Department of CSE

COURSE NAME: DBMS
COURSE CODE: 23AD2102R

Topic:

CURSORS & TRIGGERS

Session - 16

AIM OF THE SESSION

To familiarize students with the Basic concepts about cursors and triggers.

INSTRUCTIONAL OBJECTIVES

This Session is designed to:

1. Discuss about Terminology of Cursors and Triggers.
2. Describe the Syntax of Cursors and Triggers.
3. Demonstrate Cursors with suitable example.
4. Demonstrate Triggers with suitable example.

LEARNING OUTCOMES

At the end of this session, students should be able to:

1. Know the usage of cursors and triggers.
2. Know the applications and execution of cursors and triggers.

Cursor & it’s Uses

- Rather than executing a whole query at once, it is possible to set up a *cursor* that encapsulates the query, and then read the query result a few rows at a time.
- Cursor is used to avoid memory overrun when the result contains a large number of rows. It returns a reference to a cursor that a function has created, allowing the caller to read the rows.
- This provides an efficient way to return large row sets from functions.
- The major function of a cursor is to retrieve data, one row at a time, from a result set, unlike the SQL commands which operate on all the rows in the result set at one time.
- Cursors are used when the user needs to update records in a singleton fashion or in a row by row manner, in a database table.

Types of Cursors

There are 2 types of cursors. They are:

- Bound Cursor
- Unbound Cursor

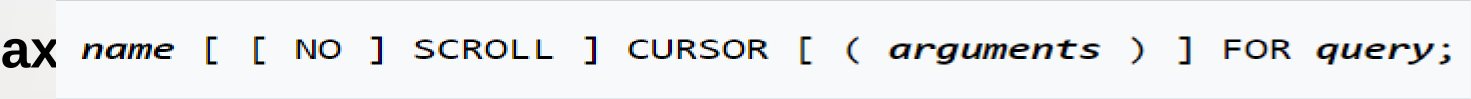
Steps involved in cursors

1. First, declare a cursor.
2. Next, open the cursor.
3. Then, fetch rows from the result set into a target.
4. After that, check if there is more row left to fetch. If yes, go to step 3, otherwise, go to step 5.
5. Finally, close the cursor.

1. Declaring Cursor Variables

- All access to cursors in PL/pgSQL goes through cursor variables, which are always of the special data type refcursor.
- One way to create a cursor variable is just to declare it as a variable of type refcursor.

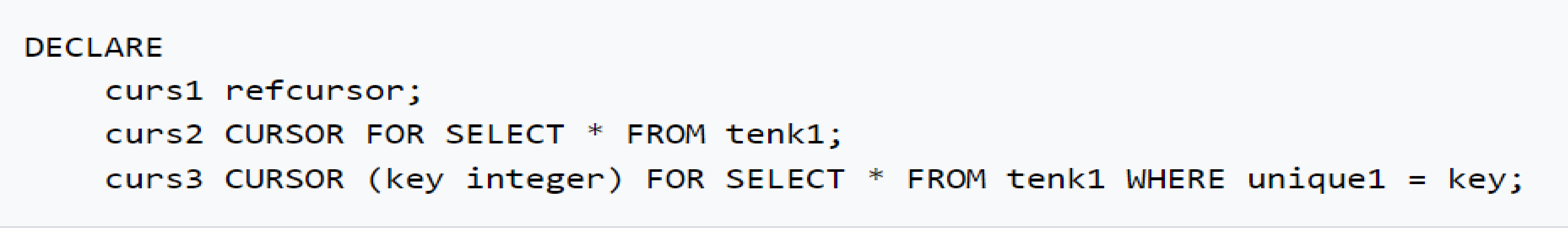
Declaration Syntax



```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

- If SCROLL is specified, the cursor will be capable of scrolling backward.
- If NO SCROLL is specified, backward fetches will be rejected.
- If neither specification appears, it is query-dependent whether backward fetches will be allowed.
- Arguments, if specified, is a comma-separated list of pairs name datatype that define names to be replaced by parameter values in the given query.
- The actual values to substitute for these names will be specified later, when the cursor is opened.

Example:



```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

- The First variable can be used with any query and is said to be unbound since it is not bound to any particular query .
- The Second variable has a fully specified query already bound to it.
- The last has a parameterized query bound to it. (key will be replaced by an integer parameter value when the cursor is opened.)

2.Opening Cursors

- Before a cursor can be used to retrieve rows, it must be opened.

- PL/pgSQL has three forms of the OPEN statement, two of which use unbound cursor variables while the third uses a bound cursor variable.

- Steps in Opening cursor:
  1. OPEN FOR Query
  2. OPEN FOR EXECUTE
  3. Opening A Bound Cursor

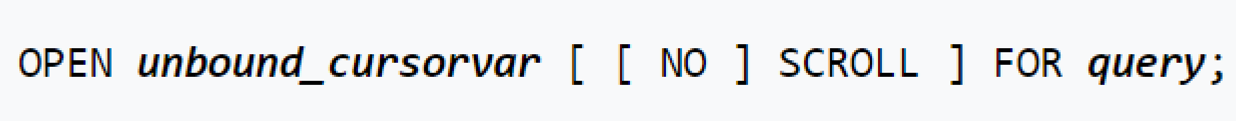
- **Note**:

  Bound cursor variables can also be used without explicitly opening the cursor, via the FOR statement.

2.1.OPEN FOR Query

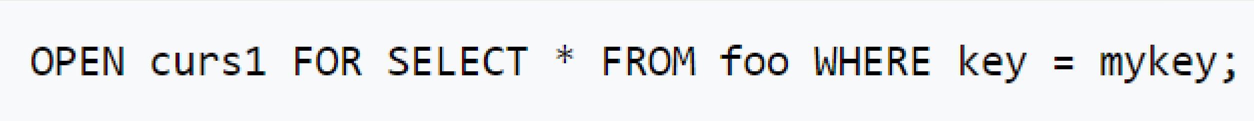
- The cursor variable is opened and given the specified query to execute.
- The cursor cannot be open already, and it must have been declared as an unbound cursor variable (that is, as a simple refcursor variable).
- When a PL/pgSQL variable is substituted into the cursor query, the value that is substituted is the one it has at the time of the OPEN; subsequent changes to the variable will not affect the cursor's behavior.

Syntax:



```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
```

Example:



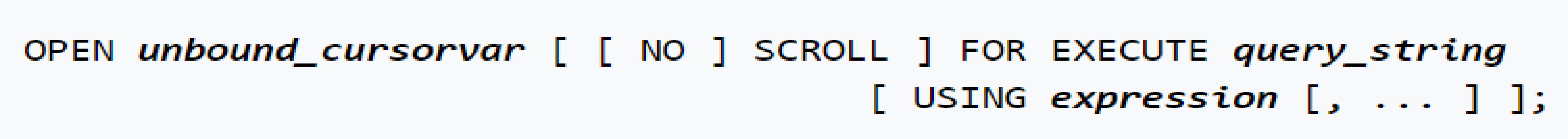
```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

2.2. OPEN FOR EXECUTE

- The query is specified as a string expression, in the same way as in the EXECUTE command.
- This gives flexibility so the query plan can vary from one run to the next and it also means that variable substitution is not done on the command string.
- As with EXECUTE, parameter values can be inserted into dynamic command via format() and USING.
- The SCROLL and NO SCROLL options have the same meanings as for a bound cursor.

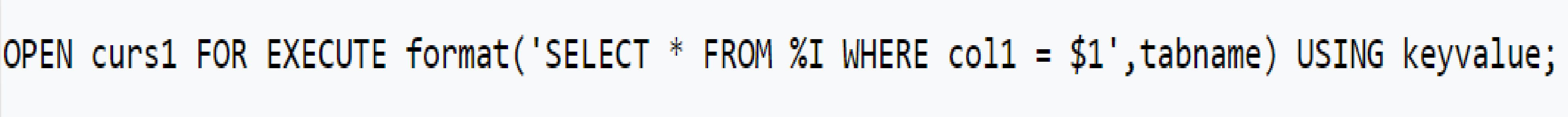
2.2. OPEN FOR EXECUTE

Syntax:



```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query_string
                                     [ USING expression [, ... ] ];
```

Example:



```
OPEN curs1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1',tabname) USING keyvalue;
```

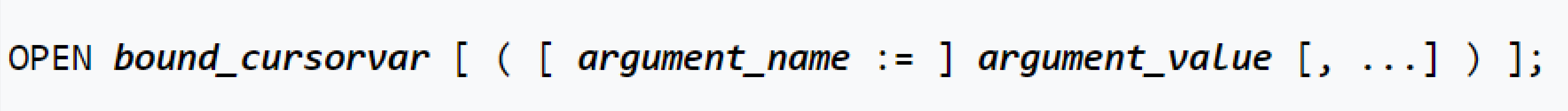
In this example, the table name is inserted into the query via format(). The comparison value for col1 is inserted via a USING parameter, so it needs no quoting.

2.3.Opening A Bound Cursor

- OPEN is used to open a cursor variable whose query was bound to it when it was declared.
- A list of actual argument value expressions must appear if and only if the cursor was declared to take arguments.
- These values will be substituted in the query.
- The query plan for a bound cursor is always considerable.
- Argument values can be passed using either positional or named notation.
- In positional notation, all arguments are specified in order.
- In named notation, each argument's name is specified using := to separate it from the argument expression.

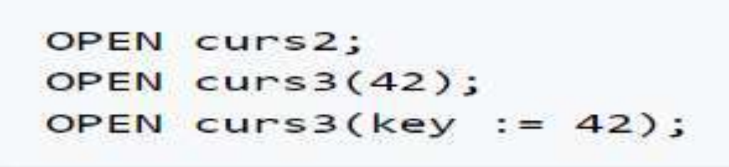
Opening A Bound Cursor

Syntax:



```
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ];
```

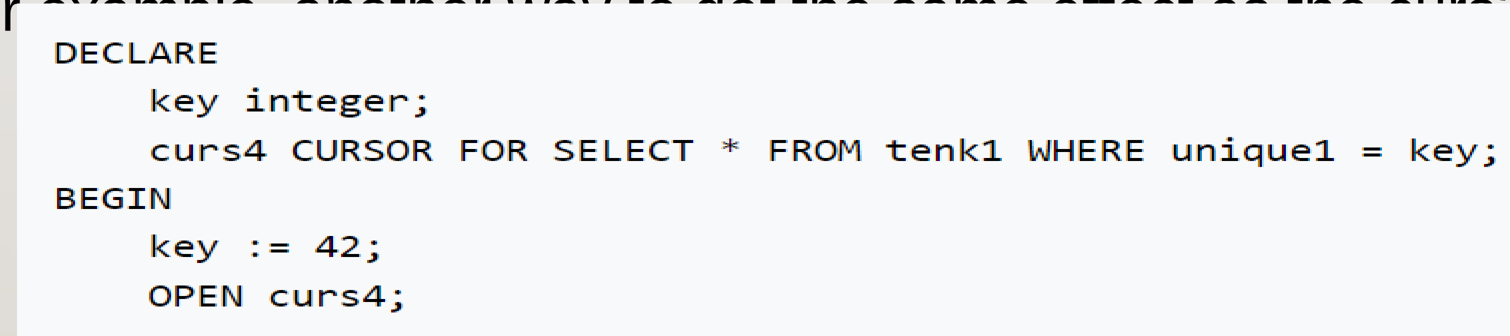
Example:



```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

However, only variables declared before the bound cursor was declared will be substituted into it. In either case the value to be passed is determined at the time of the OPEN.

