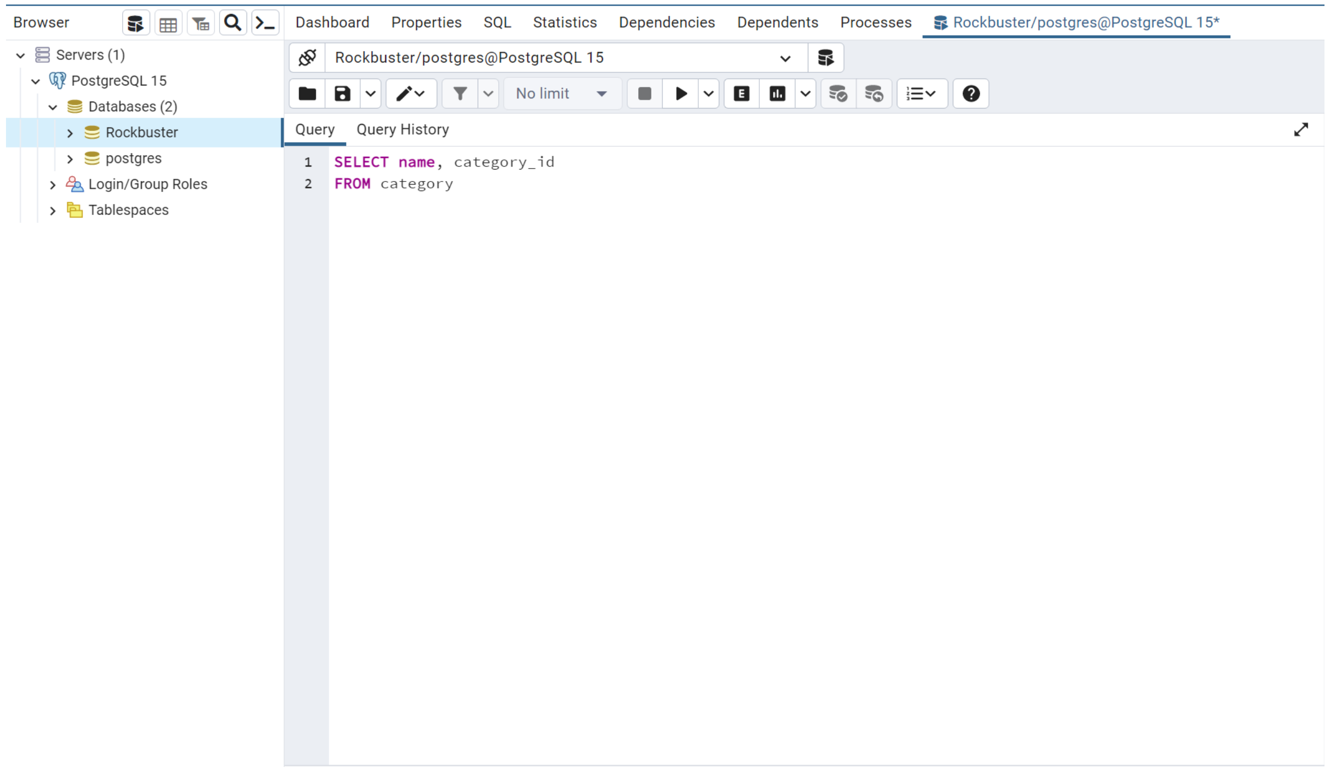
Step 1: Writing a SELECT command to find out the genres present in the category table.

SQL Statement



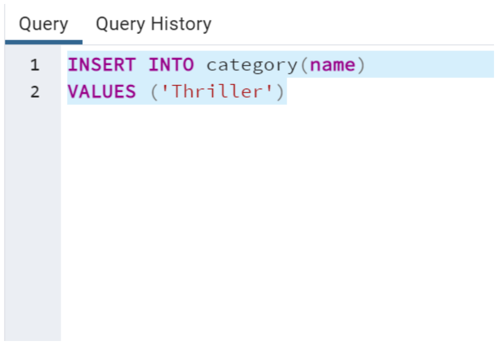
```sql
SELECT name, category_id
FROM category
```

