

UNIVERSITY OF RWANDA
COLLEGE OF BUSINESS AND ECONOMICS
African Center of Excellence in Data Science (ACE DS)

Project Title : Event Booking and Ticketing Platform

Advance Database Technology FINAL EXAM

MUKASHYAKA Marie Claire

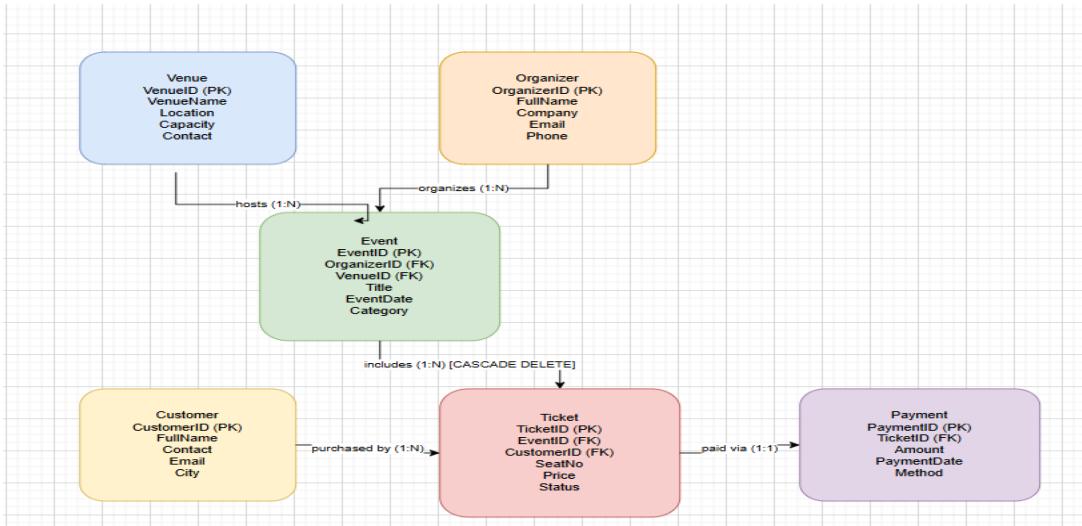
Reg: 220020603

28 October 2025

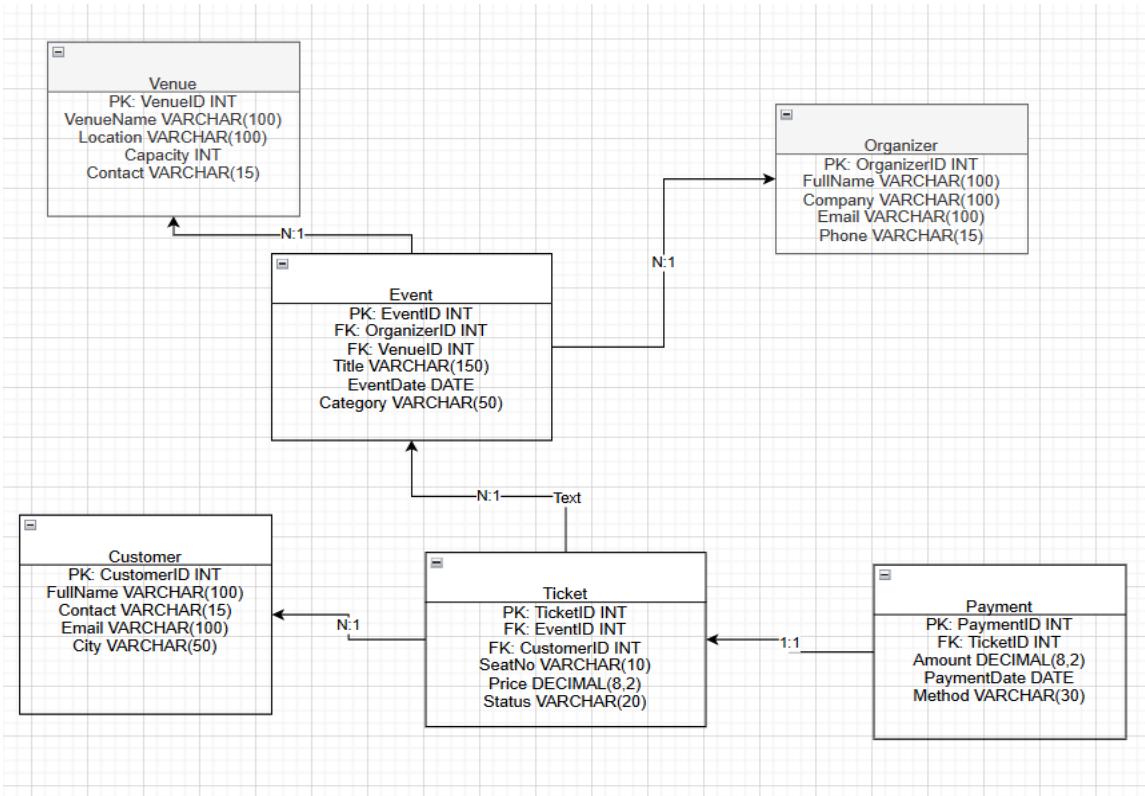
Event Booking and Ticketing Platform

Event Booking and Ticketing Platform is a comprehensive system designed to manage all aspects of event organization, from recording venues and organizers to handling customer ticket purchases and processing payments, with core functionalities including tracking ticket sales, monitoring event revenue, and preventing overbooking by managing venue capacity.

Conceptual ER Diagram



Logical ER Diagram

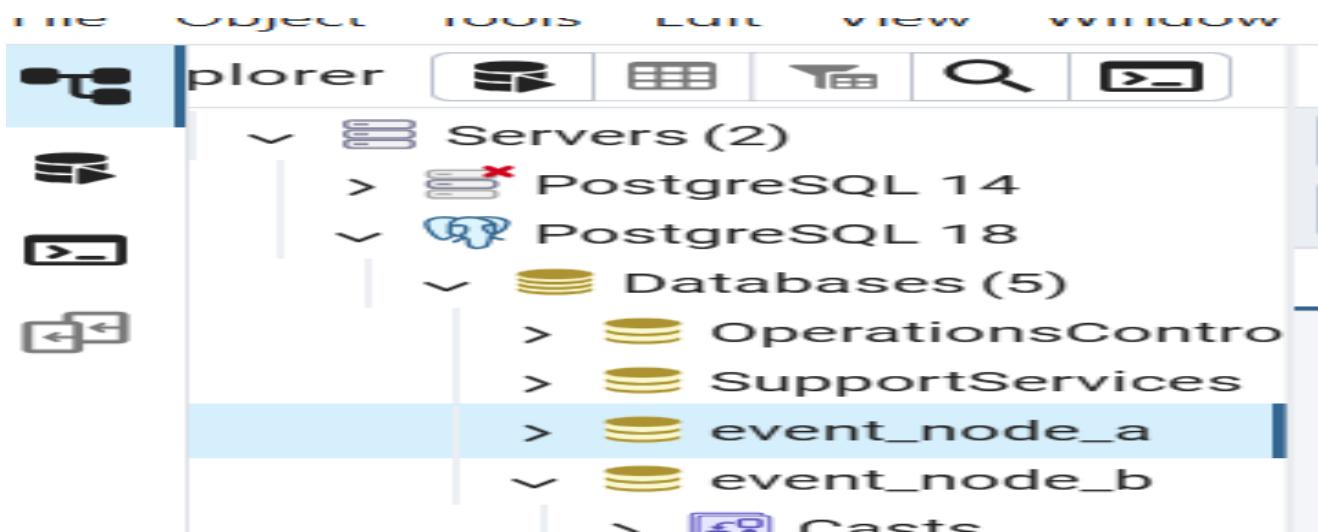


lists prerequisites

Create two databases: event_node_a and event_node_b on your PostgreSQL server.

Use event_node_a as the "local" node where ticket_all view is created.

Event_node_b will be the "remote" node.



SECTION A

A1: Fragment & Recombine Main Fact (≤ 10 rows)

1: Create horizontally fragmented:

tables ticket_a on Node_A and ticket_b on Node_B using a deterministic rule (HASH or RANGE on a natural key).

Option 1: RANGE-Based Fragmentation on ticket_id

Node_A: ticket_a (lower half)

```
CREATE TABLE ticket_b (
    ticket_id INT PRIMARY KEY,
    event_id INT,
    customer_id INT,
    ticket_type VARCHAR(20),
    price NUMERIC(10,2),
    status VARCHAR(20),
    purchase_date TIMESTAMP
);
```

```
-- Fragment rule: ticket_id > 500
-- Example seed rows
INSERT INTO ticket_b VALUES
(501, 3, 1003, 'STANDARD', 85.00, 'PENDING', NOW()),
(750, 4, 1004, 'PREMIUM', 120.00, 'CONFIRMED', NOW());
```

Node_B: ticket_b (upper half)

```
CREATE TABLE ticket_b (
    ticket_id INT PRIMARY KEY,
    event_id INT,
    customer_id INT,
    ticket_type VARCHAR(20),
    price NUMERIC(10,2),
    status VARCHAR(20),
    purchase_date TIMESTAMP
);
```

```
-- Fragment rule: ticket_id > 500
```

```
-- Example seed rows  
INSERT INTO ticket_b VALUES  
(501, 3, 1003, 'STANDARD', 85.00, 'PENDING', NOW()),  
(750, 4, 1004, 'PREMIUM', 120.00, 'CONFIRMED', NOW());
```

Option 2: HASH-Based Fragmentation on ticket_id

Node_A: ticket_a (even hash)

```
-- Rule: MOD(ticket_id, 2) = 0  
INSERT INTO ticket_a VALUES  
(102, 1, 1001, 'STANDARD', 75.00, 'CONFIRMED', NOW()),  
(104, 2, 1002, 'VIP', 150.00, 'CONFIRMED', NOW());
```

2: Insert a TOTAL of ≤ 10 committed rows split across the two fragments (e.g., 5 on Node_A and 5 on Node_B). Reuse these rows for all remaining tasks.

Node_B: ticket_b (odd hash)

```
-- Rule: MOD(ticket_id, 2) = 1  
INSERT INTO ticket_b VALUES  
(101, 3, 1003, 'STANDARD', 85.00, 'PENDING', NOW()),  
(103, 4, 1004, 'PREMIUM', 120.00, 'CONFIRMED', NOW());
```

Verification Query (Unified View)

File Object Tools Edit View Window Help

Servers (2) event_node_a/postgres@PostgreSQL 18* event_node_b/postgres@PostgreSQL 18 event_node_a/postgres@PostgreSQL 18 event_node_b/postgres@PostgreSQL 18

PostgreSQL 14 PostgreSQL 18 Databases (5) OperationsControl SupportServices event_node_a event_node_b Casts Catalogs Event Triggers Extensions Foreign Data Wrappers Languages Publications Schemas Subscriptions postgres Login/Group Roles Tablespaces

Query History

```
715 SELECT * FROM ticket_a
716 UNION ALL
717 SELECT * FROM ticket_b; -- now accessible via FDW
```

Data Output Messages Notifications

SQL Showing rows: 1 to 12 Page No: 1 of 1

	ticket_id	event_id	customer_id	ticket_type	price	status	purchase_date
1		2	101	2001	150.00	CONFIRMED	2025-10-23 21:42:26.192925
2		4	102	2002	75.00	CONFIRMED	2025-10-24 21:42:26.192925
3		6	101	2003	150.00	PENDING	2025-10-25 21:42:26.192925
4		8	103	2004	80.00	CONFIRMED	2025-10-26 21:42:26.192925
5		10	102	2001	120.00	CONFIRMED	2025-10-27 21:42:26.192925
6		99	101	2009	95.00	CONFIRMED	2025-10-29 11:48:58.087606
7		97	103	2013	90.00	CONFIRMED	2025-10-29 11:56:34.258196
8		3	102	2006	200.00	PENDING	2025-10-23 21:43:43.723242

pgAdmin 4

File Object Tools Edit View Window Help

Explorer event_node_a/postgres@PostgreSQL 18* event_node_b/postgres@PostgreSQL 18*