For example, another way to get the same effect as the curs3 example above is



```
DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

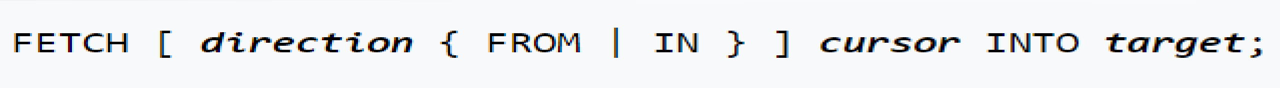
3.Using Cursors

- Once a cursor has been opened, it can be manipulated with the statements.
- These manipulations need not occur in the same function that opened the cursor to begin with.
- You can return a refcursor value out of a function and let the caller operate on the cursor.
- All portals are implicitly closed at transaction end.
- Therefore a refcursor value is usable to reference an open cursor only until the end of the transaction.

3.1 Fetch

- FETCH retrieves the next row from the cursor into a target, which might be a row variable, a record variable, or a comma-separated list of simple variables, just like SELECT INTO.

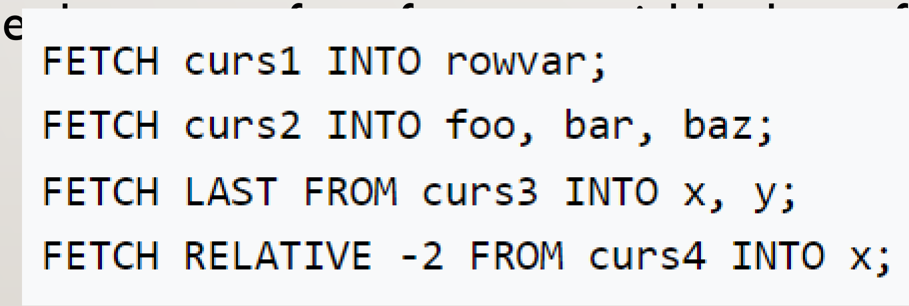
Syntax:



```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

- The direction clause can be any of the variants allowed in the SQL FETCH command except the ones that can fetch more than one row; namely, it can be NEXT, PRIOR, FIRST, LAST, ABSOLUTE count, RELATIVE count, FORWARD, or BACKWARD.
- cursor must be erences an open cursor portal.

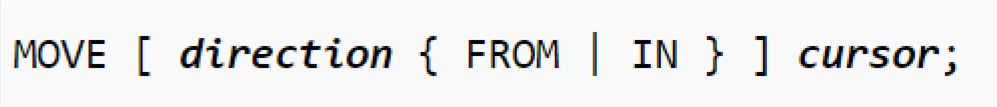
Examples:



```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

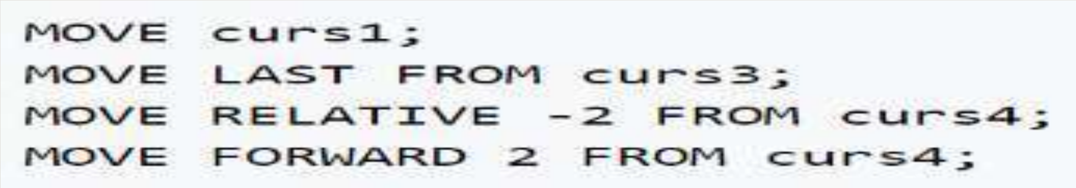
3.2 Move

- MOVE repositions a cursor without retrieving any data.
- MOVE works exactly like the FETCH command, except it only repositions the cursor and does not return the row moved to.
- As with SELECT INTO, the special variable FOUND can be checked to see whether there was a next row to move to.
- Syntax:



```
MOVE [ direction { FROM | IN } ] cursor;
```

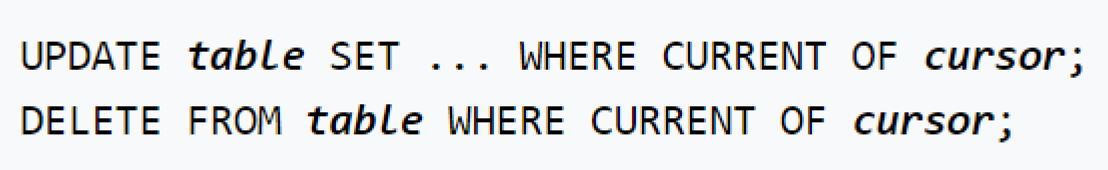
- Example:



```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

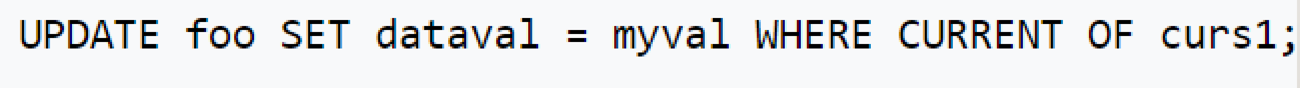
3.3 UPDATE/DELETE WHERE CURRENT OF

- When a cursor is positioned on a table row, that row can be updated or deleted using the cursor to identify the row.
- There are restrictions on what the cursor's query can be and it's best to use FOR UPDATE in the cursor.
- Syntax:



```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

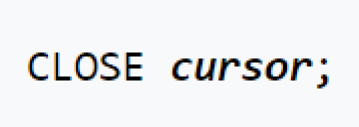
- Example:



```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

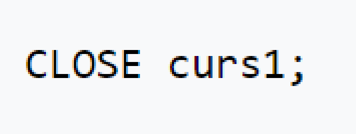
3.4 Close

- CLOSE closes the portal underlying an open cursor.
- This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.
- Syntax:



```
CLOSE cursor;
```

- Example:



```
CLOSE curs1;
```

TRIGGERS

Definition:

A trigger is a set of actions that are run automatically when a specified change operation (SQL INSERT, UPDATE, DELETE or TRUNCATE statement) is performed on a specified table. Triggers are useful for tasks such as enforcing business rules, validating input data, and keeping an audit trail.

Uses for triggers:

- Enforce business rules
- Validate input data
- Generate a unique value for a newly-inserted row in a different file.

Benefits of using triggers in business

- **Faster application development:** Database stores triggers, no need to code the trigger actions into each database application.

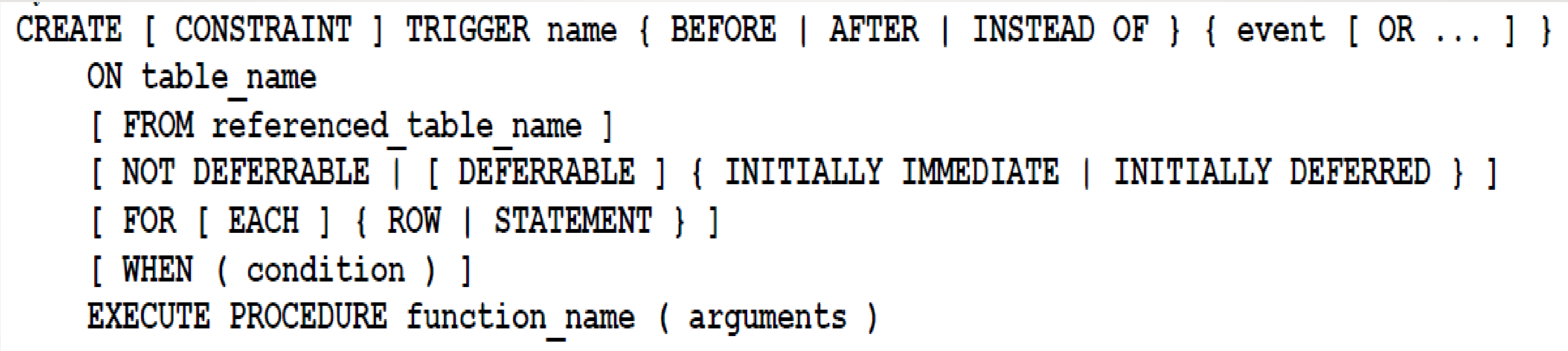
- **Easier maintenance:** If a business policy changes you need to change only the corresponding trigger program instead of each application program.

- **Improve performance in client/server environment:** All rules run on the server before the result returns.

Create trigger

A trigger is a named database object that is associated with a table, and it activates when a particular event (e.g. an insert, update or delete) occurs for the table/views. The statement CREATE TRIGGER creates a new trigger in PostgreSQL.