Output

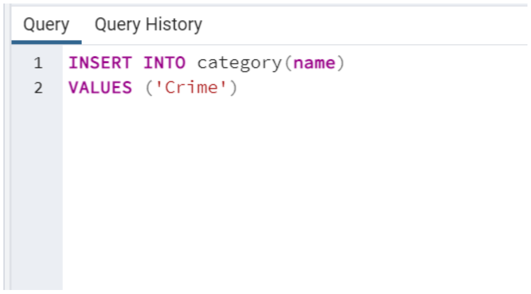
name	category_id
Action	1
Animation	2
Children	3
Classics	4
Comedy	5
Documentary	6
Drama	7
Family	8
Foreign	9

Games	10
Horror	11
Music	12
New	13
Sci-Fi	14
Sports	15
Travel	16

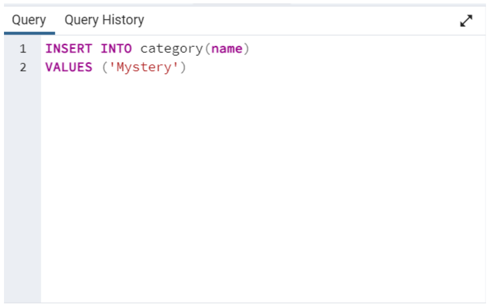
Step 2: Evaluating INSERT and CREATE TABLE Statements



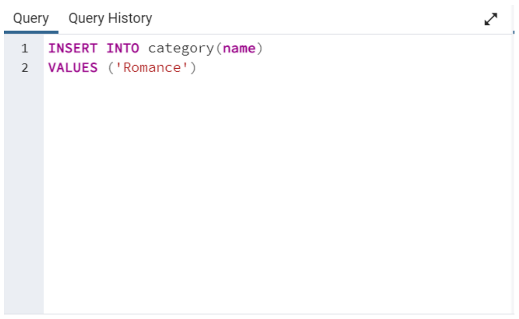
```
INSERT INTO category(name)
VALUES ('Thriller')
```



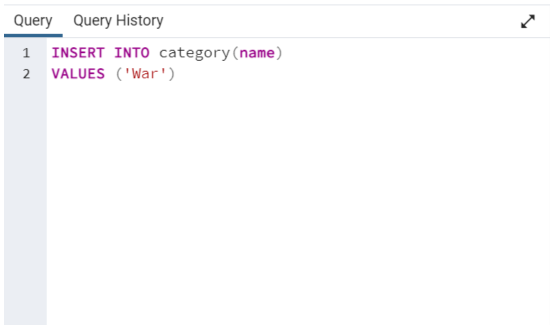
```
INSERT INTO category(name)
VALUES ('Crime')
```



```
INSERT INTO category(name)
VALUES ('Mystery')
```



```
INSERT INTO category(name)
VALUES ('Romance')
```



```
INSERT INTO category(name)
VALUES ('War')
```

```
 CREATE TABLE category
(
  category_id integer NOT NULL DEFAULT
nextval('category_category_id_seq'::regclass),
  name text COLLATE pg_catalog."default" NOT NULL,
  last_update timestamp with time zone NOT NULL DEFAULT now(),
  CONSTRAINT category_pkey PRIMARY KEY (category_id)
);
```

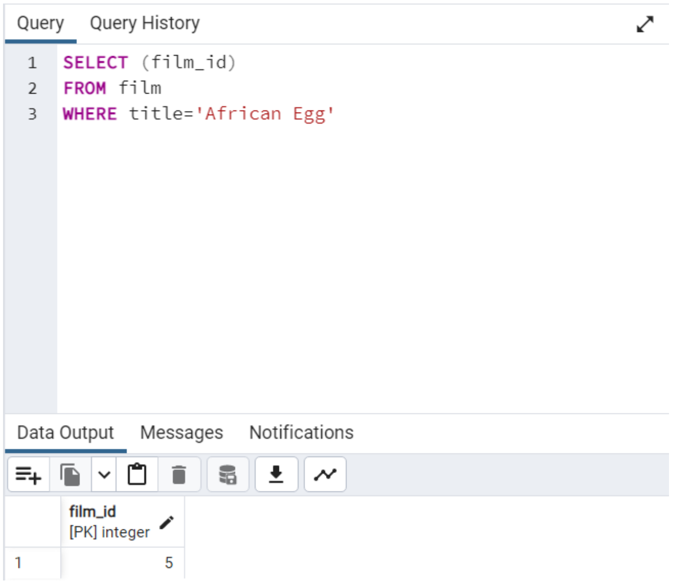
The SQL statement above was written to provide constraints on the columns within the Category table.