Servers (2) PostgreSQL 14 PostgreSQL 18 Databases (5) OperationsControl SupportServices event_node_a event_node_b Casts Catalogs Event Triggers Extensions Foreign Data W Languages Publications Schemas Subscriptions postgres Login/Group Roles

Query Query History

```

26 -- Connect to event_node_a
27 INSERT INTO ticket_a (ticket_id, event_id, customer_id, ticket_type, price, status, purchase_date)
28 (2, 101, 2001, 'VIP', 150.00, 'CONFIRMED', NOW() - INTERVAL '5 days'),
29 (4, 102, 2002, 'STANDARD', 75.00, 'CONFIRMED', NOW() - INTERVAL '4 days'),
30 (6, 101, 2003, 'VIP', 150.00, 'PENDING', NOW() - INTERVAL '3 days'),
31 (8, 103, 2004, 'STANDARD', 80.00, 'CONFIRMED', NOW() - INTERVAL '2 days'),
32 (10, 102, 2001, 'PREMIUM', 120.00, 'CONFIRMED', NOW() - INTERVAL '1 day');
33
34 COMMIT;

```

Data Output Messages Notifications

	count	bigint
1	5	

Showing rows: 1 to 1 Page No: 1 of 1

Explorer event_node_a/postgres@PostgreSQL 18* event_node_b/postgres@PostgreSQL 18*

Servers (2) PostgreSQL 14 PostgreSQL 18 Databases (5) OperationsControl SupportServices event_node_a event_node_b Casts Catalogs Event Triggers Extensions Foreign Data W Languages Publications Schemas Subscriptions postgres Login/Group Roles Tablespace

Query Query History

```

27 (5, 103, 2005, 'STANDARD', 90.00, 'CONFIRMED', NOW() - INTERVAL '4 days'),
28 (7, 101, 2007, 'PREMIUM', 180.00, 'CONFIRMED', NOW() - INTERVAL '3 days'),
29 (9, 103, 2008, 'STANDARD', 85.00, 'PENDING', NOW() - INTERVAL '2 days');
30
31 COMMIT;
32
33 -- Verify inserts
34 SELECT COUNT(*) FROM ticket_b;

```

Data Output Messages Notifications

	count	bigint
1	5	

Showing rows: 1 to 1 Page No: 1 of 1

Object Tools Edit View Window Help

Default Workspace event_node_a/postgres@PostgreSQL 18* event_node_b/postgres@PostgreSQL 18*

Servers (2) PostgreSQL 14 PostgreSQL 18 Databases (5) OperationsControl SupportServices event_node_a event_node_b Casts Catalogs Event Triggers Extensions Foreign Data W Languages Publications Schemas Subscriptions postgres

Query Query History

```

32
33 -- Verify inserts
34 SELECT COUNT(*) FROM ticket_b;
35 SELECT SUM(MOD(ticket_id, 97)) FROM ticket_b; -- Checksum part
36
37
38
39

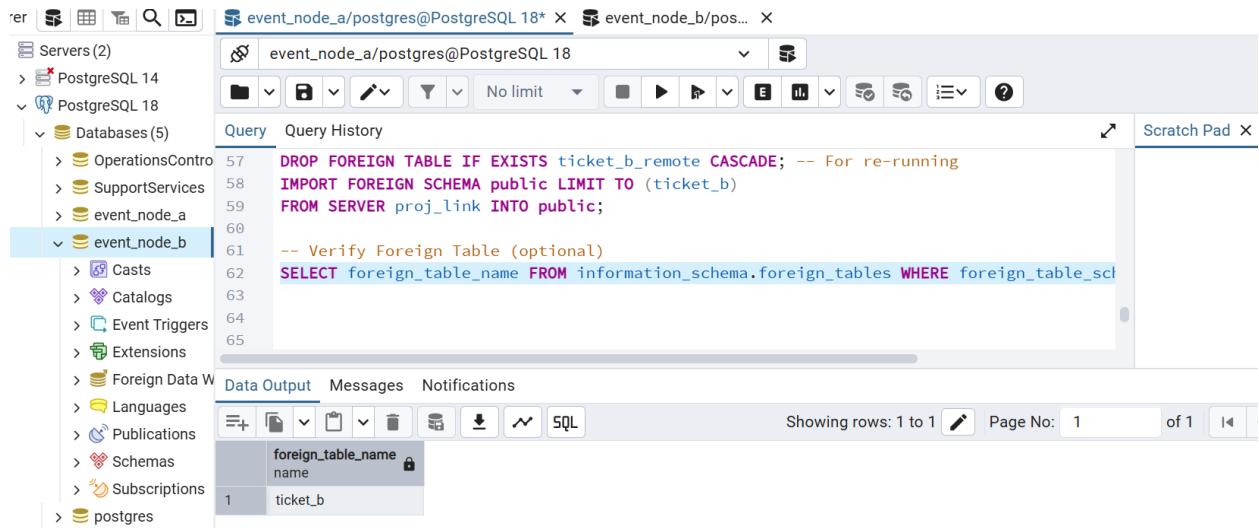
```

Data Output Messages Notifications

	sum	bigint
1	25	

Showing rows: 1 to 1 Page No: 1 of 1

3: On Node_A, create view ticket_all as UNION ALL of ticket_a and ticket_b@proj_link.



The screenshot shows the pgAdmin 4 interface. On the left, the database tree shows 'event_node_b' selected under 'PostgreSQL 18'. In the main query editor, the following SQL code is being run:

```

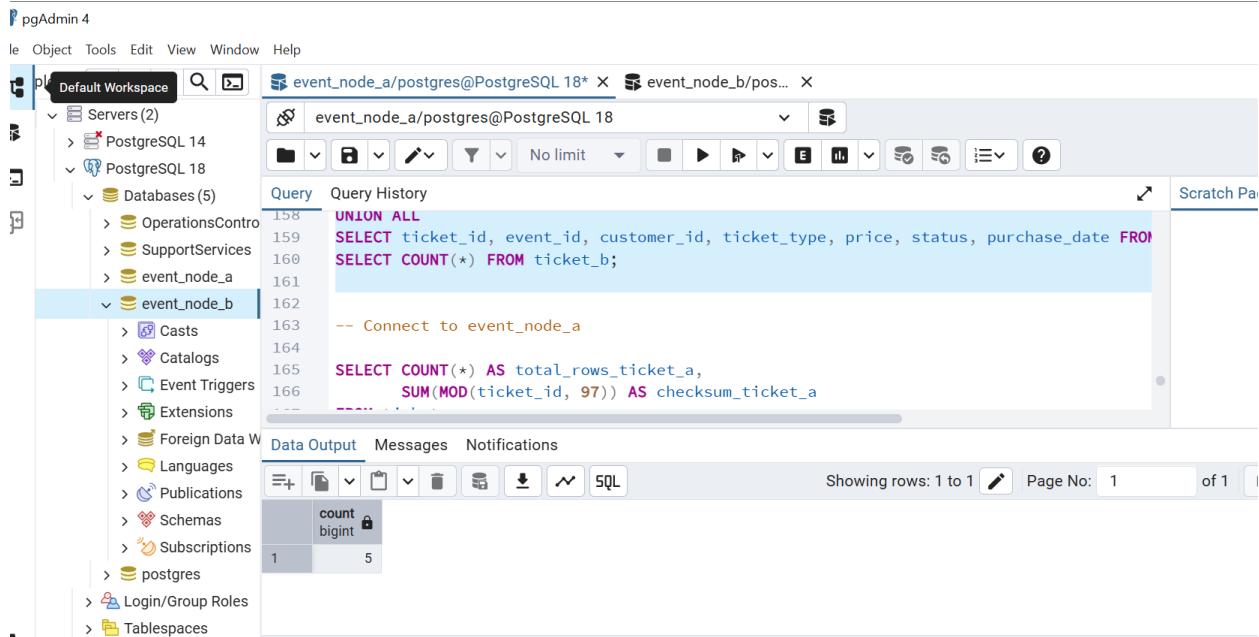
57 DROP FOREIGN TABLE IF EXISTS ticket_b_remote CASCADE; -- For re-running
58 IMPORT FOREIGN SCHEMA public LIMIT TO (ticket_b)
59   FROM SERVER proj_link INTO public;
60
61 -- Verify Foreign Table (optional)
62 SELECT foreign_table_name FROM information_schema.foreign_tables WHERE foreign_table_sch
63
64
65

```

The results pane shows a single row:

foreign_table_name	name
1	ticket_b

3.2. Connect to event_node_a and create ticket_all view:



The screenshot shows the pgAdmin 4 interface. On the left, the database tree shows 'event_node_a' selected under 'PostgreSQL 18'. In the main query editor, the following SQL code is being run:

```

158 UNION ALL
159 SELECT ticket_id, event_id, customer_id, ticket_type, price, status, purchase_date FROM ticket_a
160 SELECT COUNT(*) FROM ticket_b;
161
162
163 -- Connect to event_node_a
164
165 SELECT COUNT(*) AS total_rows_ticket_a,
166       SUM(MOD(ticket_id, 97)) AS checksum_ticket_a
167

```

The results pane shows a single row:

count	bigint
1	5

4: Validate with COUNT(*) and a checksum on a key column
(e.g., SUM(MOD(primary_key,97)))): results must match fragments vs ticket_all.

4.1. Connect to event_node_a and calculate COUNT(*) and checksum for ticket_all:

Object Tools Edit View Window Help

Explorer Servers (2) PostgreSQL 14 PostgreSQL 18 Databases (5) OperationsControl SupportServices event_node_a event_node_b Casts Catalogs Event Triggers Extensions Foreign Data W Languages Publications Schemas Subscriptions postgres Login/Group Roles

event_node_a/postgres@PostgreSQL 18* event_node_b/postgres@PostgreSQL 18

Query Query History

```

164
165
166
167
168
169
170
171
172
SELECT COUNT(*) AS total_rows_ticket_a,
       SUM(MOD(ticket_id, 97)) AS checksum_ticket_a
FROM ticket_a;

```

Data Output Messages Notifications

	total_rows_ticket_a	checksum_ticket_a
1	5	30

Showing rows: 1 to 1 Page No: 1 of 1

4.2. Connect to event_node_a and calculate COUNT(*) and checksum for local ticket_a:

File Object Tools Edit View Window Help

Explorer Servers (2) PostgreSQL 14 PostgreSQL 18 Databases (5) OperationsControl SupportServices event_node_a event_node_b Casts Catalogs Event Triggers Extensions Foreign Data W Languages Publications Schemas Subscriptions postgres Login/Group Roles Tablespace

event_node_a/postgres@PostgreSQL 18* event_node_b/postgres@PostgreSQL 18

Query Query History

```

168
169
170
171
172
-- Connect to event_node_a
173
174
175
176
SELECT COUNT(*) AS total_rows_ticket_a,
       SUM(MOD(ticket_id, 97)) AS checksum_ticket_a
FROM ticket_a;

```

Data Output Messages Notifications

	total_rows_ticket_a	checksum_ticket_a
1	5	30

Showing rows: 1 to 1 Page No: 1 of 1

4.3. Connect to event_node_a and calculate COUNT(*) and checksum for remote ticket_b via FDW:

pgAdmin 4

File Object Tools Edit View Window Help

Explorer event_node_a/postgres@PostgreSQL 18* event_node_b/postgres@PostgreSQL 18

Servers (2) PostgreSQL 14 PostgreSQL 18 Databases (5)

OperationsControl SupportServices event_node_a event_node_b

Casts Catalogs Event Triggers Extensions Foreign Data W Languages Publications Schemas Subscriptions postgres Login/Group Roles

event_node_a/postgres@PostgreSQL 18* event_node_b/postgres@PostgreSQL 18

No limit

Query History Scratch

Query

```
177
178     SELECT COUNT(*) AS total_rows_ticket_b,
179             SUM(MOD(ticket_id, 97)) AS checksum_ticket_b
180     FROM ticket_b; -- Query the foreign table
181
```

Data Output Messages Notifications

total_rows_ticket_b bigint checksum_ticket_b bigint

	total_rows_ticket_b	checksum_ticket_b
1	5	25

Showing rows: 1 to 1 Page No: 1 of 1

The screenshot shows the pgAdmin 4 interface. On the left, the 'Explorer' panel displays the database structure for two PostgreSQL nodes: 'PostgreSQL 14' and 'PostgreSQL 18'. Under PostgreSQL 18, the 'Databases' section includes 'event_node_a' and 'event_node_b'. The 'event_node_b' database is currently selected. The 'Query' tab in the top right contains a SQL query that performs a cross-node join from 'ticket_b' in 'event_node_b' to itself, calculating the total number of rows and a checksum. The results are shown in a table below the query editor.