Syntax:



```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY DEFERRED } ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )
```

Parameters in Triggers

Parameter	Description
Name	The name of the trigger. A trigger must be distinct from the name of any other trigger for the same table. The name cannot be schema-qualified — the trigger inherits the schema of its table.
BEFORE AFTER INSTEAD OF	Determines whether the function is called before, after, or instead of the event. A constraint trigger can only be specified as AFTER.
Event	One of INSERT, UPDATE, DELETE, or TRUNCATE, that will fire the trigger.
table_name	The name of the table or view the trigger is for.
referenced_table_name	The (possibly schema-qualified) name of another table referenced by the constraint. This option is used for foreign- key constraints and is not recommended for general use. This can only be specified for constraint triggers.
FOR EACH ROW FOREACH STATEMENT	Specifies whether the trigger procedure should be fired once for every row affected by the trigger event, or just once per SQL statement. If neither is specified, FOR EACH STATEMENT is the default.
condition	A Boolean expression that determines whether the trigger function will actually be executed.
function_name	A user-supplied function that is declared as taking no arguments and returning type trigger, which is executed when the trigger fires.

Events in Triggers

- Triggers that are specified to fire INSTEAD OF the trigger event must be marked FOR EACH ROW, and can only be defined on views.
- BEFORE and AFTER triggers on a view must be marked as FOR EACH STATEMENT.
- Triggers may be defined to fire for TRUNCATE, though only FOR EACH STATEMENT.
- The following table summarizes which types of triggers may be 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	----	----

After insert

Example:

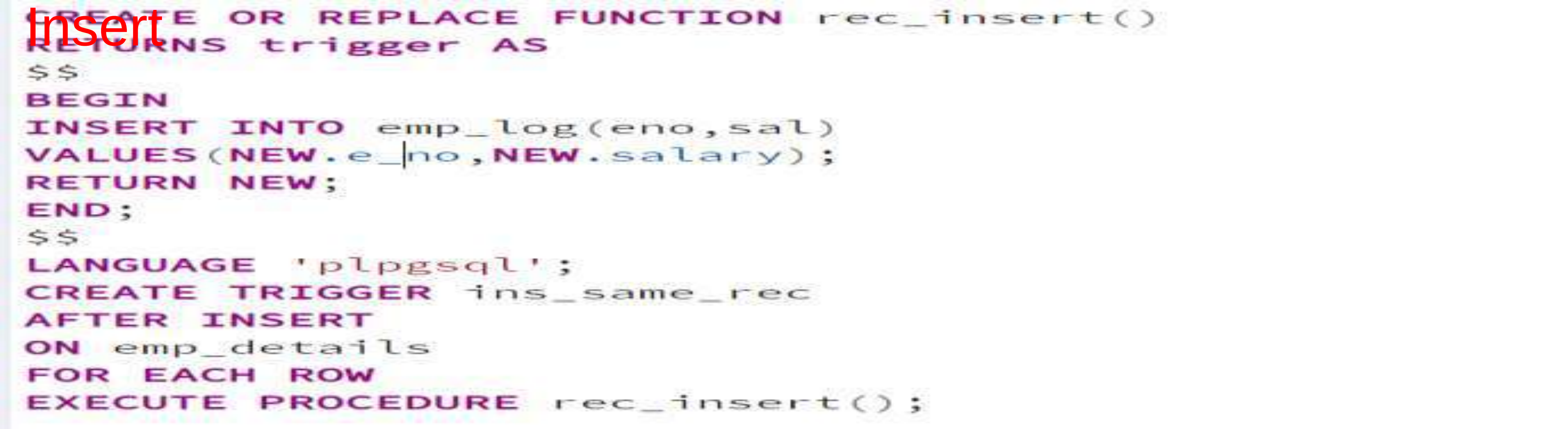
- consider a case where we want to keep audit trial for every record being inserted in COMPANY table, which we will create new two tables, emp_details and emp_log.
- To insert some information into emp_logs table (which have two fields emp_id

  and salary) every time, when an INSERT happen into emp_details table we have

  used the following trigger.
- At first a trigger function have to be created. Here is the trigger function rec_insert().

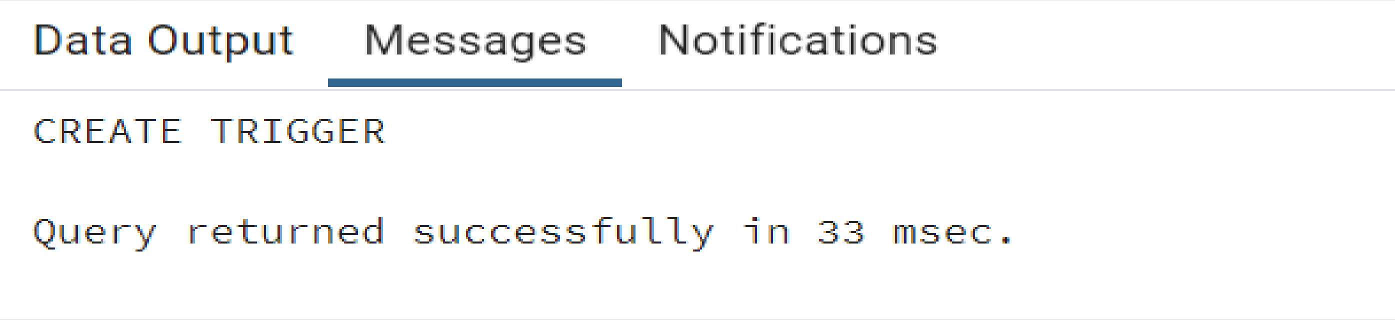
After insert

Example for After Insert



```
CREATE OR REPLACE FUNCTION rec_insert()
RETURNS trigger AS
$$
BEGIN
INSERT INTO emp_log(eno,sal)
VALUES(NEW.e_no,NEW.salary);
RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';
CREATE TRIGGER ins_same_rec
AFTER INSERT
ON emp_details
FOR EACH ROW
EXECUTE PROCEDURE rec_insert();
```

Output:

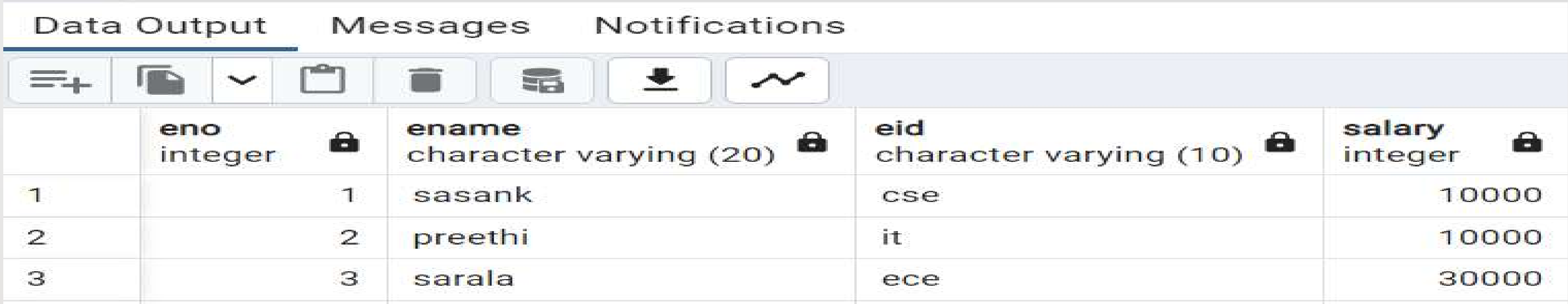


```
CREATE TRIGGER

Query returned successfully in 33 msec.
```

After insert

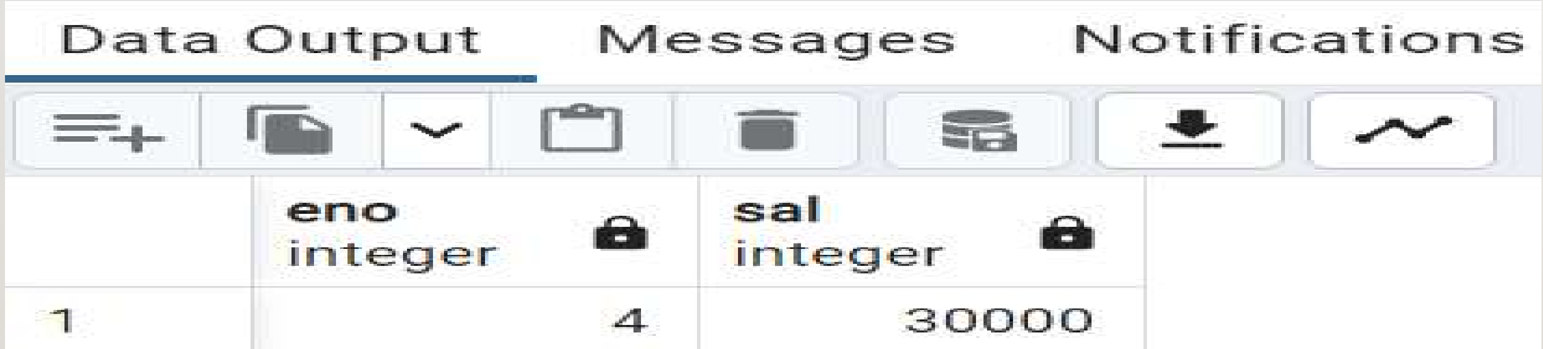
Select * from emp_details;



	eno integer	ename character varying (20)	eid character varying (10)	salary integer
1	1	sasank	cse	10000
2	2	preethi	it	10000
3	3	sarala	ece	30000

insert into emp_details values(4,'silpa','mba',30000);

Select * from emp_log;

