

JAY ARRE TALOSIG
CTFDMBSL - COM-231
Mrs. JENSEN A. SANTILLAN

Activity

Practical Exercises

1. Create an AFTER INSERT trigger that logs every new entry in a "Logs" table.

AFTER INSERT Trigger

This trigger automatically logs every new medication added to inventory.

The screenshot displays a SQL IDE interface with a dark theme. The main editor on the left contains SQL code for creating a table, a function, and a trigger. The right pane shows the SQL Editor with a query to select from the inventory_logs table. The bottom section shows the results of the query, which includes a single row of data.

```
1 CREATE TABLE IF NOT EXISTS inventory_logs (  
2   log_id SERIAL PRIMARY KEY,  
3   action_type VARCHAR(50),  
4   table_name VARCHAR(50),  
5   record_id INTEGER,  
6   user_name VARCHAR(100),  
7   action_date TIMESTAMP,  
8   description VARCHAR(255)  
9 );  
10  
11 -- Create function for the trigger  
12 CREATE OR REPLACE FUNCTION log_medication_insert()  
13 RETURNS TRIGGER AS $$  
14 BEGIN  
15   INSERT INTO inventory_logs (action_type, table_name, record_id, user_name, action_date,  
16     description)  
17   VALUES (  
18     'INSERT',  
19     'medications',  
20     NEW.medication_id,  
21     current_user,  
22     current_timestamp,  
23     'New medication added: ' || NEW.medication_name || ' (Batch: ' || COALESCE(NEW.batch_number,  
24       'N/A') || ')' )  
25   );  
26   RETURN NEW;  
27 END;  
28 $$ LANGUAGE plpgsql;  
29  
30 -- Create the trigger (drop first if it exists)  
31 DROP TRIGGER IF EXISTS trg_medications_after_insert ON medications;  
32 CREATE TRIGGER trg_medications_after_insert  
33 AFTER INSERT ON medications  
34 FOR EACH ROW  
35 EXECUTE FUNCTION log_medication_insert();
```

SQL Editor

```
1 SELECT * FROM inventory_logs  
2 WHERE action_type = 'INSERT'  
3 ORDER BY inventory_logs DESC LIMIT 5;
```