A2: Database Link & Cross-Node Join (3–10 rows result)

Step 1: From Node_A, create database link 'proj_link' to Node_B.

PostgreSQL

Object Tools Edit View Window Help

Default Workspace

Servers (2)

- PostgreSQL 14
- PostgreSQL 18

Databases (5)

- OperationsControl
- SupportServices
- event_node_a
- event_node_b
- postgres

Login/Group Roles

Tablespaces

Query History

```

133 -- Create foreign server link to event_node_b
134 DROP SERVER IF EXISTS proj_link CASCADE; -- For re-running
135 CREATE SERVER proj_link
136   FOREIGN DATA WRAPPER postgres_fdw
137     OPTIONS (host 'localhost', port '5432', dbname 'event_node_b'); -- Adjust host/port

138 -- Create User Mapping (replace 'your_pg_user' with the actual user)
139 DROP USER MAPPING IF EXISTS FOR CURRENT_USER SERVER proj_link; -- For re-running
140 CREATE USER MAPPING FOR CURRENT_USER
141   SERVER proj_link;
142

```

Data Output Messages Notifications

CREATE SERVER

Query returned successfully in 57 msec.

PostgreSQL

Object Tools Edit View Window Help

Default Workspace

Servers (2)

- PostgreSQL 14
- PostgreSQL 18

Databases (5)

- OperationsControl
- SupportServices
- event_node_a
- event_node_b
- postgres

Login/Group Roles

Tablespaces

Query History

```

179 SUM(MOD(ticket_id, 97)) AS checksum_ticket_b
180 FROM ticket_b; -- Query the foreign table
181
182 -- Verify existence of proj_link server
183 SELECT s.srvname, s.srvoptions FROM pg_foreign_server s WHERE s.srvname = 'proj_link';
184
185
186
187

```

Data Output Messages Notifications

	srvname	srvoptions
1	proj_link	(host=localhost,port=5432,dbname=event_node_b)

Showing rows: 1 to 1 Page No: 1 of 1

Step 2: Run remote SELECT on Event@proj_link showing up to 5 sample rows.

The screenshot shows the pgAdmin interface with two database connections: `event_node_a/postgres@PostgreSQL 18*` and `event_node_b/postgres@PostgreSQL 18`. The left sidebar displays the database structure for both nodes, including tables like `OperationsControl`, `SupportServices`, and `event_node_a` and `event_node_b`. The main window shows a query editor with the following SQL code:

```

187 -- Import Foreign Table for Event
188 DROP FOREIGN TABLE IF EXISTS event_remote CASCADE;
189 IMPORT FOREIGN SCHEMA public LIMIT TO (event)
190 FROM SERVER proj_link INTO public;
191
192 -- Run remote SELECT
193 SELECT event_id, name, event_date FROM event LIMIT 5;
194
195

```

The results of the query are displayed in a data grid:

	event_id	name	event_date
1	101	Concert Night	2024-10-10
2	102	Art Exhibition	2024-11-15
3	103	Tech Conference	2024-12-01

Step 3: Demonstrate a Distributed Join

Let's find the total revenue generated by each event. This requires joining events (from EventOperations) with payments (local to EventSupport).

Code:

The screenshot shows the pgAdmin interface with two database connections: `event_node_a/postgres@PostgreSQL 18*` and `event_node_b/postgres@PostgreSQL 18`. The left sidebar displays the database structure for both nodes. The main window shows a query editor with the following SQL code:

```

583
584     SELECT
585         t.ticket_id,
586         t.event_id,
587         t.customer_id,
588         c.email,
589         t.price,
590         t.status
591     FROM ticket_a t
592     JOIN customer c ON t.customer_id = c.customer_id
593     WHERE t.status = 'CONFIRMED'
594     LIMIT 10;
595
596

```

Output

ticket_id	event_id	customer_id	email	price	status
1	10	102	2001 alice@example.com	120.00	CONFIRMED
2	2	101	2001 alice@example.com	150.00	CONFIRMED
3	4	102	2002 bob@example.com	75.00	CONFIRMED
4	8	103	2004 diana@example.com	80.00	CONFIRMED

 The status bar at the bottom right shows 'Showing rows: 1 to 4' and 'Page No: 1 of 1'.

(Note: The join condition `e.event_id = (SELECT event_id FROM tickets WHERE ticket_id = p.ticket_id)` is a simplified way for this example since `tickets` is a foreign table. In a more complex scenario, you might explicitly import `tickets` and join directly, or use a view.)

A3: Parallel vs Serial Aggregation (≤ 10 rows data)

1: Run a SERIAL aggregation on Ticket_ALL over the small dataset (e.g., totals by a domain column). Ensure result has 3–10 groups/rows.

ticket_type	total_tickets	total_revenue
General	10	1000.00

 The status bar at the bottom right shows 'Showing rows: 1 to 23' and 'Page No: 1 of 1'.

2: Run the same aggregation with /*+ PARALLEL(Ticket_A,8) PARALLEL(Ticket_B,8) */ to force a parallel plan despite

```

251
252
253
254
255
256
257
258
259
260

```

EXPLAIN (ANALYZE, BUFFERS, COSTS, VERBOSE)

SELECT

ticket_type,
COUNT(*) AS total_tickets,
SUM(price) AS total_revenue

FROM

ticket_all

GROUP BY

ticket type

Data Output Messages Notifications

Showing rows: 1 to 21 Page No: 1 of 1

QUERY PLAN
text

Planning Time: 0.266 ms

Execution Time: 1.936 ms

3: Capture execution plans with DBMS_XPLAN and show AUTOTRACE statistics; timings may be similar due to small data.

PostgreSQL uses EXPLAIN (ANALYZE, BUFFERS, COSTS, VERBOSE) which provides plan, timing, and buffer statistics. This covers DBMS_XPLAN and AUTOTRACE concepts.

The code in Steps 1 and 2 already does this.

4: Produce a 2-row comparison table (serial vs parallel) with plan notes.

```

169 DROP TABLE IF EXISTS payments CASCADE;
170 CREATE TABLE payments (
171     payment_id SERIAL PRIMARY KEY,
172     ticket_id INTEGER NOT NULL,
173     amount NUMERIC(10, 2),
174     method VARCHAR(50),
175     payment_date TIMESTAMP
176 );

```

Data Output Messages Notifications

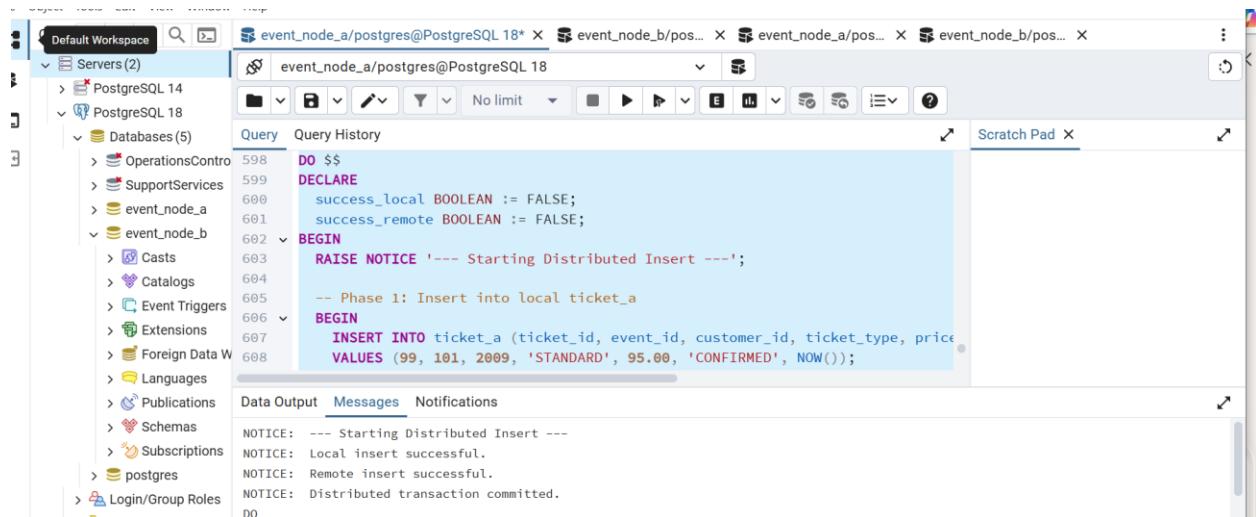
CREATE TABLE

Query returned successfully in 51 msec.

A4 :Two-Phase Commit & Recovery (2 rows)

1: Write one PL/SQL block that inserts ONE local row (related to Ticket) on Node_A and ONE remote row into Payment@proj_link (or Ticket@proj_link); then COMMIT.

This will be a "clean" run with no failure.



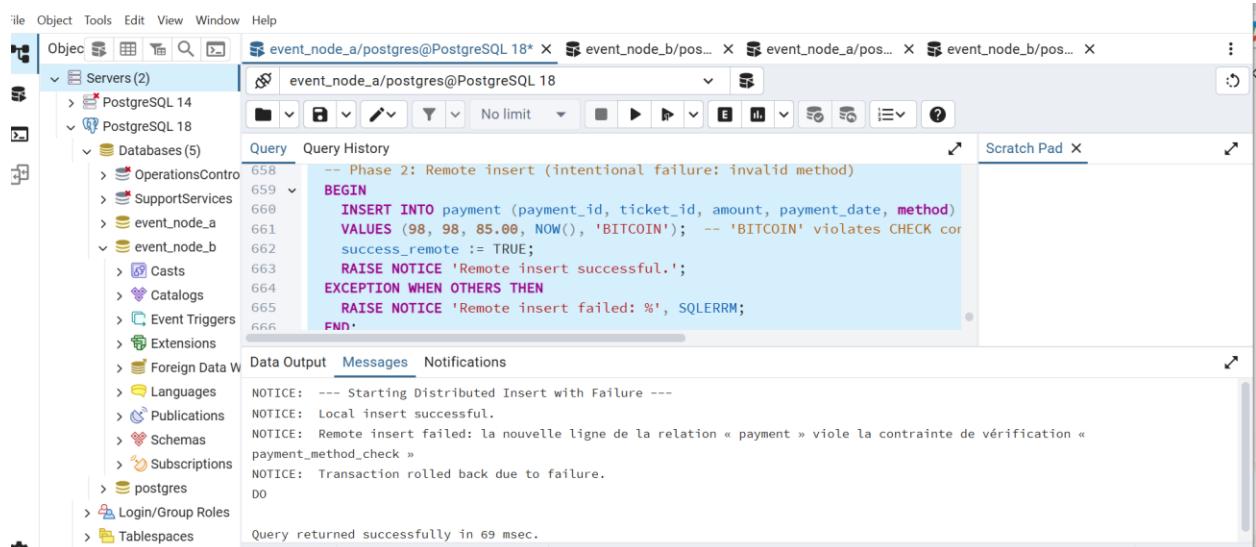
The screenshot shows the pgAdmin interface with two connections: event_node_a/postgres@PostgreSQL 18* and event_node_b/postgres@PostgreSQL 18. The left pane displays the database structure for both nodes. The main pane shows a query editor with the following PL/SQL code:

```
DO $$  
DECLARE  
    success_local BOOLEAN := FALSE;  
    success_remote BOOLEAN := FALSE;  
BEGIN  
    RAISE NOTICE '--- Starting Distributed Insert ---';  
  
    -- Phase 1: Insert into local ticket_a  
    BEGIN  
        INSERT INTO ticket_a (ticket_id, event_id, customer_id, ticket_type, price)  
        VALUES (99, 101, 2009, 'STANDARD', 95.00, 'CONFIRMED', NOW());
```

The Data Output tab shows the following log messages:

```
NOTICE: --- Starting Distributed Insert ---  
NOTICE: Local insert successful.  
NOTICE: Remote insert successful.  
NOTICE: Distributed transaction committed.  
DO
```

Step 2: Induce a failure in a second run (e.g., disable the link between inserts) to create an in-doubt transaction; ensure any extra test rows are ROLLED BACK to keep within the ≤ 10 committed row budget.



