Lab 3

User-defined functions:

1. Create a function that takes category name and returns the number of films in this category.

Query Query History

```
CREATE OR REPLACE FUNCTION get_film_count_by_category(cat_name TEXT)
RETURNS INTEGER AS $$
DECLARE
  count_result INTEGER;
BEGIN
  SELECT COUNT(*) INTO count_result
  FROM film
  JOIN film_category USING (film_id)
  JOIN category USING (category_id)
  WHERE category.name = cat_name;

  RETURN count_result;
END;
$$ LANGUAGE plpgsql
```

Data Output Messages Notifications

```
CREATE FUNCTION

Query returned successfully in 45 msec.
```

2. Create a function that takes customer_id and returns full name (first_name last_name)

Query Query History

```
CREATE OR REPLACE FUNCTION get_customer_full_name(c_id INT)
RETURNS TEXT AS $$
DECLARE
  full_name TEXT;
BEGIN
  SELECT first_name || ' ' || last_name INTO full_name
  FROM customer
  WHERE customer_id = c_id;

  RETURN full_name;
END;
$$ LANGUAGE plpgsql
```

Data Output Messages Notifications

```
CREATE FUNCTION

Query returned successfully in 34 msec.
```

3. Create a function that's takes actor_id and returns his/her film count.

Query Query History

```
CREATE OR REPLACE FUNCTION get_film_count_by_actor(a_id INT)
RETURNS INTEGER AS $$
DECLARE
  film_count INTEGER;
BEGIN
  SELECT COUNT(*) INTO film_count
  FROM film_actor
  WHERE actor_id = a_id;

  RETURN film_count;
END;
$$ LANGUAGE plpgsql
```

Data Output Messages Notifications

```
CREATE FUNCTION

Query returned successfully in 34 msec.
```

4. Create a function that returns the most recent rental date for a given customer.

Query Query History

```
CREATE OR REPLACE FUNCTION get_latest_rental_date(c_id INT)
RETURNS DATE AS $$
DECLARE
  latest_date DATE;
BEGIN
  SELECT MAX(rental_date)::DATE INTO latest_date
  FROM rental
  WHERE customer_id = c_id;

  RETURN latest_date;
END;
$$ LANGUAGE plpgsql
```

Data Output Messages Notifications

```
CREATE FUNCTION

Query returned successfully in 36 msec.
```

5. Function overload question

a. Create two versions of a function get_film_info:

i. One takes film_id and returns title

ii. One takes title and returns film_id

Query Query History

```
CREATE OR REPLACE FUNCTION get_film_info(f_id INT)
RETURNS TEXT AS $$
DECLARE
  film_title TEXT;
BEGIN
  SELECT title INTO film_title
  FROM film
  WHERE film_id = f_id;

  RETURN film_title;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get_film_info(f_title TEXT)
RETURNS INT AS $$
DECLARE
  film_id_result INT;
BEGIN
  SELECT film_id INTO film_id_result
  FROM film
  WHERE title = f_title;

  RETURN film_id_result;
END;
$$ LANGUAGE plpgsql;
```

Data Output Messages Notifications

```
CREATE FUNCTION

Query returned successfully in 32 msec.
```

Trigger

6. Create a trigger to prevent setting a negative value for rental_rate column in the film table.

Query Query History

```
CREATE OR REPLACE FUNCTION prevent_negative_rental_rate()
RETURNS TRIGGER AS $$
BEGIN
  IF NEW.rental_rate < 0 THEN
    RAISE EXCEPTION 'Rental rate cannot be negative';
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER check_rental_rate
BEFORE INSERT OR UPDATE ON film
FOR EACH ROW
EXECUTE FUNCTION prevent_negative_rental_rate()
```

Data Output Messages Notifications

```
CREATE TRIGGER

Query returned successfully in 36 msec.
```

7. Create a trigger to save a backup for newly inserted customers. (create customer_backup table first).

Creating customer_backup table first

Query Query History

```
CREATE TABLE customer_backup AS TABLE customer WITH NO DATA;
```

Data Output Messages Notifications

```
CREATE TABLE AS
```

Then trigger function

Query Query History

```sql
CREATE OR REPLACE FUNCTION backup_new_customer()
RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO customer_backup VALUES (NEW.*);
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER backup_customer_insert
AFTER INSERT ON customer
FOR EACH ROW
EXECUTE FUNCTION backup_new_customer();
```

Data Output Messages Notifications

```
CREATE TRIGGER

Query returned successfully in 32 msec.
```

8. Create a trigger on payment table that copies deleted rows into a new table deleted_payments

Query Query History

```sql
CREATE TABLE deleted_payments AS TABLE payment WITH NO DATA
```

Data Output Messages Notifications

```
ERROR:  relation "deleted_payments" already exists

SQL state: 42P07
```

Query Query History

```sql
CREATE OR REPLACE FUNCTION archive_deleted_payment()
RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO deleted_payments VALUES (OLD.*);
  RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER log_deleted_payment
BEFORE DELETE ON payment
FOR EACH ROW
EXECUTE FUNCTION archive_deleted_payment()
```

Data Output Messages Notifications

```
CREATE TRIGGER

Query returned successfully in 41 msec.
```

9. Track when a film's title is updated, and log the old and new titles along with the time of the change in a new table (film_audit).

Query Query History

```
CREATE TABLE film_audit (
  film_id INT,
  old_title TEXT,
  new_title TEXT,
  changed_at TIMESTAMP
)
```

Data Output Messages Notifications

```
CREATE TABLE

Query returned successfully in 34 msec.
```

Query Query History

```
CREATE OR REPLACE FUNCTION track_film_title_change()
RETURNS TRIGGER AS $$
BEGIN
  IF NEW.title <> OLD.title THEN
    INSERT INTO film_audit (film_id, old_title, new_title, changed_at)
    VALUES (OLD.film_id, OLD.title, NEW.title, CURRENT_TIMESTAMP);
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER log_film_title_update
AFTER UPDATE ON film
FOR EACH ROW
WHEN (OLD.title IS DISTINCT FROM NEW.title)
EXECUTE FUNCTION track_film_title_change()
```

Data Output Messages Notifications

```
CREATE TRIGGER

Query returned successfully in 33 msec.
```