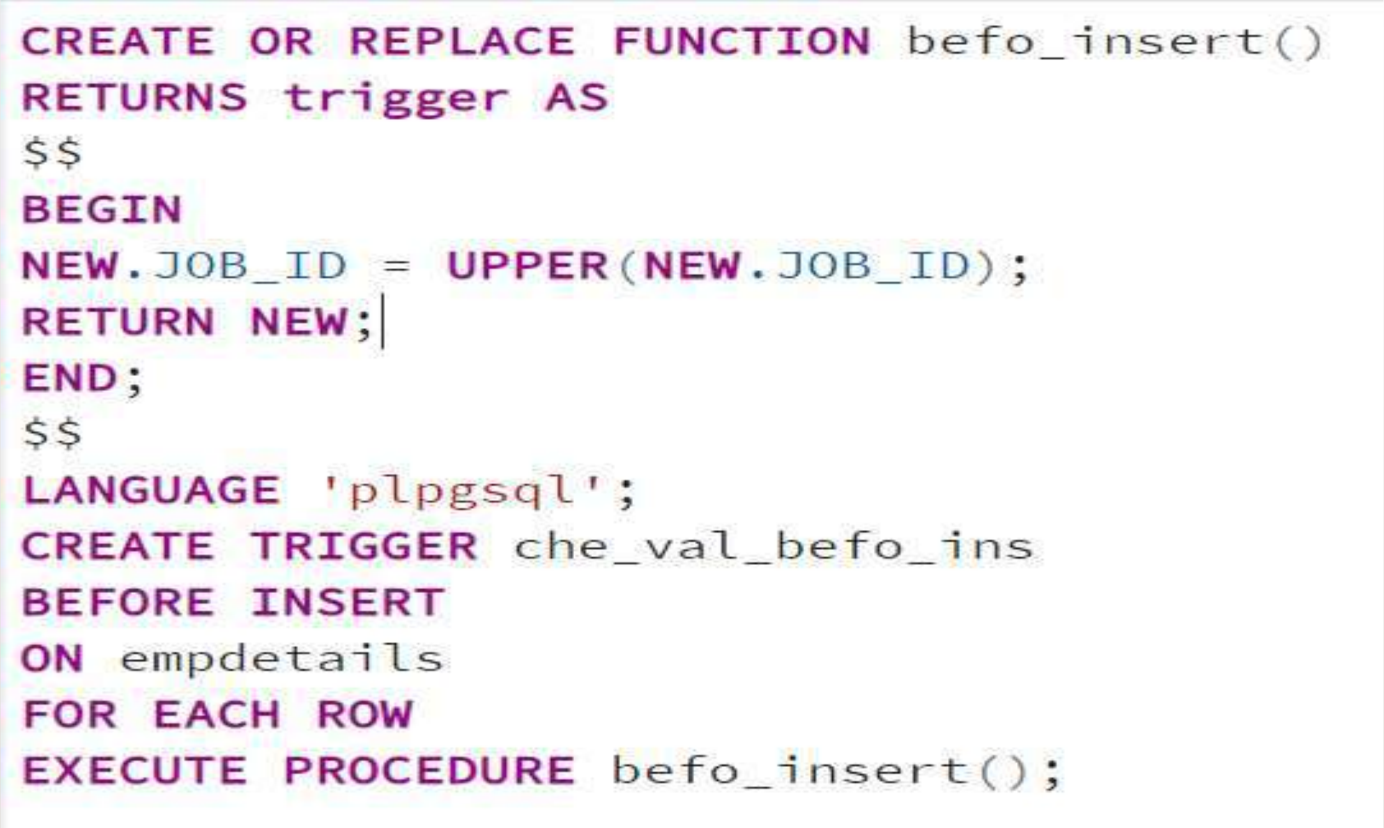


	eno integer	sal integer
1	4	30000

Before insert

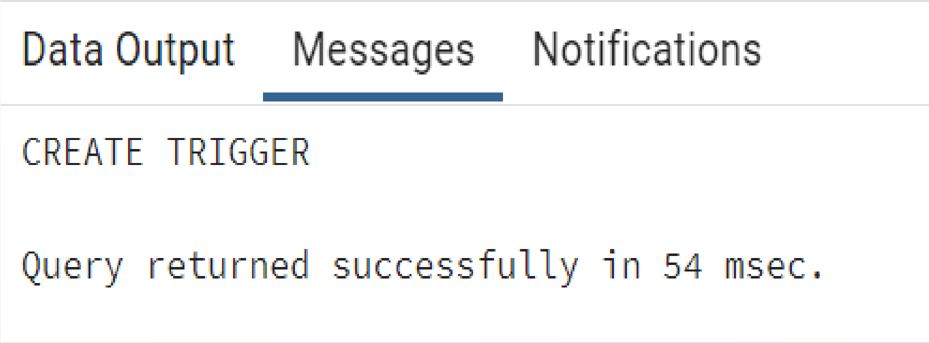
In the following example, before insert a new record in empdetails table, a trigger check the column value of JOB_ID and JOB_ID is converted to Uppercase by UPPER() function

Example for before insert



```
CREATE OR REPLACE FUNCTION befo_insert()
RETURNS trigger AS
$$
BEGIN
NEW.JOB_ID = UPPER(NEW.JOB_ID);
RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';
CREATE TRIGGER che_val_befo_ins
BEFORE INSERT
ON empdetails
FOR EACH ROW
EXECUTE PROCEDURE befo_insert();
```

Output

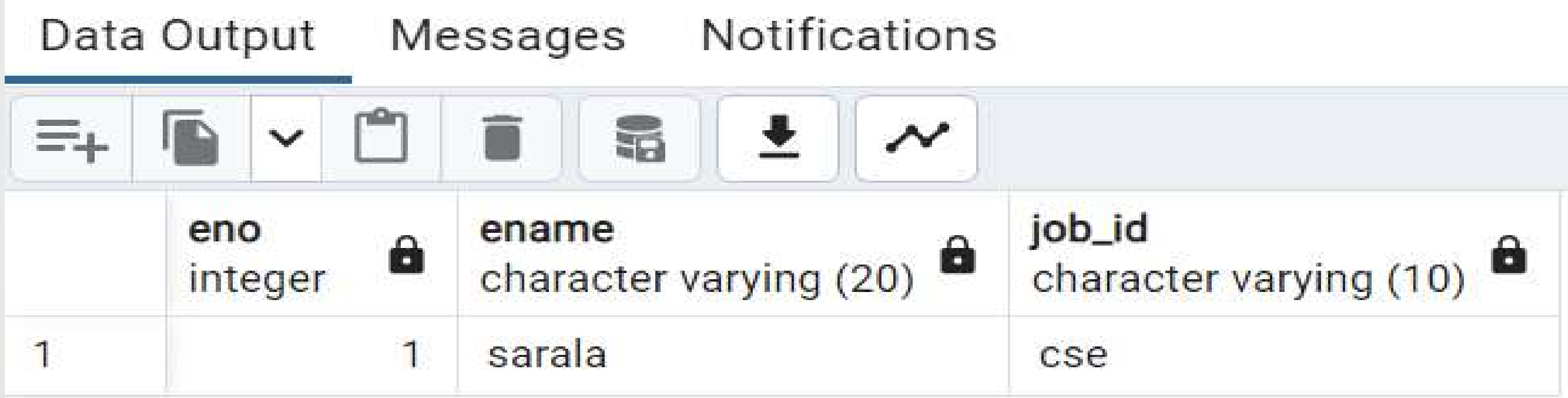


```
CREATE TRIGGER

Query returned successfully in 54 msec.
```

Before insert

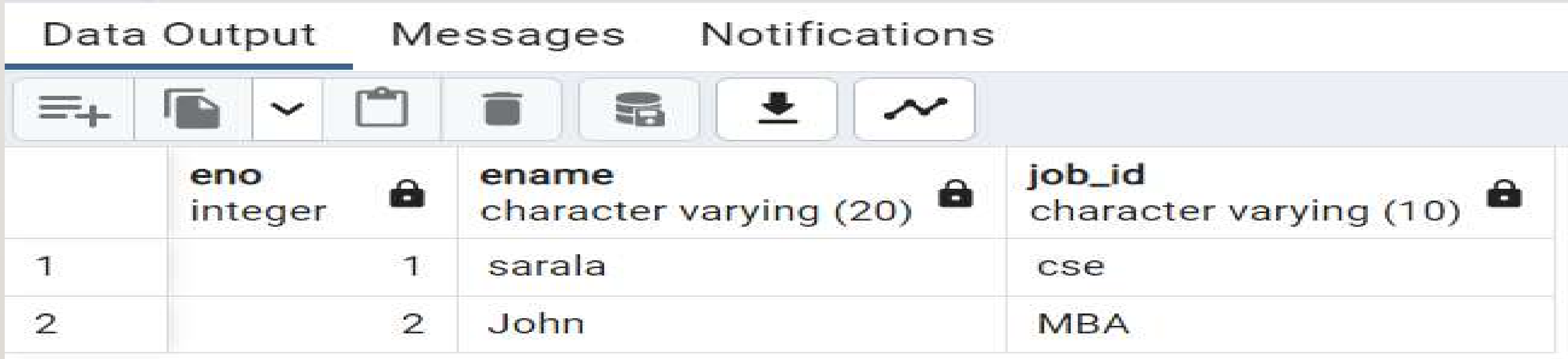
select * from empdetails;



	eno integer	ename character varying (20)	job_id character varying (10)
1	1	sarala	cse

insert into empdetails values(2,'John','mba');

select * from empdetails;