- The "NOT NULL DEFAULT" constraint ensures that there are no null values entered under the category_id column of the Category table if null values are copied over from the original table.
- The "NOT NULL DEFAULT" in the 6th line ensures that the last_update column of the Category table does not allow for the transfer of null values that may exist in the original last_update column.
- The "PRIMARY KEY" constraint ensures that the category_id column serves as the primary key, turning all the values in the category_id column into a primary key that acts as a unique identifier for an entire row in the table.

In essence, the above statement ensures that the category_id column can, indeed, serve as the primary key by preventing null values from the category_id.

Step 3: Inserting New Movie Title

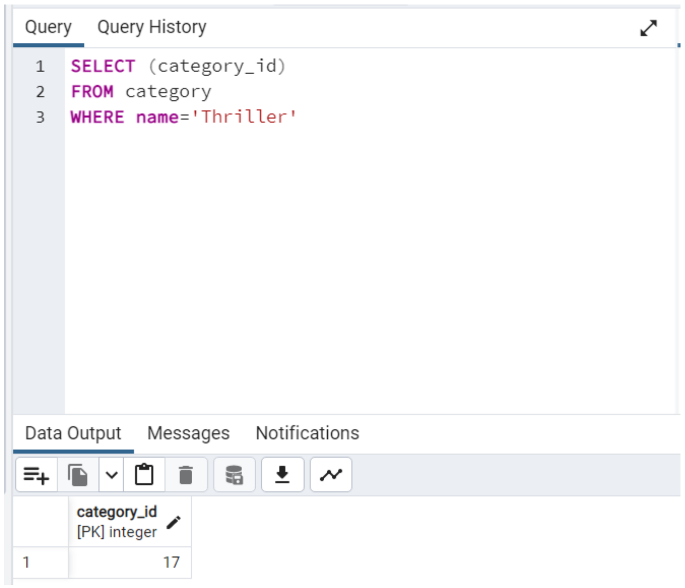
Select Statement to Find Film ID for Title



```
SELECT (film_id)
FROM film
WHERE title='African Egg'
```

	film_id [PK] integer
1	5

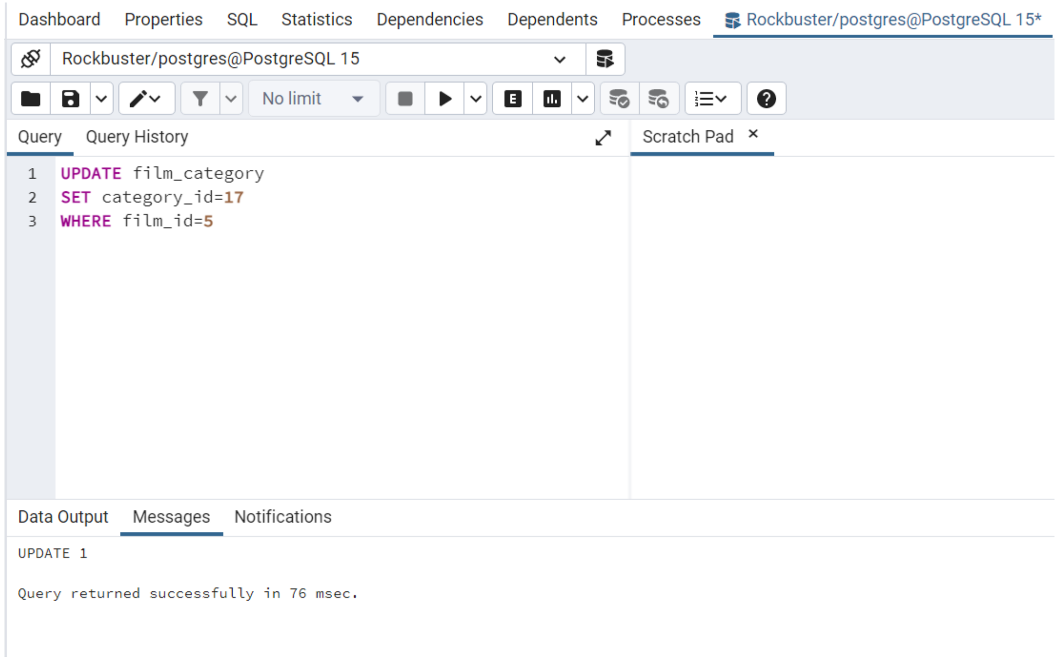
Select Statement to Find Category ID for the “Thriller” genre



```sql
SELECT (category_id)
FROM category
WHERE name='Thriller'
```

	category_id [PK] integer
1	17

SQL statement updating the category id in the film_category table to 17 (the code for Thriller movies) for African Egg.



