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

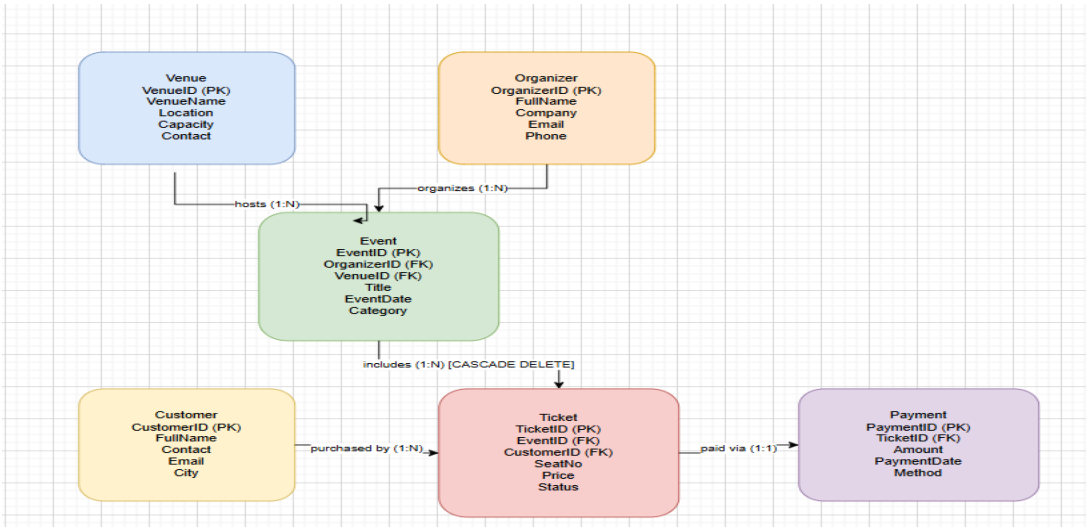
Reg: 220020603

28 October 2025

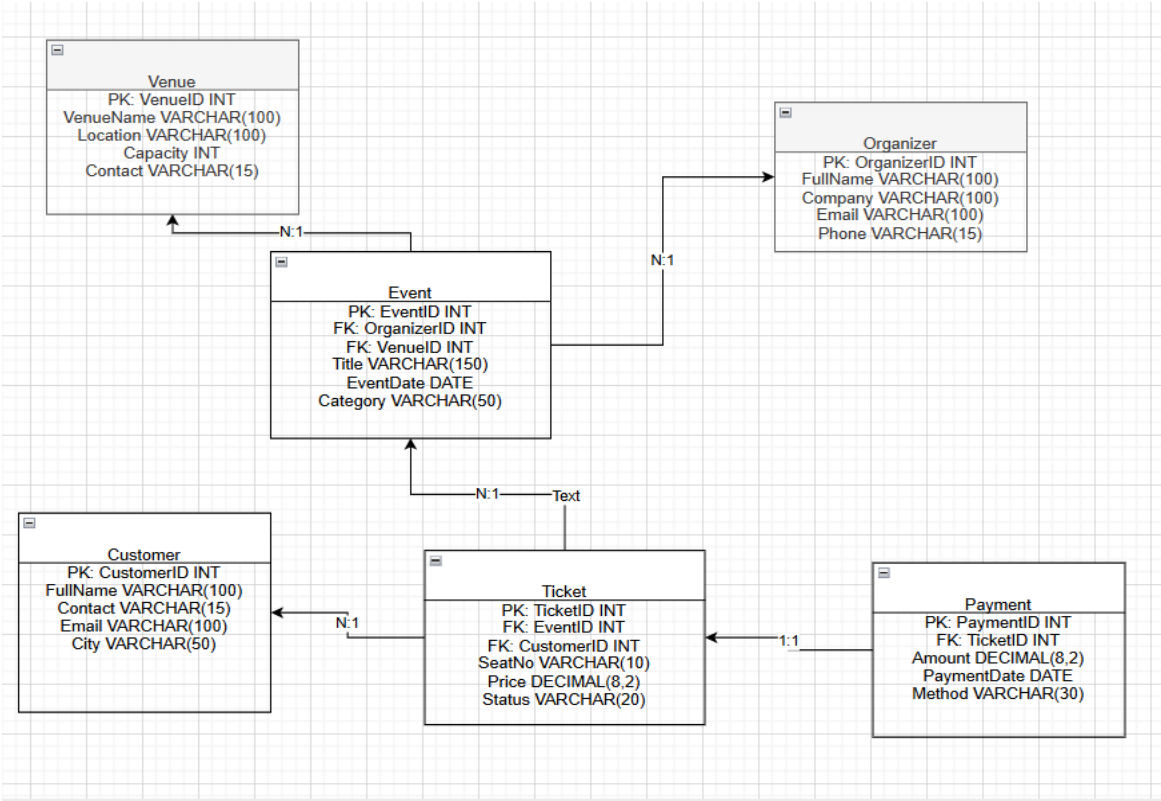
Event Booking and Ticketing Platform

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 Diagrm

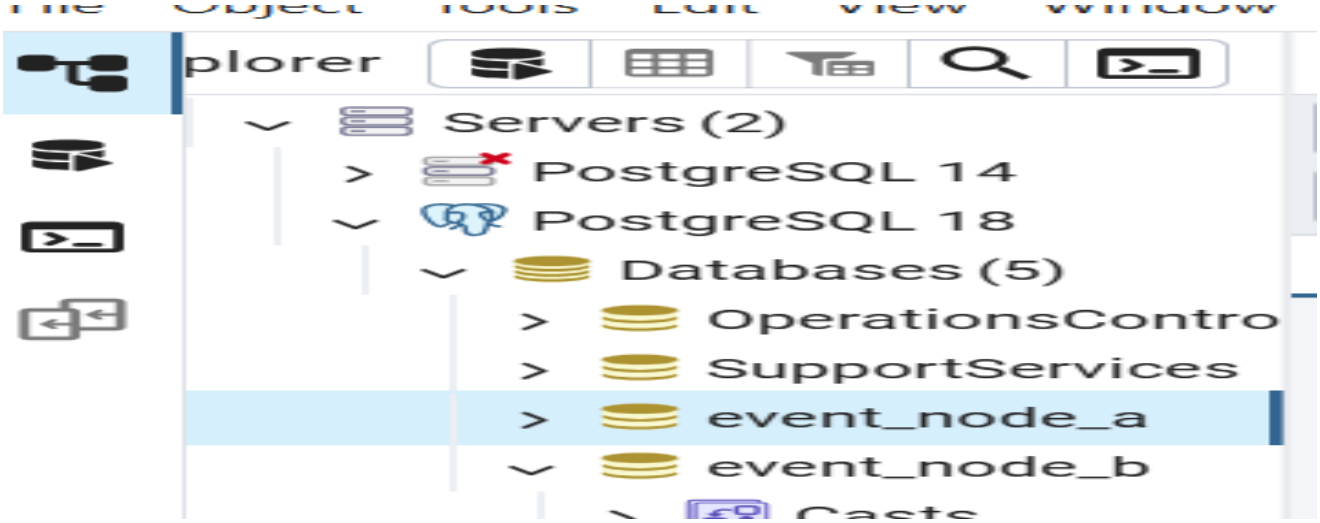


lists prerequisites

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)

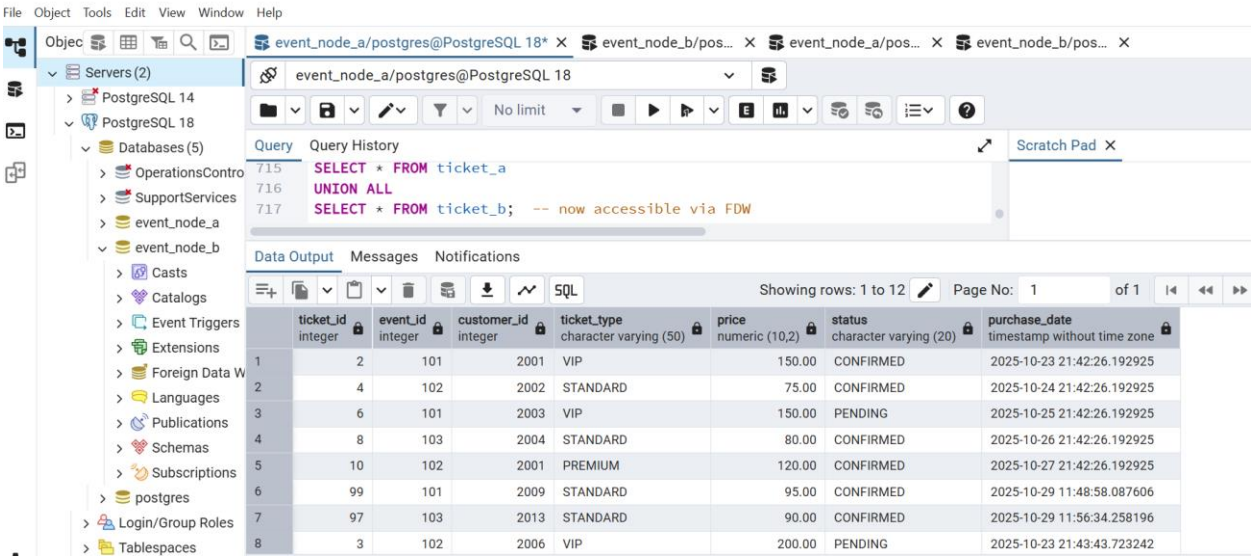
-- Rule: MOD(ticket_id, 2) = 1

INSERT INTO ticket_b VALUES

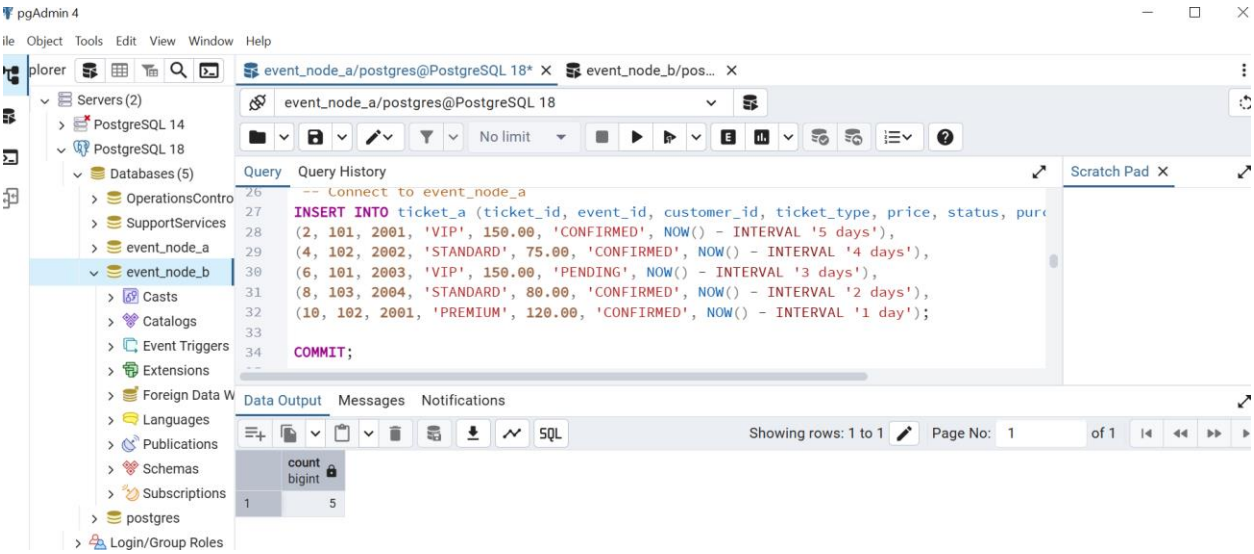
(101, 3, 1003, 'STANDARD', 85.00, 'PENDING', NOW()),

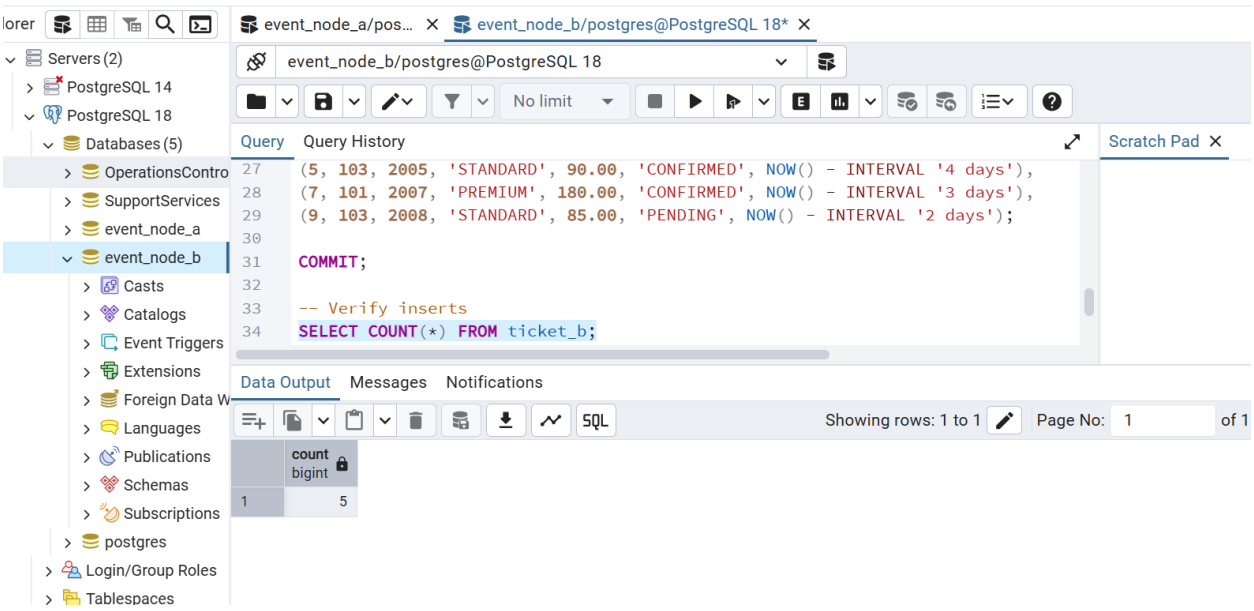
(103, 4, 1004, 'PREMIUM', 120.00, 'CONFIRMED', NOW());

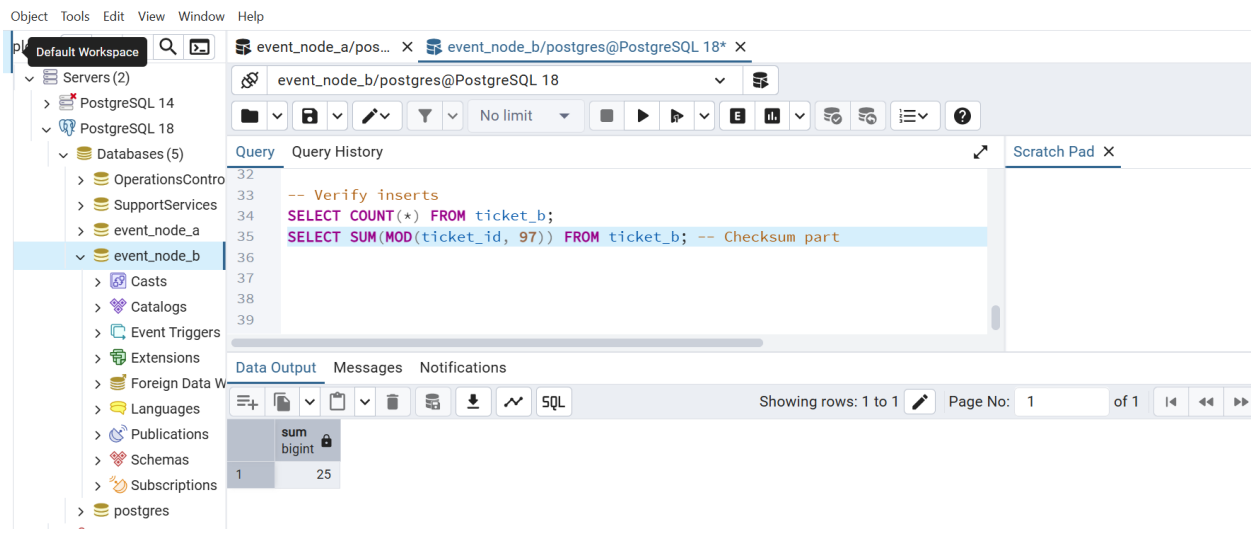
Verification Query (Unified View)



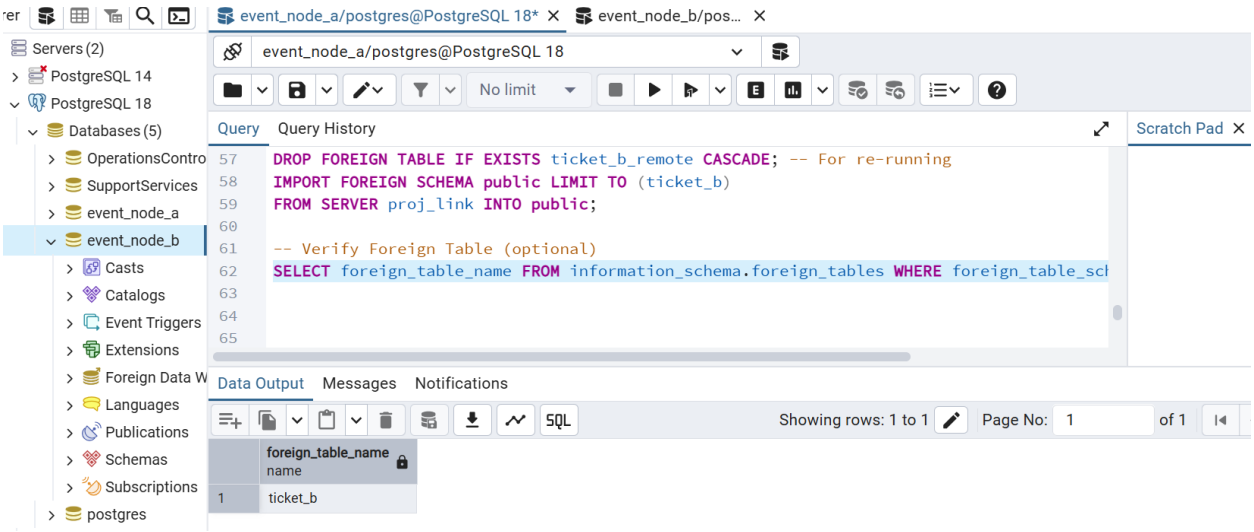
	ticket_id integer	event_id integer	customer_id integer	ticket_type character varying (50)	price numeric (10,2)	status character varying (20)	purchase_date timestamp without time zone
1	2	101	2001	VIP	150.00	CONFIRMED	2025-10-23 21:42:26.192925
2	4	102	2002	STANDARD	75.00	CONFIRMED	2025-10-24 21:42:26.192925
3	6	101	2003	VIP	150.00	PENDING	2025-10-25 21:42:26.192925
4	8	103	2004	STANDARD	80.00	CONFIRMED	2025-10-26 21:42:26.192925
5	10	102	2001	PREMIUM	120.00	CONFIRMED	2025-10-27 21:42:26.192925
6	99	101	2009	STANDARD	95.00	CONFIRMED	2025-10-29 11:48:58.087606
7	97	103	2013	STANDARD	90.00	CONFIRMED	2025-10-29 11:56:34.258196
8	3	102	2006	VIP	200.00	PENDING	2025-10-23 21:43:43.723242


pgAdmin 4
File Object Tools Edit View Window Help
event_node_a/postgres@PostgreSQL 18*
event_node_b/pos...
event_node_a/postgres@PostgreSQL 18
No limit
Query Query History
Scratch Pad
-- Connect to event_node_a
INSERT INTO ticket_a (ticket_id, event_id, customer_id, ticket_type, price, status, pur
(2, 101, 2001, 'VIP', 150.00, 'CONFIRMED', NOW() - INTERVAL '5 days'),
(4, 102, 2002, 'STANDARD', 75.00, 'CONFIRMED', NOW() - INTERVAL '4 days'),
(6, 101, 2003, 'VIP', 150.00, 'PENDING', NOW() - INTERVAL '3 days'),
(8, 103, 2004, 'STANDARD', 80.00, 'CONFIRMED', NOW() - INTERVAL '2 days'),
(10, 102, 2001, 'PREMIUM', 120.00, 'CONFIRMED', NOW() - INTERVAL '1 day');
COMMIT;
Data Output Messages Notifications
Showing rows: 1 to 1 Page No: 1 of 1
count bigint
1 5
Servers (2)
PostgreSQL 14
PostgreSQL 18
Databases (5)
OperationsContro
SupportServices
event_node_a
event_node_b
Casts
Catalogs
Event Triggers
Extensions
Foreign Data W
Languages
Publications
Schemas
Subscriptions
postgres
Login/Group Roles