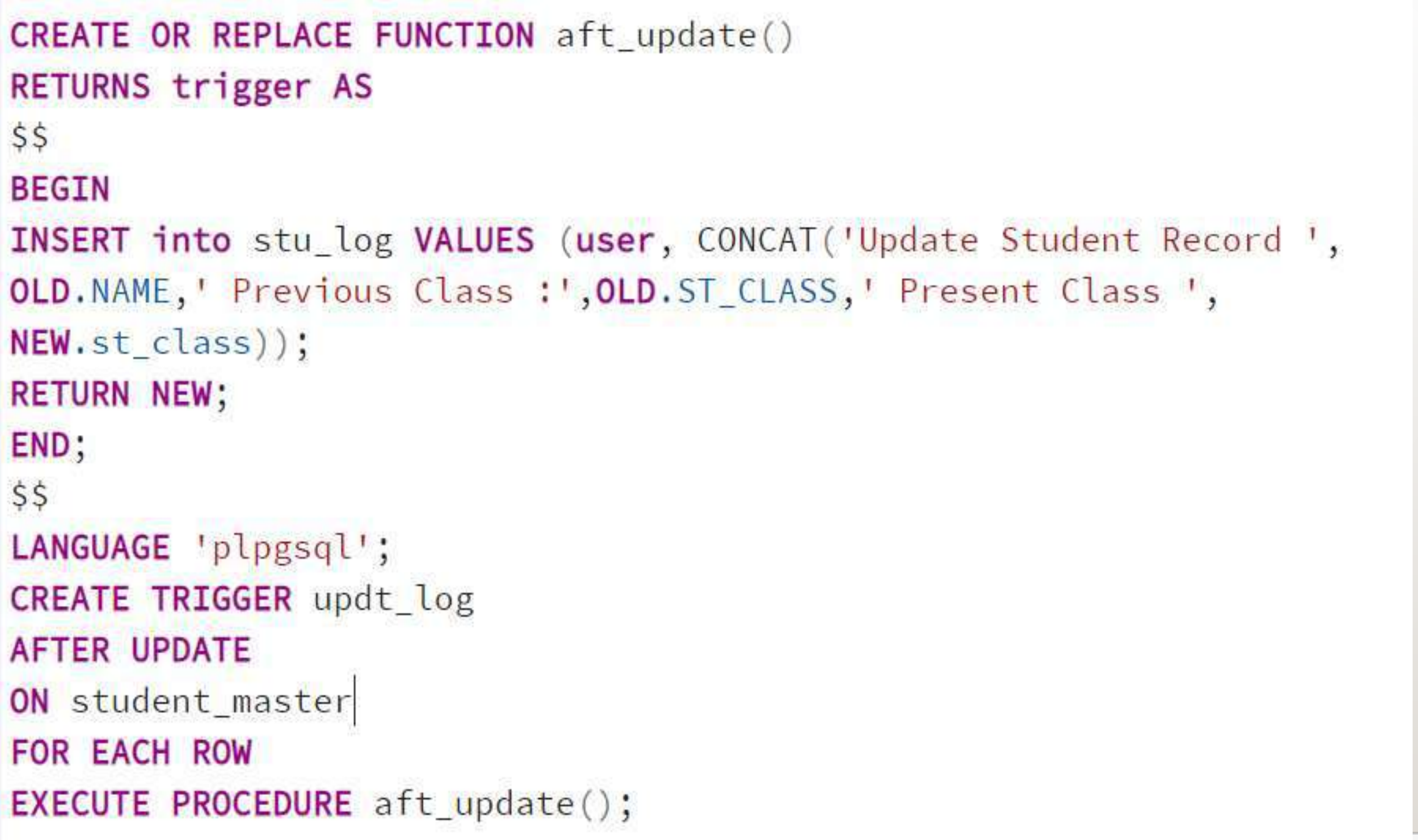


	eno integer	ename character varying (20)	job_id character varying (10)
1	1	sarala	cse
2	2	John	MBA

After Update

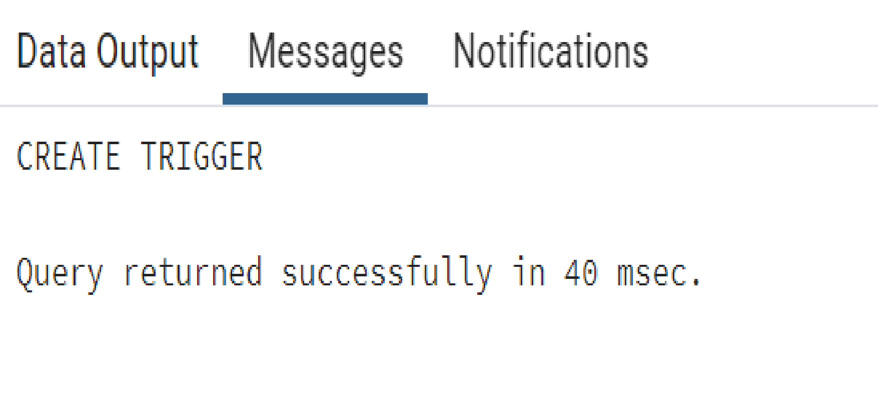
We have two tables student_mast and stu_log. student_mast have three columns STUDENT_ID, NAME, ST_CLASS. stu_log table has two columns user_id and description

Example



```
CREATE OR REPLACE FUNCTION aft_update()
RETURNS trigger AS
$$
BEGIN
INSERT into stu_log VALUES (user, CONCAT('Update Student Record ',
OLD.NAME,' Previous Class :',OLD.ST_CLASS,' Present Class ',
NEW.st_class));
RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';
CREATE TRIGGER updt_log
AFTER UPDATE
ON student_master
FOR EACH ROW
EXECUTE PROCEDURE aft_update();
```

Output

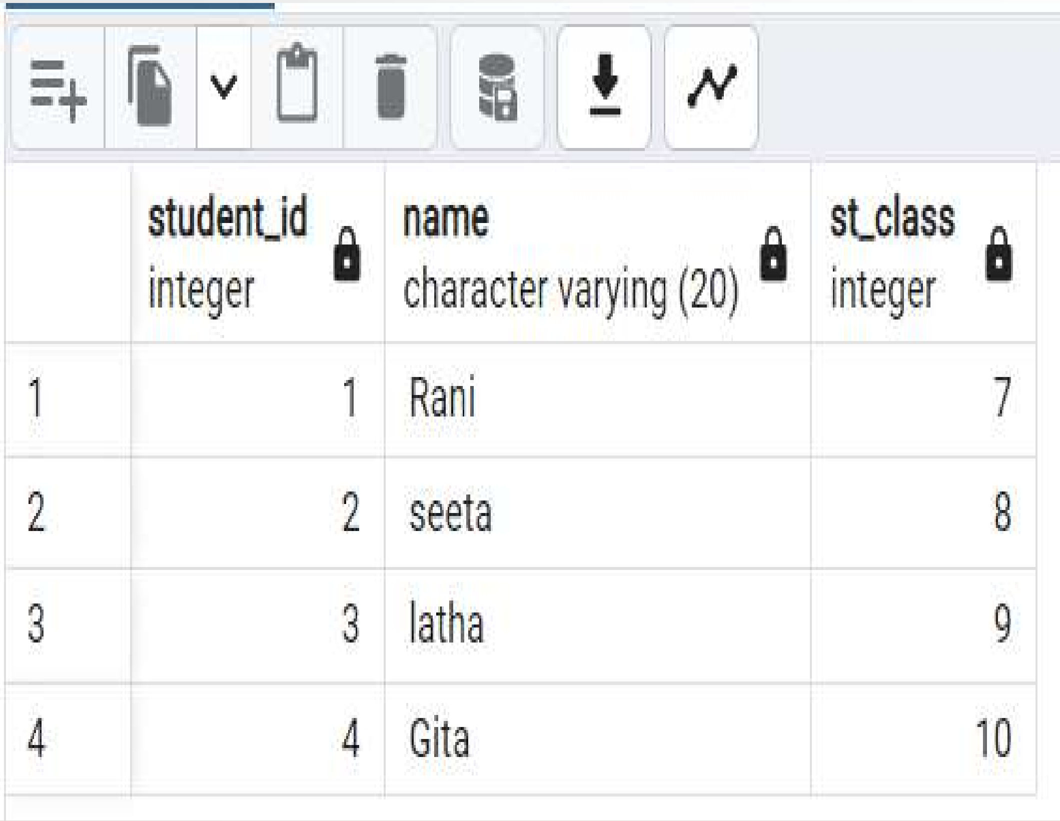


```
CREATE TRIGGER

Query returned successfully in 40 msec.
```

After Update

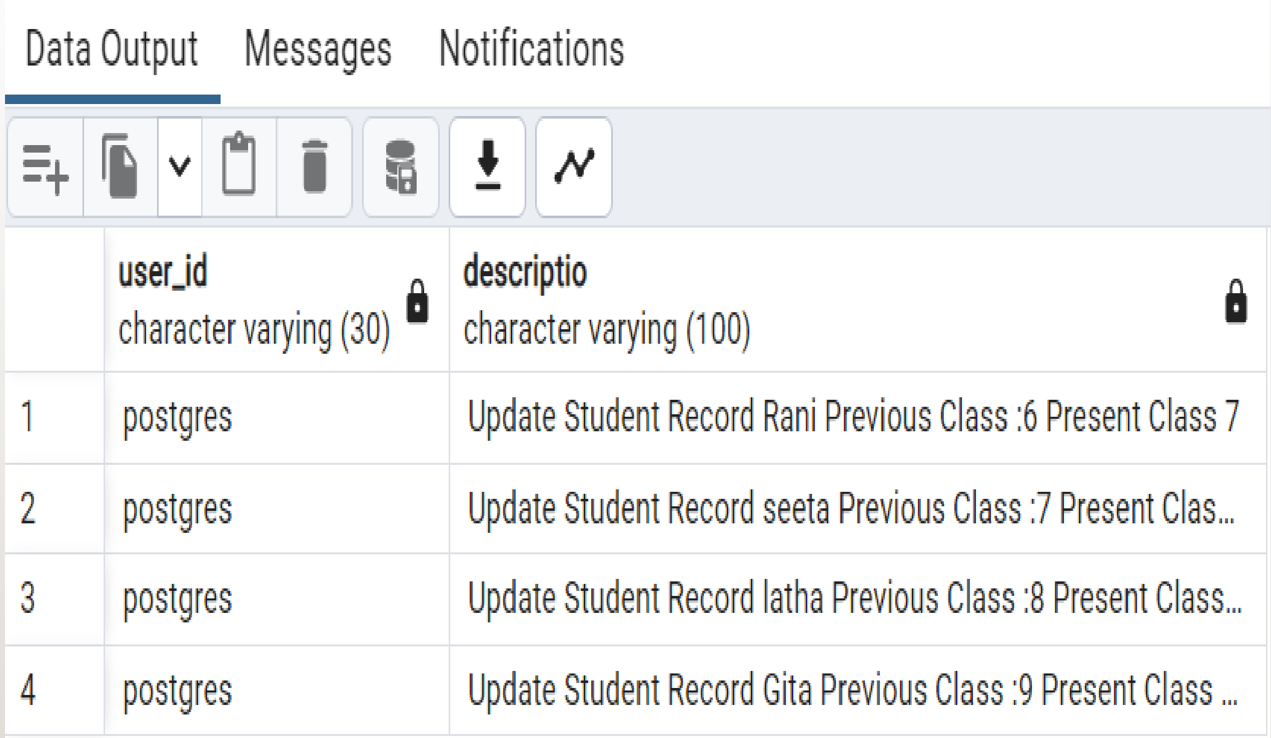
Select * from student_master;



	student_id integer	name character varying (20)	st_class integer
1	1	Rani	7
2	2	seeta	8
3	3	latha	9
4	4	Gita	10

update student_master set st_class=st_class+1;