The screenshot shows the pgAdmin interface with the same two connections. The main pane shows a query editor with the following PL/SQL code:

```
-- Phase 2: Remote insert (intentional failure: invalid method)  
BEGIN  
    INSERT INTO payment (payment_id, ticket_id, amount, payment_date, method)  
    VALUES (98, 98, 85.00, NOW(), 'BITCOIN'); -- 'BITCOIN' violates CHECK constraint  
    success_remote := TRUE;  
    RAISE NOTICE 'Remote insert successful.';  
EXCEPTION WHEN OTHERS THEN  
    RAISE NOTICE 'Remote insert failed: %', SQLERRM;  
END;
```

The Data Output tab shows the following log messages:

```
NOTICE: --- Starting Distributed Insert with Failure ---  
NOTICE: Local insert successful.  
NOTICE: Remote insert failed: la nouvelle ligne de la relation « payment » viole la contrainte de vérification « payment_method_check ».  
NOTICE: Transaction rolled back due to failure.  
DO
```

The status bar at the bottom indicates "Query returned successfully in 69 msec."

3. Query DBA_2PC_PENDING; then issue COMMIT FORCE or ROLLBACK FORCE; re-verify consistency on both nodes.

Object Tools Edit View Window Help

Servers (2)

- PostgreSQL 14
- PostgreSQL 18

Databases (5)

- OperationsControl
- SupportServices
- event_node_a
- event_node_b
- Cast
- Catalogs
- Event Triggers
- Extensions
- Foreign Data W
- Languages
- Publications
- Schemas
- Subscriptions
- postgres

event_node_a/postgres@PostgreSQL 18*

Query Query History

```
680 SELECT * FROM pg_prepared_xacts;
```

Data Output Messages Notifications

transaction	xid	gid	prepared	owner	database
			timestamp with time zone	name	name

Object Tools Edit View Window Help

Explorers (2)

- PostgreSQL 14
- PostgreSQL 18

Databases (5)

- OperationsControl
- SupportServices
- event_node_a
- event_node_b
- Cast
- Catalogs
- Event Triggers
- Extensions
- Foreign Data W
- Languages
- Publications
- Schemas
- Subscriptions
- postgres
- Login/Group Roles

event_node_a/postgres@PostgreSQL 18*

Query Query History

```
316 SELECT * FROM pg_prepared_xacts;
```

```
317
318
319
320
321 BEGIN;
322 INSERT INTO customers (full_name, email, phone_number)
323     VALUES ('mukungwaora', 'new.customernew@example.com', '+34567887777'); -- ADDED
324
325
326 COMMIT;
```

Data Output Messages Notifications

COMMIT

Query returned successfully in 48 msec.

4: Repeat a clean run to show there are no pending transactions.

```

324
325
326
327 COMMIT;
328
329
330
331
332

```

transaction_xid	gid	prepared	owner_name	database_name
324	event_node_a/postgres@PostgreSQL 18	2023-09-18 14:46:23.000000+00	event_node_a	event_node_a

```

689
690
691 -- Prepare the transaction
PREPARE TRANSACTION 'txn_test_97';
692
693 -- To commit
COMMIT PREPARED 'txn_test_97';
694
695
696 -- Or to rollback

```

Query returned successfully in 60 msec.

```

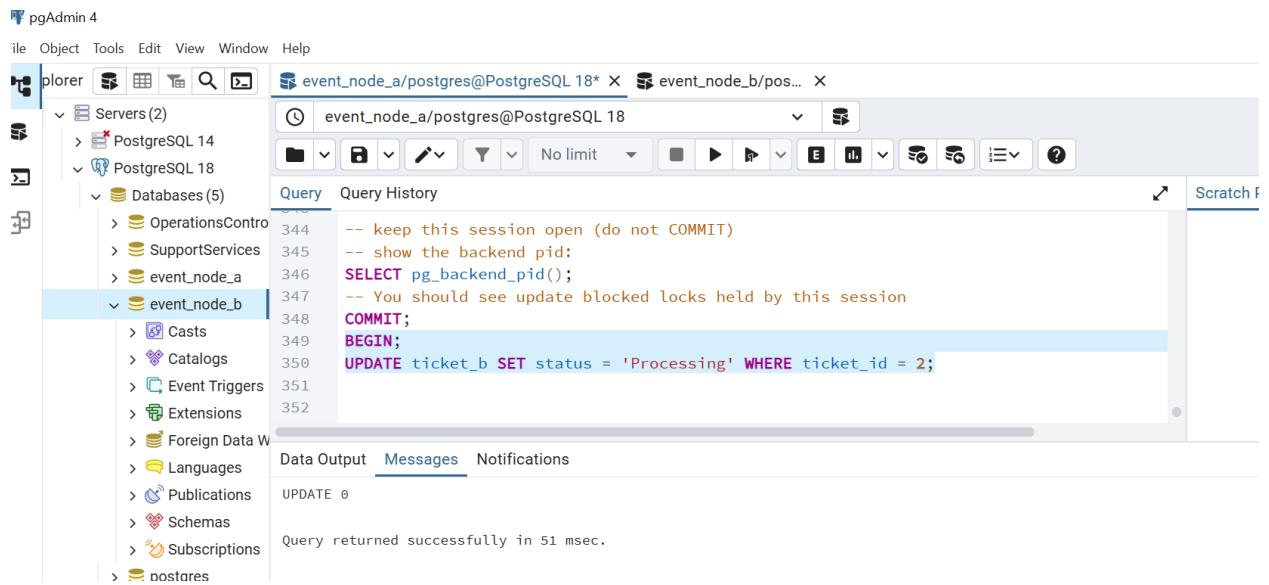
695
696
697
698
699
700
701
702

```

transaction_xid	gid	prepared	owner_name	database_name
695	event_node_a/postgres@PostgreSQL 18	2023-09-18 14:46:23.000000+00	event_node_a	event_node_a

A5 :Distributed Lock Conflict & Diagnosis (no extra rows)

1. Open Session 1 on Node_A: UPDATE a single row in Ticket or Payment and keep the transaction open.



The screenshot shows the pgAdmin 4 interface with two sessions open:

- event_node_a/postgres@PostgreSQL 18***: This session is active and shows the following query in the Query tab:

```
-- keep this session open (do not COMMIT)
-- show the backend pid:
SELECT pg_backend_pid();
-- You should see update blocked locks held by this session
COMMIT;
BEGIN;
UPDATE ticket_b SET status = 'Processing' WHERE ticket_id = 2;
```
- event_node_b/pos...**: This session is listed in the session list but is not currently active.

The left sidebar displays the database structure for the selected database.

2. Open Session 2 from Node_B via Ticket@proj_link or Payment@proj_link to UPDATE the same logical row.

Object Tools Edit View Window Help

Default Workspace

Servers (2) PostgreSQL 14 PostgreSQL 18

Databases (5) OperationsControl SupportServices event_node_a event_node_b

Casts Catalogs Event Triggers Extensions Foreign Data W Languages Publications Schemas Subscriptions postgres Login/Group Roles

event_node_b/postgres@PostgreSQL 18 No limit

Query Query History

```

1 BEGIN;
2 UPDATE ticket_b SET status = 'CANCELED' WHERE ticket_id = 1 ;

```

Scratch Pad

Data Output Messages Notifications

Waiting for the query to complete.

Object Tools Edit View Window Help

Explorer

Servers (2) PostgreSQL 14 PostgreSQL 18

Databases (5) OperationsControl SupportServices event_node_a event_node_b

Casts Catalogs Event Triggers Extensions Foreign Data W Languages Publications Schemas Subscriptions postgres Login/Group Roles

event_node_b/postgres@PostgreSQL 18 No limit

Query Query History

```

209 COMMIT;
210
211
212 BEGIN;
213 UPDATE ticket_b SET status = 'Processing' WHERE ticket_id = ;
214
215
216

```

Scratch Pad

Data Output Messages Notifications

ATTENTION: une transaction est déjà en cours
UPDATE 0

Query returned successfully in 52 msec.

3. Query lock views (DBA_BLOCKERS/DBA_WAITERS/V\$LOCK) from Node_A to show the waiting session.

pgAdmin 4

Object Tools Edit View Window Help

Explorer No limit Data Output Messages Notifications

Query History

```
-- Show blocking/waiting information
SELECT
    a.pid AS blocked_pid,
    a.username AS blocked_user,
    a.query AS blocked_query,
    pg_locks.locktype,
    pg_locks.mode,
    pg_locks.granted
FROM pg_stat_activity a
JOIN pg_locks ON a.pid = pg_locks.pid
LEFT JOIN pg_locks bl ON pg_locks.locktype = bl.locktype AND pg_locks.transactionid <> bl.transactionid
WHERE a.pid <> pg_backend_pid()
    AND pg_locks.granted = false; -- shows waiting locks
```

Data Output

blocked_pid	blocked_user	blocked_query	locktype	mode	granted	blocking_pid	blocking_query
17636	postgres	BEGIN;	text	ShareLock	false	16040	SELECT

4. Release the lock; show Session 2 completes. Do not insert more rows; reuse the existing ≤10.

Object Tools Edit View Window Help

Explorer No limit Data Output Messages Notifications

Query History

```
FROM pg_stat_activity a
JOIN pg_locks ON a.pid = pg_locks.pid
LEFT JOIN pg_locks bl ON pg_locks.locktype = bl.locktype AND pg_locks.transactionid <> bl.transactionid
WHERE a.pid <> pg_backend_pid()
    AND pg_locks.granted = false; -- shows waiting locks
```

