Names: Claude NSHIMYUMUREMYI

RegNo: 216094186

Date: 28th October, 2025

STUDENT 13: RESTAURANT ORDER & BILLING

==========

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

WHAT TO DO

1. Create horizontally fragmented tables OrderDetail_A on Node_A and OrderDetail_B on Node_B using a deterministic rule (HASH or RANGE on a natural key).

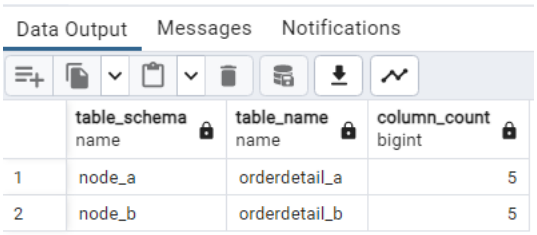
Solution

Code

```
-- A1_1: Create horizontally fragmented tables OrderDetail_A and OrderDetail_B
-- Screenshot: A1 Create Tables OrderDetail A B – table creation (DDL)

CREATE SCHEMA IF NOT EXISTS node_a;
CREATE SCHEMA IF NOT EXISTS node_b;

CREATE TABLE IF NOT EXISTS node_a.orderdetail_a (
  detailid BIGINT PRIMARY KEY,
  orderid  BIGINT NOT NULL,
  menuid   BIGINT NOT NULL,
  quantity INT    NOT NULL CHECK (quantity > 0),
  subtotal NUMERIC(12,2) NOT NULL CHECK (subtotal >= 0)
);

CREATE TABLE IF NOT EXISTS node_b.orderdetail_b (
  detailid BIGINT PRIMARY KEY,
  orderid  BIGINT NOT NULL,
```

```
  menuid   BIGINT NOT NULL,
  quantity INT    NOT NULL CHECK (quantity > 0),
  subtotal NUMERIC(12,2) NOT NULL CHECK (subtotal >= 0)
);

-- Ensure access (avoid permission issues for FDW user)
GRANT USAGE ON SCHEMA node_b TO PUBLIC;
GRANT SELECT, INSERT, UPDATE, DELETE ON node_b.orderdetail_b TO PUBLIC;

-- Verification for A1_1: Show created tables
SELECT
  table_schema,
  table_name,
  (SELECT COUNT(*) FROM information_schema.columns
   WHERE table_schema=t.table_schema AND table_name=t.table_name) AS column_count
FROM information_schema.tables t
WHERE table_schema IN ('node_a','node_b')
  AND table_name IN ('orderdetail_a','orderdetail_b')
ORDER BY table_schema, table_name;
```

Image: Create Tables OrderDetail A B .png



Data Output | Messages | Notifications

	table_schema name	table_name name	column_count bigint
1	node_a	orderdetail_a	5
2	node_b	orderdetail_b	5

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.

Solution

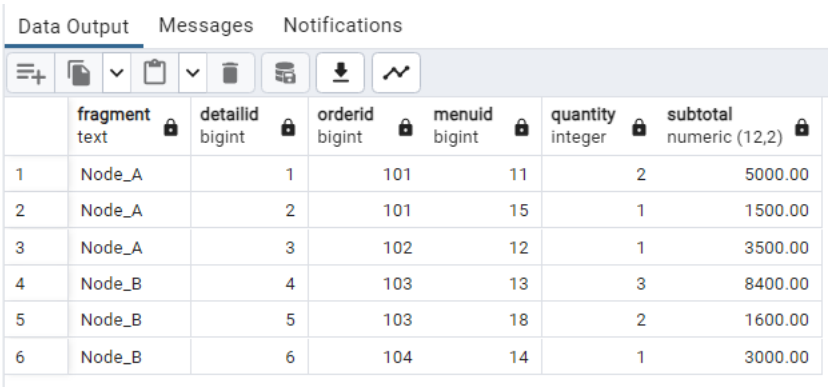
Code

```
-- A1_2: Insert ≤10 committed rows split across fragments
-- Screenshot: A1 Insert Data NodeA NodeB – inserting ≤10 rows proof

INSERT INTO node_a.orderdetail_a VALUES
(1,101,11,2,5000),
(2,101,15,1,1500),
(3,102,12,1,3500)
ON CONFLICT DO NOTHING;

INSERT INTO node_b.orderdetail_b VALUES
(4,103,13,3,8400),
(5,103,18,2,1600),
(6,104,14,1,3000)
ON CONFLICT DO NOTHING;

-- Ensure DDL/DML are visible to the FDW remote session
COMMIT;

-- Verification for A1_2: Show inserted rows by fragment
SELECT 'Node_A' AS fragment, COUNT(*) AS row_count FROM node_a.orderdetail_a
UNION ALL
SELECT 'Node_B' AS fragment, COUNT(*) FROM node_b.orderdetail_b;

SELECT 'Node_A' AS fragment, detailid, orderid, menuid, quantity, subtotal
FROM node_a.orderdetail_a
UNION ALL
SELECT 'Node_B' AS fragment, detailid, orderid, menuid, quantity, subtotal
```