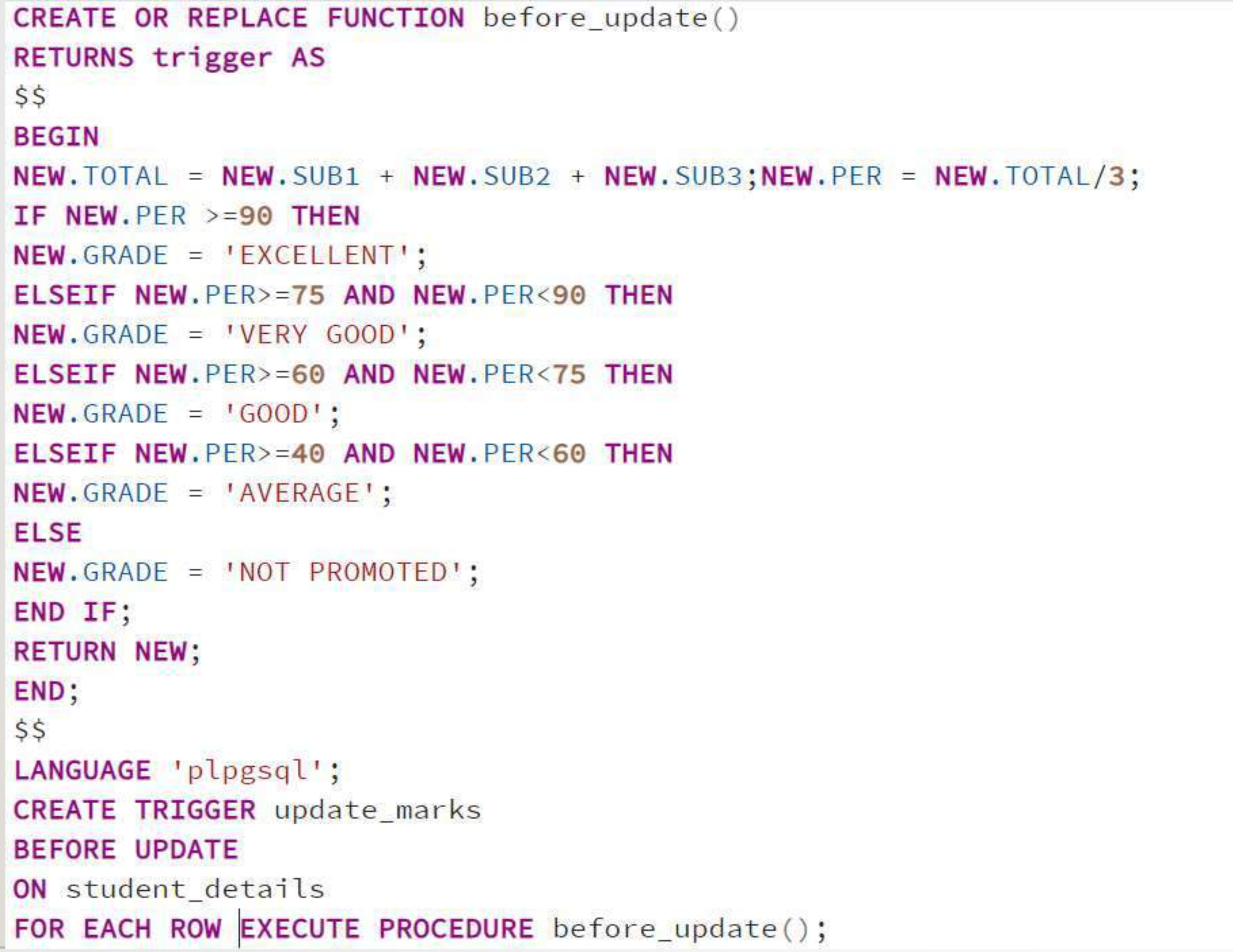
Select * from student_log;



	user_id character varying (30)	descriptio character varying (100)
1	postgres	Update Student Record Rani Previous Class :6 Present Class 7
2	postgres	Update Student Record seeta Previous Class :7 Present Clas...
3	postgres	Update Student Record latha Previous Class :8 Present Class...
4	postgres	Update Student Record Gita Previous Class :9 Present Class ...

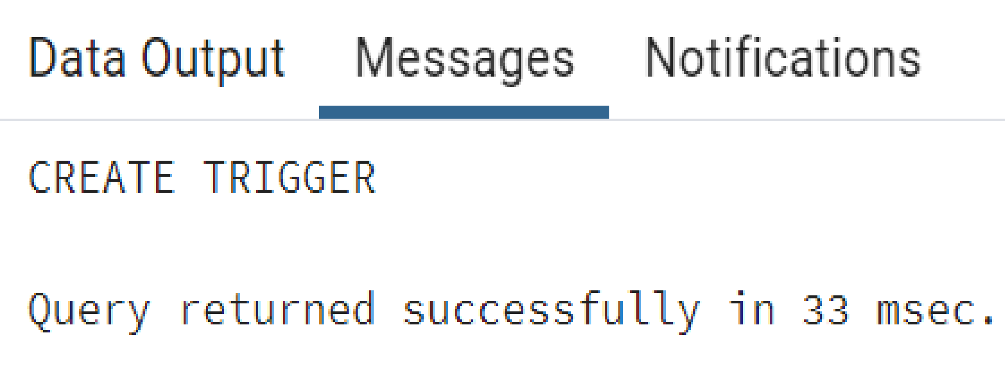
Before Update

Example:



```
CREATE OR REPLACE FUNCTION before_update()
RETURNS trigger AS
$$
BEGIN
NEW.TOTAL = NEW.SUB1 + NEW.SUB2 + NEW.SUB3;NEW.PER = NEW.TOTAL/3;
IF NEW.PER >=90 THEN
NEW.GRADE = 'EXCELLENT';
ELSEIF NEW.PER>=75 AND NEW.PER<90 THEN
NEW.GRADE = 'VERY GOOD';
ELSEIF NEW.PER>=60 AND NEW.PER<75 THEN
NEW.GRADE = 'GOOD';
ELSEIF NEW.PER>=40 AND NEW.PER<60 THEN
NEW.GRADE = 'AVERAGE';
ELSE
NEW.GRADE = 'NOT PROMOTED';
END IF;
RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';
CREATE TRIGGER update_marks
BEFORE UPDATE
ON student_details
FOR EACH ROW EXECUTE PROCEDURE before_update();
```

Output:

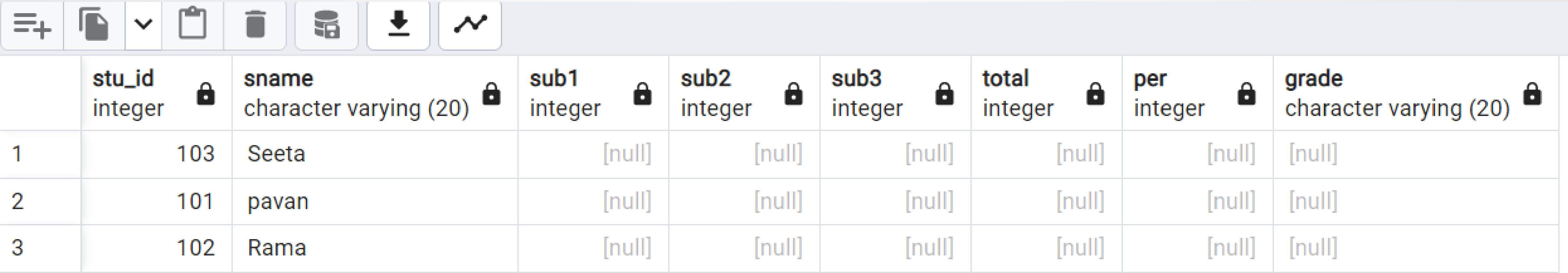


```
CREATE TRIGGER

Query returned successfully in 33 msec.
```

Before Update

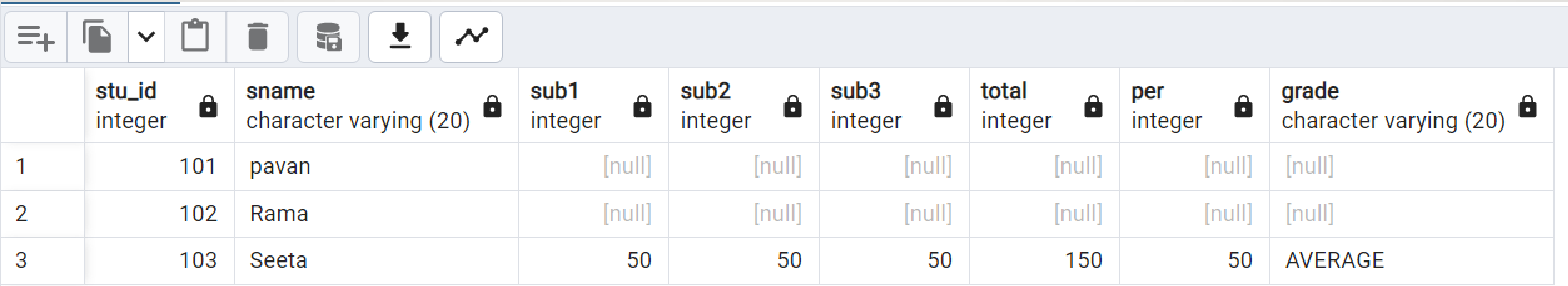
Before inserting marks



	stu_id integer	sname character varying (20)	sub1 integer	sub2 integer	sub3 integer	total integer	per integer	grade character varying (20)
1	103	Seeta	[null]	[null]	[null]	[null]	[null]	[null]
2	101	pavan	[null]	[null]	[null]	[null]	[null]	[null]
3	102	Rama	[null]	[null]	[null]	[null]	[null]	[null]

```
UPDATE STUDENT_MARKS SET SUB1 = 50, SUB2 = 50, SUB3 = 50 WHERE STU_ID = 103;
```