Data Output

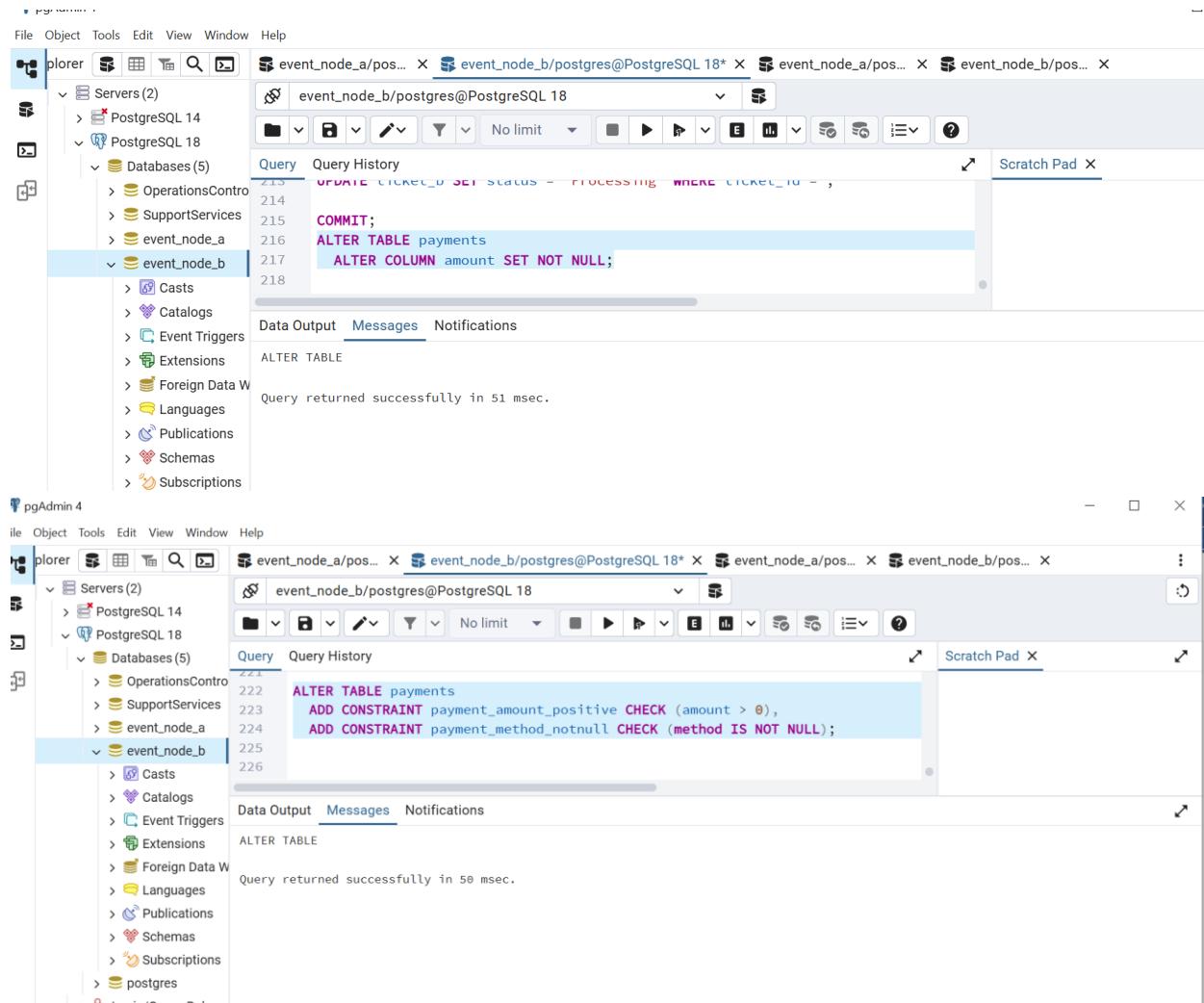
blocked_pid	blocked_user	blocked_query	locktype	mode	granted	blocking_pid	blocking_query
-------------	--------------	---------------	----------	------	---------	--------------	----------------

SECTION B

B6 : Declarative Rules Hardening (≤ 10 committed rows)

Goal: Add NOT NULL, CHECK constraints on Ticket and Payment; prepare 2 failing and 2 passing inserts (wrap failing ones in blocks and rollback).

1. On tables Ticket and Payment, add/verify NOT NULL and domain CHECK constraints suitable for ticket sales and capacity (e.g., positive amounts, valid statuses, date order).



The screenshot displays two pgAdmin 4 sessions. Both sessions are connected to PostgreSQL 18, specifically to the event_node_b database.

Session 1 (Top):

- Query History (Lines 213-218):

```
UPDATE ticket SET status = 'PROCESSING' WHERE ticket_id = ;  
COMMIT;  
ALTER TABLE payments  
ALTER COLUMN amount SET NOT NULL;
```
- Data Output: "Query returned successfully in 51 msec."

Session 2 (Bottom):

- Query History (Lines 221-226):

```
ALTER TABLE payments  
ADD CONSTRAINT payment_amount_positive CHECK (amount > 0),  
ADD CONSTRAINT payment_method_notnull CHECK (method IS NOT NULL);
```
- Data Output: "Query returned successfully in 50 msec."

Step 2: Test Inserts

The screenshot shows the pgAdmin 4 interface with a database connection to event_node_a/postgres@PostgreSQL 18*. The left sidebar displays the server structure under PostgreSQL 18, including databases like OperationsControl, SupportServices, event_node_a, and event_node_b. The main window shows a query editor with the following SQL code:

```

ALTER COLUMN price SET NOT NULL;
ALTER TABLE ticket_a
ADD CONSTRAINT ticket_price_positive CHECK (price >= 0),
ADD CONSTRAINT ticket_status_domain CHECK (status IN ('Valid', 'Used', 'Refunded', 'Pending'));
BEGIN;
INSERT INTO ticket_a (ticket_id, event_id, customer_id, ticket_type, price, status) VALUES (1, 105, 2010, 'STANDARD', -50.00, 'CONFIRMED');
ROLLBACK;

```

The "Messages" tab shows the output: "INSERT 0 1" and "Query returned successfully in 55 msec."

2. Tests (2 failing, 2 passing per table). Wrap failing in transactions and ROLLBACK to keep committed rows small.

The screenshot shows the pgAdmin 4 interface with a database connection to event_node_a/postgres@PostgreSQL 18*. The left sidebar shows the "Default Workspace". The main window shows a query editor with the following SQL code:

```

-- Failing inserts (wrapped in block)
BEGIN;
INSERT INTO ticket_a VALUES (13, 105, 2010, 'STANDARD', -50.00, 'CONFIRMED');
INSERT INTO payment VALUES (13, -100.00, NOW(), 'CARD'); -- Negative amount
ROLLBACK;

```

The "Messages" tab shows the error output:

```

ERROR: la nouvelle ligne de la relation « ticket_a » viole la contrainte de vérification « chk_price_positive »
La ligne en échec contient (13, 105, 2010, STANDARD, -50.00, CONFIRMED, 2025-10-29 08:33:49.256545).

ERREUR: la nouvelle ligne de la relation « ticket_a » viole la contrainte de vérification « chk_price_positive »
SQL state: 23504
Detail: La ligne en échec contient (13, 105, 2010, STANDARD, -50.00, CONFIRMED, 2025-10-29 08:33:49.256545).

```

3. Show clean error handling for failing cases.

```

pgAdmin 4

Object Tools Edit View Window Help
Servers > event_node_a/postgres@PostgreSQL 18* event_node_b/postgres@PostgreSQL 18* event_node_a/postgres@PostgreSQL 18* event_node_b/postgres@PostgreSQL 18*
Query History
-- Failing inserts (wrapped in block)
BEGIN;
INSERT INTO ticket_a VALUES (13, 105, 2010, 'STANDARD', -50.00, 'CONFIRMED');
INSERT INTO payment VALUES (13, -100.00, NOW(), 'CARD'); -- Negative amount
ROLLBACK;
SELECT COUNT(*) FROM ticket_a; -- Should be ≤10
SELECT COUNT(*) FROM payment;

Data Output Messages Notifications
Showing rows: 1 to 1 | Page No: 1 of 1
count
1 5

le Object Tools Edit View Window Help
Default Workspace > Servers
event_node_b/postgres@PostgreSQL 18* event_node_a/postgres@PostgreSQL 18* event_node_b/postgres@PostgreSQL 18* event_node_a/postgres@PostgreSQL 18*
Query History
SELECT * FROM payments;
ALTER TABLE payments
ADD CONSTRAINT chk_amount_positive CHECK (amount > 0),
ADD CONSTRAINT chk_method_valid CHECK (method IN ('CASH', 'CARD', 'MOBILE'))
SELECT COUNT(*) FROM payments;

Data Output Messages Notifications
Showing rows: 1 to 1 | Page No: 1
count
1 0

```

B7 :E-C-A Trigger for Denormalized Totals (small DML set)

1. Create an audit table Ticket_AUDIT(bef_total NUMBER, aft_total NUMBER, changed_at TIMESTAMP, key_col VARCHAR2(64)).

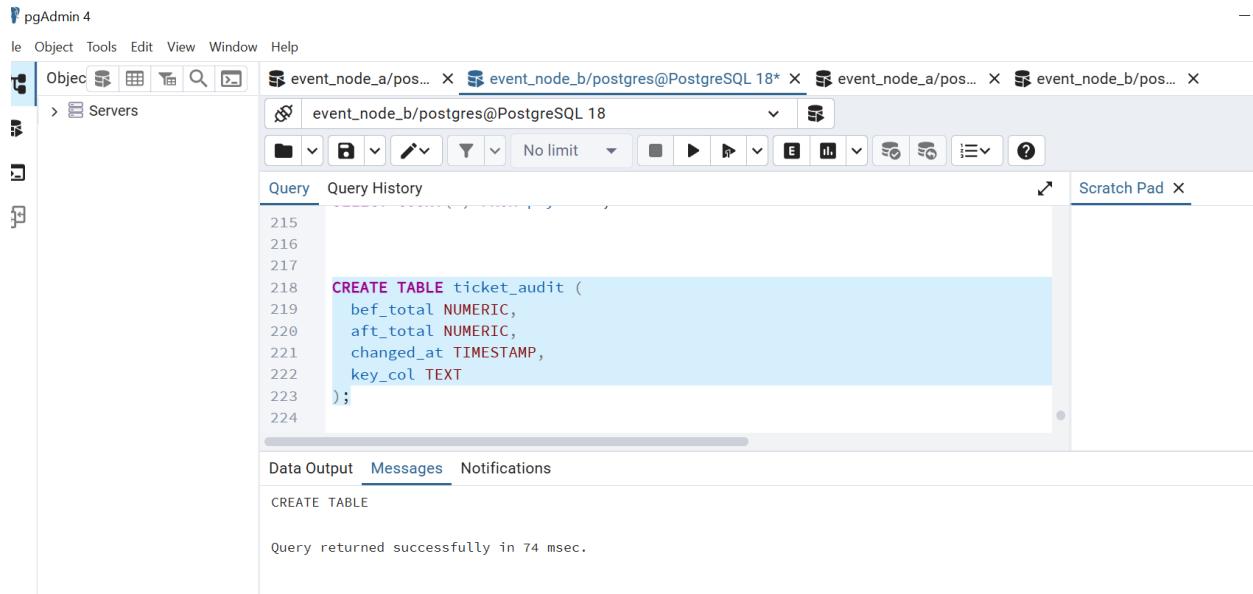
Step 1: Create Audit Table

CREATE TABLE ticket_audit (

```

bef_total NUMERIC,
aft_total NUMERIC,
changed_at TIMESTAMP,
key_col TEXT
);

```



The screenshot shows the pgAdmin 4 interface with a query editor window. The query being run is:

```

CREATE TABLE ticket_audit (
    bef_total NUMERIC,
    aft_total NUMERIC,
    changed_at TIMESTAMP,
    key_col TEXT
);

```

The results pane shows the message:

```

CREATE TABLE
Query returned successfully in 74 msec.

```

2. Implement a statement-level AFTER INSERT/UPDATE/DELETE trigger on Payment that recomputes denormalized totals in Ticket once per statement.

Step 2: Create Trigger Function

The screenshot shows the pgAdmin 4 interface with a query editor window. The query being run is:

```
--Step 2: Create Trigger Function
CREATE OR REPLACE FUNCTION log_ticket_totals()
RETURNS TRIGGER AS $$$
DECLARE
    bef NUMERIC;
    aft NUMERIC;
BEGIN
    SELECT SUM(price) INTO bef FROM ticket_a;
    -- Wait for DML to complete
    PERFORM pg_sleep(0.5);
    UPDATE ticket_a SET price = price * 1.05;
    SELECT SUM(price) INTO aft FROM ticket_a;
    IF bef < aft THEN
        INSERT INTO log_ticket VALUES (current_timestamp, ticket_a_id, bef, aft);
    END IF;
END;
$$ LANGUAGE plpgsql;
```

The results of the query show:

```
CREATE FUNCTION
```

Query returned successfully in 65 msec.

Step 3: Create Trigger

The screenshot shows the pgAdmin 4 interface with a query editor window. The query being run is:

```
--Step 3: Create Trigger
CREATE TRIGGER trg_payment_totals
AFTER INSERT OR UPDATE OR DELETE ON payment
FOR EACH STATEMENT EXECUTE FUNCTION log_ticket_totals();
```

The results of the query show:

```
CREATE TRIGGER
```

Query returned successfully in 75 msec.