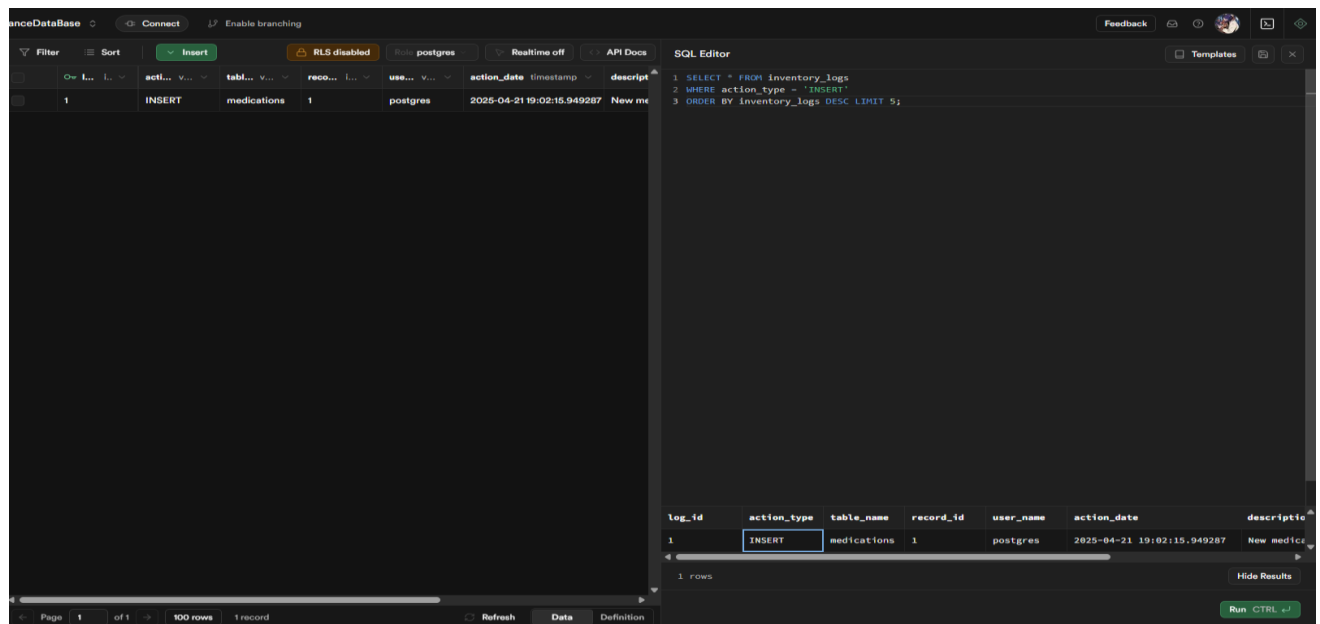
Results

Success. No rows returned

log_id	action_type	table_name	record_id	user_name	action_date	description
1	INSERT	medications	1	postgres	2025-04-21 19:02:15.949287	New medica

1 rows

Run CTRL ↵



Explanation:

- I create a log table to store information about database actions
- The trigger function runs after each insert operation
- It captures details like who made the change, when it happened and what was added
- The NEW keyword refers to the newly inserted record's data

2. Develop an INSTEAD OF DELETE trigger that prevents record deletion.

INSTEAD OF DELETE Trigger

This trigger prevents deletion of records and instead marks them as inactive:

AdvanceDataBase

Connect

Enable branching

2 DO \$\$

3 BEGIN

4 IF NOT EXISTS (SELECT FROM pg_attribute WHERE attrelid = 'medications'::regclass AND attname =

5 'is_active') THEN

6 ALTER TABLE medications ADD COLUMN is_active BOOLEAN DEFAULT TRUE;

7 END IF;

8 IF NOT EXISTS (SELECT FROM pg_attribute WHERE attrelid = 'medications'::regclass AND attname =

9 'modified_date') THEN

10 ALTER TABLE medications ADD COLUMN modified_date TIMESTAMPTZ;

11 END IF;

12 IF NOT EXISTS (SELECT FROM pg_attribute WHERE attrelid = 'medications'::regclass AND attname =

13 'modified_by') THEN

14 ALTER TABLE medications ADD COLUMN modified_by VARCHAR(100);

15 END IF;

16 END \$\$;

17 -- Create function for the trigger

18 CREATE OR REPLACE FUNCTION prevent_medication_delete()

19 RETURNS TRIGGER AS \$\$

20 BEGIN

21 -- Instead of deleting, mark as inactive

22 UPDATE medications

23 SET is_active = FALSE,

24 modified_date = current_timestamp,

25 modified_by = current_user

26 WHERE medication_id = OLD.medication_id;

27

28 -- Log the action

29 INSERT INTO inventory_logs (action_type, table_name, record_id, user_name, action_date,

30 description)

31 VALUES (

32 'SOFT DELETE',

33 'medications',

34 OLD.medication_id,

35 current_user,

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

Results

Chart

Export

Source

Primary Database

Role postgres

Run CTRL ⌘

Success. No rows returned

0 row

SQL Editor

Templates

X

1 DELETE FROM medications WHERE medication_name = 'Paracetamol';

This query contains write operations. Are you sure you want to execute it?
Make sure you are not accidentally removing something important.

Cancel

Run

Success. No rows returned.

