of departments  
7   select count(*) into deptcount from department;  
8   -- display a message  
9   raise notice 'The number of departments is %', deptcount;  
10 end first_block $$  
  
Data Output Explain Messages Notifications  
NOTICE: The number of departments is 7  
DO  
  
Query returned successfully in 108 msec.
```

Variables Declaration

- Syntax of declaring a variable

variable_name data_type [:= expression];

- How to declare a variable with the data type of a table column:

variable_name table_name.column_name%type;

- Example illustrates how to declare and initialize variables:

```
do $$
declare
  counter integer := 1;
  first_name varchar(50) := 'John';
  last_name varchar(50) := 'Doe';
  payment numeric(11,2) := 20.5;
begin
  raise notice '%%% has been paid % USD',
    counter,
    first_name,
    last_name,
    payment;
end $$;
```

Create Function

- The **create function** statement allows you to define a new user-defined function.

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

- **Drop Function**

```
drop function function_name;
```

Example

Create a function that, given the name of a department, returns the count of the number of instructors in that department

```
create or replace function dept_count(deptname varchar(20))
```

```
    returns integer
```

```
    language plpgsql
```

```
as
```

```
$$
```

```
declare
```

```
    d_count integer;
```

```
begin
```

```
    select count(*) into d_count from instructor where instructor.dept_name=deptname;
```

```
    return d_count;
```

```
end;
```

```
$$;
```

Write query that returns names and budgets of all departments with more than 12 instructors

```
select distinct dept_name from instructor where dept_count(dept_name) > 12;
```


Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called table functions

```
create or replace function function_name (  
    parameter_list  
)  
returns table ( column_list )  
language plpgsql  
as $$  
declare  
    -- variable declaration  
begin  
    -- body  
end; $$
```

Example – Table functions

- Create function that returns a table containing all the instructors of a particular department.

```
create or replace function instructors_of (deptname varchar(20))
    returns table(
        inst_ID varchar(5),
        inst_NAME varchar(20),
        inst_DEPT varchar(20),
        inst_SALARY numeric(8,2))
    language plpgsql
as $$
begin
    return query
        select id, name, dept_name, salary from instructor
            where instructor.dept_name = instructors_of.deptname;
end;$$
```

- Query returns all instructors of the 'Finance' department

```
select * from instructors_of('Finance');
```

Control statements

While loop

```
while boolean expression do  
    sequence of statements;  
end loop;
```

For loop

```
for loop_counter in from.. to by step loop  
    statements  
end loop;
```

If then elseif

```
if condition_1 then  
    statement_1;  
elseif condition_2 then  
    statement_2  
...  
elseif condition_n then  
    statement_n;  
else  
    else-statement;  
end if;
```