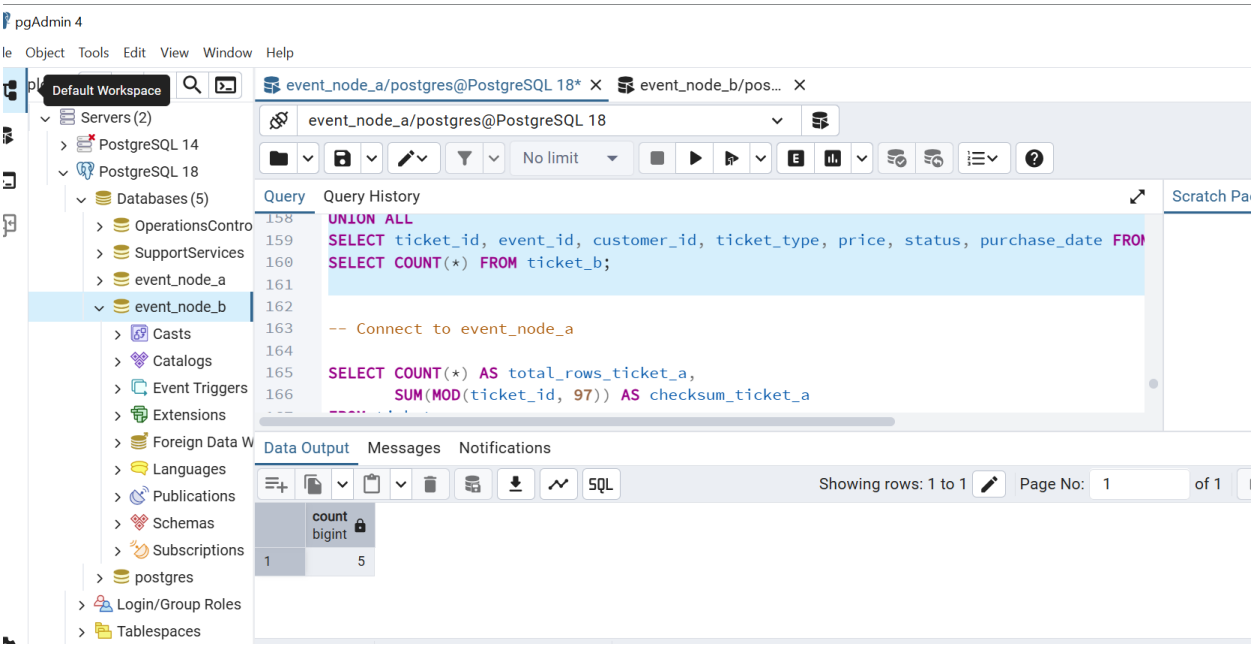

event_node_a/pos...
event_node_b/postgres@PostgreSQL 18*
event_node_b/postgres@PostgreSQL 18
No limit
Query Query History
Scratch Pad
(5, 103, 2005, 'STANDARD', 90.00, 'CONFIRMED', NOW() - INTERVAL '4 days'),
(7, 101, 2007, 'PREMIUM', 180.00, 'CONFIRMED', NOW() - INTERVAL '3 days'),
(9, 103, 2008, 'STANDARD', 85.00, 'PENDING', NOW() - INTERVAL '2 days');
COMMIT;
-- Verify inserts
SELECT COUNT(*) FROM ticket_b;
Data Output Messages Notifications
Showing rows: 1 to 1 Page No: 1 of 1
count bigint
1 5
Servers (2)
PostgreSQL 14
PostgreSQL 18
Databases (5)
OperationsContro
SupportServices
event_node_a
event_node_b
Casts
Catalogs
Event Triggers
Extensions
Foreign Data W
Languages
Publications
Schemas
Subscriptions
postgres
Login/Group Roles
Tablespaces


Object Tools Edit View Window Help
Default Workspace
event_node_a/pos...
event_node_b/postgres@PostgreSQL 18*
event_node_b/postgres@PostgreSQL 18
No limit
Query Query History
Scratch Pad
-- Verify inserts
SELECT COUNT(*) FROM ticket_b;
SELECT SUM(MOD(ticket_id, 97)) FROM ticket_b; -- Checksum part
Data Output Messages Notifications
Showing rows: 1 to 1 Page No: 1 of 1
sum bigint
1 25
Servers (2)
PostgreSQL 14
PostgreSQL 18
Databases (5)
OperationsContro
SupportServices
event_node_a
event_node_b
Casts
Catalogs
Event Triggers
Extensions
Foreign Data W
Languages
Publications
Schemas
Subscriptions
postgres

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

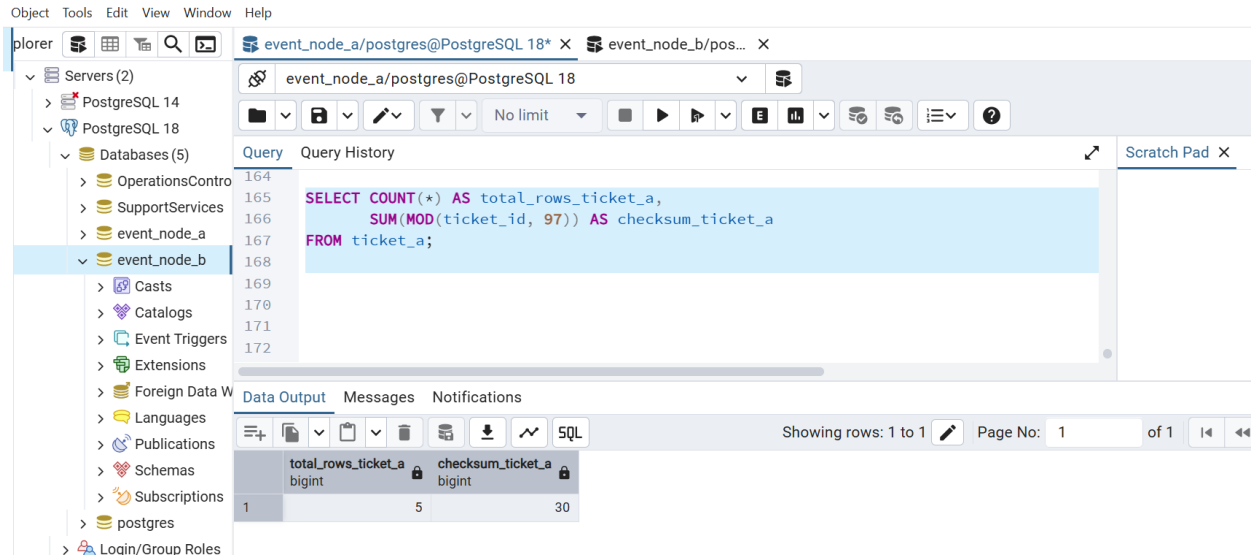


3.2. Connect to event_node_a and create ticket_all view:

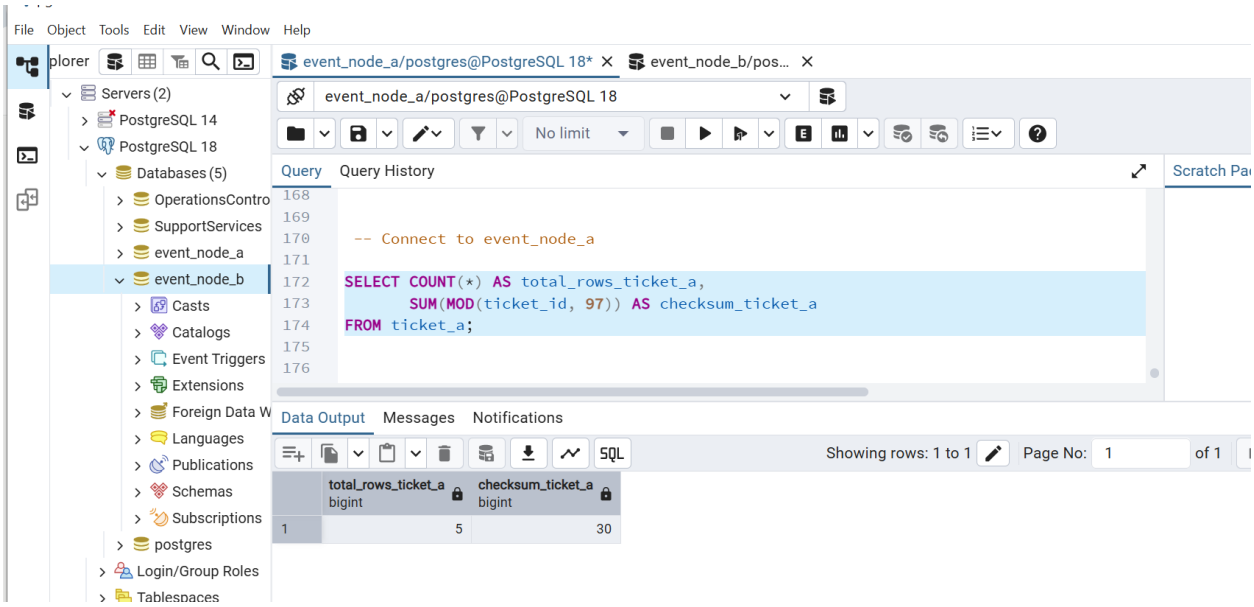


4: Validate with COUNT(*) and a checksum on a key column (e.g., SUM(MOD(primary_key,97)))): results must match fragments vs ticket_all.

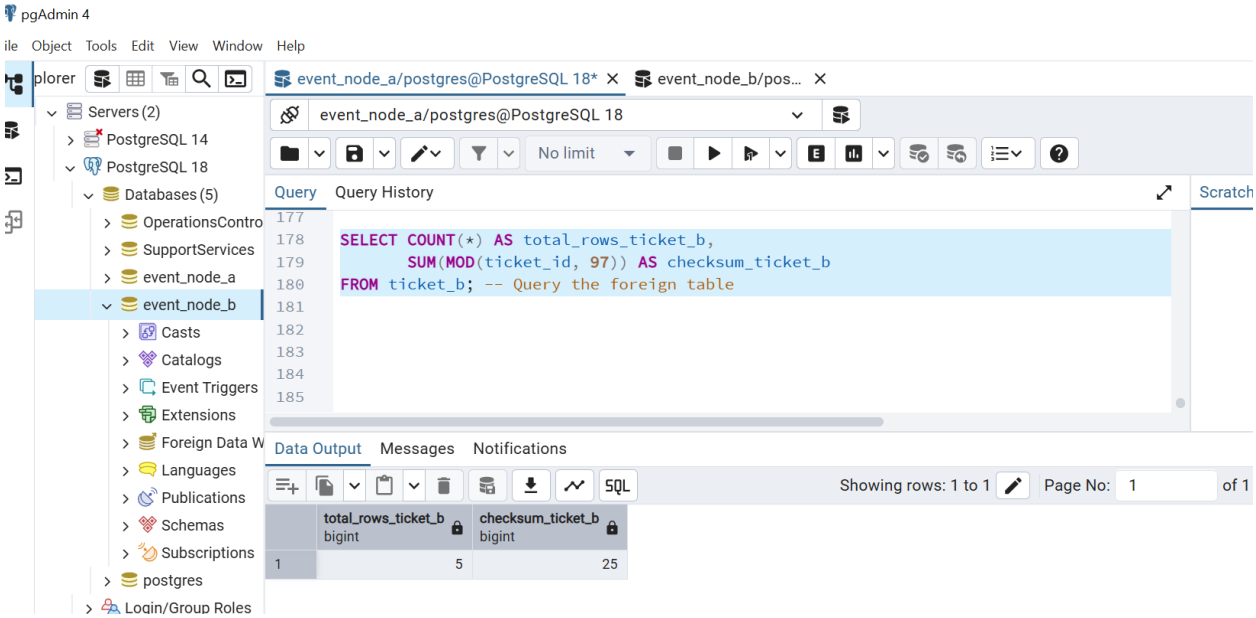
4.1. Connect to event_node_a and calculate COUNT(*) and checksum for ticket_all:



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

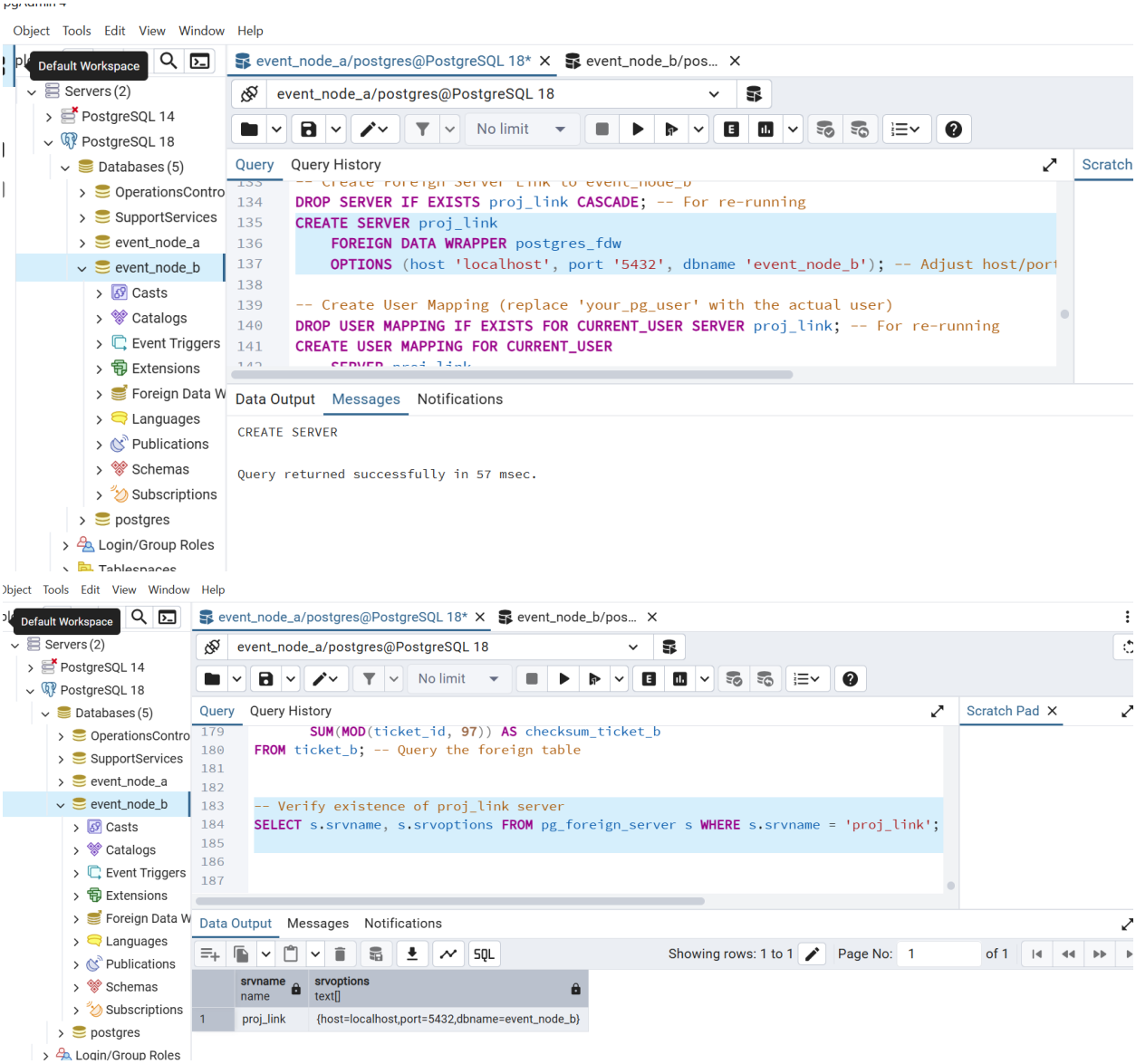


4.3. Connect to event_node_a and calculate COUNT(*) and checksum for remote ticket_b via FDW:

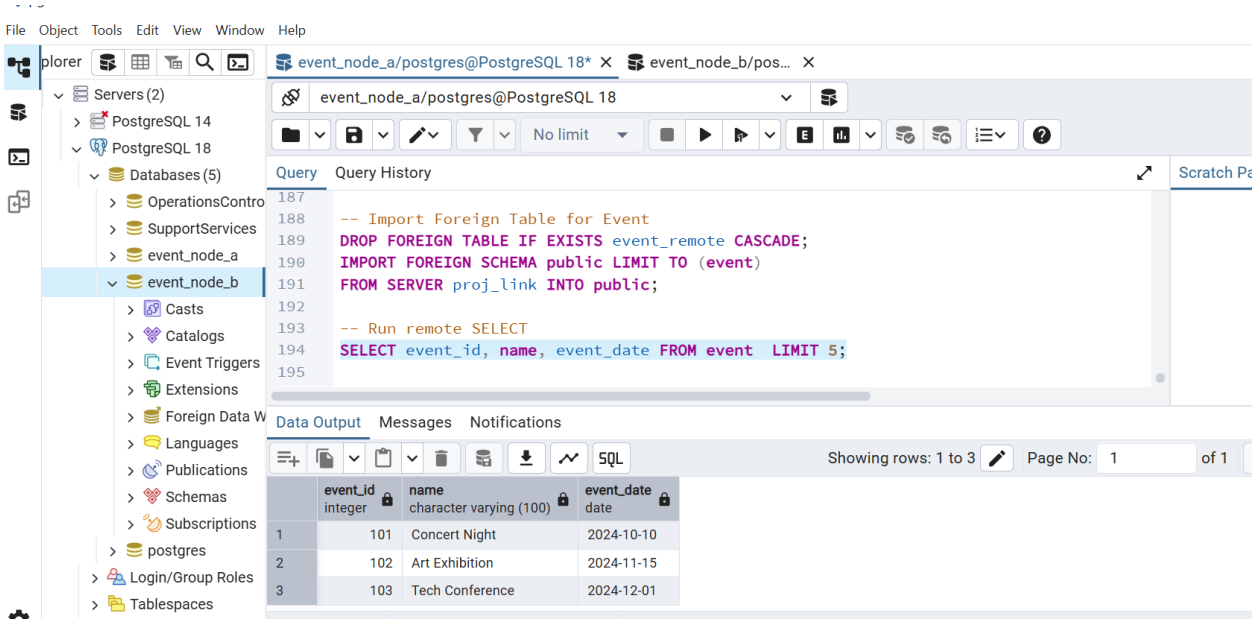


A2: Database Link & Cross-Node Join (3–10 rows result)

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



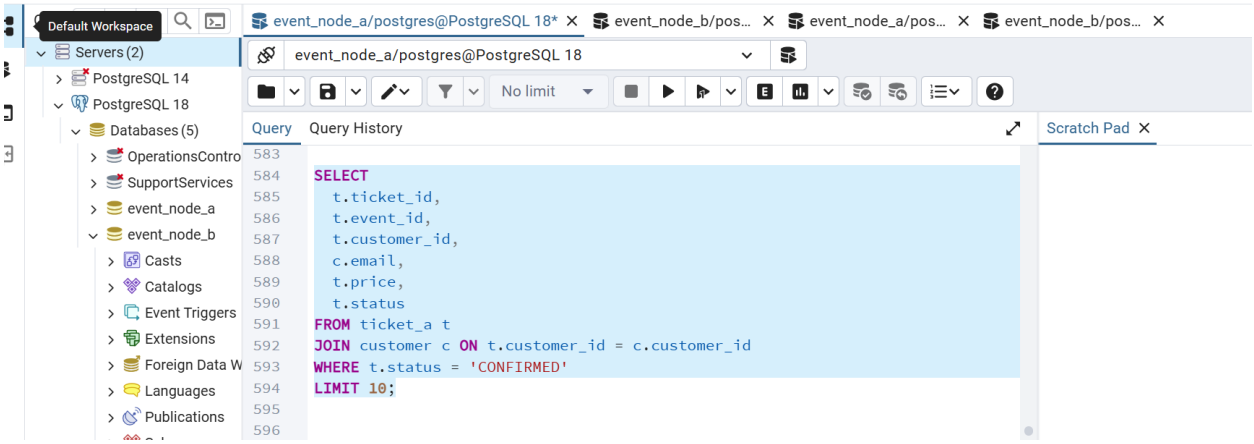
Step 2: Run remote SELECT on Event@proj_link showing up to 5 sample rows.