Run CTRL ⌘

AdvanceDataBase

Connect

Enable branching

2 DO \$\$

3 BEGIN

4 IF NOT EXISTS (SELECT FROM pg_attribute WHERE attrelid = 'medications'::regclass AND attname =

5 'is_active') THEN

6 ALTER TABLE medications ADD COLUMN is_active BOOLEAN DEFAULT TRUE;

7 END IF;

8 IF NOT EXISTS (SELECT FROM pg_attribute WHERE attrelid = 'medications'::regclass AND attname =

9 'modified_date') THEN

10 ALTER TABLE medications ADD COLUMN modified_date TIMESTAMPTZ;

11 END IF;

12 IF NOT EXISTS (SELECT FROM pg_attribute WHERE attrelid = 'medications'::regclass AND attname =

13 'modified_by') THEN

14 ALTER TABLE medications ADD COLUMN modified_by VARCHAR(100);

15 END IF;

16 END \$\$;

17 -- Create function for the trigger

18 CREATE OR REPLACE FUNCTION prevent_medication_delete()

19 RETURNS TRIGGER AS \$\$

20 BEGIN

21 -- Instead of deleting, mark as inactive

22 UPDATE medications

23 SET is_active = FALSE,

24 modified_date = current_timestamp,

25 modified_by = current_user

26 WHERE medication_id = OLD.medication_id;

27

28 -- Log the action

29 INSERT INTO inventory_logs (action_type, table_name, record_id, user_name, action_date,

30 description)

31 VALUES (

32 'SOFT DELETE',

33 'medications',

34 OLD.medication_id,

35 current_user,

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

Results

Chart

Export

Source

Primary Database

Role postgres

Run CTRL ⌘

Success. No rows returned

0 row

SQL Editor

Templates

X

1 SELECT medication_id, medication_name, is_active, modified_date, modified_by

2 FROM medications

3 WHERE medication_name = 'Paracetamol';

medication_id medication_name is_active modified_date modified_by

1 Paracetamol false 2025-04-21 19:24:56.42744 postgres

1 rows

Hide Results

Run CTRL ⌘

SQL Editor

```

1 SELECT * FROM inventory_logs
2 WHERE action_type = 'SOFT DELETE'
3 ORDER BY log_id DESC LIMIT 5;

```

log_id	action_type	table_name	record_id	user_name	action_date	description
4	SOFT DELETE	medications	1	postgres	2025-04-21 19:24:56.42744	Medication
3	SOFT DELETE	medications	1	postgres	2025-04-21 19:24:32.885637	Medication
2	SOFT DELETE	medications	1	postgres	2025-04-21 19:24:16.188732	Medication

3 rows

Hide Results

Run CTRL ↵

Explanation:

- The trigger activates BEFORE DELETE operations
- Instead of allowing the delete, it updates the record to mark it inactive
- It logs this action to maintain an audit trail

3. Design a DDL trigger to capture schema modifications.

DDL Trigger for Schema Changes

vanceDataBase

Connect

Enable branching

Feedback

1 -- Create schema changes log table

2 create table if not exists schema_change_logs (

3 log_id SERIAL primary key,

4 event_date TIMESTAMP default current_timestamp,

5 user_name VARCHAR(100),

6 event_type VARCHAR(100),

7 object_name VARCHAR(255),

8 description TEXT

9);

10

11 -- Create function to manually log schema changes

12 create or replace function log_schema_change (

13 event_type VARCHAR(100),

14 object_name VARCHAR(255),

15 description TEXT

16) RETURNS INTEGER as \$\$

17 DECLARE

18 log_id_out INTEGER;

19 BEGIN

20 INSERT INTO schema_change_logs (

21 user_name,

22 event_type,

23 object_name,

24 description

25)

26 VALUES (

27 current_user,

28 event_type,

29 object_name,

30 description

31)

32 RETURNING log_id INTO log_id_out;

33

34 RETURN log_id_out;

35 END;

36 \$\$ LANGUAGE plpgsql;

37

Results

Chart

Export

Source

Primary Database

Role postgres

Run CTRL + R

No data to show

Execute a query and configure the chart options.