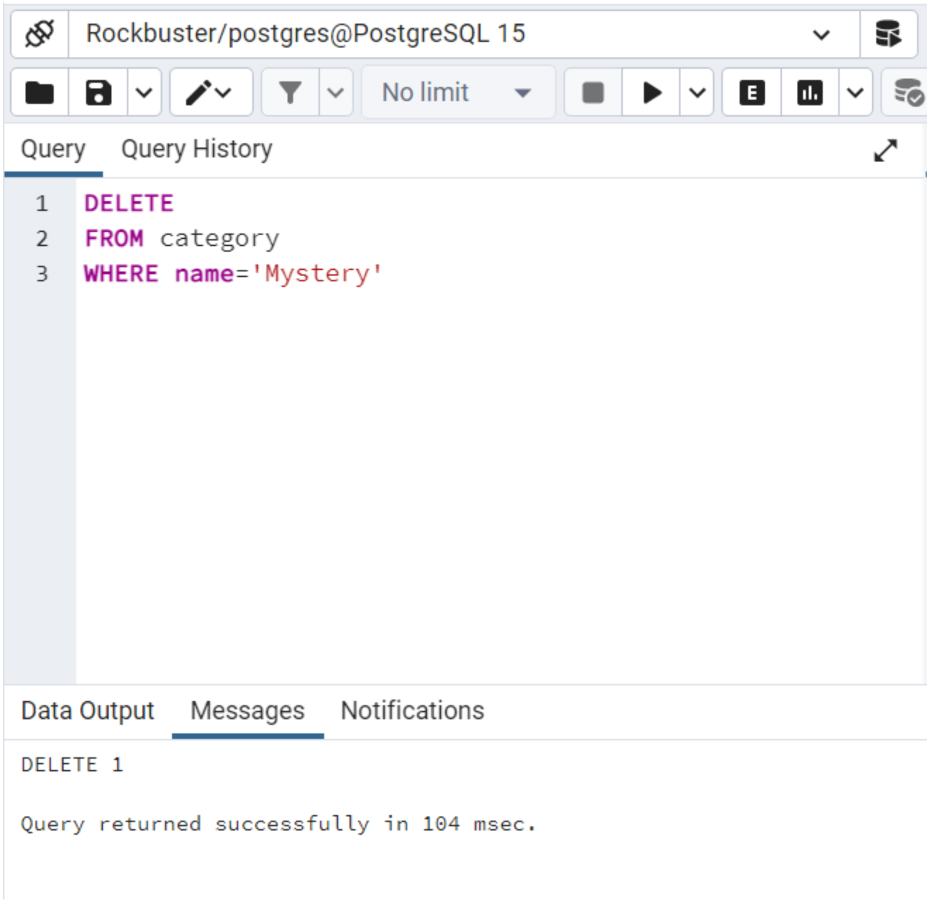
```sql
UPDATE film_category
SET category_id=17
WHERE film_id=5
```

```
UPDATE 1

Query returned successfully in 76 msec.
```

Step 4: Deleting A Category

SQL Statement to Delete all Columns in the Category Table where “Mystery” ia as an option from the name column



```sql
DELETE
FROM category
WHERE name='Mystery'
```

```
DELETE 1

Query returned successfully in 104 msec.
```

Step 5: Reflection on Excel vs SQL

If I had carried out the above tasks in Excel, I would have had to begun by creating a Pivot Table to uncover the genres present. I also would have had to have pulled all pivot table data from their respective source table(s) by going directly into said table(s). Much of the rest of the work would have been carried out in the Pivot Tables by making use of functions like filters, the find and replace tool, etc. One of the major advantages of using SQL in this particular use case is the ease of navigating between different tables without having to physically navigate to and fro the actual tables. It’s easier to visualize. The main con is that one has to have a fairly strong command of the SQL language to take advantage of the confusion- and time-savings.