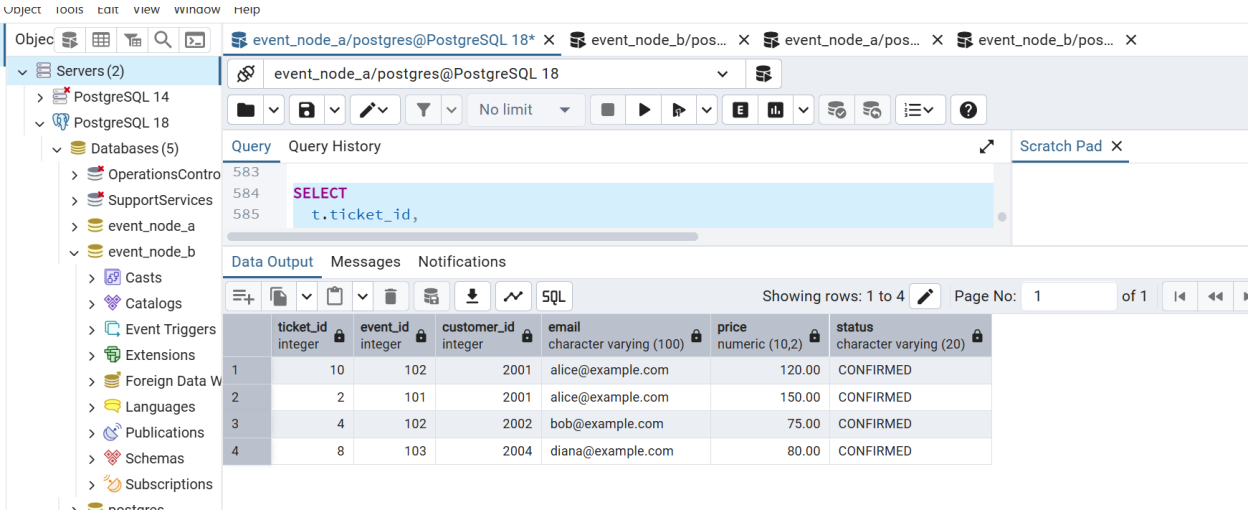
Step 3: Demonstrate a Distributed Join

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

Code:



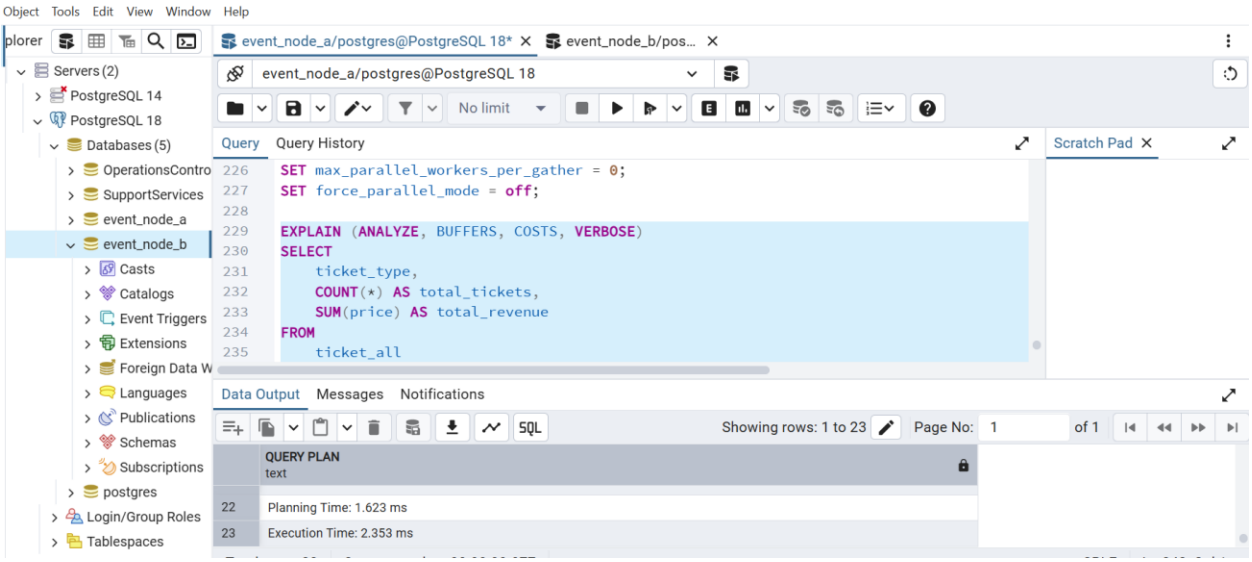
Output



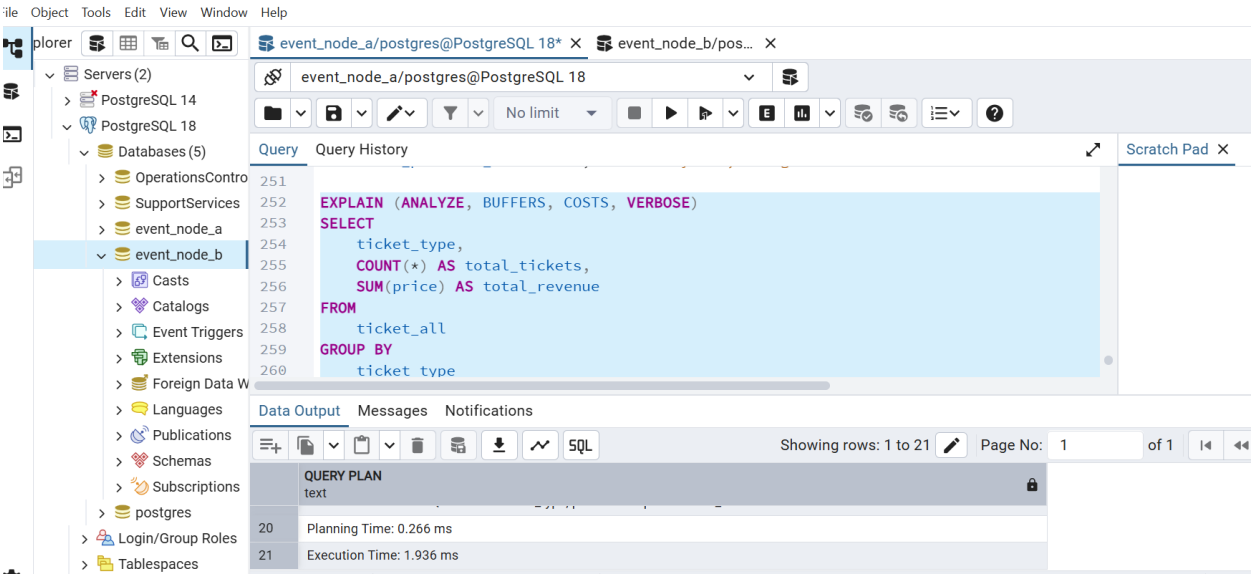
(Note: The join condition e.event_id = (SELECT event_id FROM tickets WHERE ticket_id = p.ticket_id) is a simplified way for this example since tickets is a foreign table. In a more complex scenario, you might explicitly import tickets and join directly, or use a view.)

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.



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

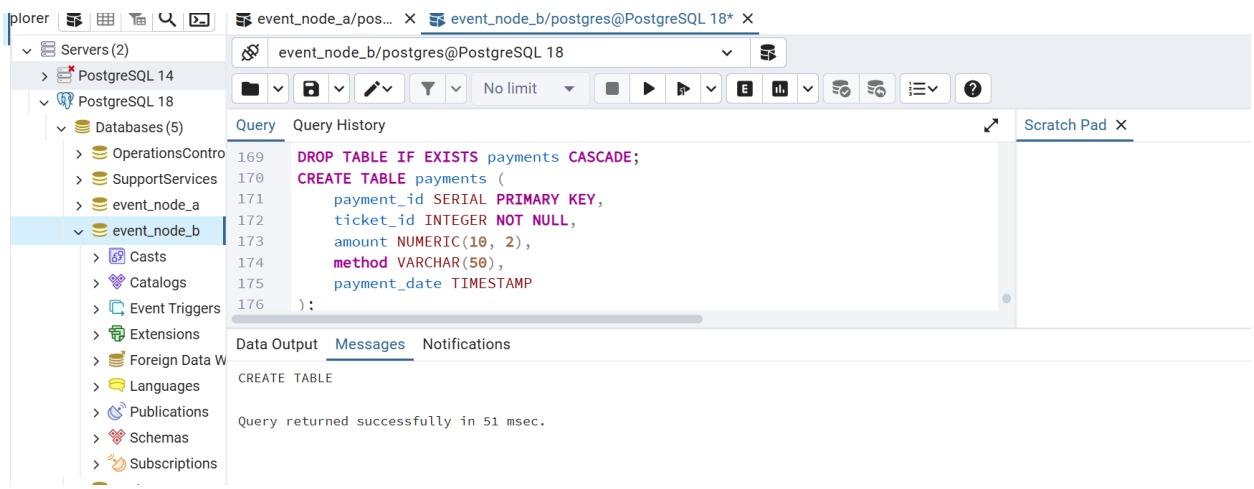


3: Capture execution plans with DBMS_XPLAN and show AUTOTRACE statistics; timings may be similar due to small data.

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.

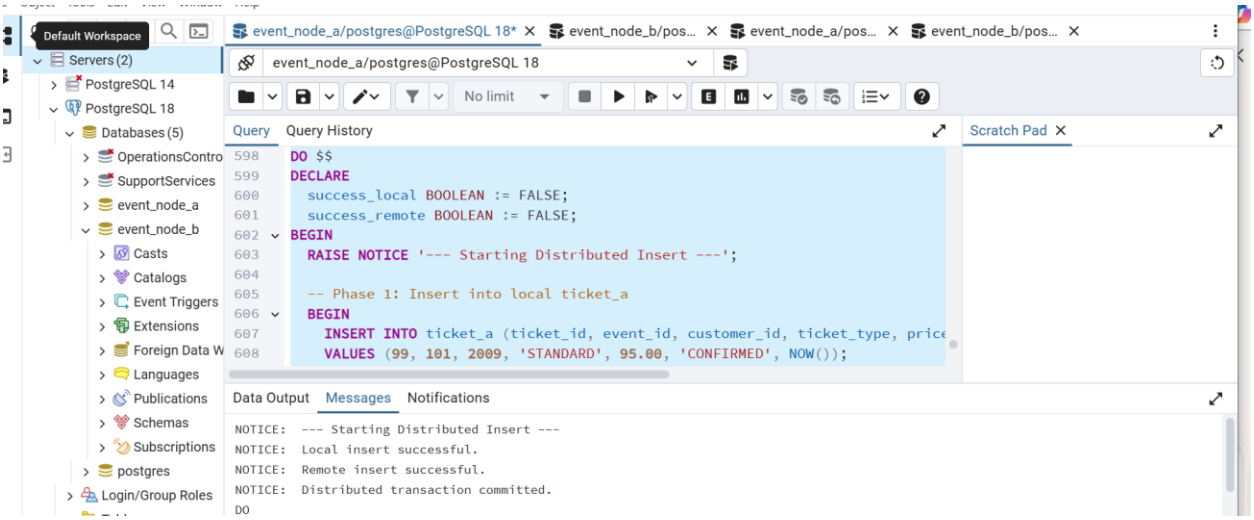
4: Produce a 2-row comparison table (serial vs parallel) with plan notes.



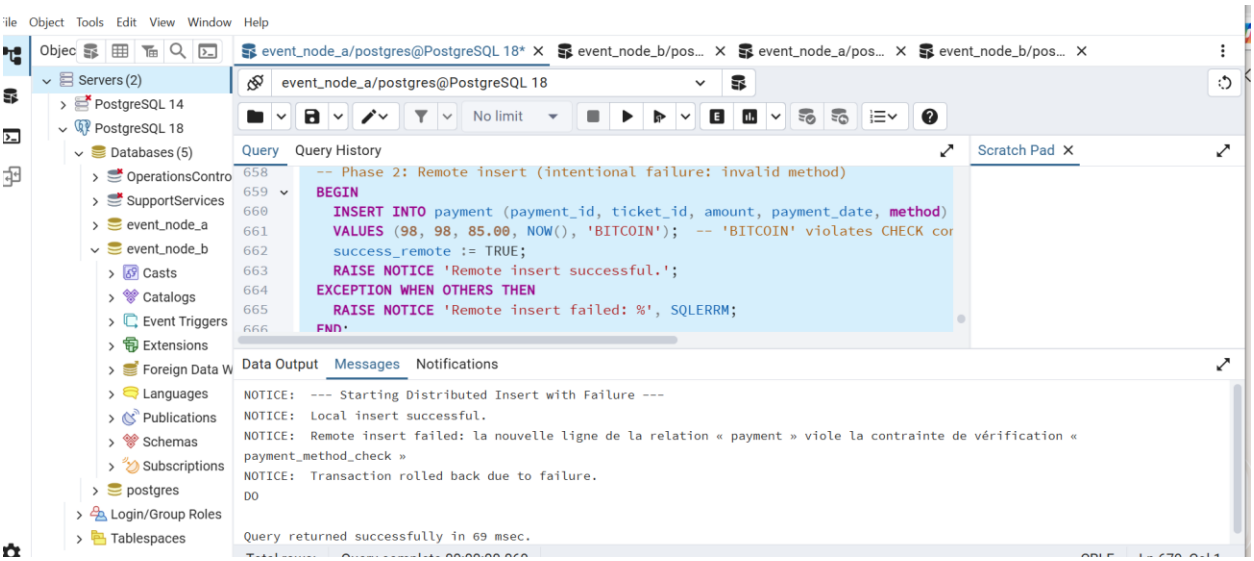
A4 :Two-Phase Commit & Recovery (2 rows)

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.

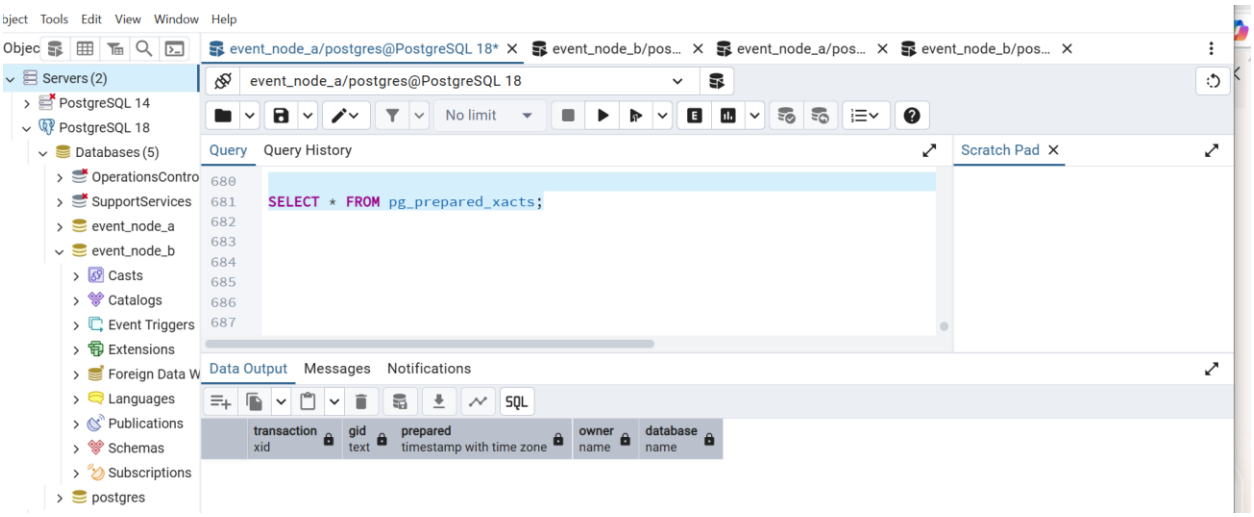
This will be a "clean" run with no failure.

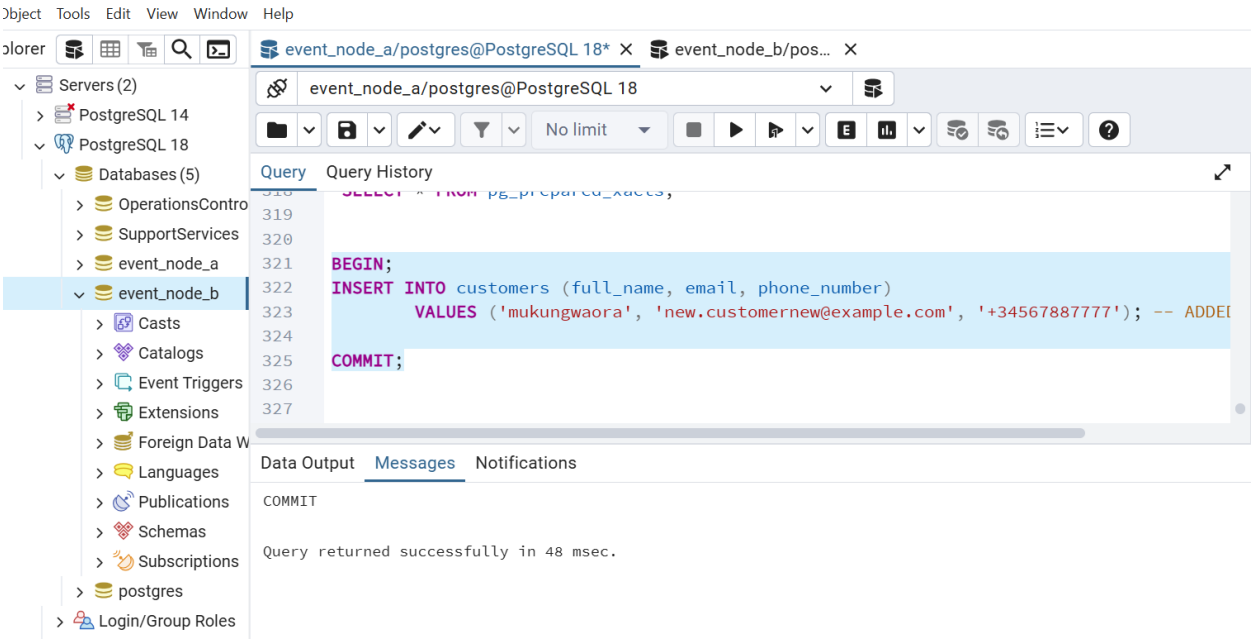


Step 2: Induce a failure in a second run (e.g., disable the link between inserts) to create an in-doubt transaction; ensure any extra test rows are ROLLED BACK to keep within the ≤10 committed row budget.