After inserting marks

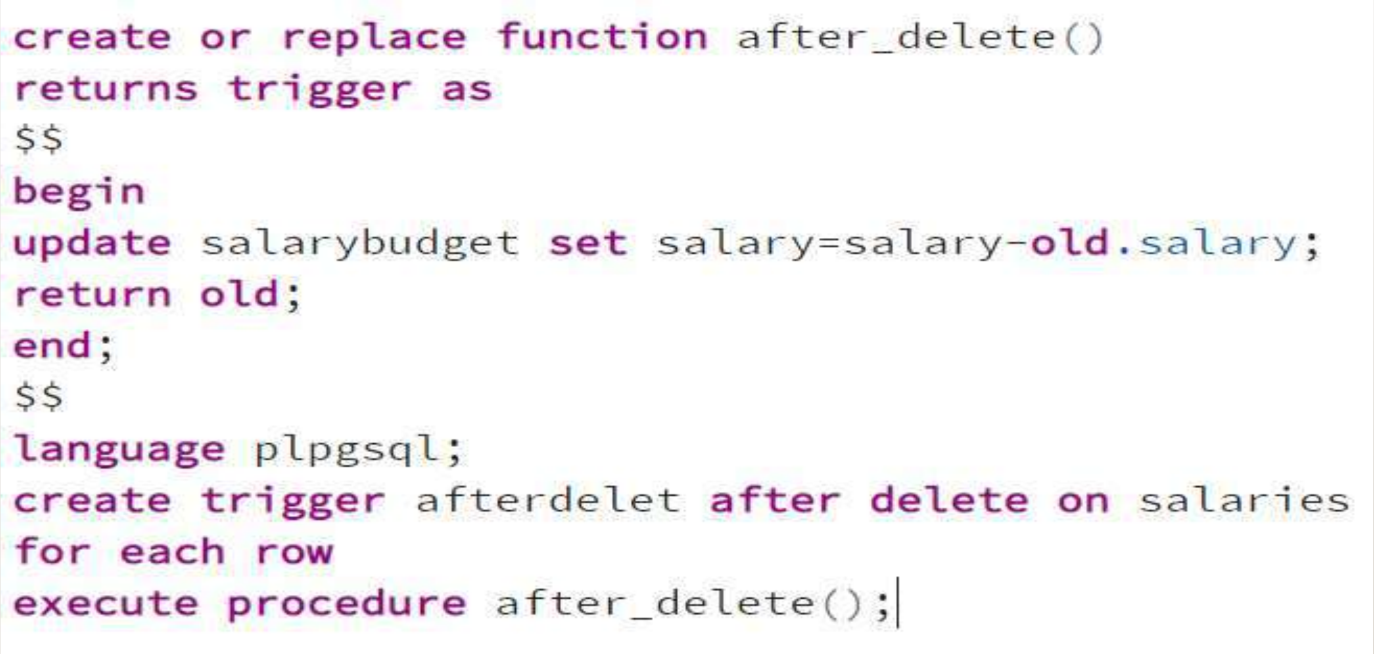


	stu_id integer	sname character varying (20)	sub1 integer	sub2 integer	sub3 integer	total integer	per integer	grade character varying (20)
1	101	pavan	[null]	[null]	[null]	[null]	[null]	[null]
2	102	Rama	[null]	[null]	[null]	[null]	[null]	[null]
3	103	Seeta	50	50	50	150	50	AVERAGE

After Delete

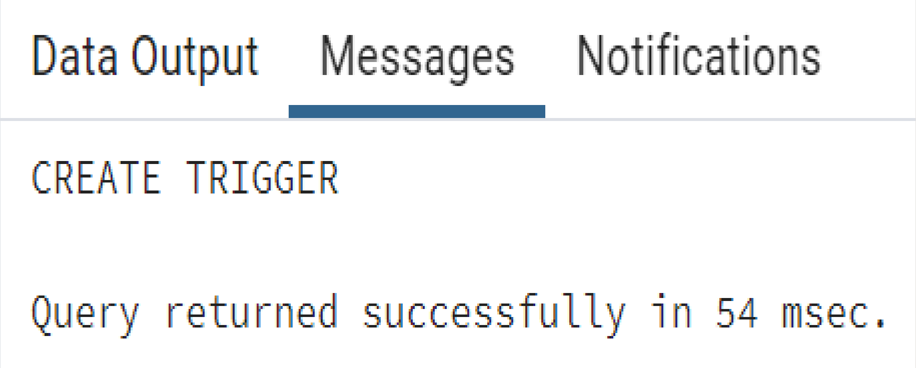
- Consider two tables named salaries and salary budget, salaries table contains the salary information of employees and salary budget contains sum of salaries in salaries table.
- Whenever an employee is delete then automatically from salarybudget the total amount will be deducted.

Example:



```
create or replace function after_delete()
returns trigger as
$$
begin
update salarybudget set salary=salary-old.salary;
return old;
end;
$$
language plpgsql;
create trigger afterdelet after delete on salaries
for each row
execute procedure after_delete();
```

Output

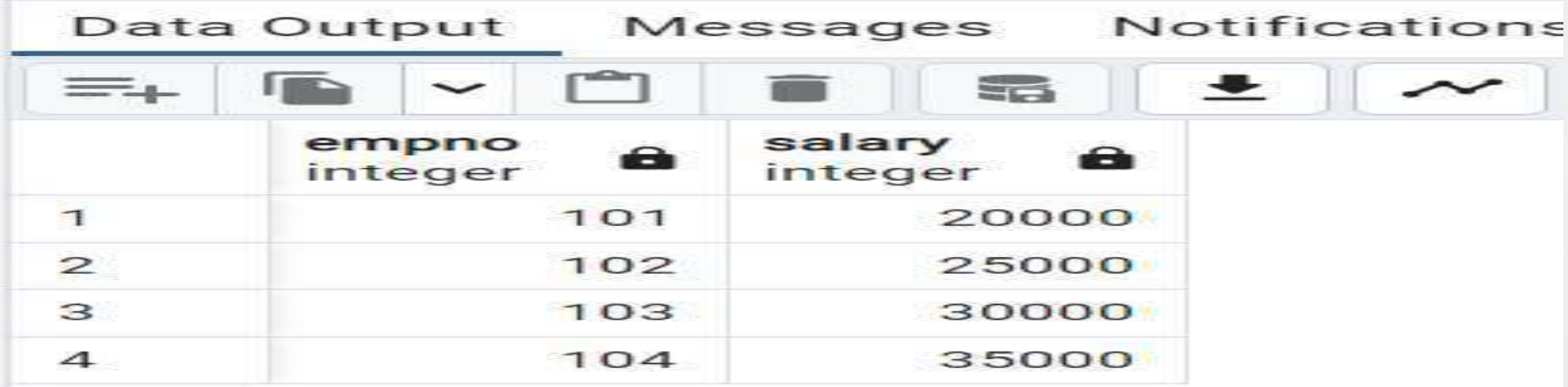


```
CREATE TRIGGER


Query returned successfully in 54 msec.
```

After Delete

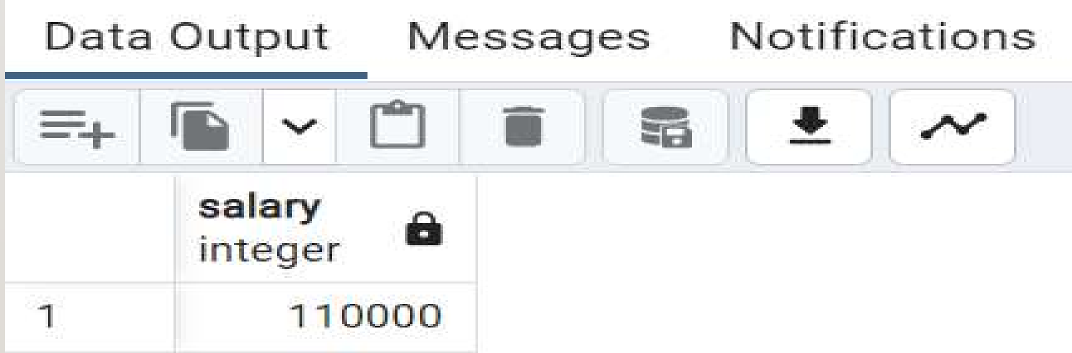
Select * from salaries



	empno integer	salary integer
1	101	20000
2	102	25000
3	103	30000
4	104	35000

insert into salarybudget (select sum(salary) from salaries);

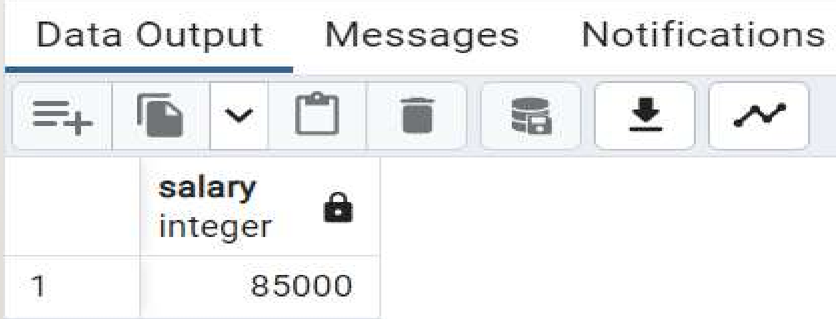
Select * from salarybudget;



	salary integer
1	110000

delete from salaries where empno=102;

select * from salarybudget;