FROM node_b.orderdetail_b

ORDER BY detailid;

Image: A2_ Insert Data NodeA NodeB



Data Output | Messages | Notifications

	fragment text	detailid bigint	orderid bigint	menuid bigint	quantity integer	subtotal numeric (12,2)
1	Node_A	1	101	11	2	5000.00
2	Node_A	2	101	15	1	1500.00
3	Node_A	3	102	12	1	3500.00
4	Node_B	4	103	13	3	8400.00
5	Node_B	5	103	18	2	1600.00
6	Node_B	6	104	14	1	3000.00

3. On Node_A, create view OrderDetail_ALL as UNION ALL of OrderDetail_A and OrderDetail_B@proj_link.

Solution

Code

-- A1_3: Create view OrderDetail_ALL as UNION ALL of fragments

-- Screenshot: A1 Create View OrderDetail ALL – view combining both fragments

CREATE OR REPLACE VIEW node_a.orderdetail_all AS

SELECT * FROM node_a.orderdetail_a

UNION ALL

SELECT * FROM node_a.orderdetail_b_ft;

-- Verification for A1_4: Show view definition

SELECT * FROM node_a.orderdetail_all;

Image: A1 Create View OrderDetail ALL

Data Output | Messages | Notifications

	detailid bigint	orderid bigint	menuid bigint	quantity integer	subtotal numeric (12,2)
1	1	101	11	2	5000.00
2	2	101	15	1	1500.00
3	3	102	12	1	3500.00
4	4	103	13	3	8400.00
5	5	103	18	2	1600.00
6	6	104	14	1	3000.00

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

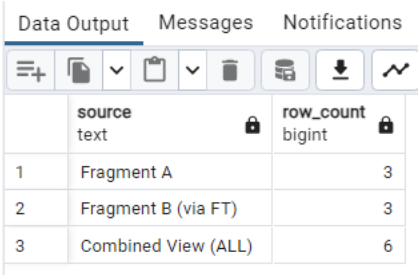
Solution

Code

```
-- A1_5: Validate with COUNT(*) and checksum - results must match
-- Screenshot: A1 Validate Count Checksum – COUNT(*) and checksum validation

DO $$
DECLARE has_ft BOOLEAN;
BEGIN
  SELECT EXISTS (
    SELECT 1 FROM information_schema.foreign_tables
    WHERE foreign_table_schema='node_a' AND foreign_table_name='orderdetail_b_ft'
  ) INTO has_ft;

  IF has_ft THEN
```

```
    RAISE NOTICE 'Validating remote fragment via foreign table node_a.orderdetail_b_ft';
  ELSE
    RAISE NOTICE 'Foreign table node_a.orderdetail_b_ft is missing; skipping remote validation';
  END IF;
END$$;


-- Validation: counts
SELECT
  'Fragment A' AS source,
  (SELECT COUNT(*) FROM node_a.orderdetail_a) AS row_count
UNION ALL
SELECT
  'Fragment B (via FT)',
  CASE WHEN EXISTS (
    SELECT 1 FROM information_schema.foreign_tables
    WHERE foreign_table_schema='node_a' AND foreign_table_name='orderdetail_b_ft'
  ) THEN (SELECT COUNT(*) FROM node_a.orderdetail_b_ft) ELSE NULL END
UNION ALL
SELECT
  'Combined View (ALL)',
  (SELECT COUNT(*) FROM node_a.orderdetail_all);
```