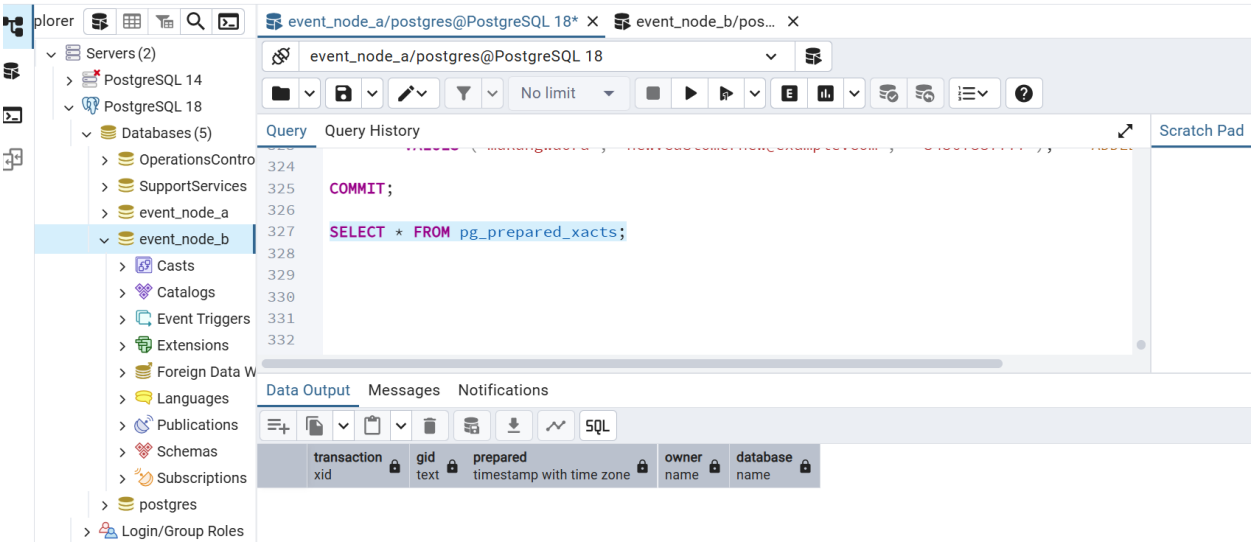


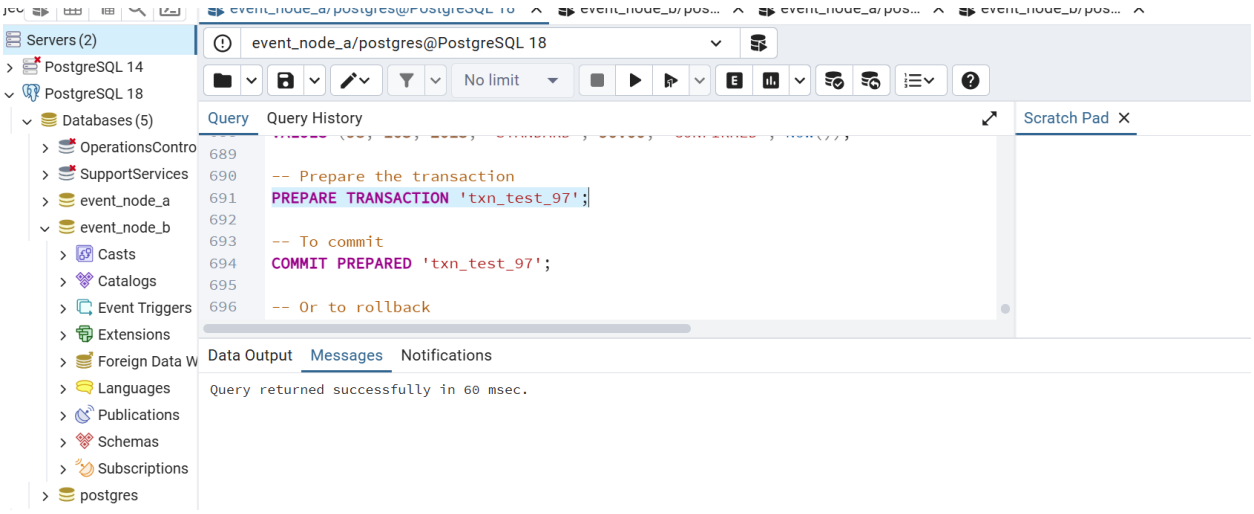
3. Query DBA_2PC_PENDING; then issue COMMIT FORCE or ROLLBACK FORCE; re-verify consistency on both nodes.

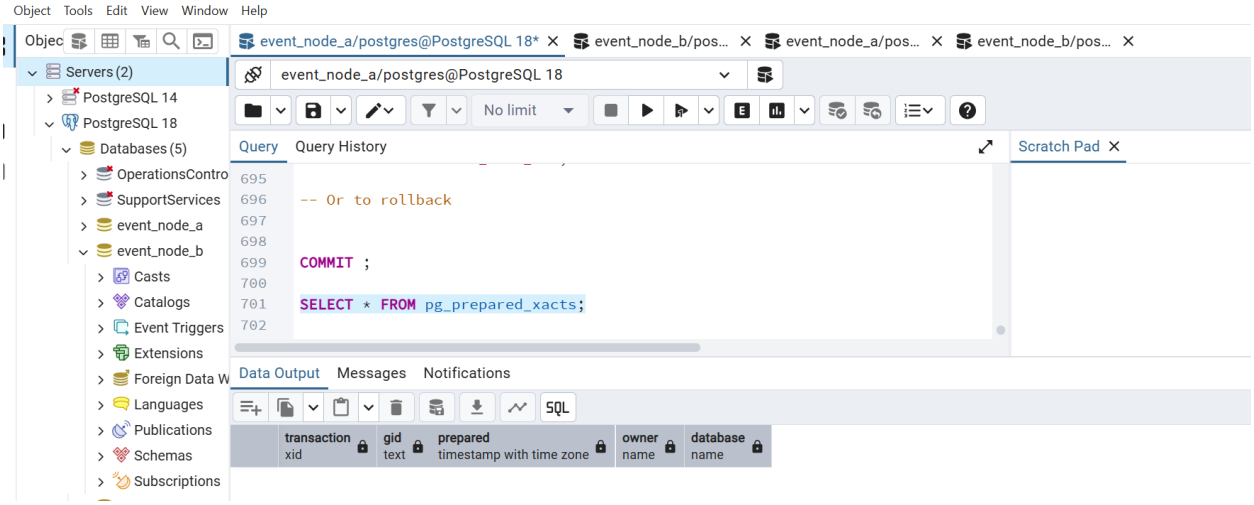




4: Repeat a clean run to show there are no pending transactions.

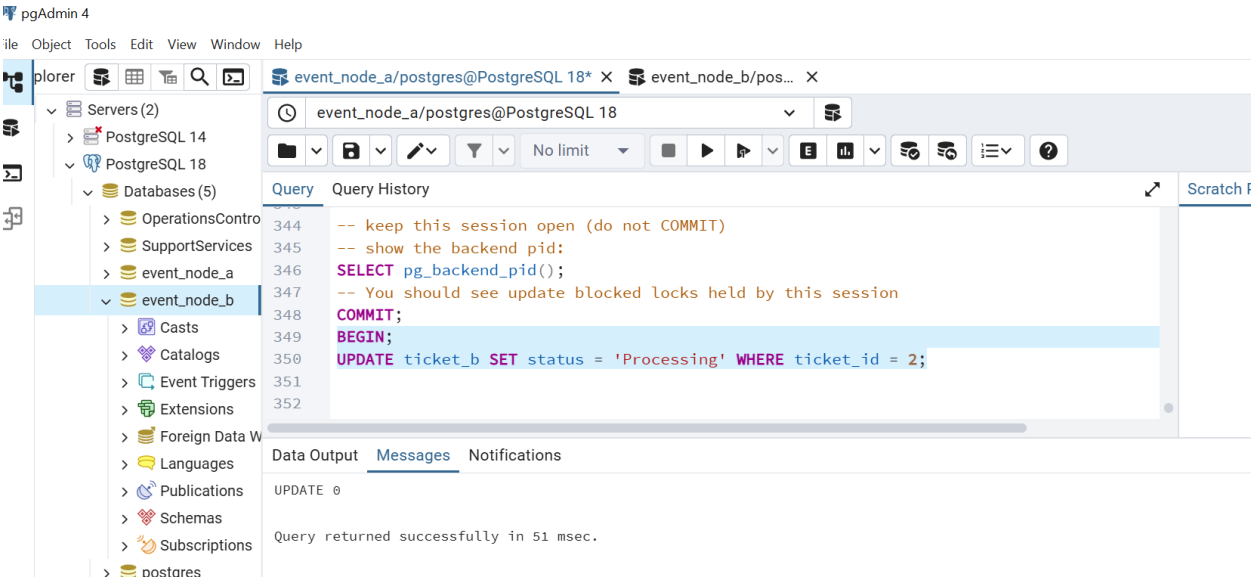




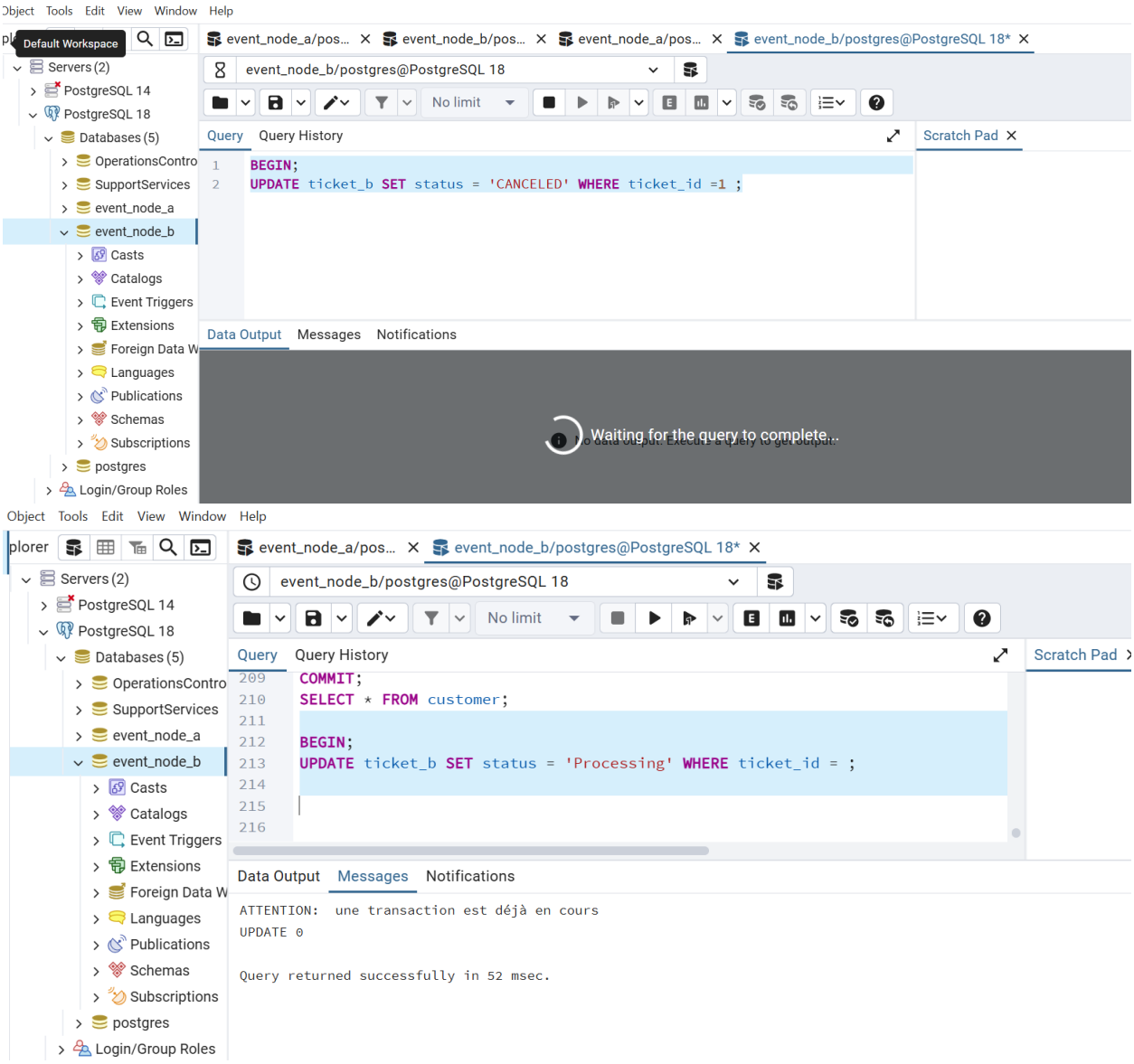


A5 :Distributed Lock Conflict & Diagnosis (no extra rows)

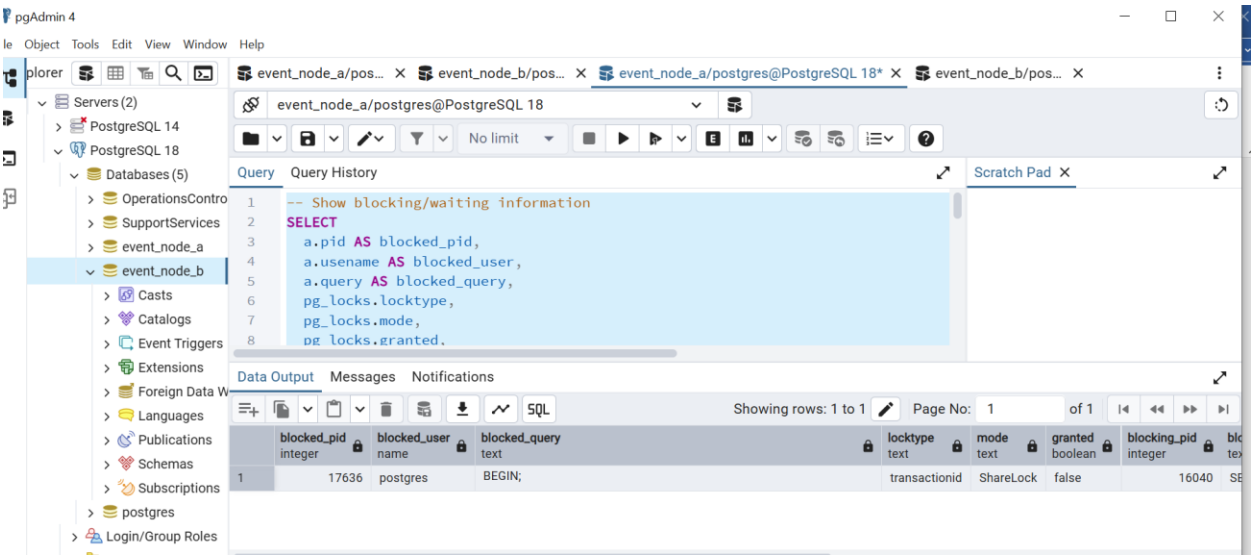
1.Open Session 1 on Node_A: UPDATE a single row in Ticket or Payment and keep the transaction open.



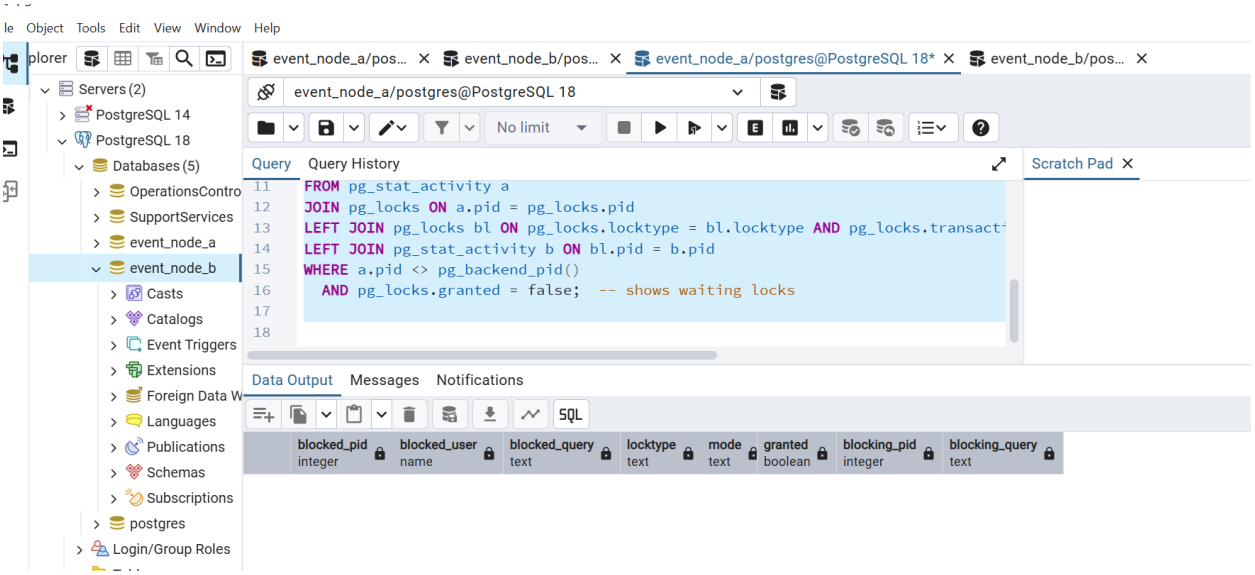
2. Open Session 2 from Node_B via Ticket@proj_link or Payment@proj_link to UPDATE the same logical row.



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



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

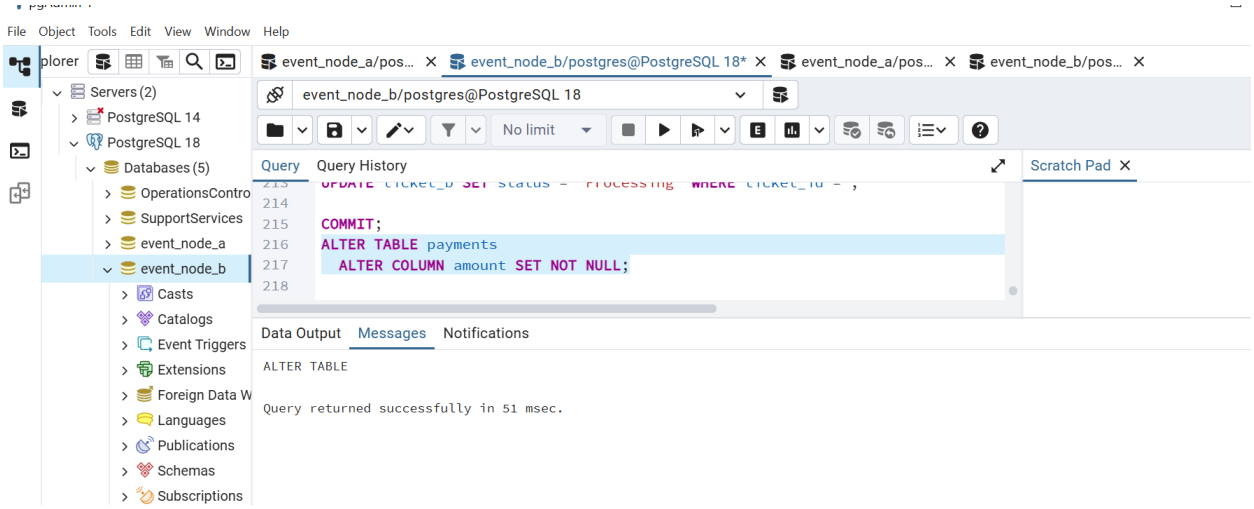


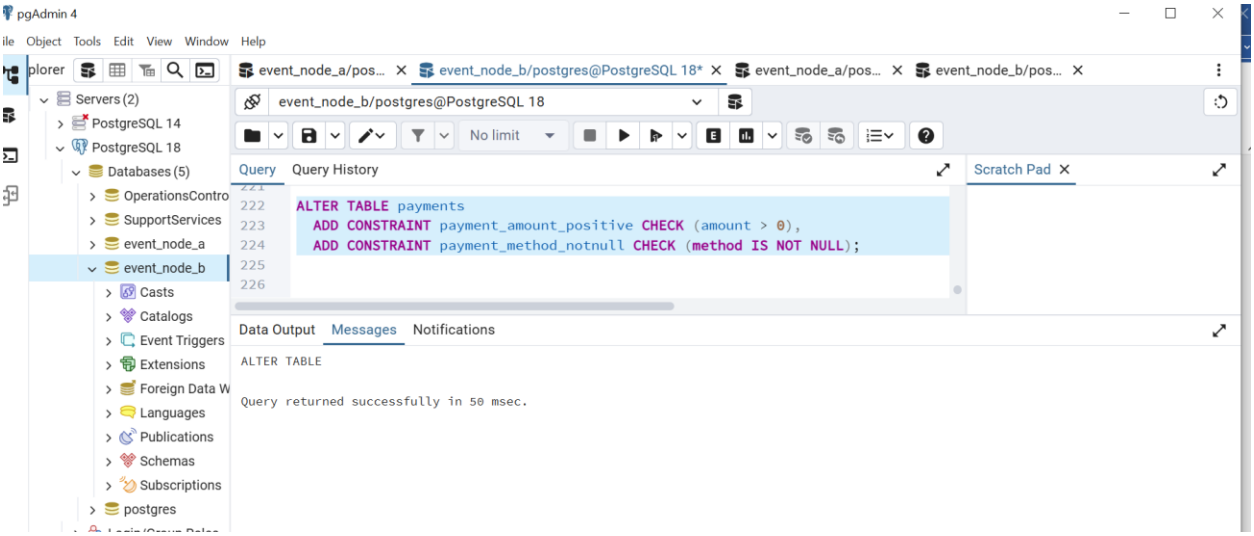
SECTION B

B6 : Declarative Rules Hardening (≤10 committed rows)

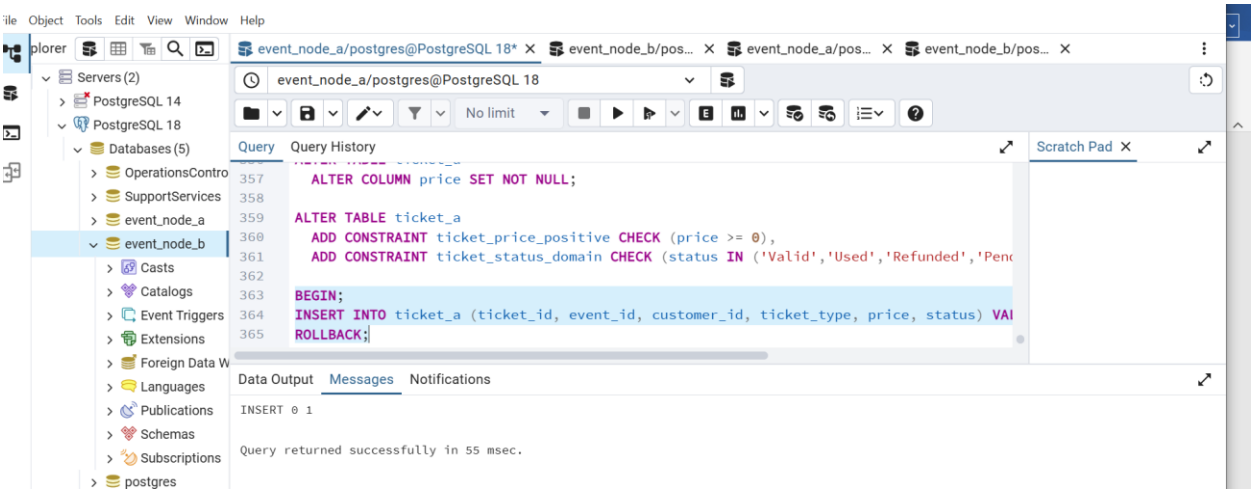
Goal: Add NOT NULL, CHECK constraints on Ticket and Payment; prepare 2 failing and 2 passing inserts (wrap failing ones in blocks and rollback).