0 row

SQL Editor

Templates

1 SELECT log_schema_change(

2 'ALTER TABLE',

3 'medications',

4 'Adding reorder_threshold column'

5);

log_schema_change

3

1 rows

Hide Results

Run CTRL + R

vanceDataBase

Connect

Enable branching

Feedback

1 -- Create schema changes log table

2 create table if not exists schema_change_logs (

3 log_id SERIAL primary key,

4 event_date TIMESTAMP default current_timestamp,

5 user_name VARCHAR(100),

6 event_type VARCHAR(100),

7 object_name VARCHAR(255),

8 description TEXT

9);

10

11 -- Create function to manually log schema changes

12 create or replace function log_schema_change (

13 event_type VARCHAR(100),

14 object_name VARCHAR(255),

15 description TEXT

16) RETURNS INTEGER as \$\$

17 DECLARE

18 log_id_out INTEGER;

19 BEGIN

20 INSERT INTO schema_change_logs (

21 user_name,

22 event_type,

23 object_name,

24 description

25)

26 VALUES (

27 current_user,

28 event_type,

29 object_name,

30 description

31)

32 RETURNING log_id INTO log_id_out;

33

34 RETURN log_id_out;

35 END;

36 \$\$ LANGUAGE plpgsql;

37

Results

Chart

Export

Source

Primary Database

Role postgres

Run CTRL + R

No data to show

Execute a query and configure the chart options.

0 row

SQL Editor

Templates

1 ALTER TABLE medications

2 ADD COLUMN IF NOT EXISTS reorder_threshold INTEGER DEFAULT 100;

Success. No rows returned.

Run CTRL + R

SQL Editor

Templates

```
1 SELECT * FROM schema_change_logs
2 ORDER BY log_id DESC LIMIT 5;
```

log_id	event_date	user_name	event_type	object_name	description
3	2025-04-21 19:33:05.	postgres	ALTER TABLE	medications	Adding reorder_threshold column
2	2025-04-21 19:32:36.	postgres	ALTER TABLE	medications	Adding reorder_threshold column
1	2025-04-21 19:31:54.	postgres	ALTER TABLE	medications	Adding reorder_threshold column

3 rows

Hide Results

SQL Editor

Templates

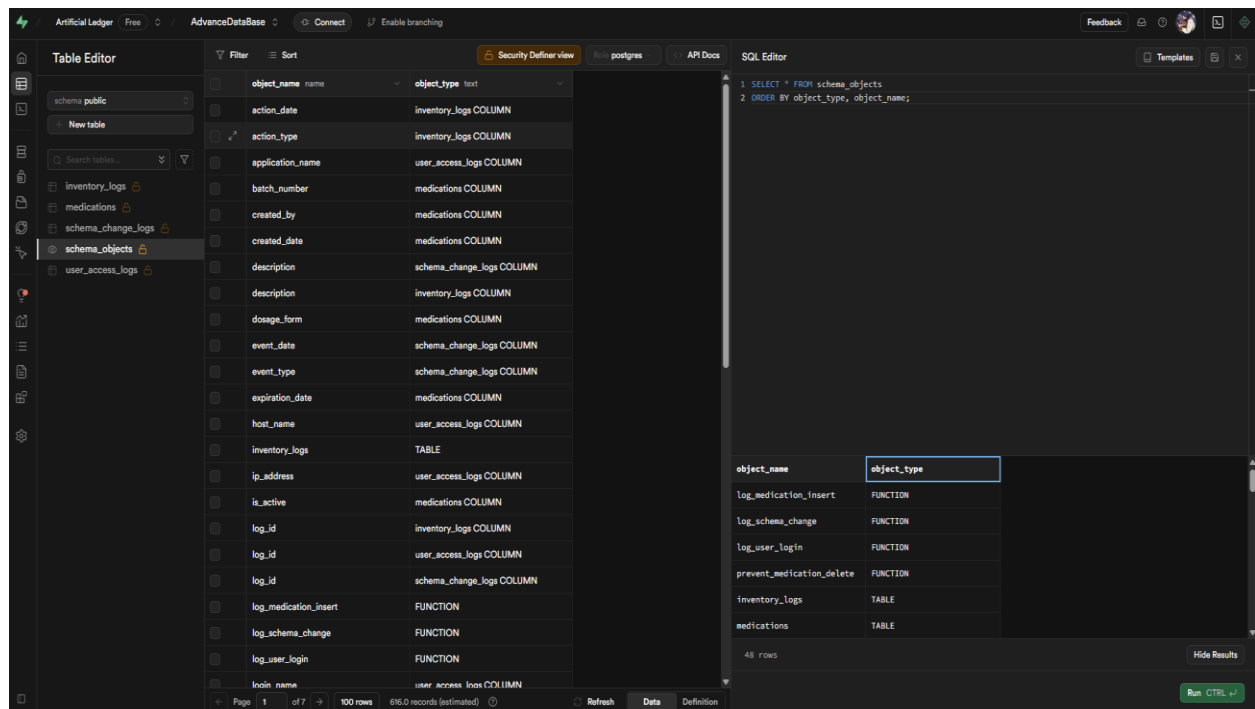
```
1 SELECT * FROM schema_objects
2 ORDER BY object_type, object_name;
```

