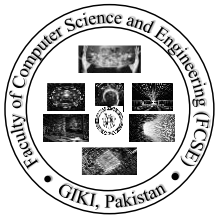
CS232L – Database Management System

LAB 09

TRIGGERS, VIEWS, INDEXES

Prepared By: Amna Arooj FCSE Computer Engineer

Ghulam Ishaq Khan Institute of Engineering Sciences



Faculty of Computer Science & Engineering

CS232L – Database Management system Lab

Lab 9 – Triggers, Views, Indexes

Objective

The objective of this session is to learn about PostgreSQL triggers, views and Indexes. What they are and how they are used using different postgresql examples.

Instructions

- Open the handout/lab manual in front of a computer in the lab during the session.
- Practice each new command by completing the examples and exercises.
- Turn in the answers for all the exercise problems as your lab report.
- When answering problems, indicate the commands you entered, and the output displayed.
- Try to practice and revise all the concepts covered in all previous sessions before coming to the lab to avoid unnecessary ambiguities.

9.1 PostgreSQL CREATE TRIGGER

To create a new trigger in PostgreSQL, you follow these steps:

- First, create a trigger function using CREATE FUNCTION statement.
- Second, bind the trigger function to a table by using CREATE TRIGGER statement.

Create trigger function syntax

A trigger function is similar to a regular user-defined function. However, a trigger function does not take any arguments and has a return value with the type trigger.

The following illustrates the syntax of creating trigger function:

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

A trigger function receives data about its calling environment through a special structure called Trigger Data which contains a set of local variables. For example, OLD and NEW represent the states of the row in the table before or after the triggering event. Once you define a trigger function, you can bind it to one or more trigger events such as INSERT, UPDATE, and DELETE.

Introduction to PostgreSQL CREATE TRIGGER statement

The CREATE TRIGGER statement creates a new trigger. The following illustrates the basic syntax of the CREATE TRIGGER statement:

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

In this syntax: First, specify the name of the trigger after the TRIGGER keywords. Second, specify the timing that causes the trigger to fire. It can be BEFORE or AFTER an event occurs. Third, specify the event that invokes the trigger. The event can be INSERT , DELETE, UPDATE or TRUNCATE. Fourth, specify the name of the table associated with the trigger after the ON keyword. Fifth, specify the type of triggers which can be:

- Row-level trigger that is specified by the FOR EACH ROW clause.
- Statement-level trigger that is specified by the FOR EACH STATEMENT clause.

A row-level trigger is fired for each row while a statement-level trigger is fired for each transaction. Suppose a table has 100 rows and two triggers that will be fired when a DELETE event occurs.

If the DELETE statement deletes 100 rows, the row-level trigger will fire 100 times, once for each deleted row. On the other hand, a statement-level trigger will be fired for one time regardless of how many rows are deleted.

Finally, specify the name of the trigger function after the EXECUTE PROCEDURE keywords.

PostgreSQL CREATE TRIGGER example

The following statement create a new table called employees:

```
DROP TABLE IF EXISTS employees;

CREATE TABLE employees(
   id INT GENERATED ALWAYS AS IDENTITY,
   first_name VARCHAR(40) NOT NULL,
   last_name VARCHAR(40) NOT NULL,
   PRIMARY KEY(id)
);
```

Suppose that when the name of an employee changes, you want to log the changes in a separate table called employee_audits :

```
CREATE TABLE employee_audits (
   id INT GENERATED ALWAYS AS IDENTITY,
   employee_id INT NOT NULL,
   last_name VARCHAR(40) NOT NULL,
   changed_on TIMESTAMP(6) NOT NULL
);
```

First, create a new function called log_last_name_changes:

```
CREATE OR REPLACE FUNCTION log_last_name_changes()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
  AS
$$
BEGIN
        IF NEW.last_name <> OLD.last_name THEN
                INSERT INTO
employee_audits(employee_id,last_name,changed_on)
                VALUES(OLD.id,OLD.last_name,now());
        END IF;

        RETURN NEW;
END;
$$
```

The function inserts the old last name into the employee_audits table including employee id, last name, and the time of change if the last name of an employee changes.