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).

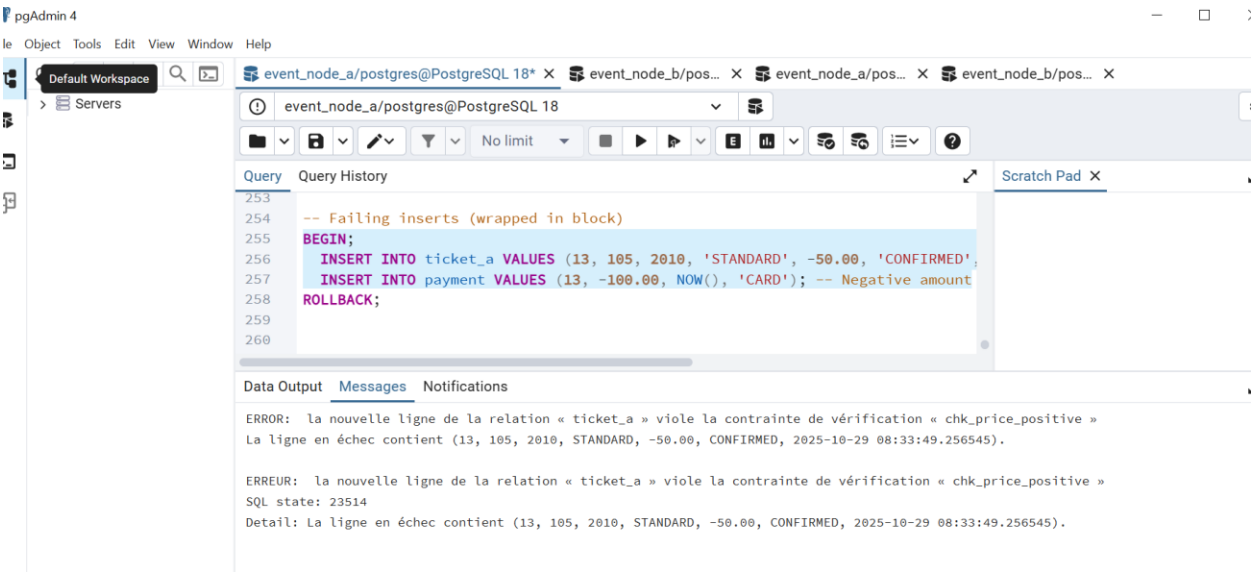




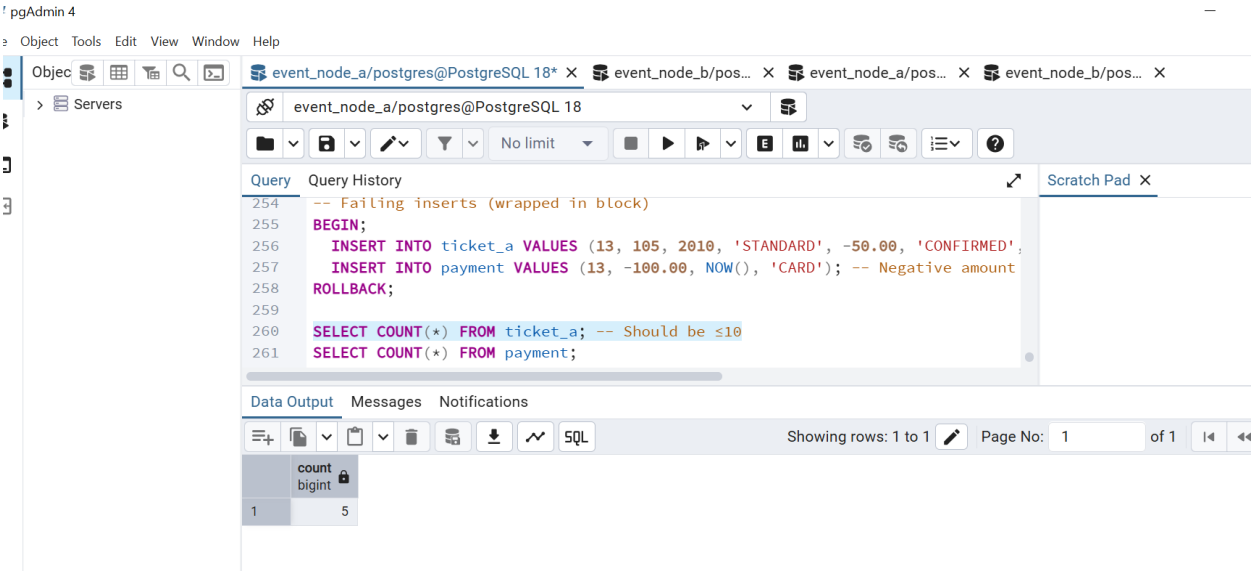
Step 2: Test Inserts

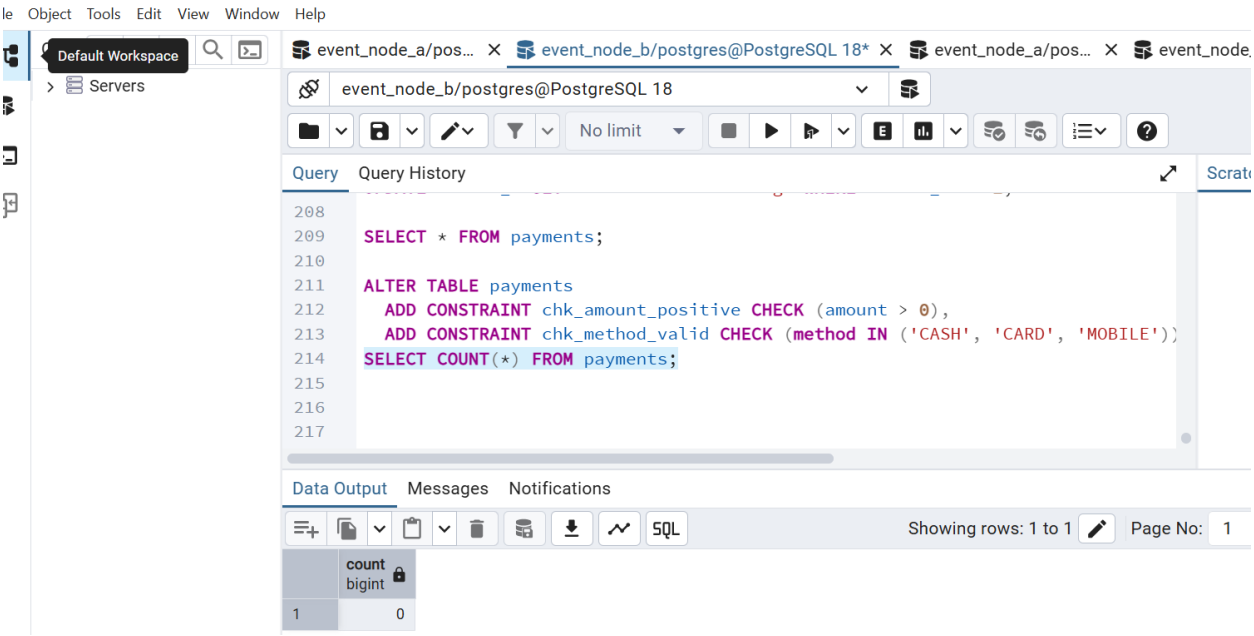


2.Tests (2 failing, 2 passing per table). Wrap failing in transactions and ROLLBACK to keep committed rows small.



3. Show clean error handling for failing cases.





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 (

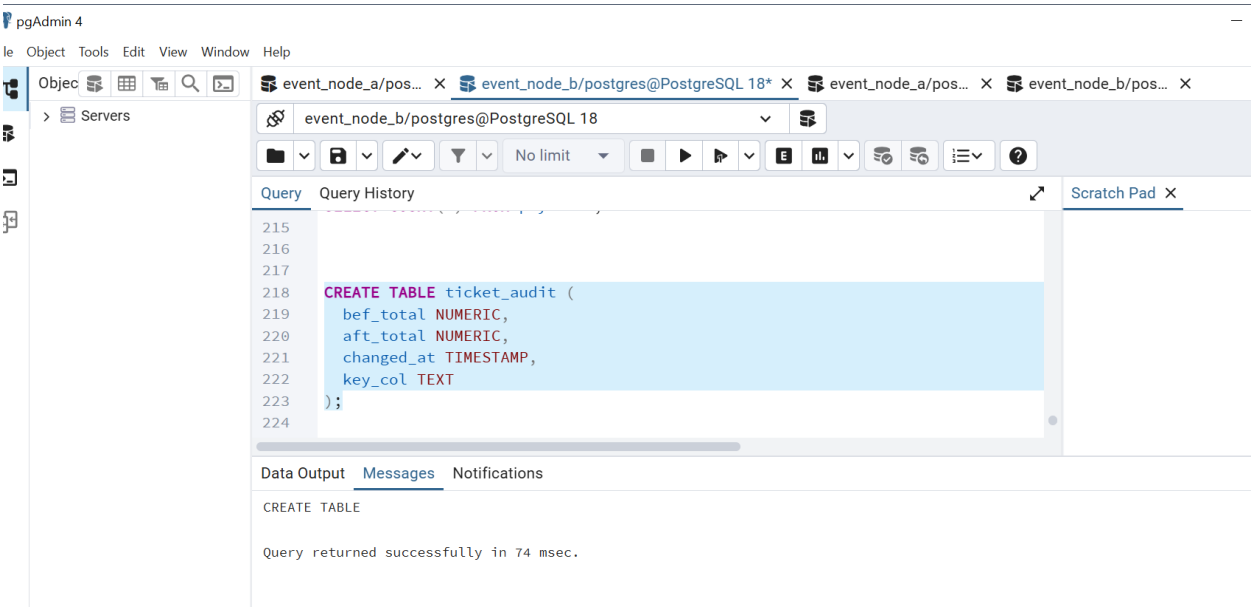
bef_total NUMERIC,

aft_total NUMERIC,

changed_at TIMESTAMP,

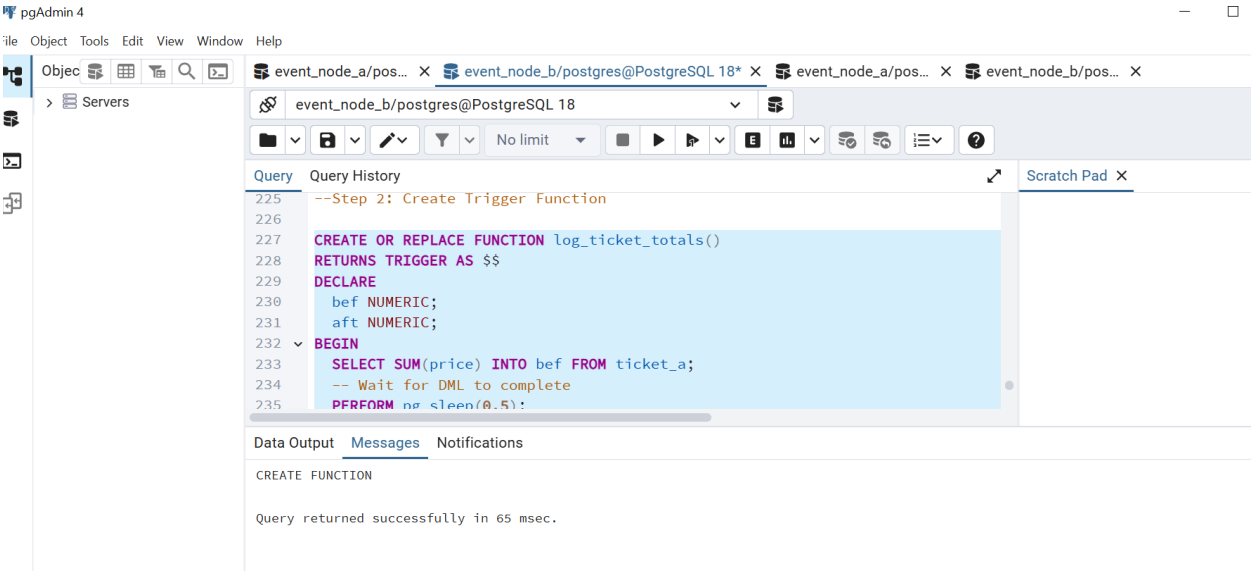
key_col TEXT

);

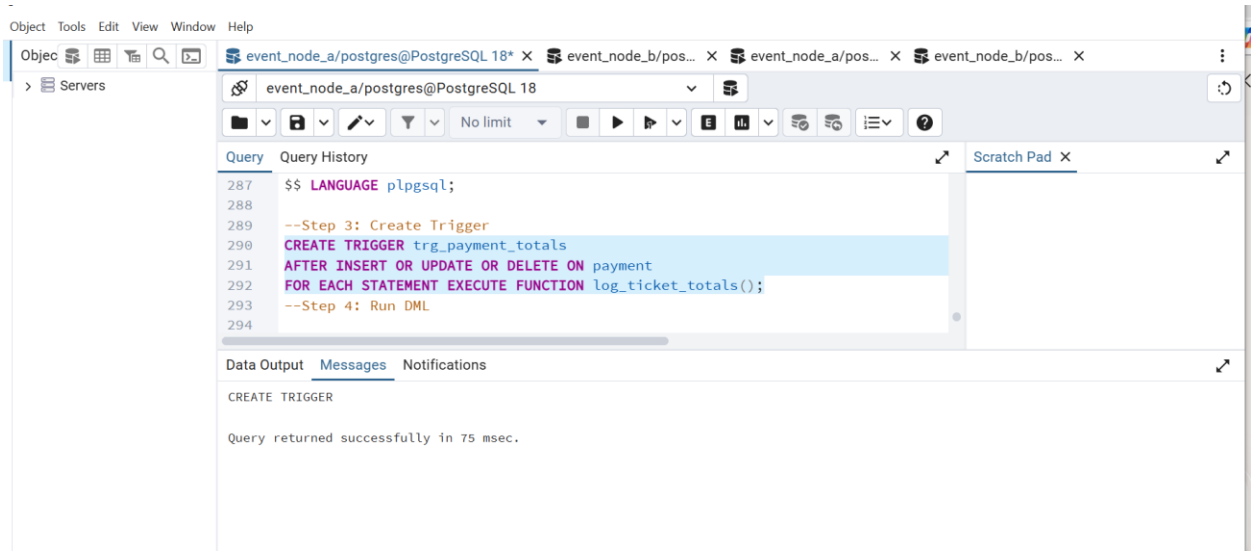


2. Implement a statement-level AFTER INSERT/UPDATE/DELETE trigger on Payment that recomputes denormalized totals in Ticket once per statement.

Step 2: Create Trigger Function



Step 3: Create Trigger



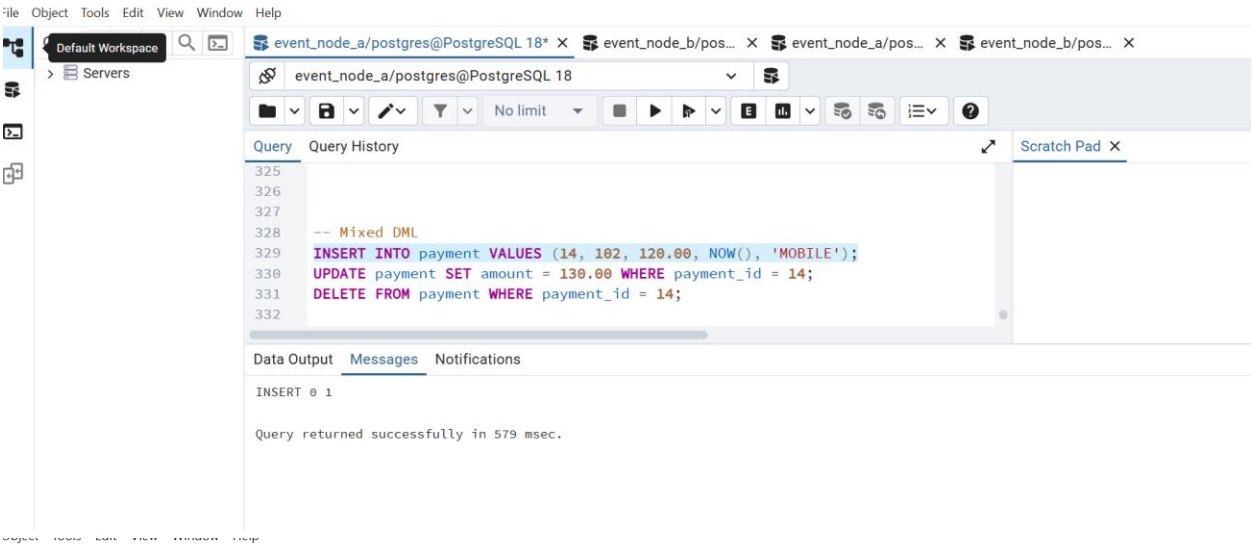
3. Execute a small mixed DML script on CHILD affecting at most 4 rows in total; ensure net committed rows across the project remain ≤10.