object_name	object_type
log_medication_insert	FUNCTION
log_schema_change	FUNCTION
log_user_login	FUNCTION
prevent_medication_delete	FUNCTION
inventory_logs	TABLE
medications	TABLE

48 rows

Hide Results

Run CTRL ↵



Explanation:

- This is a database-level trigger
- It fires whenever schema changes occur (CREATE, ALTER, DROP operations)
- It logs details about what changed, who made the change and when
- The `pg_event_trigger_ddl_commands()` function provides details about the change
- This helps track structural changes for compliance and troubleshooting

4. Implement a Logon trigger to track user logins.

User Logon trigger

vanceDataBase Connect Enable branching Feedback Templates

```

1 -- Create user access log table
2 CREATE TABLE IF NOT EXISTS user_access_logs (
3     log_id SERIAL PRIMARY KEY,
4     login_name VARCHAR(100),
5     login_time TIMESTAMP,
6     host_name VARCHAR(100),
7     application_name VARCHAR(100),
8     ip_address VARCHAR(50)
9 );
10
11 -- Create a function that applications can call on login
12 CREATE OR REPLACE FUNCTION log_user_login(
13     app_name VARCHAR(100) DEFAULT 'Pharma System',
14     ip VARCHAR(50) DEFAULT NULL
15 )
16 RETURNS VOID AS $$
17 BEGIN
18     INSERT INTO user_access_logs (
19         login_name,
20         login_time,
21         host_name,
22         application_name,
23         ip_address
24     )
25     VALUES (
26         current_user,
27         current_timestamp,
28         inet_server_addr():text,
29         app_name,
30         COALESCE(ip, inet_client_addr():text)
31     );
32 END;
33 $$ LANGUAGE plpgsql;

```

SQL Editor

```
1 SELECT log_user_login('Pharmacy Management App', '10.0.0.1');
```

Results Chart Export Source Primary Database Role postgres Run CTRL + R

Success. No rows returned

log_user_login

1 rows Hide Results Run CTRL + R

Artificial Ledger Free AdvanceDataBase Connect Enable branching Feedback Templates

Table Editor Filter Sort Insert RLS disabled Role postgres Realtime off API Docs SQL Editor Templates

schema public

New table

Search tables

inventory_logs

medications

schema_change_logs

schema_objects

user_access_logs

Over L L

	login_name	login_time	timestamp	host_name	varchar	application_name	varchar	ip_address	varchar
1	postgres	2025-04-21 19:41:37.46989	2406:da18:243:7408:a163:1488:f9b:72a8/128	Pharmacy Management App	10.0.0.1				