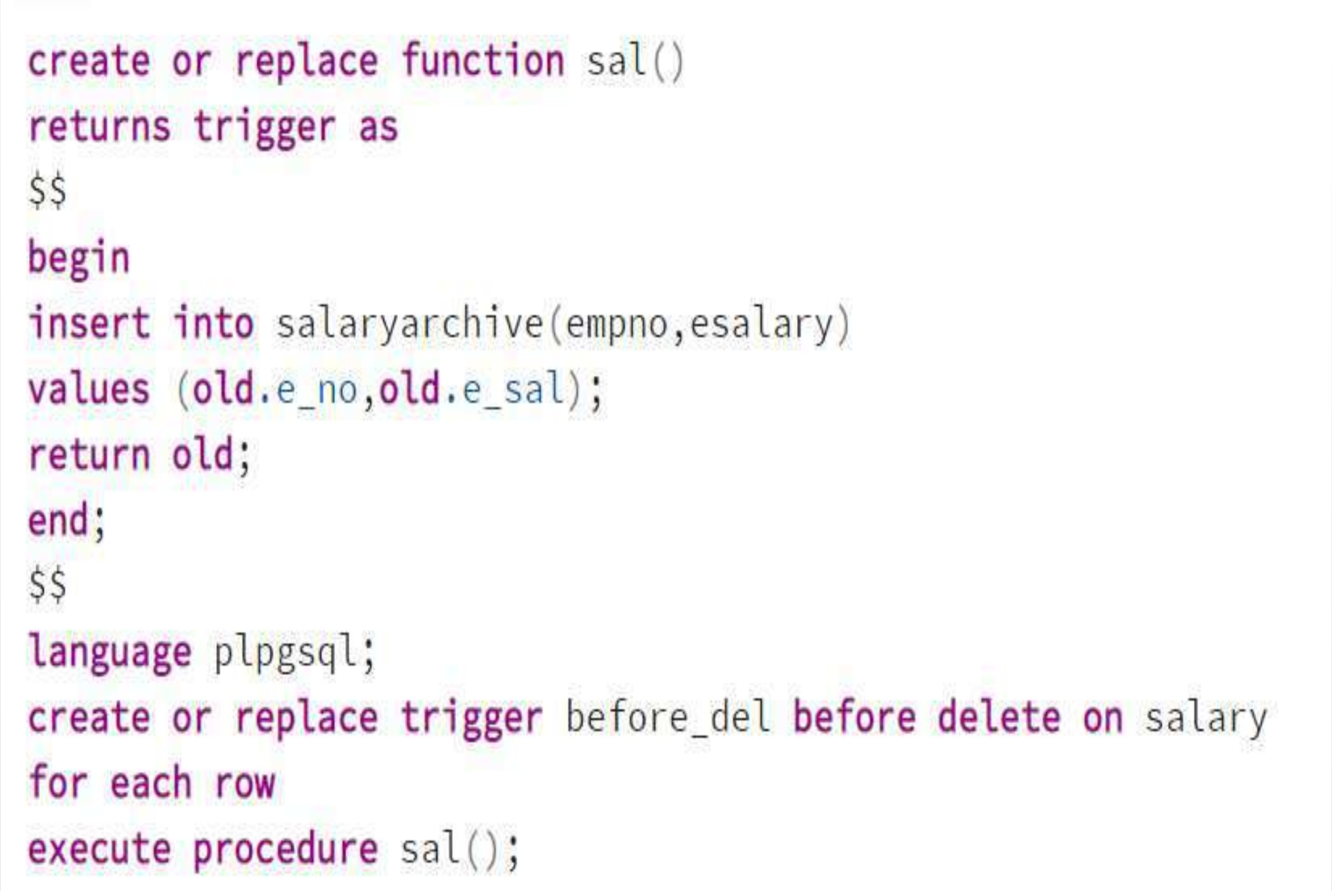


	salary integer
1	85000

Before Delete

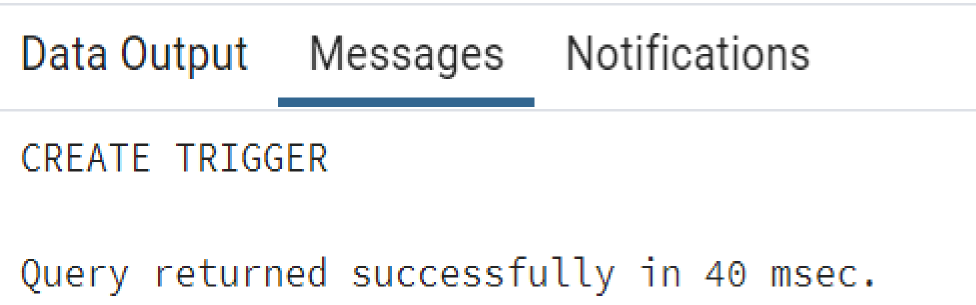
- Consider two tables salary and salaryarchive. When a row is deleted from salary table before deletion it should be inserted in salaryarchive table.

Example



```
create or replace function sal()
returns trigger as
$$
begin
insert into salaryarchive(empno,esalary)
values (old.e_no,old.e_sal);
return old;
end;
$$
language plpgsql;
create or replace trigger before_del before delete on salary
for each row
execute procedure sal();
```

Output

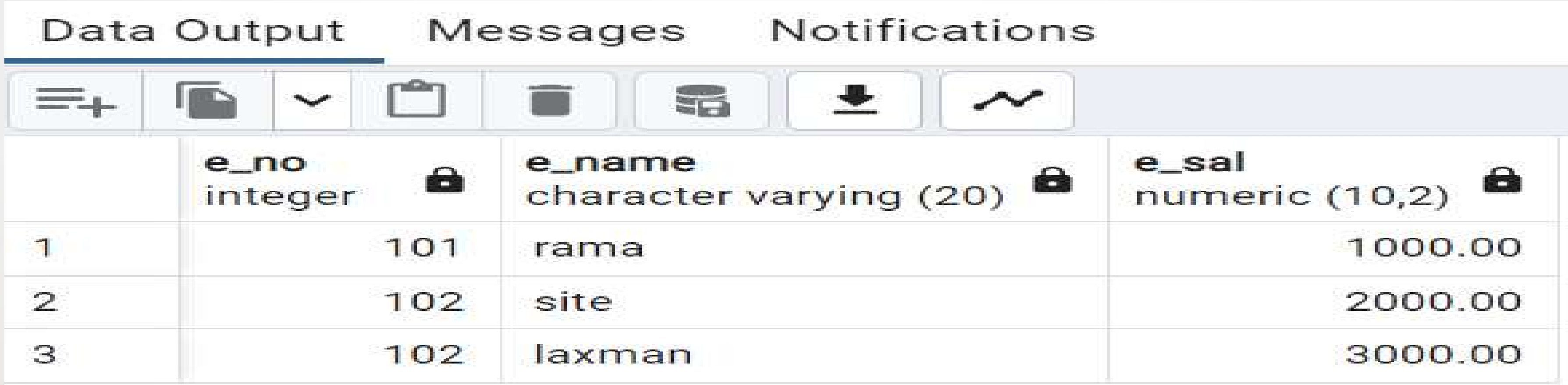


```
CREATE TRIGGER

Query returned successfully in 40 msec.
```

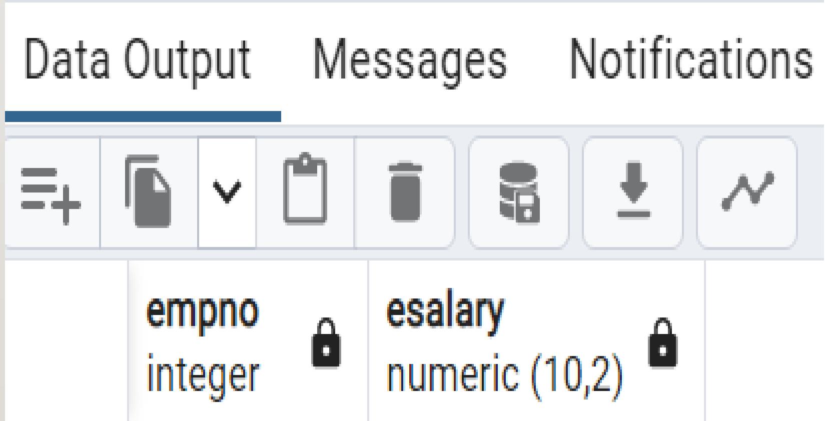
Before Delete

select * from salary;



	e_no integer	e_name character varying (20)	e_sal numeric (10,2)
1	101	rama	1000.00
2	102	site	2000.00
3	102	laxman	3000.00

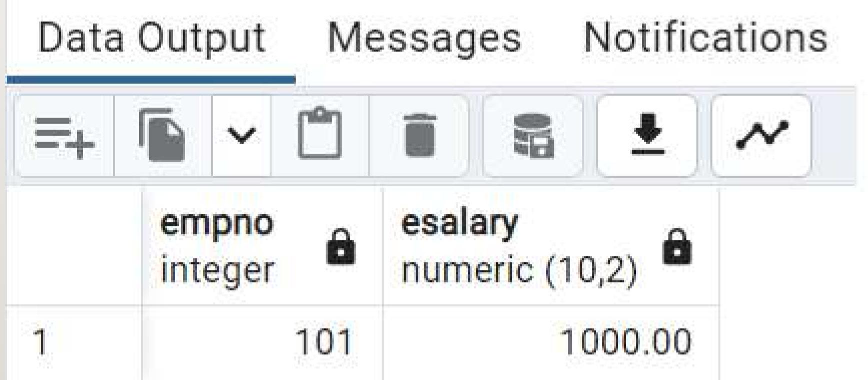
select * from salaryarchive;



	empno integer	esalary numeric (10,2)

select * from salaryarchive;

delete from salary where e_no=101;



	empno integer	esalary numeric (10,2)
1	101	1000.00

SUMMARY

A cursor keeps track of the position in the result set, and allows you to perform multiple operations row by row against a result set, with or without returning to the original table. A trigger is designed to check or change data based on a data modification or definition statement; it should't return data to the user.

SELF-ASSESSMENT QUESTIONS

1. Which of the following is a valid triggering event

(a) After Drop

(b) Before Update

(c) Instead of create

(d) After Select

Answer: B

2. An SQL _____ refers to a program that retrieves and processes one row at a time, based on the results of the SQL statement.

(a) Cursor.

(b) Function.

(c) Procedure

(d) View

Answer: A

TERMINAL QUESTIONS

1. **Explain usage of Triggers.**

2. **List out the types of cursors in Postgresql**

3. **Analyze**

4. **Summarize**