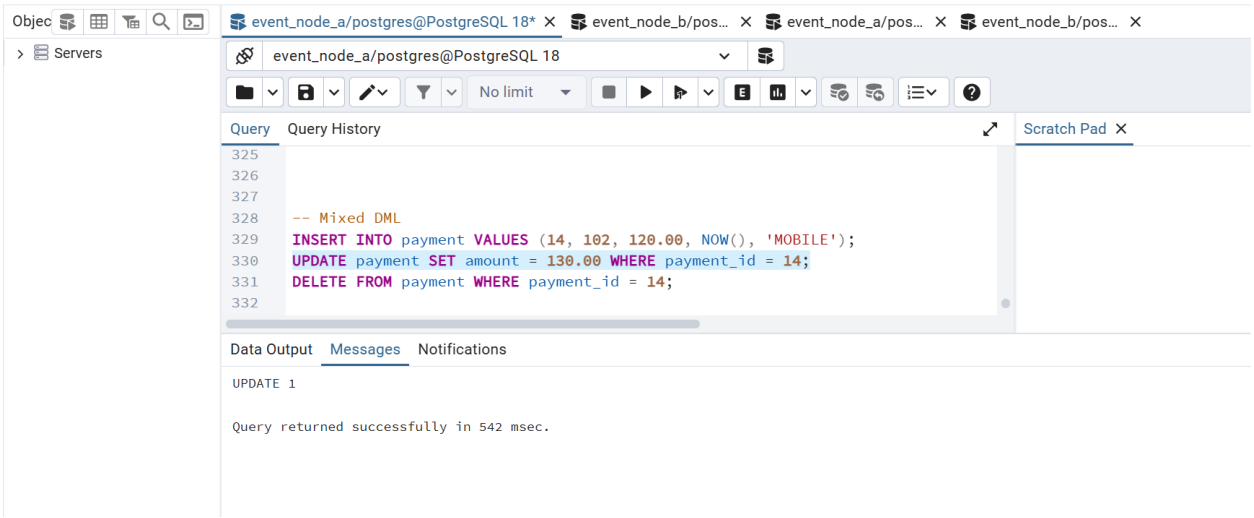
Step 4: Run DML

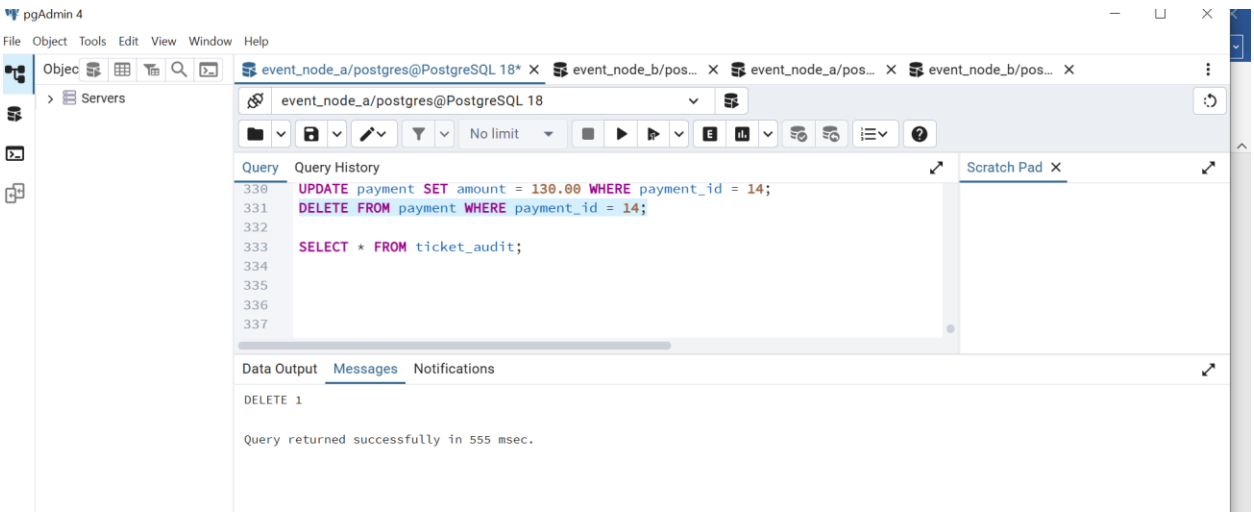
INSERT INTO payment VALUES (14, 120.00, NOW(), 'MOBILE');

UPDATE payment SET amount = 130.00 WHERE payment_id = 14;

DELETE FROM payment WHERE payment_id = 14;

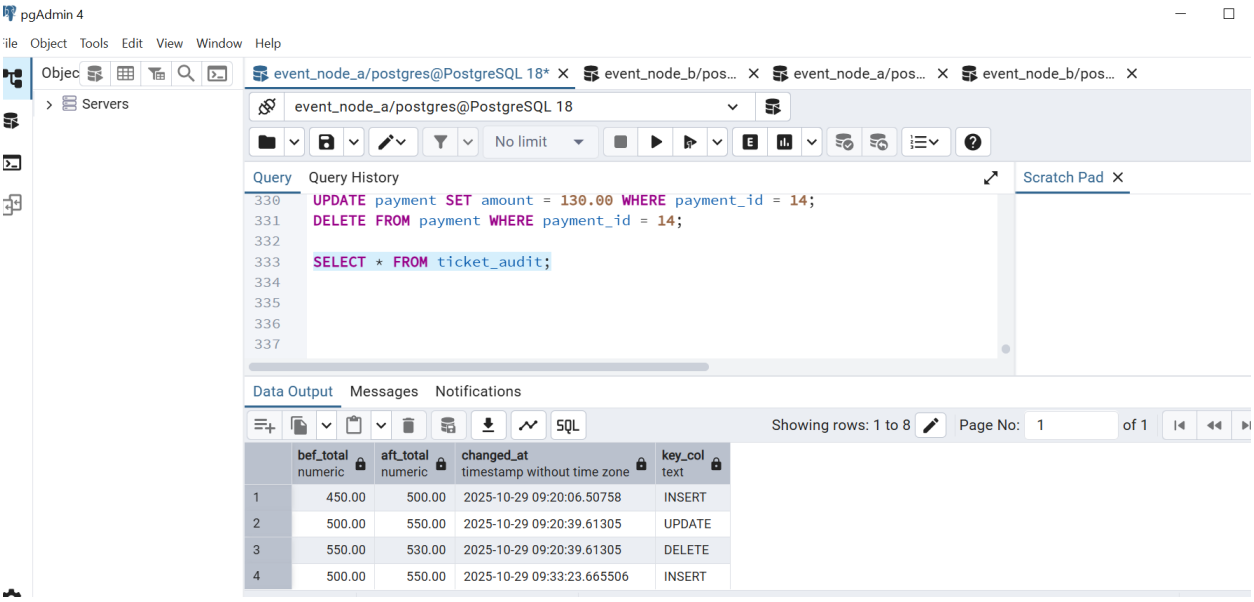

File Object Tools Edit View Window Help
Default Workspace
Servers
event_node_a/postgres@PostgreSQL 18*
event_node_b/pos...
event_node_a/pos...
event_node_b/pos...
event_node_a/postgres@PostgreSQL 18
No limit
Query
Query History
Scratch Pad
325
326
327
328 -- Mixed DML
329 INSERT INTO payment VALUES (14, 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.


Servers
event_node_a/postgres@PostgreSQL 18*
event_node_b/pos...
event_node_a/pos...
event_node_b/pos...
event_node_a/postgres@PostgreSQL 18
No limit
Query
Query History
Scratch Pad
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
Servers
event_node_a/postgres@PostgreSQL 18*
event_node_b/pos...
event_node_a/pos...
event_node_b/pos...
event_node_a/postgres@PostgreSQL 18
No limit
Query
Query History
Scratch Pad
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).

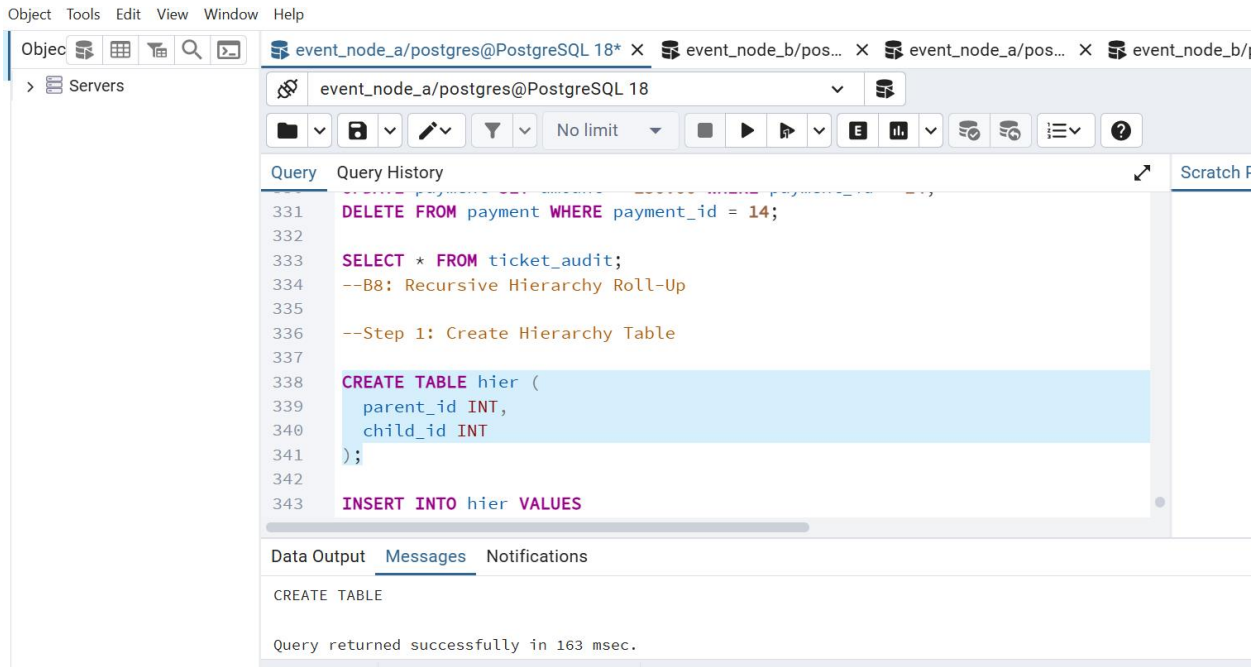
Step 5: Verify Audit



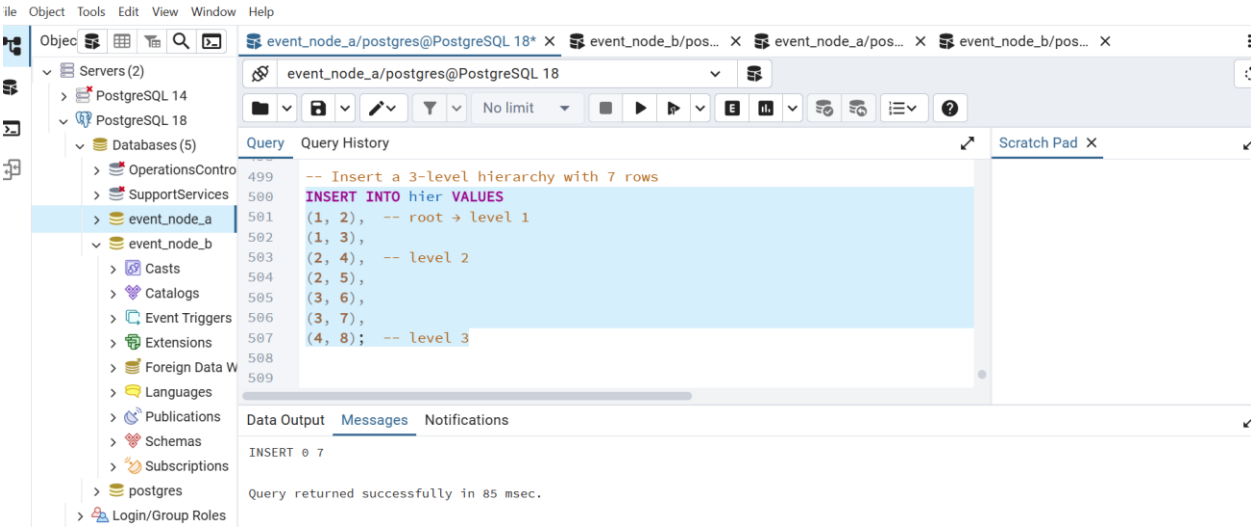
B8: Recursive Hierarchy Roll-Up

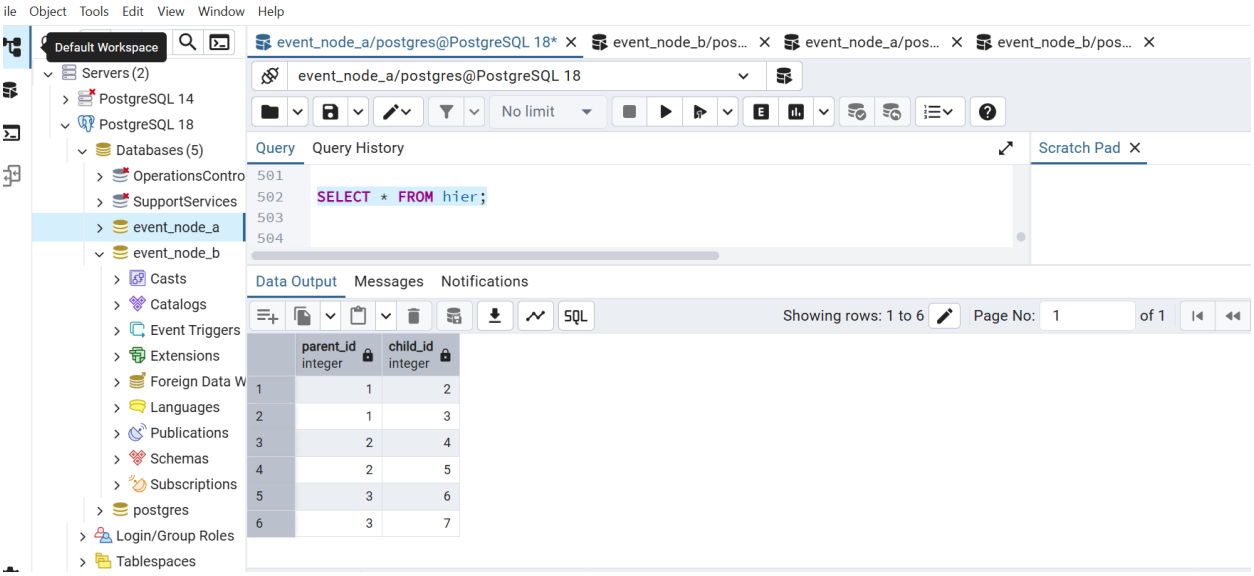
1. Create table HIER(parent_id, child_id) for a natural hierarchy (domain-specific).

Step 1: Create Hierarchy Table



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

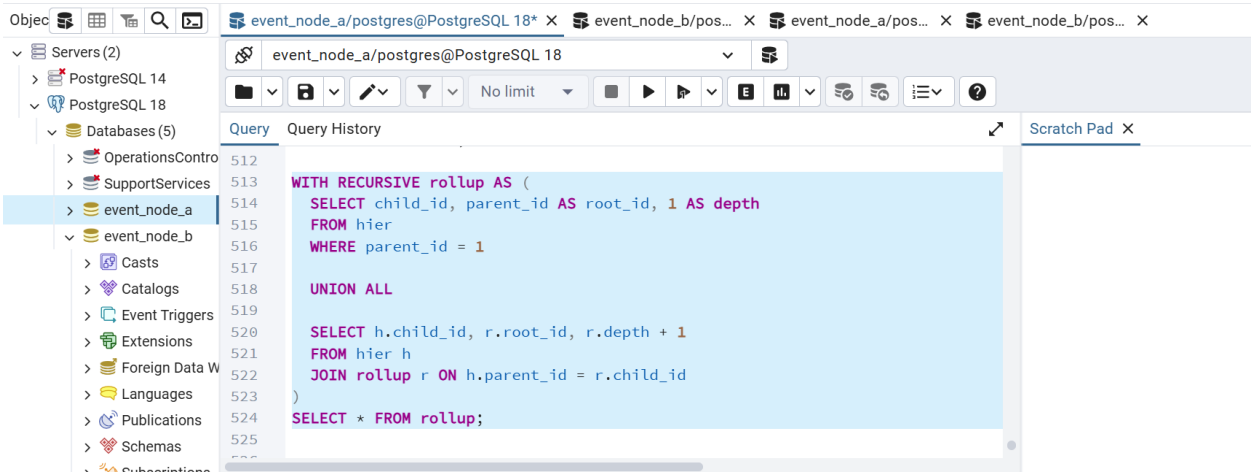




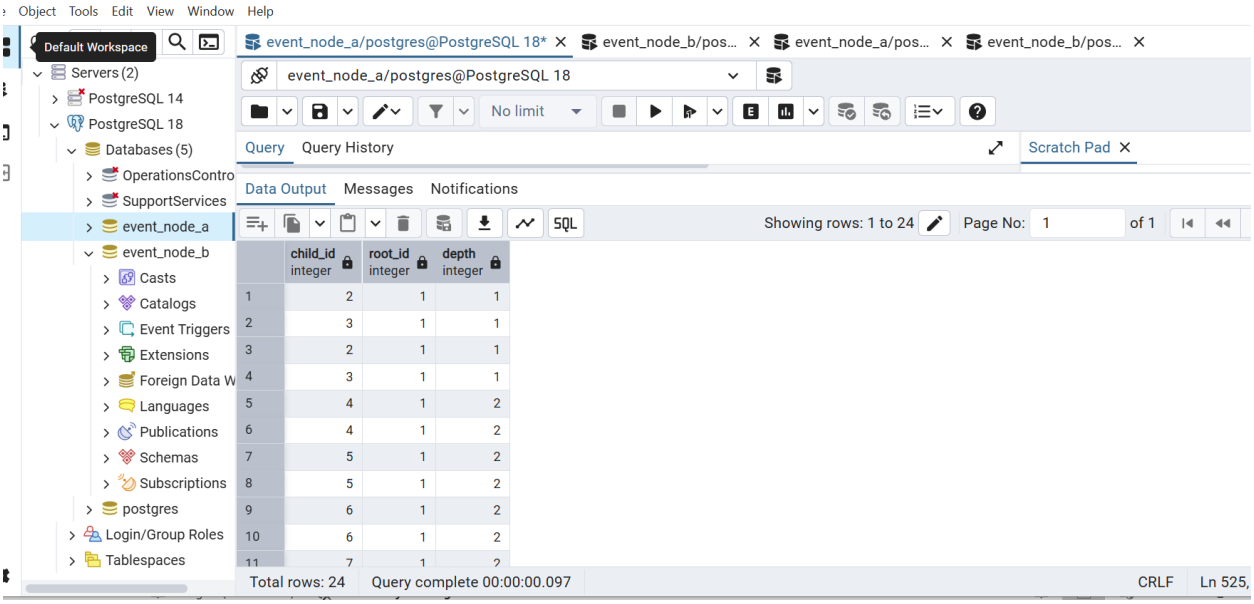
3. Write a recursive WITH query to produce (child_id, root_id, depth) and join to Ticket or its parent to compute rollups; return 6–10 rows total.