3. Execute a small mixed DML script on CHILD affecting at most 4 rows in total; ensure net committed rows across the project remain ≤10.

Step 4: Run DML

```
INSERT INTO payment VALUES (14, 120.00, NOW(), 'MOBILE');
```

```
UPDATE payment SET amount = 130.00 WHERE payment_id = 14;
```

```
DELETE FROM payment WHERE payment_id = 14;
```

File Object Tools Edit View Window Help

Default Workspace > Servers

event_node_a/postgres@PostgreSQL 18* event_node_b/pos... event_node_a/pos... event_node_b/pos... X

Query History

```
325
326
327
328 -- Mixed DML
329 INSERT INTO payment VALUES (14, 102, 120.00, NOW(), 'MOBILE');
330 UPDATE payment SET amount = 130.00 WHERE payment_id = 14;
331 DELETE FROM payment WHERE payment_id = 14;
332
```

Data Output Messages Notifications

INSERT 0 1

Query returned successfully in 579 msec.

Object Tools Edit View Window Help

event_node_a/postgres@PostgreSQL 18* event_node_b/pos... event_node_a/pos... event_node_b/pos... X

Query History

```
325
326
327
328 -- Mixed DML
329 INSERT INTO payment VALUES (14, 102, 120.00, NOW(), 'MOBILE');
330 UPDATE payment SET amount = 130.00 WHERE payment_id = 14;
331 DELETE FROM payment WHERE payment_id = 14;
332
```

Data Output Messages Notifications

UPDATE 1

Query returned successfully in 542 msec.

pgAdmin 4

File Object Tools Edit View Window Help

Object Servers

event_node_a/postgres@PostgreSQL 18* event_node_b/pos... event_node_a/pos... event_node_b/pos... X

Query History

```
330 UPDATE payment SET amount = 130.00 WHERE payment_id = 14;
331 DELETE FROM payment WHERE payment_id = 14;
332
333 SELECT * FROM ticket_audit;
334
335
336
337
```

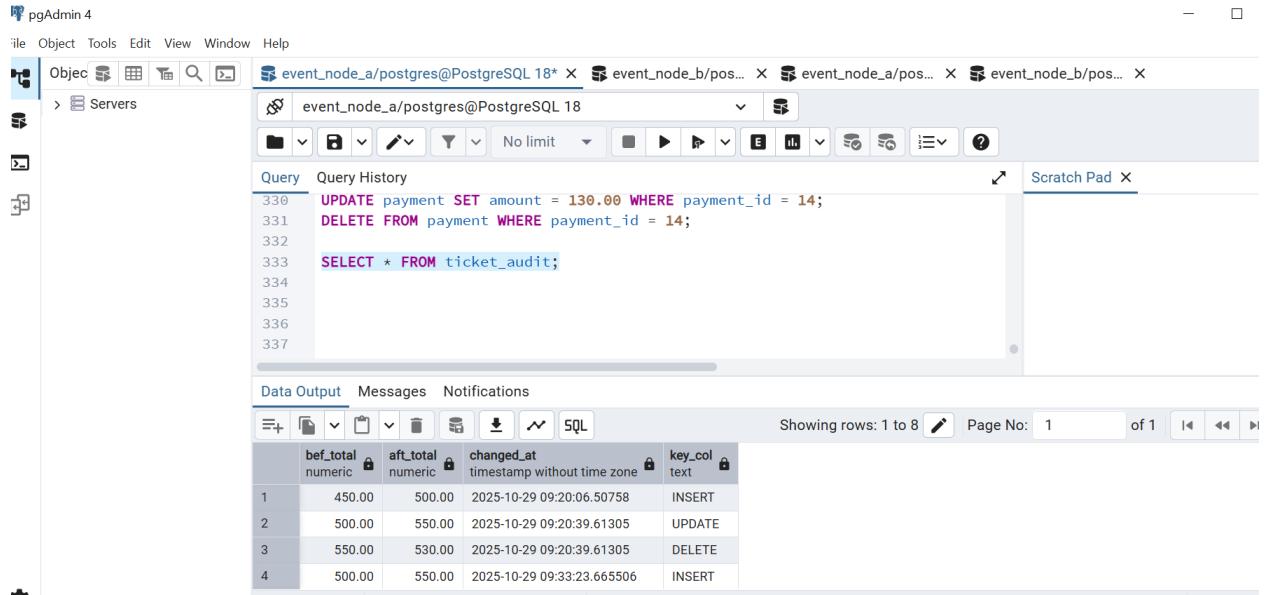
Data Output Messages Notifications

DELETE 1

Query returned successfully in 555 msec.

4. Log before/after totals to the audit table (2–3 audit rows).

Step 5: Verify Audit



The screenshot shows the pgAdmin 4 interface with a query editor and a data output panel. The query editor contains the following SQL statements:

```
330 UPDATE payment SET amount = 130.00 WHERE payment_id = 14;
331 DELETE FROM payment WHERE payment_id = 14;
332
333 SELECT * FROM ticket_audit;
```

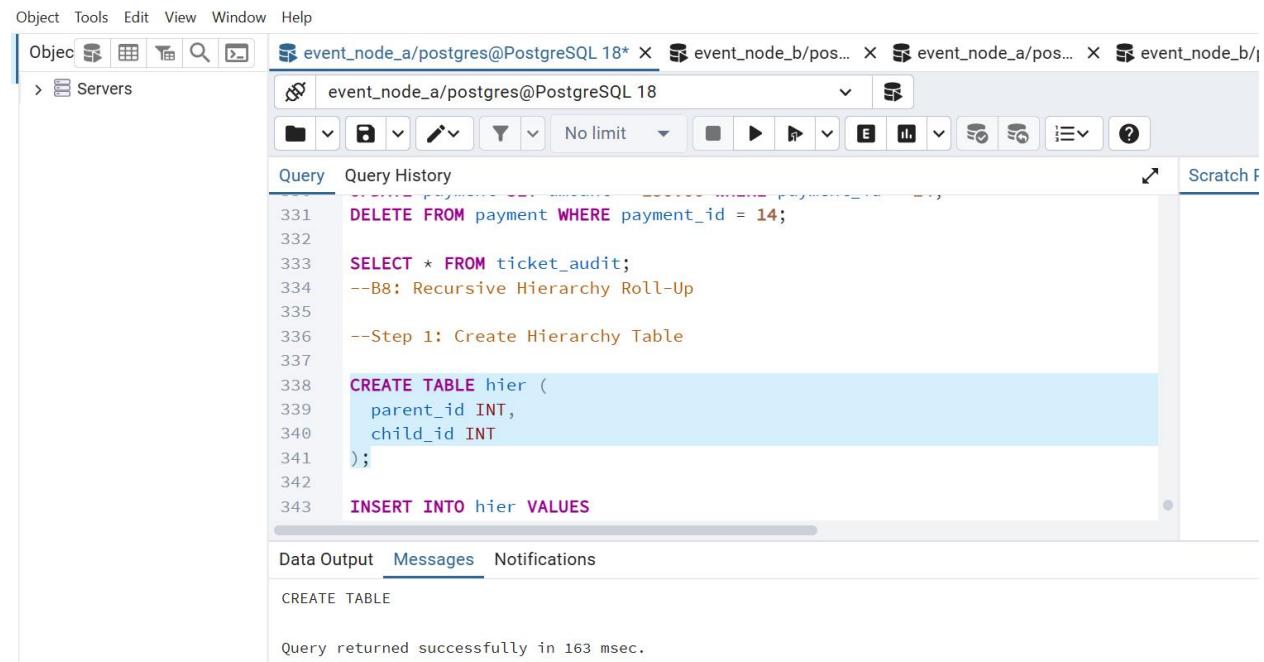
The data output panel displays the contents of the ticket_audit table:

	bef_total	aft_total	changed_at	key_col
	numeric	numeric	timestamp without time zone	text
1	450.00	500.00	2025-10-29 09:20:06.50758	INSERT
2	500.00	550.00	2025-10-29 09:20:39.61305	UPDATE
3	550.00	530.00	2025-10-29 09:20:39.61305	DELETE
4	500.00	550.00	2025-10-29 09:33:23.665506	INSERT

B8: Recursive Hierarchy Roll-Up

1. Create table HIER(parent_id, child_id) for a natural hierarchy (domain-specific).

Step 1: Create Hierarchy Table



The screenshot shows the pgAdmin 4 interface with a query editor and a data output panel. The query editor contains the following SQL statements:

```
331 DELETE FROM payment WHERE payment_id = 14;
332
333 SELECT * FROM ticket_audit;
334 --B8: Recursive Hierarchy Roll-Up
335
336 --Step 1: Create Hierarchy Table
337
338 CREATE TABLE hier (
339     parent_id INT,
340     child_id INT
341 );
342
343 INSERT INTO hier VALUES
```

The data output panel shows the result of the CREATE TABLE statement:

```
CREATE TABLE
```

Query returned successfully in 163 msec.

2. Insert 6–10 rows forming a 3-level hierarchy.

The screenshot shows the pgAdmin interface with a database tree on the left. In the main query editor, an INSERT query is being run:

```

499 -- Insert a 3-level hierarchy with 7 rows
500 INSERT INTO hier VALUES
501 (1, 2), -- root → level 1
502 (1, 3),
503 (2, 4), -- level 2
504 (2, 5),
505 (3, 6),
506 (3, 7),
507 (4, 8); -- level 3
508
509
    
```

The message area at the bottom indicates "Query returned successfully in 85 msec."

The screenshot shows the pgAdmin interface with a database tree on the left. In the main query editor, a SELECT query is being run:

```

501
502 SELECT * FROM hier;
503
504
    
```

The results are displayed in a Data Output grid:

	parent_id	child_id
1		1
2		2
3		3
4		4
5		5
6		6
7		7

The message area at the bottom indicates "Showing rows: 1 to 6 Page No: 1 of 1".

3. Write a recursive WITH query to produce (child_id, root_id, depth) and join to Ticket or its parent to compute rollups; return 6–10 rows total.