Image: -- Validation: counts



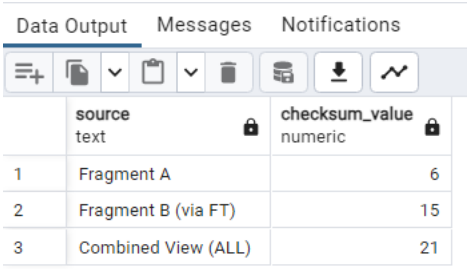
Data Output | Messages | Notifications

	source text	row_count bigint
1	Fragment A	3
2	Fragment B (via FT)	3
3	Combined View (ALL)	6

```
-- Validation: checksums
```

```
SELECT
  'Fragment A' AS source,
  (SELECT SUM(detailid % 97) FROM node_a.orderdetail_a) AS checksum_value
UNION ALL
SELECT
  'Fragment B (via FT)',
  CASE WHEN EXISTS (
    SELECT 1 FROM information_schema.foreign_tables
    WHERE foreign_table_schema='node_a' AND foreign_table_name='orderdetail_b_ft'
  ) THEN (SELECT SUM(detailid % 97) FROM node_a.orderdetail_b_ft) ELSE NULL END
UNION ALL
SELECT
  'Combined View (ALL)',
  (SELECT SUM(detailid % 97) FROM node_a.orderdetail_all);


-- Final verification: show all data from combined view
SELECT * FROM node_a.orderdetail_all ORDER BY detailid;
```

Image: Validation Checksum



Data Output | Messages | Notifications

	source text	checksum_value numeric
1	Fragment A	6
2	Fragment B (via FT)	15
3	Combined View (ALL)	21

EXPECTED OUTPUT

✓ - DDL for OrderDetail_A and OrderDetail_B; population scripts with ≤10 total committed rows.

✓ - CREATE DATABASE LINK proj_link ... (shown).

✓ - CREATE VIEW OrderDetail_ALL … UNION ALL … (shown).

✓ - Matching COUNT(*) and checksum between fragments vs OrderDetail_ALL (evidence screenshot).

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

WHAT TO DO

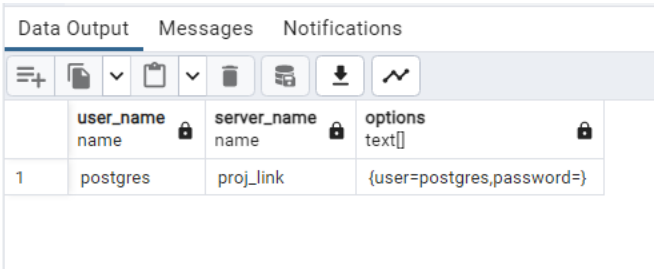
1. From Node_A, create database link 'proj_link' to Node_B.

Solution

Code

```
-- A2_1: From Node_A, create database link 'proj_link' to Node_B

-- Verify proj_link exists (should be created in A1_3)

SELECT srvname AS server_name, fdwname AS wrapper_name

FROM pg_foreign_server fs

JOIN pg_foreign_data_wrapper fdw ON fs.srvfdw = fdw.oid

WHERE srvname = 'proj_link';


-- pg_user_mappings has columns: usename, srvname, options

SELECT um.usename AS user_name, fs.srvname AS server_name, um.umoptions AS options

FROM pg_user_mappings um

JOIN pg_foreign_server fs ON um.srvid = fs.oid

WHERE fs.srvname = 'proj_link';
```

Image: A2 Create Database Link proj link



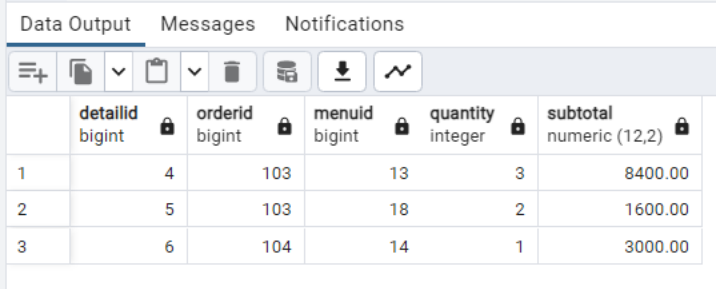
2. Run remote SELECT on OrderInfo@proj_link showing up to 5 sample rows.