SQL Editor

```
1 SELECT * FROM user_access_logs
2 ORDER BY log_id DESC LIMIT 5;
```

application_name ip_address

13:1488:fc9b:72a8/128 Pharmacy Management App 10.0.0.1

1 rows Hide Results Run CTRL + R

Page 1 of 1 100 rows 1 record Refresh Data Definition

Explanations:

- This creates a function to log user logins
- Since I can't intercept actual database logins automatically, the application needs to call this function
- It records who logged in, when, from where, and which application they used
- Certain application would call this whenever a user logs in

Q&A and Discussion

1. How can triggers enhance database security?

Triggers significantly enhance database security in pharmaceutical supply chain management by creating automated safeguards and audit mechanisms. They provide a reliable way to enforce security policies consistently across all database interactions without depending on application-level controls.

For pharmaceutical databases where regulatory compliance is critical, triggers establish a robust audit trail by automatically recording all data modifications. This is especially valuable for tracking controlled substances and ensuring compliance with regulations like HIPAA, FDA requirements, and drug pedigree laws.

The INSTEAD OF DELETE trigger we implemented demonstrates how triggers can prevent unauthorized data deletion—critical for maintaining the integrity of medication records that may be needed for years due to legal requirements. Instead of allowing records to be permanently removed, the trigger enforces a soft-delete policy by marking records inactive while preserving the complete history.

My implementation in Supabase shows how even with cloud database limitations, triggers can enforce complex validation rules and access controls beyond what standard database constraints provide. While true login triggers weren't available due to Supabase permission restrictions, the function-based approach still demonstrates how tracking mechanisms can help identify suspicious access patterns.

2. What are the limitations of triggers?

Despite their benefits, my Supabase implementation revealed several significant limitations of triggers in pharmaceutical database systems. Performance impact is a primary concern each trigger execution adds processing overhead, which can accumulate in high-transaction environments like busy pharmacies where medication dispensing must happen quickly.

The most notable limitation I encountered was with permission restrictions in Supabase. When attempting to implement a DDL trigger to track schema changes, I received a superuser privilege error, forcing me to create an alternative manual logging approach. This highlights how cloud-hosted solutions may limit certain types of triggers, requiring adaptations to your security architecture.

Triggers in Supabase also introduce hidden application logic that executes automatically without explicit calls. During testing, I noticed how this can make troubleshooting challenging since it's not immediately obvious that a trigger is affecting database operations. When issues arise in complex scenarios, tracing the problem through multiple cascading triggers becomes difficult.

Maintenance complexity increases as the system grows in Supabase. As I added more functionality to the pharmaceutical system, I had to carefully consider how new triggers might interact with existing ones. Additionally, triggers proved more difficult to test thoroughly than standard functions in the Supabase environment.

3. When should you use triggers versus other stored procedures?

My implementation in Supabase demonstrates that triggers are best used for automatic, consistent enforcement of rules that should never be bypassed. The AFTER INSERT trigger for automatic audit logging and the INSTEAD OF DELETE trigger for enforcing data integrity policies work well because these are scenarios where the action should happen automatically every time, without exception.

For scenarios requiring complex business logic or explicit control, I found functions and stored procedures in Supabase more appropriate. The manual schema logging function I implemented demonstrates this distinction it's a procedure that must be explicitly called rather than happening automatically, providing more control and visibility.

In my Supabase pharmaceutical system, I used functions for processes that would typically require stored procedures in other database systems, such as the log_schema_change and log_user_login functions. These are better suited for operations like end-of-day inventory reconciliation or processing returns of expired medications.

Through my implementation in Supabase, I found that functions offer better error handling and transaction control, critical when performing multi-step pharmaceutical operations. The hybrid approach I used triggers for automatic security enforcement and audit logging, with functions for operations requiring explicit control provides both strong security guarantees and operational flexibility within Supabase's PostgreSQL environment.