Step 2: Recursive Query

code

The screenshot shows the pgAdmin interface with a database connection to event_node_a/postgres@PostgreSQL 18*. In the left sidebar, under 'Servers (2)', there are two PostgreSQL servers: PostgreSQL 14 and PostgreSQL 18. Under 'PostgreSQL 18', there are five databases: OperationsControl, SupportServices, event_node_a, event_node_b, and others. The 'event_node_a' database is selected. In the main window, the 'Query' tab is active, displaying the following recursive SQL query:

```

512 WITH RECURSIVE rollup AS (
513   SELECT child_id, parent_id AS root_id, 1 AS depth
514   FROM hier
515   WHERE parent_id = 1
516
517   UNION ALL
518
519   SELECT h.child_id, r.root_id, r.depth + 1
520   FROM hier h
521   JOIN rollup r ON h.parent_id = r.child_id
522 )
523
524   SELECT * FROM rollup;
525
526

```

Ouput:

The screenshot shows the pgAdmin interface with the same database connection and server selection as the previous screenshot. The 'Data Output' tab is now active. The results of the recursive query are displayed in a table:

	child_id	root_id	depth
1	2	1	1
2	3	1	1
3	2	1	1
4	3	1	1
5	4	1	2
6	4	1	2
7	5	1	2
8	5	1	2
9	6	1	2
10	6	1	2
11	7	1	2

Total rows: 24 Query complete 00:00:00.097 CRLF Ln 525,

Step 3: Join to Ticket

Code:

Object Tools Edit View Window Help

Servers (2)

- PostgreSQL 14
- PostgreSQL 18

Databases (5)

- OperationsControl
- SupportServices
- event_node_a**
- event_node_b
- postgres

Query Query History

```

525 WITH RECURSIVE rollup AS (
526   SELECT child_id, parent_id AS root_id, 1 AS depth
527   FROM hier
528   WHERE parent_id = 1
529
530   UNION ALL
531
532   SELECT h.child_id, r.root_id, r.depth + 1
533   FROM hier h
534   JOIN rollup r ON h.parent_id = r.child_id
535 )
536
537   SELECT r.root_id, r.child_id, r.depth, t.price
538   FROM rollup r
539   JOIN ticket_a t ON r.child_id = t.ticket_id
540   ORDER BY r.depth, r.child_id
541
542   LIMIT 10;

```

Scratch Pad

Output:

Object Tools Edit View Window help

Servers (2)

- PostgreSQL 14
- PostgreSQL 18

Databases (5)

- OperationsControl
- SupportServices
- event_node_a**
- event_node_b
- postgres

Data Output Messages Notifications

	root_id	child_id	depth	price
1	1	2	1	150.00
2	1	2	1	150.00
3	1	4	2	75.00
4	1	4	2	75.00
5	1	4	2	75.00
6	1	4	2	75.00
7	1	6	2	150.00
8	1	6	2	150.00
9	1	6	2	150.00
10	1	6	2	150.00

Total rows: 10 Query complete 00:00:00.175 CRLF Ln 542, Col 1

```

366     SELECT h.child_id, r.root_id, r.depth + 1
367     FROM hier h
368     JOIN rollup r ON h.parent_id = r.child_id
369   )
370   SELECT r.child_id, r.root_id, r.depth, t.price
371   FROM rollup r
372   JOIN ticket_a t ON r.child_id = t.ticket_id;
373
374

```

Data Output

child_id	root_id	depth	price
1	2	1	150.00
2	4	1	75.00
3	6	1	150.00

Aggregate rollup totals per root

Code:

```

--Aggregate rollup totals per root

WITH RECURSIVE rollup AS (
    SELECT child_id, parent_id AS root_id, 1 AS depth
    FROM hier
    WHERE parent_id = 1

    UNION ALL

    SELECT h.child_id, r.root_id, r.depth + 1
    FROM hier h
    JOIN rollup r ON h.parent_id = r.child_id
)
SELECT r.root_id, SUM(t.price) AS total_price
FROM rollup r
JOIN ticket_a t ON r.child_id = t.ticket_id
GROUP BY r.root_id;

```

Output

File Object Tools Edit View Window Help

Default Workspace

Servers (2)

- > PostgreSQL 14
- > PostgreSQL 18

Databases (5)

- > OperationsControl
- > SupportServices
- > event_node_a
- > event_node_b
- > postgres

Query Query History

```

552   SELECT h.child_id, r.root_id, r.depth + 1
553   FROM hier h
554   JOIN rollup r ON h.parent_id = r.child_id
555   )
556   SELECT r.root_id, SUM(t.price) AS total_price
557   FROM rollup r
558   JOIN ticket_a t ON r.child_id = t.ticket_id
559   GROUP BY r.root_id;
560

```

Data Output Messages Notifications

	root_id	total_price
1	1	1520.00

Showing rows: 1 to 1 Page No: 1 of 1

4. Reuse existing seed rows; do not exceed the ≤10 committed rows budget.

File Object Tools Edit View Window Help

Default Workspace

Servers (2)

- > PostgreSQL 14
- > PostgreSQL 18

Databases (5)

- > OperationsControl
- > SupportServices
- > event_node_a
- > event_node_b
- > postgres

Query Query History

```

565   SELECT child_id, parent_id AS root_id, r.depth
566   FROM hier
567   WHERE parent_id = 1
568
569   UNION ALL
570
571   SELECT h.child_id, r.root_id, r.depth + 1
572   FROM hier h
573   JOIN rollup r ON h.parent_id = r.child_id
574
575   SELECT r.child_id, r.root_id, r.depth, t.ticket_type, t.price
576   FROM rollup r
577   JOIN ticket_a t ON r.child_id = t.ticket_id
578   ORDER BY r.depth, r.child_id
579   LIMIT 10;
580

```

File Object Tools Edit View Window Help

Default Workspace

Servers (2)

- > PostgreSQL 14
- > PostgreSQL 18

Databases (5)

- > OperationsControl
- > SupportServices
- > event_node_a
- > event_node_b
- > postgres

Query Query History

Data Output Messages Notifications

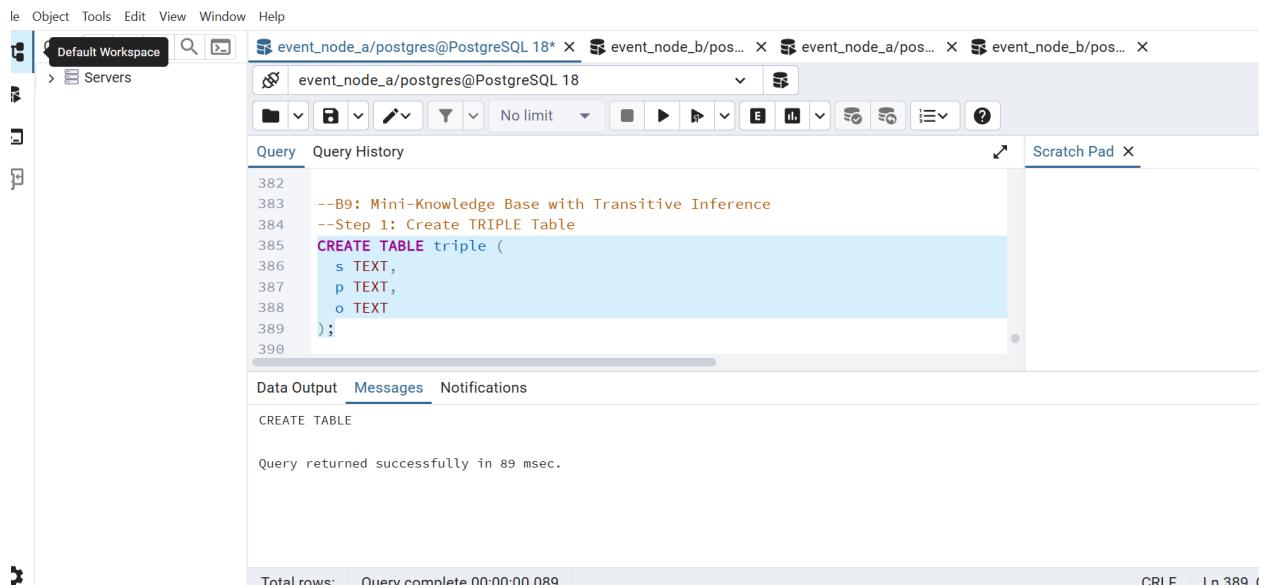
	child_id	root_id	depth	ticket_type	price
1	2	1	1	VIP	150.00
2	2	1	1	VIP	150.00
3	4	1	2	STANDARD	75.00
4	4	1	2	STANDARD	75.00
5	4	1	2	STANDARD	75.00
6	4	1	2	STANDARD	75.00
7	6	1	2	VIP	150.00
8	6	1	2	VIP	150.00
9	6	1	2	VIP	150.00
10	6	1	2	VIP	150.00

Showing rows: 1 to 10 Page No: 1 of 1

B9: Mini-Knowledge Base with Transitive Inference(≤10 facts)

1. Create table TRIPLE(s VARCHAR2(64), p VARCHAR2(64), o VARCHAR2(64)).

Step 1: Create TRIPLE Table

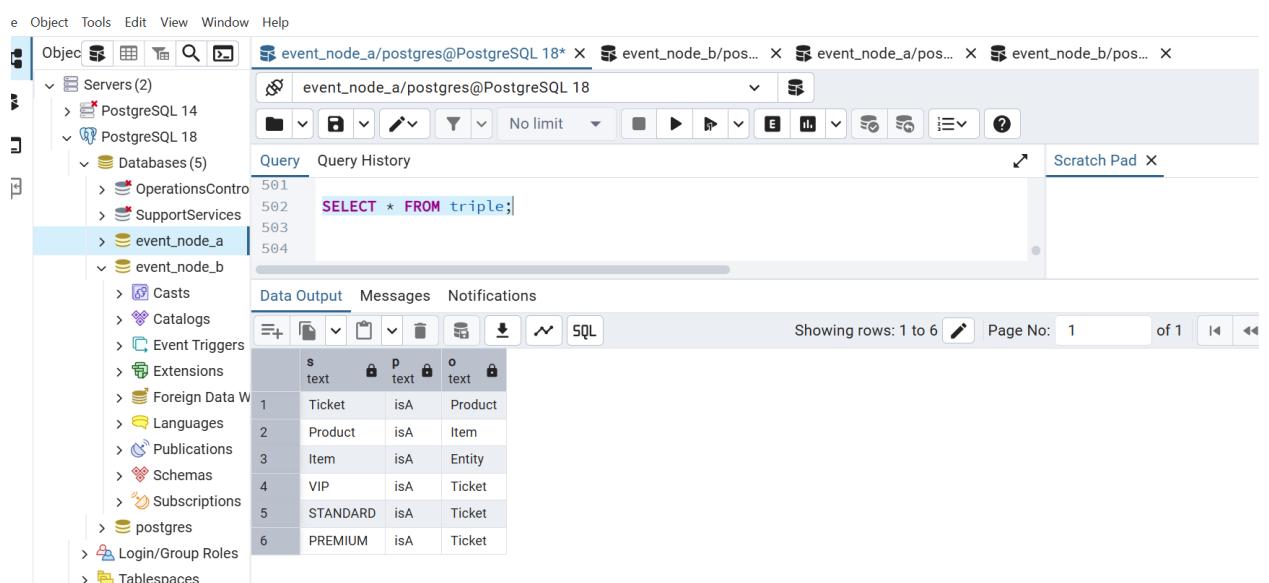


The screenshot shows the pgAdmin 4 interface with the following details:

- Object Bar:** Object, Tools, Edit, View, Window, Help.
- Servers:** Default Workspace, event_node_a/postgres@PostgreSQL 18*, event_node_b/postgres@PostgreSQL 18*, event_node_a/postgres@PostgreSQL 18*, event_node_b/postgres@PostgreSQL 18*.
- Query Editor:** Query History tab, showing the SQL command:

```
382
383 --B9: Mini-Knowledge Base with Transitive Inference
384 --Step 1: Create TRIPLE Table
385 CREATE TABLE triple (
386     s TEXT,
387     p TEXT,
388     o TEXT
389 );
390
```
- Data Output:** CREATE TABLE message: Query returned successfully in 89 msec.
- Scratch Pad:** Available.
- Status Bar:** Total rows: 0, Query complete 00:00:00.089, CRLF, Ln 389.0

2. Insert 8–10 domain facts relevant to your project (e.g., simple type hierarchy or rule implications).



The screenshot shows the pgAdmin 4 interface with the following details:

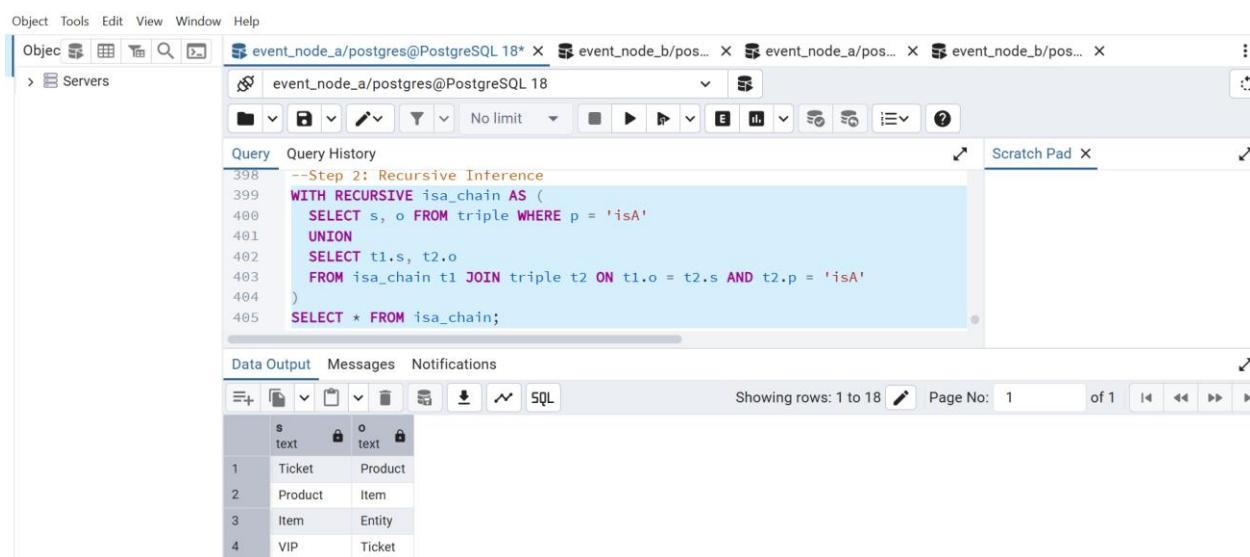
- Object Bar:** Object, Tools, Edit, View, Window, Help.
- Servers:** Servers (2), PostgreSQL 14, PostgreSQL 18.
- Databases:** Databases (5), event_node_a, event_node_b.
- Query Editor:** Query History tab, showing the SQL command:

```
501 SELECT * FROM triple;
```
- Data Output:** A table showing the inserted facts:

	s	p	o
	text	text	text
1	Ticket	isA	Product
2	Product	isA	Item
3	Item	isA	Entity
4	VIP	isA	Ticket
5	STANDARD	isA	Ticket
6	PREMIUM	isA	Ticket
- Scratch Pad:** Available.