Step 2: Recursive Query

code

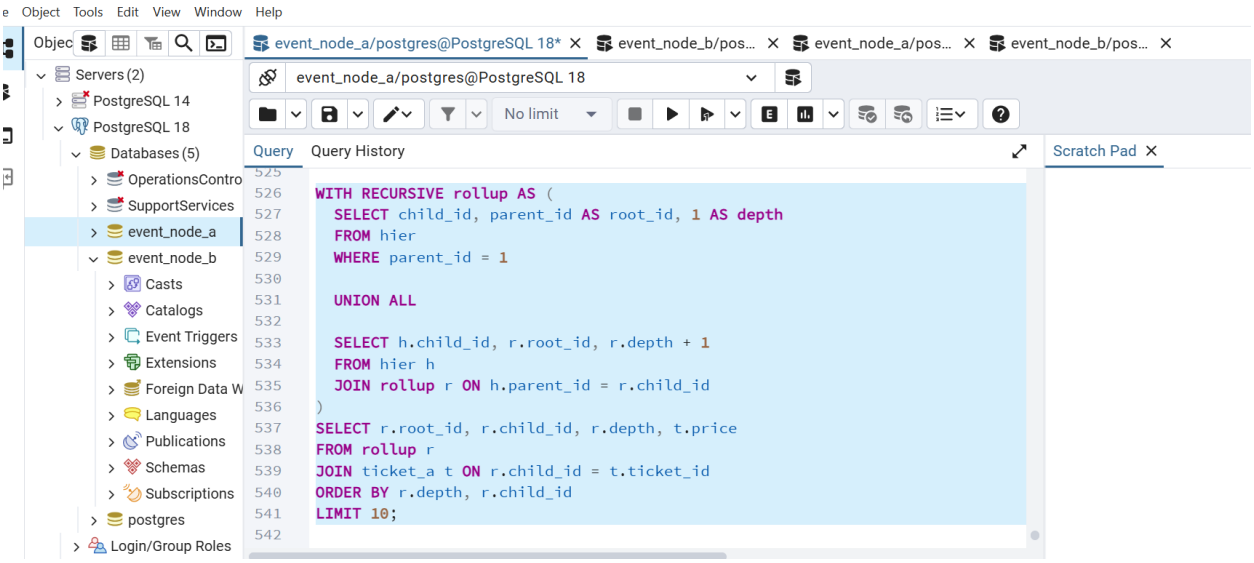


Ouput:

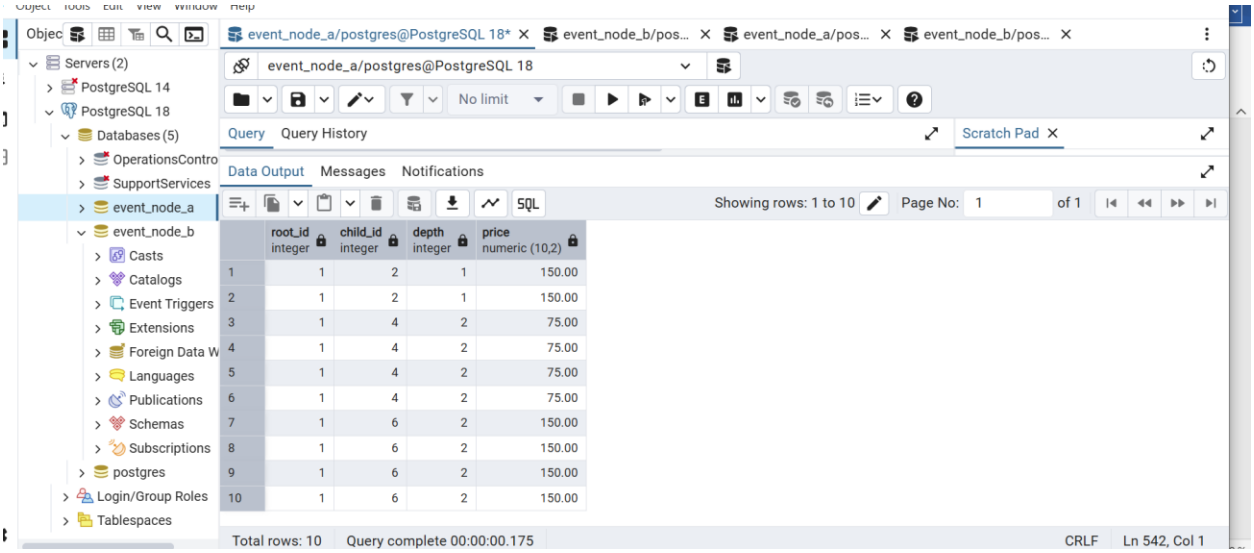


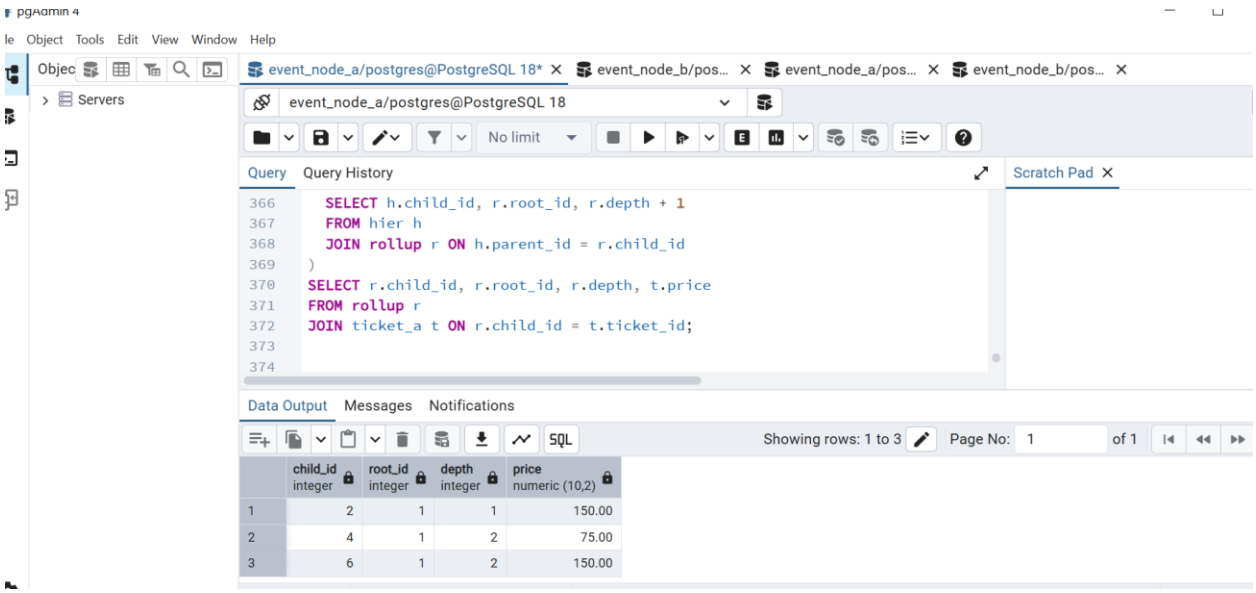
Step 3: Join to Ticket

Code:


Object Tools Edit View Window Help
event_node_a/postgres@PostgreSQL 18*
event_node_b/pos...
event_node_a/pos...
event_node_b/pos...
Servers (2)
PostgreSQL 14
PostgreSQL 18
Databases (5)
OperationsContro
SupportServices
event_node_a
event_node_b
Casts
Catalogs
Event Triggers
Extensions
Foreign Data W
Languages
Publications
Schemas
Subscriptions
postgres
Login/Group Roles
No limit
Query
Query History
Scratch Pad
525
526 WITH RECURSIVE rollup AS (
527 SELECT child_id, parent_id AS root_id, 1 AS depth
528 FROM hier
529 WHERE parent_id = 1
530
531 UNION ALL
532
533 SELECT h.child_id, r.root_id, r.depth + 1
534 FROM hier h
535 JOIN rollup r ON h.parent_id = r.child_id
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 LIMIT 10;
542

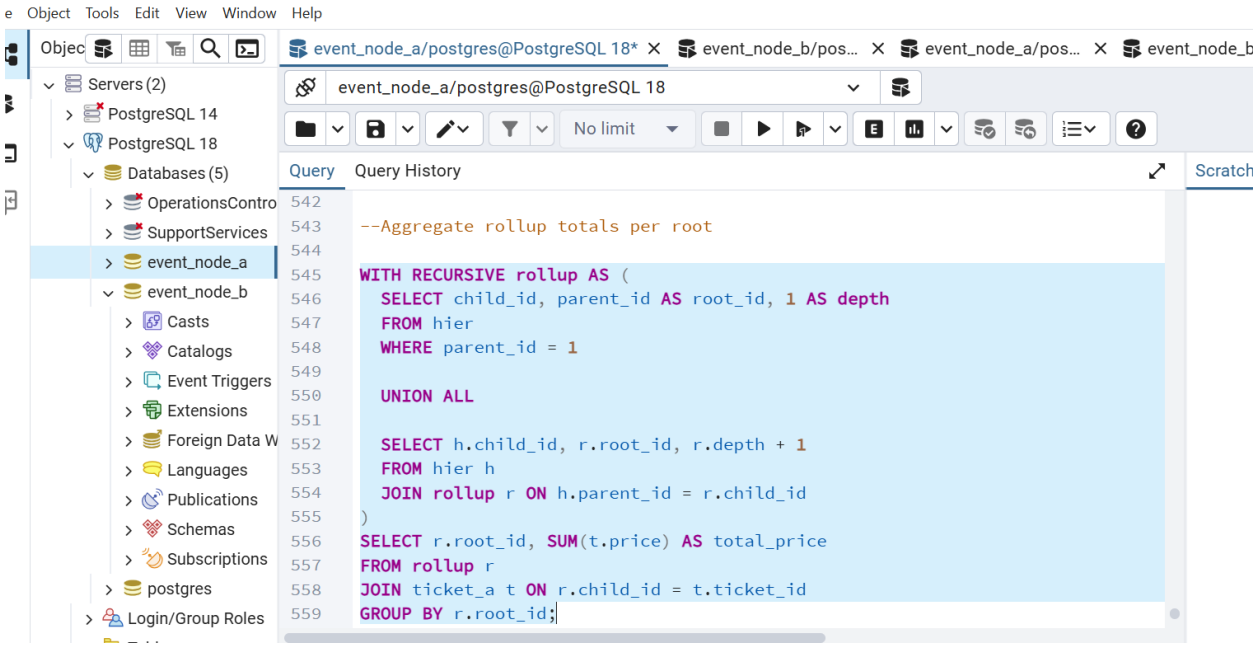
Output:


Data Output
Messages
Notifications
SQL
Showing rows: 1 to 10
Page No: 1
of 1
root_id integer
child_id integer
depth integer
price numeric (10,2)
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
Tablespaces
Total rows: 10
Query complete 00:00:00.175
CRLF
Ln 542, Col 1

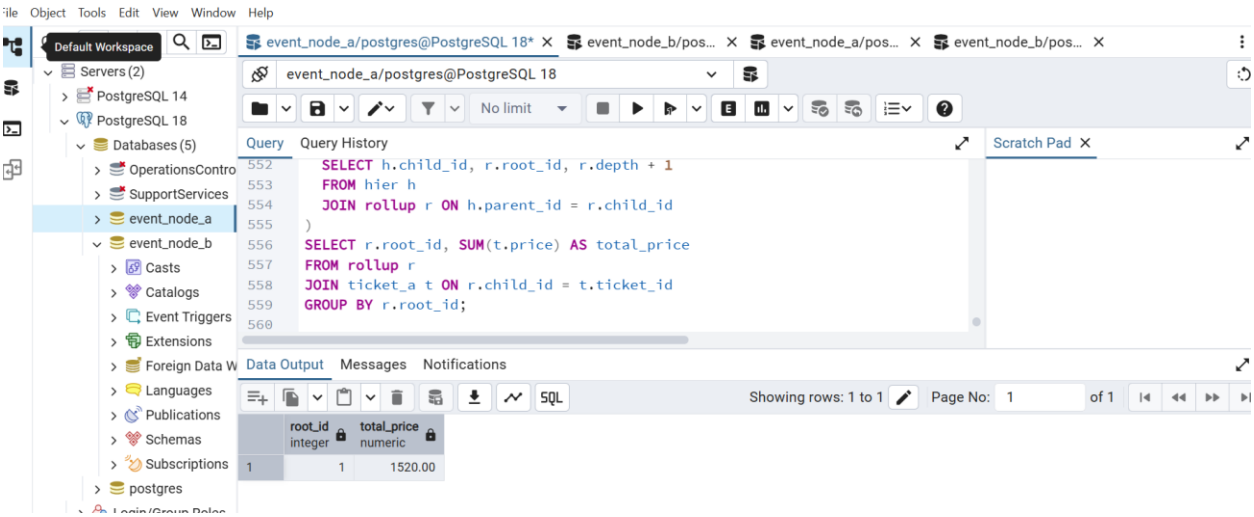


Aggregate rollup totals per root

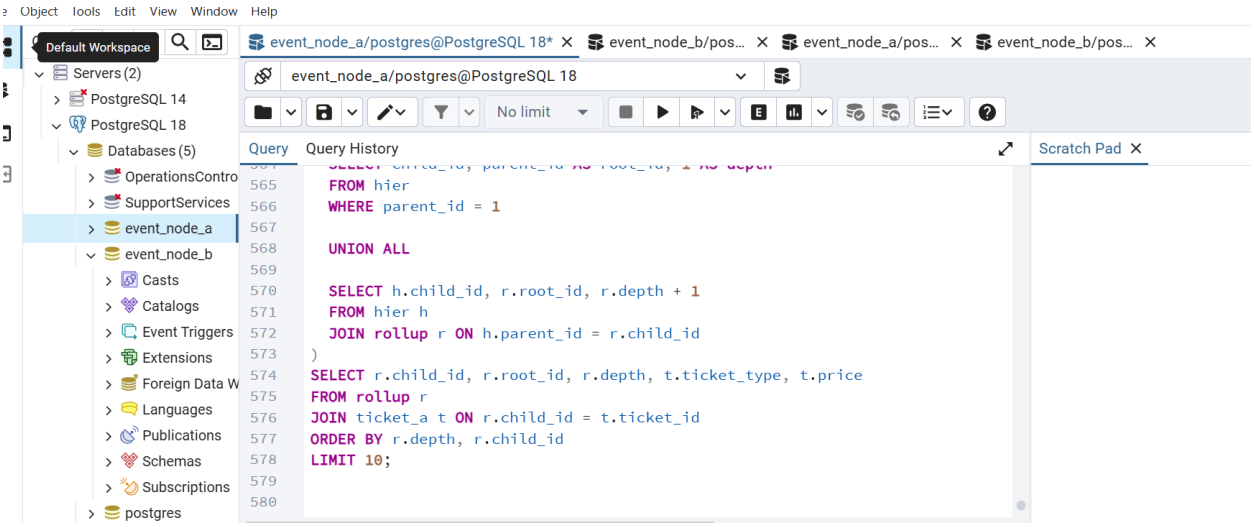
Code:

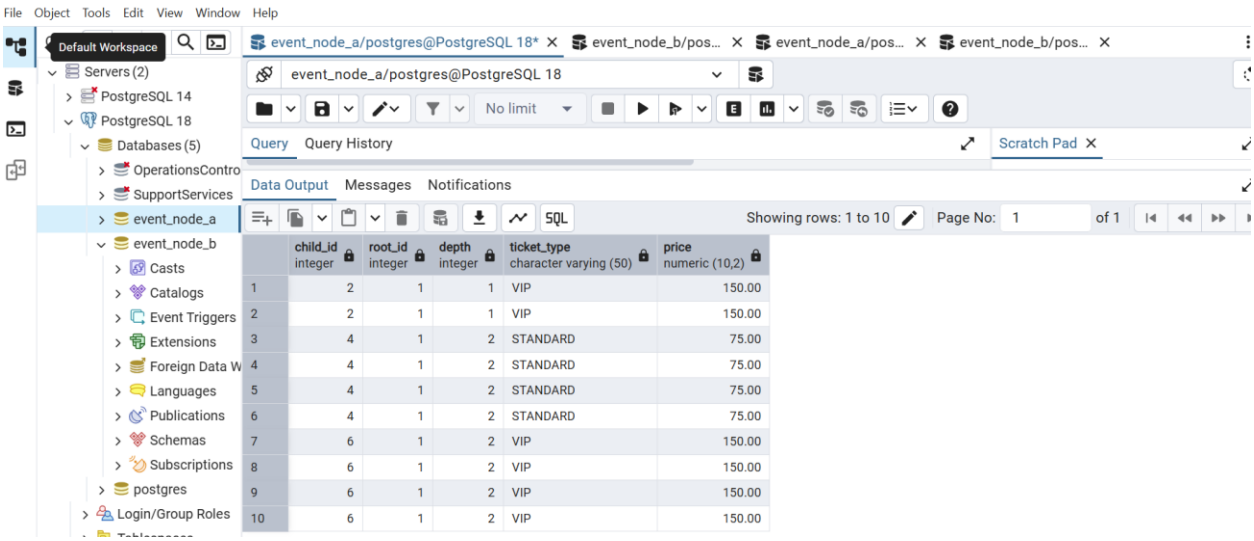


Output



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

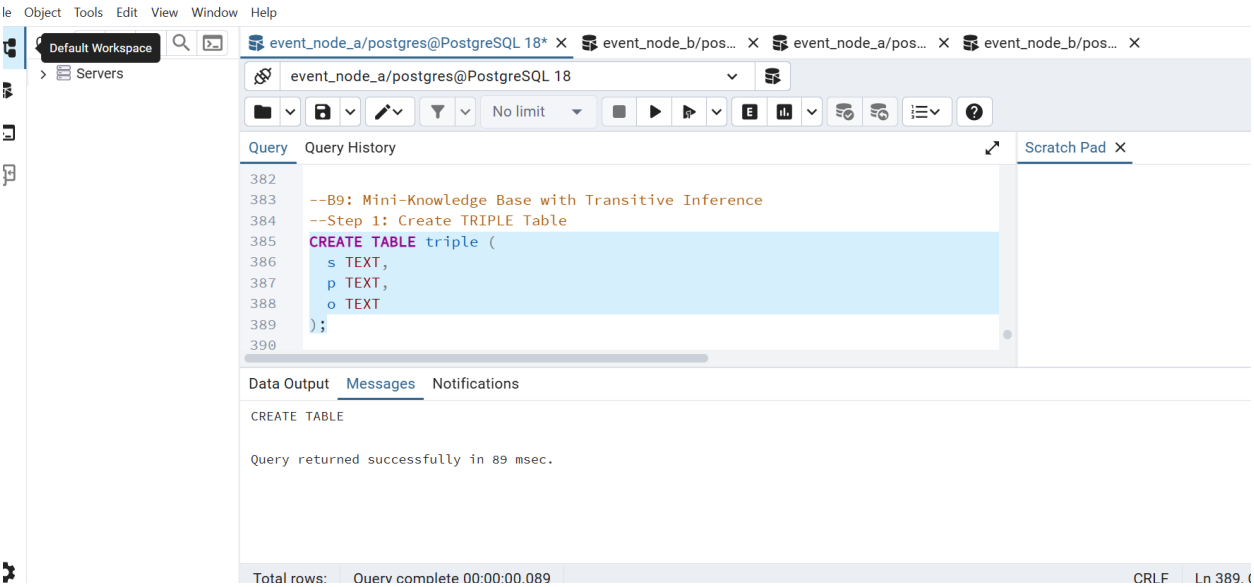




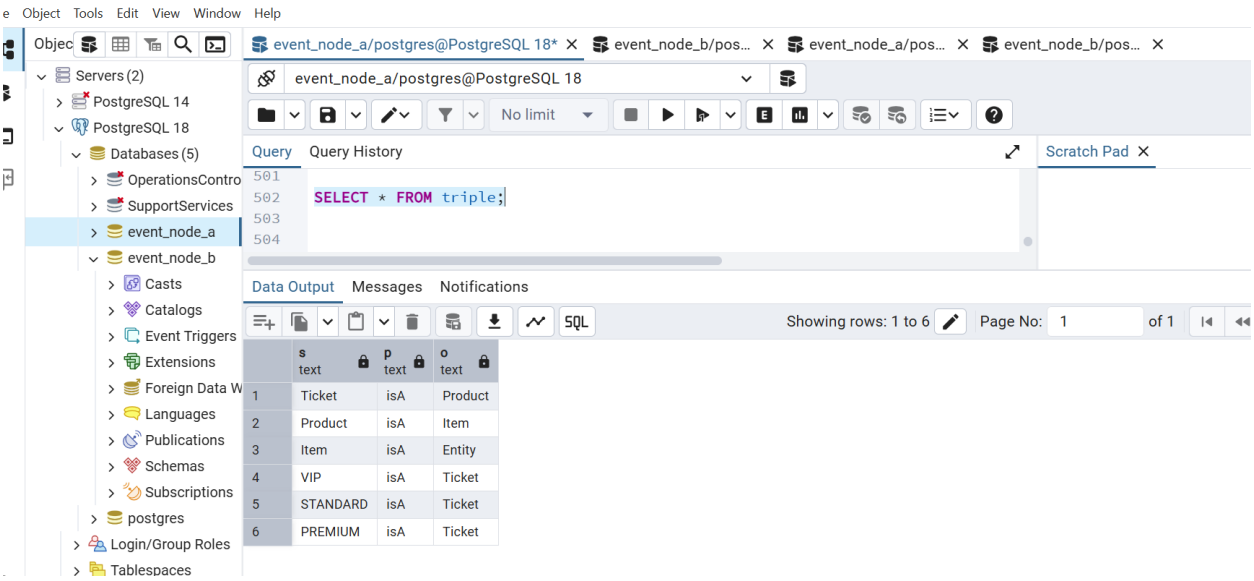
B9: Mini-Knowledge Base with Transitive Inference(≤10 facts)