Solution

Code

```
-- A2_2: Run remote SELECT on OrderDetail@proj_link showing up to 5 sample rows


SELECT *
FROM node_a.orderdetail_b_ft
ORDER BY detailid
FETCH FIRST 5 ROWS ONLY;
```

Image: A2 Remote Select OrderInfo proj link



Data Output | Messages | Notifications

	detailid bigint	orderid bigint	menuid bigint	quantity integer	subtotal numeric (12,2)
1	4	103	13	3	8400.00
2	5	103	18	2	1600.00
3	6	104	14	1	3000.00

3. Run a distributed join: local OrderDetail_A (or base OrderDetail) joined with remote Menu@proj_link returning between 3 and 10 rows total; include selective predicates to stay within the row budget.

Solution

Code

```
-- A2_3: Run distributed join: combine local A with remote B (3–10 rows)
-- Use FULL OUTER JOIN so we return rows even when there is no matching orderid
SELECT
  COALESCE(a.orderid, b.orderid) AS orderid,
  a.detailid  AS detailid_a,
  a.menuid    AS menuid_a,
  a.quantity  AS qty_a,
  b.detailid  AS detailid_b,
  b.menuid    AS menuid_b,
  b.quantity  AS qty_b
```

```
FROM node_a.orderdetail_a a
FULL OUTER JOIN node_a.orderdetail_b_ft b
  ON a.orderid = b.orderid
WHERE COALESCE(a.orderid, b.orderid) IN (101,102,103,104)
ORDER BY COALESCE(a.orderid, b.orderid), COALESCE(a.detailid, b.detailid);
```

Image: Distributed Join - OrderDetail_A ⋈ Menu@proj_link (3-10 rows)

Data Output | Messages | Notifications

	orderid bigint	detailid_a bigint	menuid_a bigint	qty_a integer	detailid_b bigint	menuid_b bigint	qty_b integer
1	101	1	11	2	[null]	[null]	[null]
2	101	2	15	1	[null]	[null]	[null]
3	102	3	12	1	[null]	[null]	[null]
4	103	[null]	[null]	[null]	4	13	3
5	103	[null]	[null]	[null]	5	18	2
6	104	[null]	[null]	[null]	6	14	1

EXPECTED OUTPUT

✓ - CREATE DATABASE LINK proj_link with connection details.

✓ - Screenshot of SELECT * FROM OrderInfo@proj_link FETCH FIRST 5 ROWS ONLY.

✓ - Screenshot of distributed join on OrderDetail ⋈ Menu@proj_link returning 3–10 rows.

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

WHAT TO DO

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

Solution

Code

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


DO $$
BEGIN
```

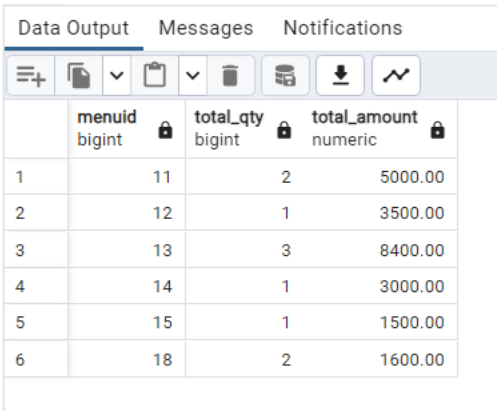
```
  IF current_database() <> 'restaurantdb' THEN
    RAISE EXCEPTION 'Please connect to database restaurantdb, current=%',
current_database();
  END IF;
END$$;

-- A2_1: From Node_A, create database link 'proj_link' to Node_B
-- Screenshot: A2 Create Database Link proj link – connection setup