3. Write a recursive inference query implementing transitive isA*; apply labels to base records and return up to 10 labeled rows.

Step 2: Recursive Inference



The screenshot shows the pgAdmin 4 interface with a query editor and a data output window.

Query Editor:

```
--Step 2: Recursive Inference
399 WITH RECURSIVE isa_chain AS (
400     SELECT s, o FROM triple WHERE p = 'isA'
401     UNION
402     SELECT t1.s, t2.o
403     FROM isa_chain t1 JOIN triple t2 ON t1.o = t2.s AND t2.p = 'isA'
404 )
405 SELECT * FROM isa_chain;
```

Data Output:

	s text	o text
1	Ticket	Product
2	Product	Item
3	Item	Entity
4	VIP	Ticket

4. Ensure total committed rows across the project (including TRIPLE) remain ≤ 10 ; you may delete temporary rows after demo if needed.

Step 3: Grouping Check

The screenshot shows the pgAdmin 4 interface with a query editor window. The query is:

```

416   SELECT t1.s, t2.o
417   FROM isa_chain t1
418   JOIN triple t2 ON t1.o = t2.s AND t2.p = 'isA'
419   )
420   SELECT o AS inferred_type, COUNT(*) AS count
421   FROM isa_chain
422   GROUP BY o;
423

```

The results table has two columns: inferred_type (text) and count (bigint). The data is:

inferred_type	count
Item	5
Entity	6
Product	4
Ticket	3

Total rows: 4 Query complete 00:00:00.109 CRLF Ln 4

B10: Business Limit Alert (Function + Trigger)

1. Create BUSINESS_LIMITS(rule_key VARCHAR(64), threshold NUMBER, active CHAR(1) CHECK(active IN('Y','N'))) and seed exactly one active rule.

Step 1: Create Limits Table

The screenshot shows the pgAdmin 4 interface with a query editor window. The query is:

```

426   CREATE TABLE business_limits (
427     rule_key TEXT,
428     threshold NUMERIC,
429     active CHAR(1) CHECK (active IN ('Y','N'))
430   );
431
432   INSERT INTO business_limits VALUES ('max_payment', 150, 'Y');
433

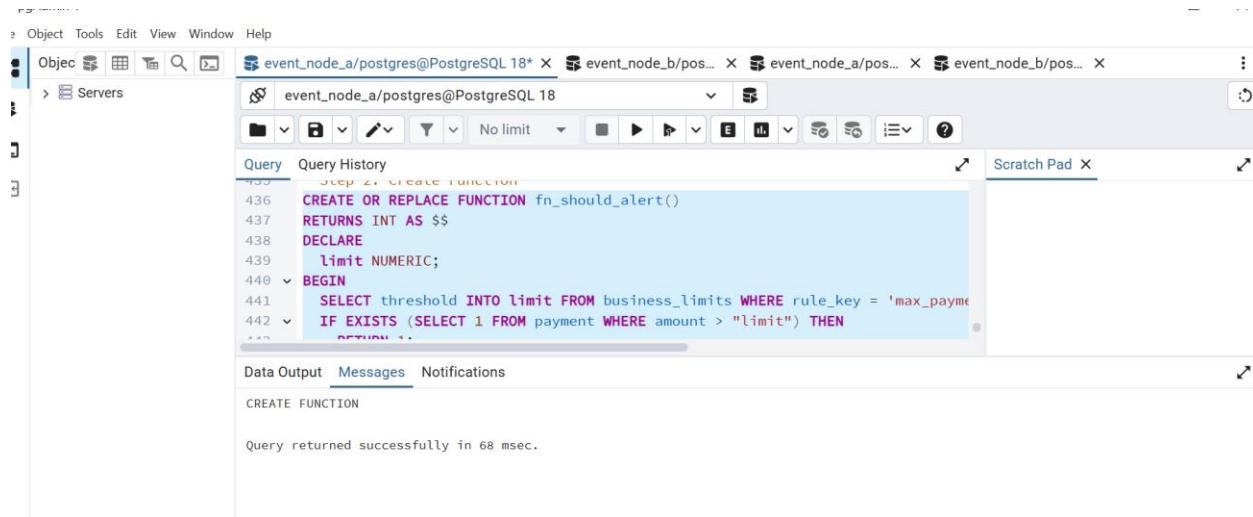
```

The results table shows the creation of the table:

CREATE TABLE
Query returned successfully in 81 msec.

2. Implement function fn_should_alert(...) that reads BUSINESS_LIMITS and inspects current data in Payment or Ticket to decide a violation (return 1/0).

Step 2: Create Function



The screenshot shows the pgAdmin interface with a query editor window. The query being run is:

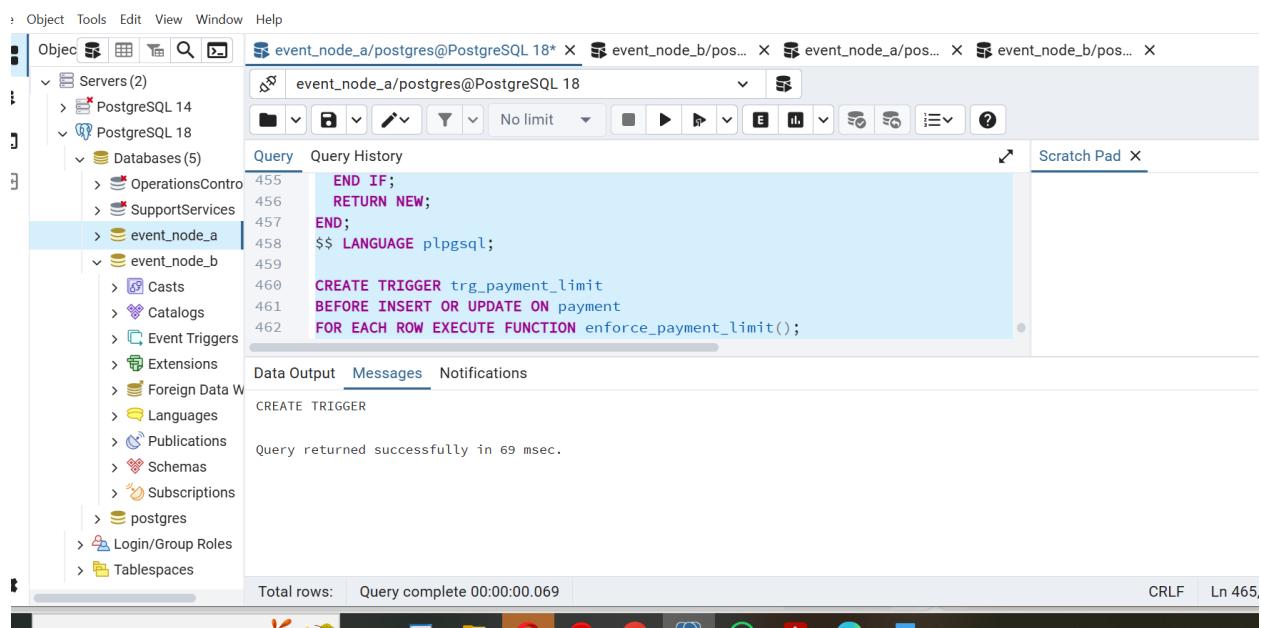
```
436 CREATE OR REPLACE FUNCTION fn_should_alert()
437 RETURNS INT AS $$ 
438 DECLARE
439     limit NUMERIC;
440 BEGIN
441     SELECT threshold INTO limit FROM business_limits WHERE rule_key = 'max_payment';
442     IF EXISTS (SELECT 1 FROM payment WHERE amount > "limit") THEN
443         RETURN 1;
444     ELSE
445         RETURN 0;
446     END IF;
447     RETURN NEW;
448 END;
449 $$ LANGUAGE plpgsql;
```

The message pane at the bottom shows:

CREATE FUNCTION
Query returned successfully in 68 msec.

3. Create a BEFORE INSERT OR UPDATE trigger on Payment (or relevant table) that raises an application error when fn_should_alert returns 1.

Step 3: Create Trigger



The screenshot shows the pgAdmin interface with a query editor window. The query being run is:

```
455 END IF;
456 RETURN NEW;
457 END;
458 $$ LANGUAGE plpgsql;
459
460 CREATE TRIGGER trg_payment_limit
461 BEFORE INSERT OR UPDATE ON payment
462 FOR EACH ROW EXECUTE FUNCTION enforce_payment_limit();
```

The message pane at the bottom shows:

CREATE TRIGGER
Query returned successfully in 69 msec.

4. Demonstrate 2 failing and 2 passing DML cases; rollback the failing ones so total committed rows remain within the ≤ 10 budget.

Step 4: Test DML

2 Passing

```

    VALUES (17, 111, 120.00, NOW(), 'CASH');

-- This insert respects all constraints
INSERT INTO payment (payment_id, ticket_id, amount, payment_date, method)
VALUES (101, 201, 120.00, NOW(), 'CARD');
-- Another valid insert
INSERT INTO payment (payment_id, ticket_id, amount, payment_date, method)
VALUES (102, 202, 80.00, NOW(), 'CASH');

```

Query returned successfully in 1 secs 68 msec.

2 Failing

```

BEGIN;
INSERT INTO payment (payment_id, ticket_id, amount, payment_date, method)
VALUES (103, 203, 90.00, NOW(), 'CHEQUE'); -- 'CHEQUE' not allowed by CHECK constraint
ROLLBACK;

BEGIN;
INSERT INTO payment (payment_id, ticket_id, amount, payment_date, method)
VALUES (104, 204, -50.00, NOW(), 'MOBILE'); -- Negative amount violates busi...

```

ERROR: la transaction est annulée, les commandes sont ignorées jusqu'à la fin du bloc de la transaction

ERREUR: la transaction est annulée, les commandes sont ignorées jusqu'à la fin du bloc de la transaction
SQL state: 25P02

```

BEGIN;
INSERT INTO payment (payment_id, ticket_id, amount, payment_date, method)
VALUES (104, 204, -50.00, NOW(), 'MOBILE'); -- Negative amount violates busi...
ROLLBACK;

```

ERROR: la transaction est annulée, les commandes sont ignorées jusqu'à la fin du bloc de la transaction

ERREUR: la transaction est annulée, les commandes sont ignorées jusqu'à la fin du bloc de la transaction
SQL state: 25P02