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

Step 1: Create TRIPLE Table

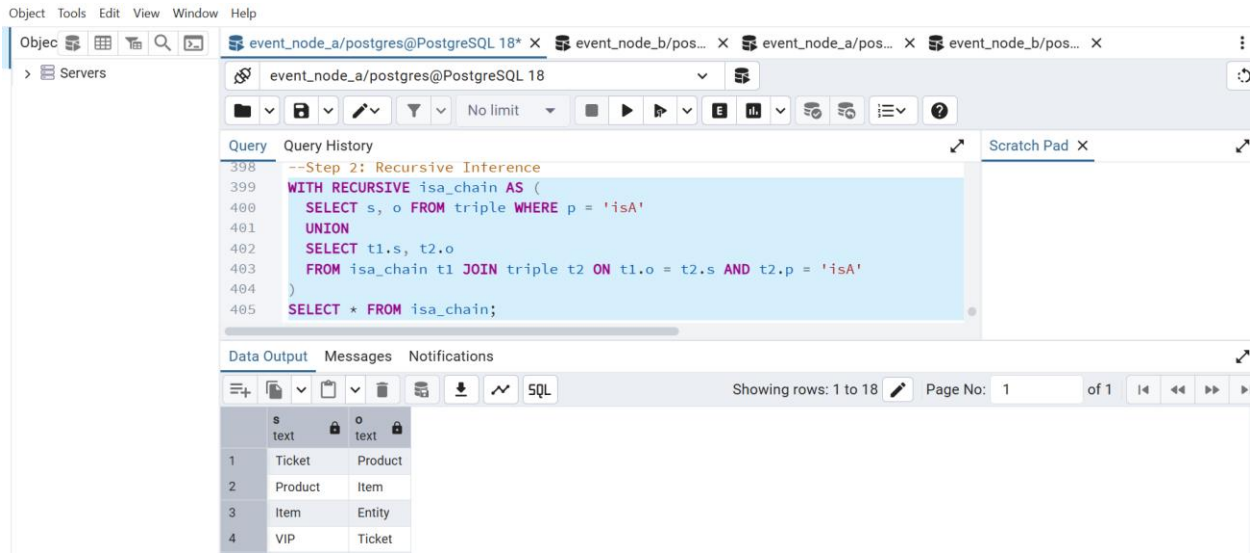


2. Insert 8–10 domain facts relevant to your project (e.g., simple type hierarchy or rule implications).



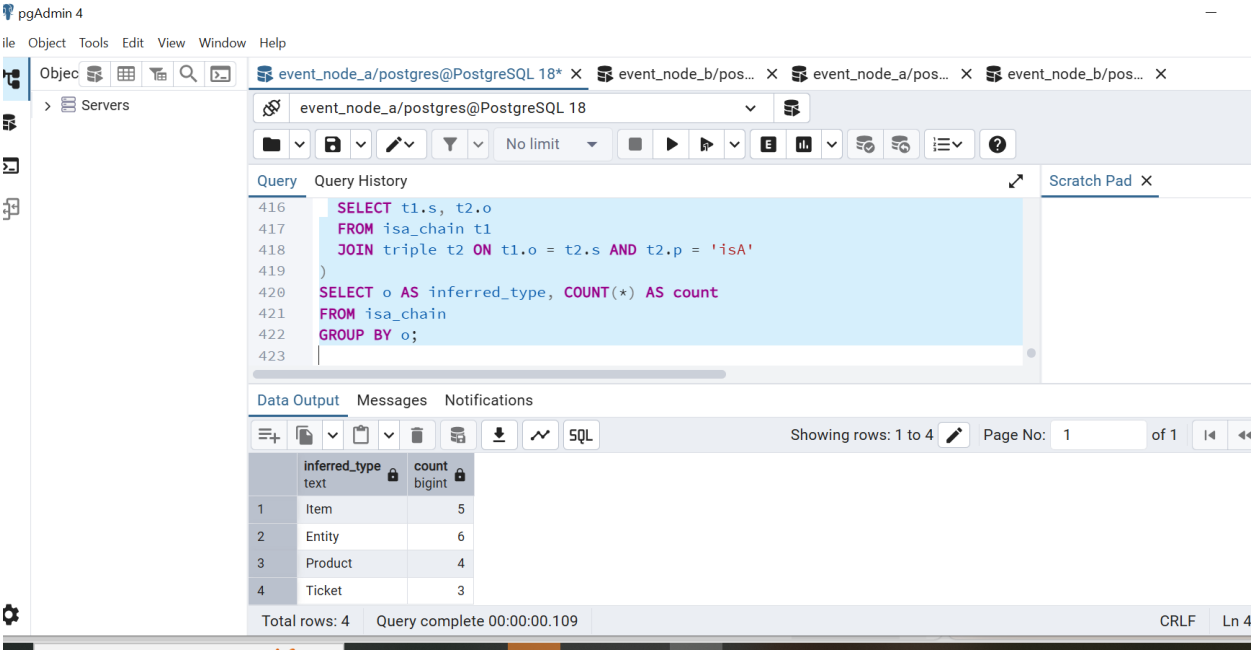
3. Write a recursive inference query implementing transitive isA*; apply labels to base records and return up to 10 labeled rows.

Step 2: Recursive Inference



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

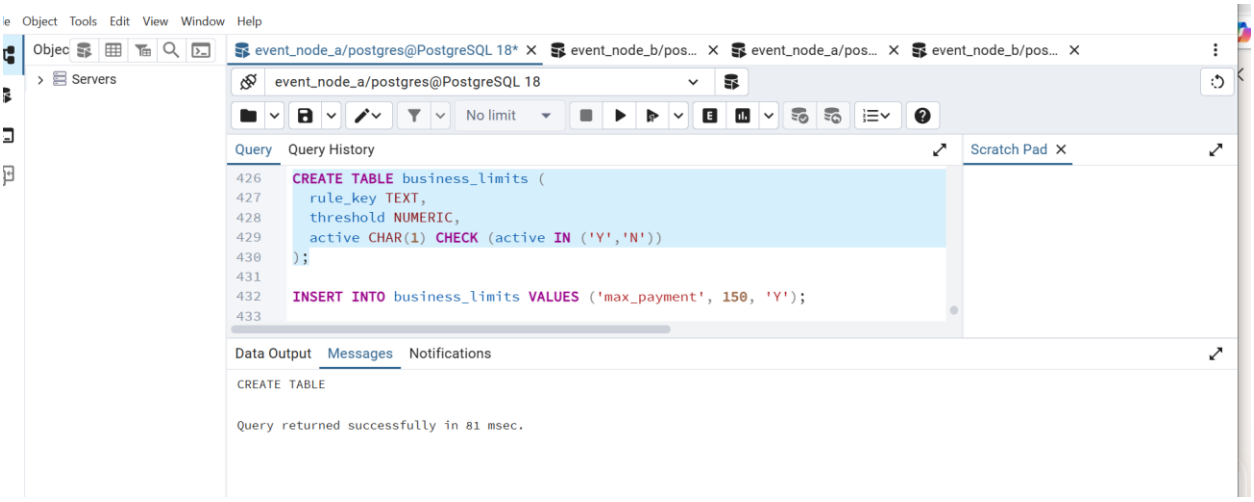
Step 3: Grouping Check



B10: Business Limit Alert (Function + Trigger)

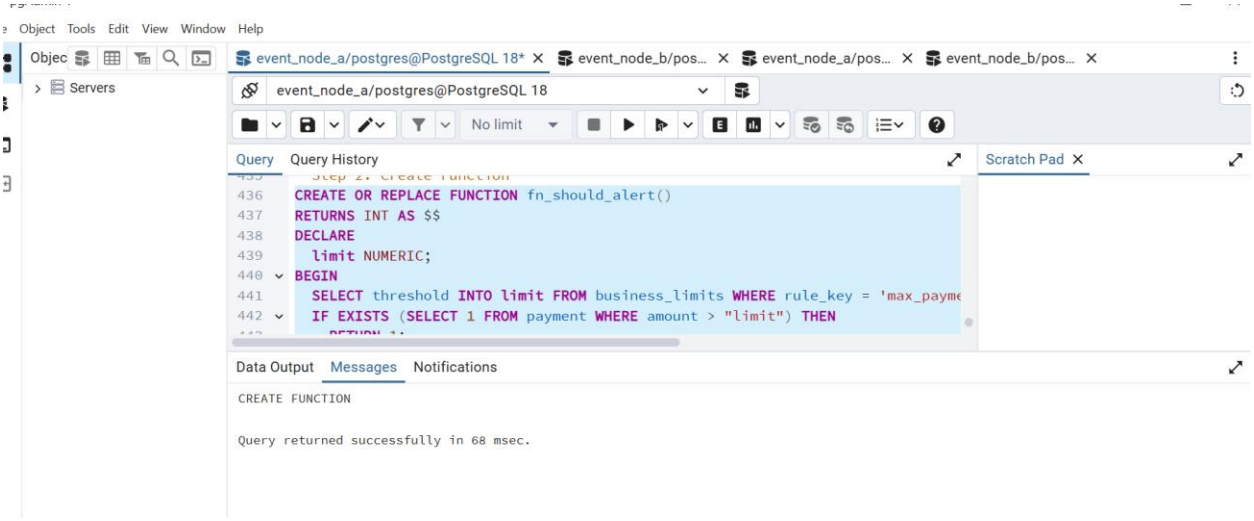
1. Create BUSINESS_LIMITS(rule_key VARCHAR2(64), threshold NUMBER, active CHAR(1) CHECK(active IN('Y','N'))) and seed exactly one active rule.

Step 1: Create Limits Table



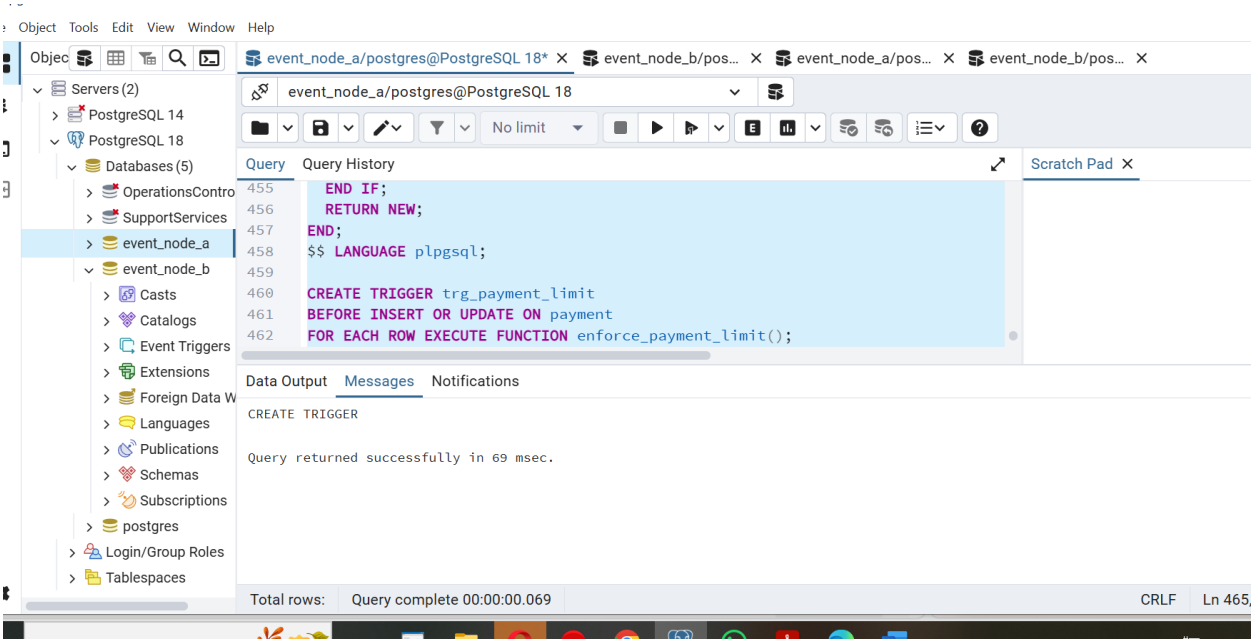
2. Implement function fn_should_alert(...) that reads BUSINESS_LIMITS and inspects current data in Payment or Ticket to decide a violation (return 1/0).

Step 2: Create Function



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

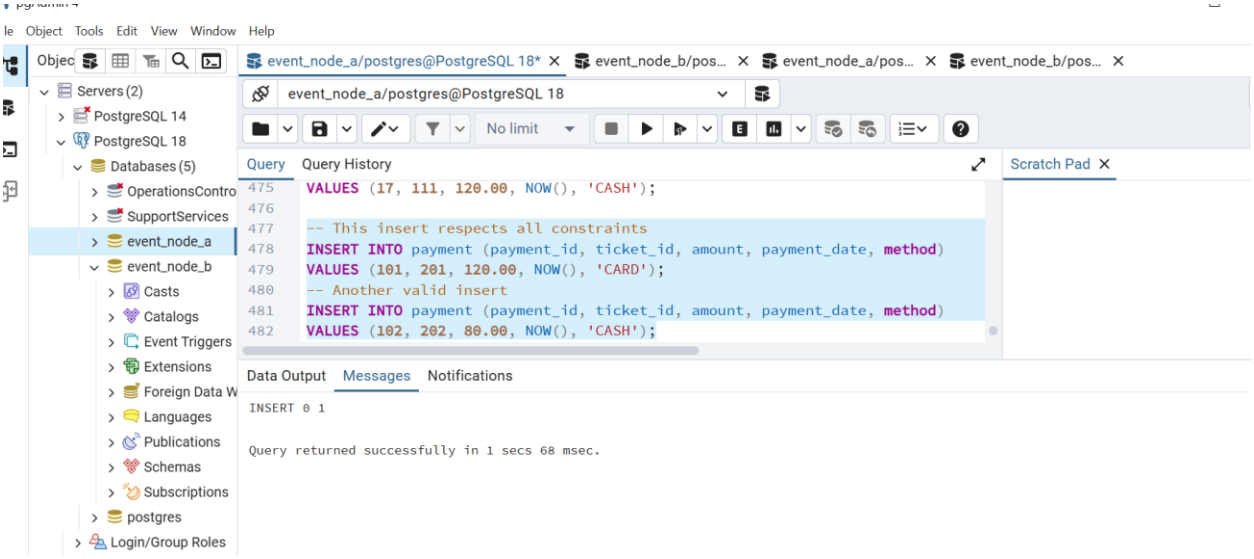
Step 3: Create Trigger



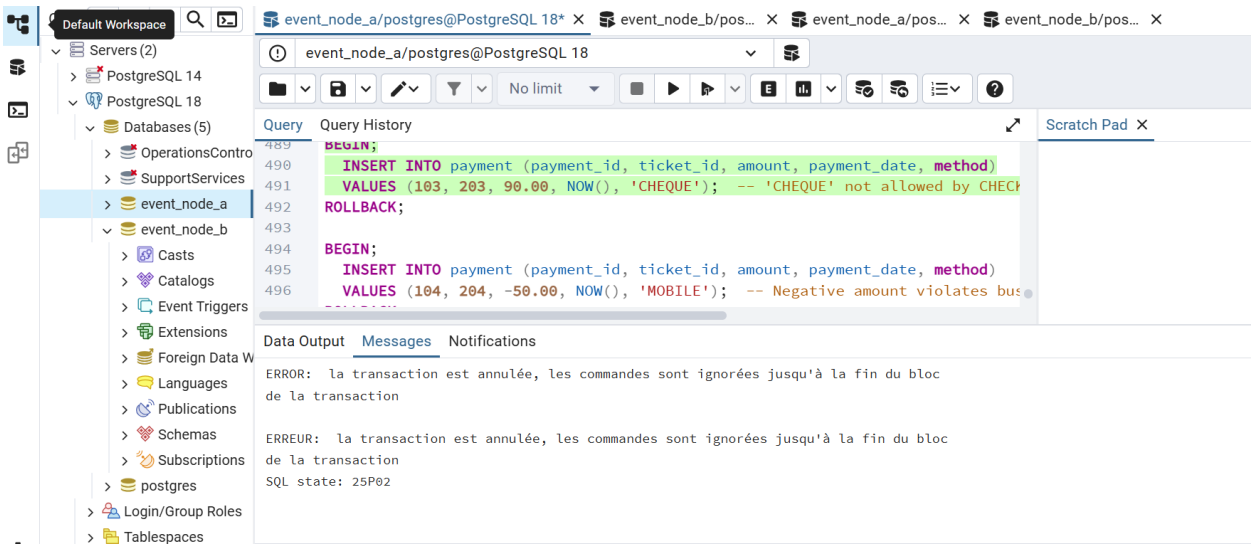
4. Demonstrate 2 failing and 2 passing DML cases; rollback the failing ones so total committed rows remain within the ≤10 budget.

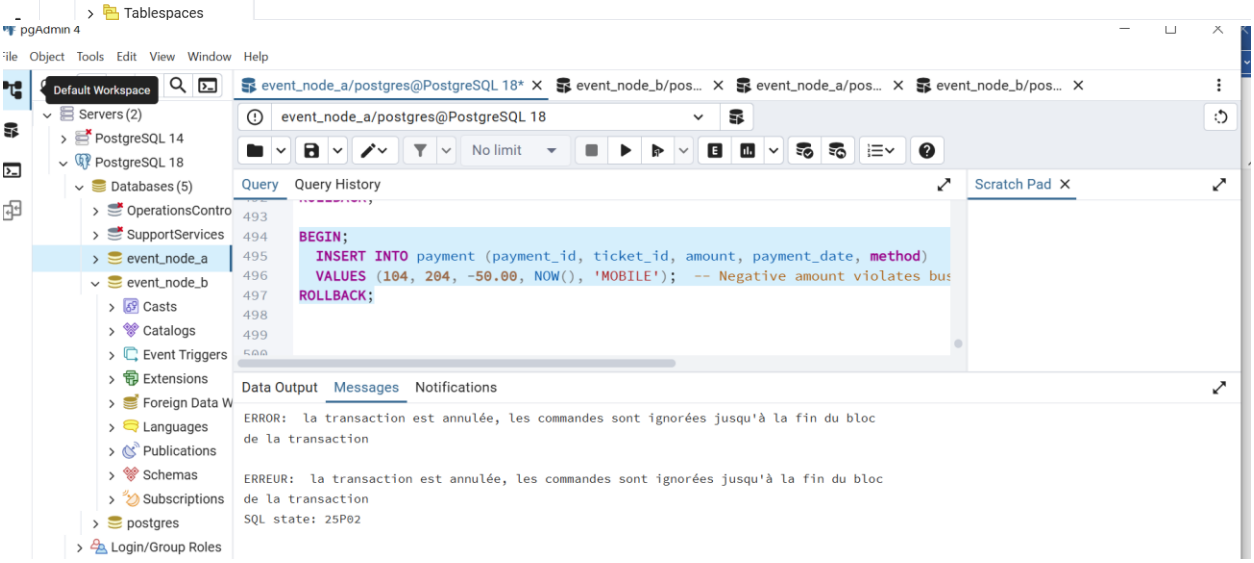
Step 4: Test DML

2 Passing


event_node_a/postgres@PostgreSQL 18*
event_node_b/pos...
event_node_a/pos...
event_node_b/pos...
event_node_a/postgres@PostgreSQL 18
No limit
Query
Query History
Scratch Pad
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');
Data Output
Messages
Notifications
INSERT 0 1
Query returned successfully in 1 secs 68 msec.

2 Failing


Default Workspace
event_node_a/postgres@PostgreSQL 18*
event_node_b/pos...
event_node_a/pos...
event_node_b/pos...
event_node_a/postgres@PostgreSQL 18
No limit
Query
Query History
Scratch Pad
BEGIN;
INSERT INTO payment (payment_id, ticket_id, amount, payment_date, method)
VALUES (103, 203, 90.00, NOW(), 'CHEQUE'); -- 'CHEQUE' not allowed by CHECK
ROLLBACK;
BEGIN;
INSERT INTO payment (payment_id, ticket_id, amount, payment_date, method)
VALUES (104, 204, -50.00, NOW(), 'MOBILE'); -- Negative amount violates bus
Data Output
Messages
Notifications
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


pgAdmin 4
Default Workspace
event_node_a/postgres@PostgreSQL 18*
event_node_b/pos...
event_node_a/pos...
event_node_b/pos...
event_node_a/postgres@PostgreSQL 18
No limit
Query
Query History
Scratch Pad
BEGIN;
INSERT INTO payment (payment_id, ticket_id, amount, payment_date, method)
VALUES (104, 204, -50.00, NOW(), 'MOBILE'); -- Negative amount violates bus
ROLLBACK;
Data Output
Messages
Notifications
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