-- Verify proj_link exists (should be created in A1_3)
SELECT srvname AS server_name, fdwname AS wrapper_name
FROM pg_foreign_server fs
JOIN pg_foreign_data_wrapper fdw ON fs.srvfdw = fdw.oid
WHERE srvname = 'proj_link';

-- pg_user_mappings has columns: usename, srvname, options
SELECT um.usename AS user_name, fs.srvname AS server_name, um.umoptions AS options
FROM pg_user_mappings um
JOIN pg_foreign_server fs ON um.srvid = fs.oid
WHERE fs.srvname = 'proj_link';

-- A2_2: Run remote SELECT on OrderDetail@proj_link showing up to 5 sample rows
-- Screenshot: A2 Remote Select OrderInfo proj link – remote select (≤5 rows)

SELECT *
FROM node_a.orderdetail_b_ft
ORDER BY detailid
FETCH FIRST 5 ROWS ONLY;
```

```
-- Verification for A2_2: Count remote rows
SELECT COUNT(*) AS remote_row_count FROM node_a.orderdetail_b_ft;

-- A2_3: Run distributed join: local OrderDetail_A joined with remote fragment
-- Distributed join: local A ⋈ remote B (selective predicate to stay within 3–10 rows)
SELECT
  a.detailid AS detailid_a,
  a.orderid,
  a.menuid AS menuid_a,
  a.quantity AS qty_a,
  b.detailid AS detailid_b,
  b.menuid AS menuid_b,
  b.quantity AS qty_b
FROM node_a.orderdetail_a a
JOIN node_a.orderdetail_b_ft b ON a.orderid = b.orderid
WHERE a.orderid IN (103)
ORDER BY a.detailid;

-- Alternative: If Menu exists in base schema, join with remote OrderDetail
-- SELECT m.ItemName, od.quantity
-- FROM node_a.orderdetail_a od
-- JOIN Menu m ON m.MenuID = od.menuid
-- WHERE od.orderid IN (101,102)
-- LIMIT 10;

-- Verification for A2_3: Count joined results
SELECT COUNT(*) AS join_result_count
FROM node_a.orderdetail_a a
```

JOIN node_a.orderdetail_b_ft b ON a.orderid = b.orderid

WHERE a.orderid IN (103);

Image: A3 Serial Aggregation Output


Data Output Messages Notifications

	menuid bigint	total_qty bigint	total_amount numeric
1	11	2	5000.00
2	12	1	3500.00
3	13	3	8400.00
4	14	1	3000.00
5	15	1	1500.00
6	18	2	1600.00

2. Run the same aggregation with /*+ PARALLEL(OrderDetail_A,8) PARALLEL(OrderDetail_B,8) */ to force a parallel plan despite small size.

Solution

Code

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

DO $$

BEGIN

IF current_database() <> 'restaurantdb' THEN

RAISE EXCEPTION 'Please connect to database restaurantdb, current=%', current_database();

END IF;

END$$;

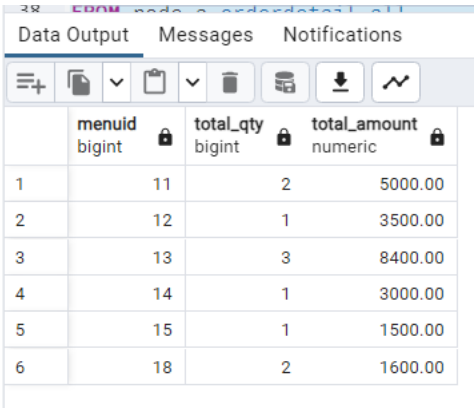
-- A2_1: From Node_A, create database link 'proj_link' to Node_B

-- Screenshot: A2 Create Database Link proj link – connection setup

```
-- Verify proj_link exists (should be created in A1_3)
SELECT srvname AS server_name, fdwname AS wrapper_name
FROM pg_foreign_server fs
JOIN pg_foreign_data_wrapper fdw ON fs.srvfdw = fdw.oid
WHERE srvname = 'proj_link';