The OLD represents the row before update while the NEW represents the new row that will be updated. The OLD.last_name returns the last name before the update and the NEW.last_name returns the new last name.

Second, bind the trigger function to the employees table. The trigger name is last_name_changes. Before the value of the last_name column is updated, the trigger function is automatically invoked to log the changes.

```
CREATE TRIGGER last_name_changes
  BEFORE UPDATE
  ON employees
  FOR EACH ROW
  EXECUTE PROCEDURE log_last_name_changes();
```

Third, insert some rows into the employees table:

```
INSERT INTO employees (first_name, last_name)
VALUES ('John', 'Doe');

INSERT INTO employees (first_name, last_name)
VALUES ('Lily', 'Bush');
```

Fourth, examine the contents of the employees table:

```
SELECT * FROM employees;
```

	id integer	first_name character varying (40)	last_name character varying (40)
1	1	John	Doe
2	2	Lily	Bush

Suppose that Lily Bush changes her last name to Lily Brown. Fifth, update Lily's last name to the new one:

```
UPDATE employees
SET last_name = 'Brown'
WHERE ID =2;
```

Seventh, check if the last name of Lily has been updated:

```
SELECT * FROM employees;
```

	id integer	first_name character varying (40)	last_name character varying (40)
1	1	John	Doe
2	2	Lily	Brown

As you can see from the output, Lily's last name has been updated.

Eighth, verify the contents of the employee_audits table:

```
SELECT * FROM employee_audits;
```

	id integer	employee_id integer	last_name character varying (40)	changed_on timestamp without time zone
1	1	2	Bush	2020-07-30 17:29:04.248925

The change was logged in the employee_audits table by the trigger.

PostgreSQL DROP TRIGGER example

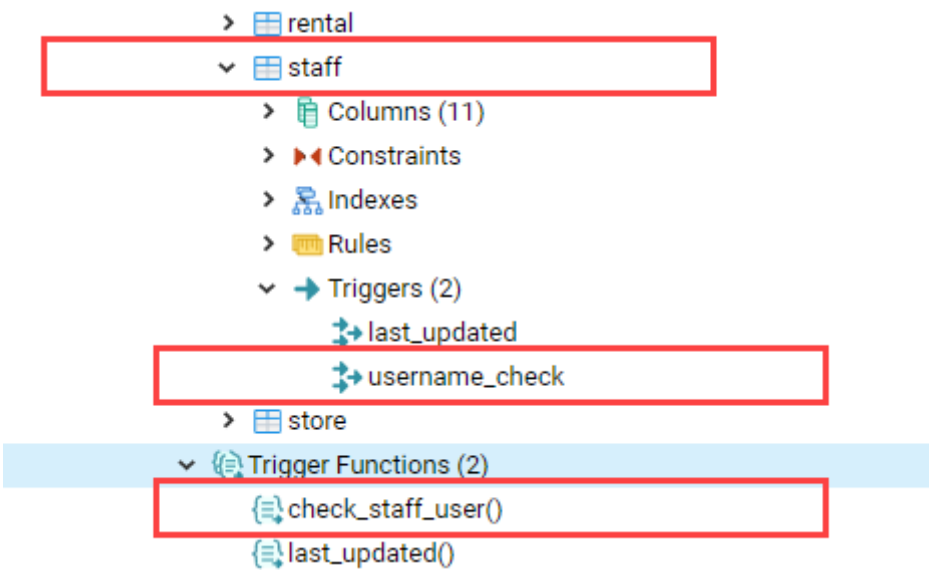
First, create a function that validates the username of a staff. The username of staff must not be null and its length must be at least 8.

```
CREATE FUNCTION check_staff_user()
    RETURNS TRIGGER
AS $$
BEGIN
    IF length(NEW.username) < 8 OR NEW.username IS NULL THEN
        RAISE EXCEPTION 'The username cannot be less than 8
characters';
    END IF;
    IF NEW.NAME IS NULL THEN
        RAISE EXCEPTION 'Username cannot be NULL';
    END IF;
    RETURN NEW;
END;
$$
LANGUAGE plpgsql;
```

Code language: SQL (Structured Query Language) (sql)

Second, create a new trigger on the staff table to check the username of a staff. This trigger will fire whenever you insert or update a row in the staff table (from the sample database):

```
CREATE TRIGGER username_check
    BEFORE INSERT OR UPDATE
ON staff
FOR EACH ROW
    EXECUTE PROCEDURE check_staff_user();
```



Third, use the DROP TRIGGER statement to delete the username_check trigger:

```
DROP TRIGGER username_check
ON staff;
```

9.2 Introduction to the PostgreSQL Views

A view is a stored query. A view can be accessed as a virtual table in PostgreSQL. In other words, a PostgreSQL view is a logical table that represents data of one or more underlying tables through a SELECT statement.