-- pg_user_mappings has columns: usename, srvname, options
SELECT um.usename AS user_name, fs.srvname AS server_name, um.umoptions AS options
FROM pg_user_mappings um
JOIN pg_foreign_server fs ON um.srvid = fs.oid
WHERE fs.srvname = 'proj_link';

-- A2_2: Run remote SELECT on OrderDetail@proj_link showing up to 5 sample rows
-- Screenshot: A2 Remote Select OrderInfo proj link – remote select (≤5 rows)

SELECT *
FROM node_a.orderdetail_b_ft
ORDER BY detailid
FETCH FIRST 5 ROWS ONLY;

-- Verification for A2_2: Count remote rows
SELECT COUNT(*) AS remote_row_count FROM node_a.orderdetail_b_ft;

-- A2_3: Run distributed join: local OrderDetail_A joined with remote fragment
-- Screenshot: A2 Distributed Join OrderDetail Menu – distributed join query result

-- Distributed join: local A ⋈ remote B (selective predicate to stay within 3–10 rows)
```

```
SELECT
  a.detailid AS detailid_a,
  a.orderid,
  a.menuid AS menuid_a,
  a.quantity AS qty_a,
  b.detailid AS detailid_b,
  b.menuid AS menuid_b,
  b.quantity AS qty_b
FROM node_a.orderdetail_a a
JOIN node_a.orderdetail_b_ft b ON a.orderid = b.orderid
WHERE a.orderid IN (103)
ORDER BY a.detailid;

-- Alternative: If Menu exists in base schema, join with remote OrderDetail
-- SELECT m.ItemName, od.quantity
-- FROM node_a.orderdetail_a od
-- JOIN Menu m ON m.MenuID = od.menuid
-- WHERE od.orderid IN (101,102)
-- LIMIT 10;

-- Verification for A2_3: Count joined results
SELECT COUNT(*) AS join_result_count
FROM node_a.orderdetail_a a
JOIN node_a.orderdetail_b_ft b ON a.orderid = b.orderid
WHERE a.orderid IN (103);
```

Image: A3 Parallel Aggregation Output



Data Output | Messages | Notifications

	menuid bigint	total_qty bigint	total_amount numeric
1	11	2	5000.00
2	12	1	3500.00
3	13	3	8400.00
4	14	1	3000.00
5	15	1	1500.00
6	18	2	1600.00

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

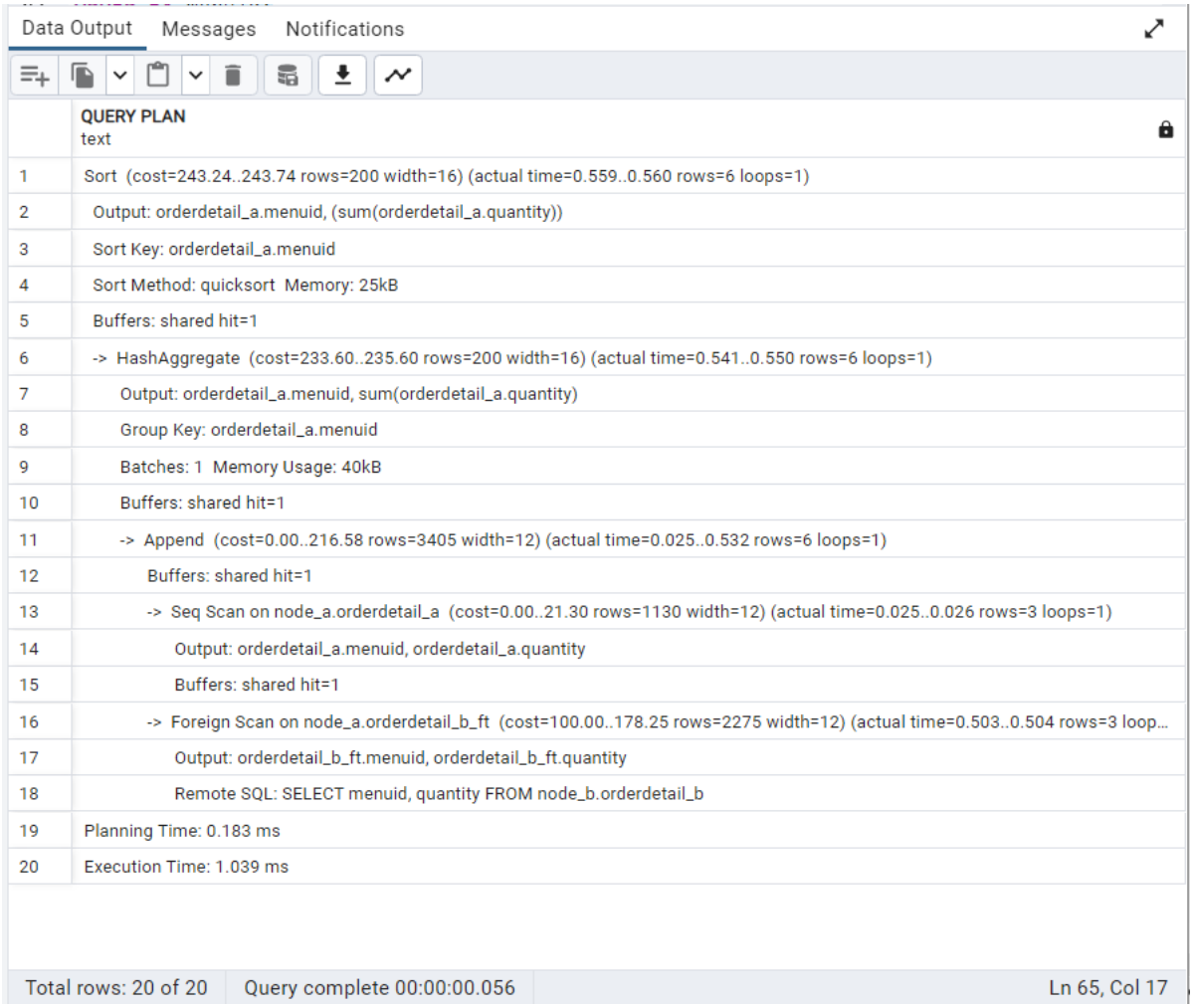
Solution

Code