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 the underlying table change.

Creating PostgreSQL Views

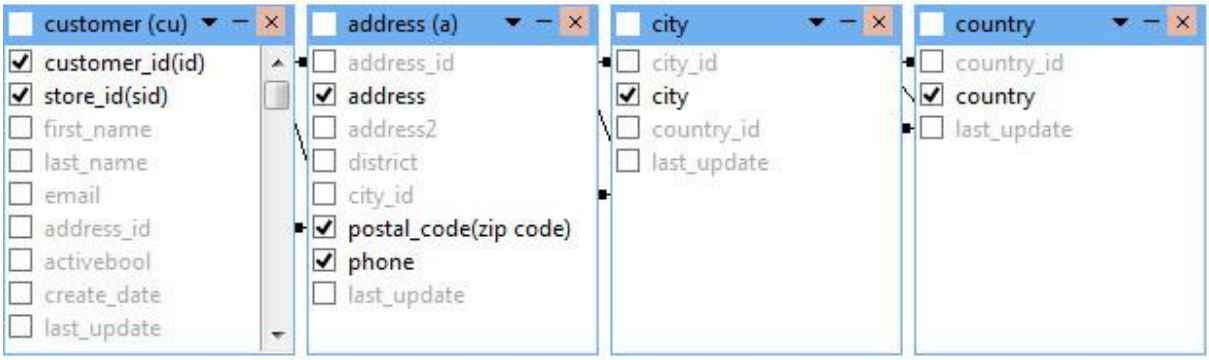
To create a view, we use CREATE VIEW statement. The simplest syntax of the CREATE VIEW statement is as follows:

```
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.

For example, in our sample database i.e. dvd rental we have four tables:

1. customer – stores all customer data
2. address – stores address of customers
3. city – stores city data
4. country– stores country data



If you want to get complete customers data, you normally construct a join statement as follows:

```
SELECT cu.customer_id AS id,
  cu.first_name || ' ' || cu.last_name AS name,
  a.address,
  a.postal_code AS "zip code",
  a.phone,
  city.city,
  country.country,
     CASE
        WHEN cu.activebool THEN 'active'
        ELSE ''
     END AS notes,
  cu.store_id AS sid
 FROM customer cu
   INNER JOIN address a USING (address_id)
   INNER JOIN city USING (city_id)
   INNER JOIN country USING (country_id);
```

The result of the query is as shown in the screenshot below:

id	name	address	zip code	phone	city	country	notes	sid
1	Mary Smith	1913 Hanoi Way	35200	28303384290	Sasebo	Japan	active	1
2	Patricia Johnson	1121 Loja Avenue	17886	838635286649	San Bernardino	United States	active	1
3	Linda Williams	692 Joliet Street	83579	448477190408	Athenai	Greece	active	1
4	Barbara Jones	1566 Inegl Manor	53561	705814003527	Myingyan	Myanmar	active	2
5	Elizabeth Brown	53 Idfu Parkway	42399	10655648674	Nantou	Taiwan	active	1
6	Jennifer Davis	1795 Santiago de Composte	18743	860452626434	Laredo	United States	active	2
7	Maria Miller	900 Santiago de Compostela	93896	716571220373	Kragujevac	Yugoslavia	active	1
8	Susan Wilson	478 Joliet Way	77948	657282285970	Hamilton	New Zealand	active	2
9	Margaret Moore	613 Korolev Drive	45844	380657522649	Masqat	Oman	active	2

This query is quite complex. However, you can create a view named customer_master as follows:

```
CREATE VIEW customer_master AS
  SELECT cu.customer_id AS id,
    cu.first_name || ' ' || cu.last_name AS name,
    a.address,
    a.postal_code AS "zip code",
    a.phone,
    city.city,
    country.country,
       CASE
          WHEN cu.activebool THEN 'active'
          ELSE ''
       END AS notes,
    cu.store_id AS sid
   FROM customer cu
     INNER JOIN address a USING (address_id)
     INNER JOIN city USING (city_id)
     INNER JOIN country USING (country_id);
```

From now on, whenever you need to get complete customer data, you just query it from the view by executing the following simple SELECT statement:

```
SELECT
        *
FROM
        customer_master;
```

This query produces the same result as the complex one with the joins above.

To change the definition of a view, you use the ALTER VIEW statement. For example, you can change the name of the view from customer_master to customer_info by using the following statement:

```
ALTER VIEW customer_master RENAME TO customer_info;
```

PostgreSQL allows you to set a default value for a column name, change the view's schema, set or reset options of a view. For detailed information on the altering view's definition, check it out the PostgreSQL ALTER VIEW statement.

Removing PostgreSQL Views

To remove an existing view in PostgreSQL, you use DROP VIEW statement as follows:

```
DROP VIEW [ IF EXISTS ] view_name;
```

You specify the name of the view that you want to remove after DROP VIEW clause. Removing a view that does not exist in the database will result in an error. To avoid this, you normally add IF EXISTS option to the statement to instruct PostgreSQL to remove the view if it exists, otherwise, do nothing.

For example, to remove the customer_info view that you have created, you execute the following query:

```
DROP VIEW IF EXISTS customer_info;
```

The view customer_infois removed from the database.

Creating PostgreSQL Updatable Views

First, create a new updatable view name usa_cities using CREATE VIEW statement. This view contains all cities in the city table locating in the USA whose country id is 103.

```
CREATE VIEW usa_cities AS SELECT
        city,
        country_id
FROM
        city
WHERE
        country_id = 103;
```

Next, check the data in the usa_cities view by executing the following SELECT statement:

```
SELECT
        *
FROM
        usa_cities;
```

usa_cities view using the following INSERT statement:

```
INSERT INTO usa_cities (city, country_id)
VALUES('San Jose', 103);
```

After that, check the contents of the city table:

```
SELECT
        city,
        country_id
FROM
        city
WHERE
        country_id = 103
ORDER BY
        last_update DESC;
```

We have a newly entry added to the city table.

city	country_id
San Jose	103
Arlington	103
Augusta-Richmond County	103
Aurora	103
Bellevue	103
Brockton	103
Cape Coral	103
Citrus Heights	103
Clarksville	103

Finally, delete the entry that has been added through the usa_cities view.

```
DELETE
FROM
        usa_cities
WHERE
        city = 'San Jose';
```

The entry has been deleted from the city table through the usa_cities view.

9.3 PostgreSQL CREATE INDEX

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

For example, if you want to reference all pages in a book that discusses a certain topic, you must first refer to the index, which lists all topics alphabetically and then refer to one or more specific page numbers.

An index helps to speed up SELECT queries and WHERE clauses; however, it slows down data input, with UPDATE and INSERT statements. Indexes can be created or dropped with no effect on the data.

Phonebook analogy and index

Suppose you need to look up John Doe's phone number in a phone book. Assuming that the names on the phone book are in alphabetical order. To find John Doe's phone number, you first look for the page where the last name is Doe, then look for the first name John, and finally, get his phone number. If the names on the phone book were not ordered alphabetically, you would have to go through all pages and check every name until you find John Doe's phone number. This is called a sequential scan in which you go over all entries until you find the one that you are looking for. Like a phonebook, the data stored in the table should be organized in a particular order to speed up various searches. This is why indexes come into play. By definition, An index is a separated data structure that speeds up the data retrieval on a table at the cost of additional writes and storage to maintain the index.

PostgreSQL CREATE INDEX Syntax
The syntax to create an index using the CREATE INDEX statement in PostgreSQL is:

```
CREATE [UNIQUE] INDEX [CONCURRENTLY] index_name
  [ USING BTREE | HASH | GIST | SPGIST | GIN ]
  ON table_name
    (index_col1 [ASC | DESC] [NULLS {FIRST | LAST }],
     index_col2 [ASC | DESC] [NULLS {FIRST | LAST }],
     ...
     index_col_n [ASC | DESC] [NULLS {FIRST | LAST }]);
```

UNIQUE
Optional. The *UNIQUE* modifier indicates that the combination of values in the indexed columns must be unique.

CONCURRENTLY
Optional. When the index is created, it will not lock the table. By default, the table is locked while the index is being created.

index_name
The name to assign to the index.

table_name
The name of the table in which to create the index.

index_col1, index_col2, ... index_col_n
The columns to use in the index.

ASC
Optional. The index is sorted in ascending order for that column.

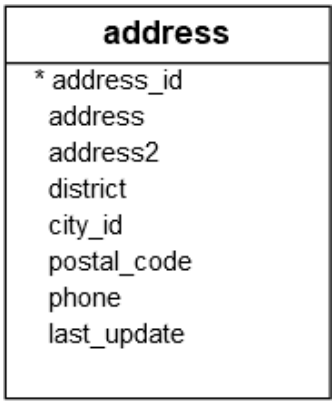
DESC
Optional. The index is sorted in descending order for that column.

If a column contains NULL, you can specify NULLS FIRST or NULLS LAST option. The NULLS FIRST is the default when DESC is specified and NULLS LAST is the default when DESC is not specified.

To check if a query uses an index or not, you use the EXPLAIN statement.

PostgreSQL CREATE INDEX statement example

We will use the address table from the sample database for the demonstration.



The following query finds the address whose phone number is 223664661973:

```
SELECT * FROM address
WHERE phone = '223664661973';
```

It is obvious that the database engine has to scan the whole address table to look for the address because there is no index available for the phone column. To show the query plan, you use the EXPLAIN statement as follows:

```
EXPLAIN SELECT *
FROM address
WHERE phone = '223664661973';
```

QUERY PLAN
Seq Scan on address (cost=0.00..15.54 rows=1 width=61)
Filter: ((phone)::text = '223664661973'::text)

To create an index for the values in the phone column of the address table, you use the following statement:

```
CREATE INDEX idx_address_phone
ON address(phone);
```

Now, if you execute the query again, you will find that the database engine uses the index for lookup:

```
EXPLAIN SELECT *
FROM address
WHERE phone = '223664661973';
```

QUERY PLAN
Index Scan using idx_address_phone on address (cost=0.28..8.29 rows=1 width=61)
Index Cond: ((phone)::text = '223664661973'::text)

PostgreSQL Index Types

PostgreSQL has several index types: B-tree, Hash, GiST, SP-GiST, GIN, and BRIN. Each index type uses a different storage structure and algorithm to cope with different kinds of

queries. When you use the CREATE INDEX statement without specifying the index type, PostgreSQL uses the B-tree index type by default because it is best to fit the most common queries.

B-tree indexes

B-tree is a self-balancing tree that maintains sorted data and allows searches, insertions, deletions, and sequential access in logarithmic time. PostgreSQL query planner will consider using a B-tree index whenever index columns are involved in a comparison. In addition, the query planner can use a B-tree index for queries that involve a pattern-matching operator LIKE and ~ if the pattern is a constant and is anchor at the beginning of the pattern.

Hash indexes

Hash indexes can handle only simple equality comparison (=). It means that whenever an indexed column is involved in a comparison using the equal(=) operator, the query planner will consider using a hash index. To create a hash index, you use the CREATE INDEX statement with the HASH index type in the USING clause as follows:

```
CREATE INDEX index_name

ON table_name USING HASH (indexed_column);
```

GIN indexes

GIN stands for generalized inverted indexes. These are most useful when you have *multiple values stored in a single column*, for example, hstore, array, jsonb, and range types.

BRIN

BRIN stands for block range indexes. BRIN is much smaller and less costly to maintain in comparison with a B-tree index. BRIN allows the use of an index on a very large table that would previously be impractical using a B-tree without horizontal partitioning. BRIN is often used on a column that has a linear sort order, for example, the created date column of the sales order table.

GiST Indexes

GiST stands for Generalized Search Tree. GiST indexes allow the building of general tree structures. GiST indexes are useful in indexing *geometric data types and full-text searches.*

SP-GiST Indexes

SP-GiST stands for space-partitioned GiST. SP-GiST supports partitioned search trees that facilitate the development of a wide range of different *non-balanced data structures.* SP-GiST indexes are most useful for data that has a *natural clustering element* to it and is also not an equally balanced tree, for example, GIS, multimedia, phone routing, and IP routing.

PostgreSQL UNIQUE index

When you define a UNIQUE index for a column, the column cannot store multiple rows with the same values. If you define a UNIQUE index for two or more columns, the combined values in these columns cannot be duplicated in multiple rows. PostgreSQL treats NULLs as distinct values, therefore, you can have multiple NULL values in a column with a UNIQUE index.

When you define a primary key or a unique constraint for a table, PostgreSQL automatically creates a corresponding UNIQUE index. Note that only B-tree indexes can be declared as unique indexes.

The following statement creates a table called employees :

```
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(255) NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE
);
```

In this statement, the employee_id is the primary key column and email column has a unique constraint, therefore, PostgreSQL created two UNIQUE indexes, one for each column. To show indexes of the employees table, you use the following statement:

```
SELECT
    tablename,
    indexname,
    indexdef
FROM
    pg_indexes
WHERE
    tablename = 'employees';
```

Here is the output:

tablename	indexname	indexdef
employees	employees_pkey	CREATE UNIQUE INDEX employees_pkey ON public.employees USING btree (employee_id)
employees	employees_email_key	CREATE UNIQUE INDEX employees_email_key ON public.employees USING btree (email)

PostgreSQL UNIQUE index – single column example

The following statement adds the mobile_phone column to the employees table:

```
ALTER TABLE employees
ADD mobile_phone VARCHAR(20);
```

To ensure that the mobile phone numbers are distinct for all employees, you define a UNIQUE index for the mobile_phone column as follows:

```
CREATE UNIQUE INDEX idx_employees_mobile_phone
ON employees(mobile_phone);
```

Let’s take a test.

First, insert a new row into the employees table:

```
INSERT INTO employees(first_name, last_name, email, mobile_phone)
VALUES ('John','Doe','john.doe@postgresqltutorial.com', '(408)-555-1234');
```

Second, attempt to insert another row with the same phone number::

```
INSERT INTO employees(first_name, last_name, email, mobile_phone)
VALUES ('Mary','Jane','mary.jane@postgresqltutorial.com', '(408)-555-
1234');
```

PostgreSQL issues the following error due to the duplicate mobile phone number:

```
ERROR:  duplicate key value violates unique constraint
"idx_employees_mobile_phone"
DETAIL:  Key (mobile_phone)=((408)-555-1234) already exists
```

PostgreSQL UNIQUE index – multiple columns example

The following statement adds two new columns called work_phone and extension to the employees table:

```
ALTER TABLE employees
ADD work_phone VARCHAR(20),
ADD extension VARCHAR(5);
```

Multiple employees can share the same work phone number. However, they cannot have the same extension number. To enforce this rule, you can define a UNIQUE index on both work_phone and extension columns:

```
CREATE UNIQUE INDEX idx_employees_workphone
ON employees(work_phone, extension);
```

To test this index, first, insert a row into the employees table:

```
INSERT INTO employees(first_name, last_name, work_phone, extension)
VALUES('Lily', 'Bush', '(408)-333-1234','1212');
```

Second, insert another employee with the same work phone number but a different extension:

```
INSERT INTO employees(first_name, last_name, work_phone, extension)
VALUES('Joan', 'Doe', '(408)-333-1234','1211');
```

The statement works because the combination of values in the work_phone and extension column is unique. Third, attempt to insert a row with the same values in both work_phone and extension columns that already exist in the employees table:

```
INSERT INTO employees(first_name, last_name, work_phone, extension)
VALUES('Tommy', 'Stark', '(408)-333-1234','1211');
```

PostgreSQL issued the following error:

```
ERROR:  duplicate key value violates unique constraint
"idx_employees_workphone"
```

```
DETAIL:  Key (work_phone, extension)=((408)-333-1234, 1211) already exists.
```