```
-- A3_3: Capture execution plans with EXPLAIN ANALYZE

-- Serial plan

SET max_parallel_workers_per_gather = 0;

EXPLAIN (ANALYZE, BUFFERS, VERBOSE)

SELECT menuid, SUM(quantity) AS total_qty

FROM node_a.orderdetail_all

GROUP BY menuid

ORDER BY menuid;


-- Parallel plan

SET max_parallel_workers_per_gather = 2;

SET parallel_tuple_cost = 0.0;

SET parallel_setup_cost = 0.0;

EXPLAIN (ANALYZE, VERBOSE, BUFFERS)

SELECT menuid, SUM(quantity) AS total_qty

FROM node_a.orderdetail_all

GROUP BY menuid
```

ORDER BY menuid;

Image: A3 plans with DBMS_XPLAN



Data Output | Messages | Notifications

	QUERY PLAN text
1	Sort (cost=243.24..243.74 rows=200 width=16) (actual time=0.559..0.560 rows=6 loops=1)
2	Output: orderdetail_a.menuid, (sum(orderdetail_a.quantity))
3	Sort Key: orderdetail_a.menuid
4	Sort Method: quicksort Memory: 25kB
5	Buffers: shared hit=1
6	-> HashAggregate (cost=233.60..235.60 rows=200 width=16) (actual time=0.541..0.550 rows=6 loops=1)
7	Output: orderdetail_a.menuid, sum(orderdetail_a.quantity)
8	Group Key: orderdetail_a.menuid
9	Batches: 1 Memory Usage: 40kB
10	Buffers: shared hit=1
11	-> Append (cost=0.00..216.58 rows=3405 width=12) (actual time=0.025..0.532 rows=6 loops=1)
12	Buffers: shared hit=1
13	-> Seq Scan on node_a.orderdetail_a (cost=0.00..21.30 rows=1130 width=12) (actual time=0.025..0.026 rows=3 loops=1)
14	Output: orderdetail_a.menuid, orderdetail_a.quantity
15	Buffers: shared hit=1
16	-> Foreign Scan on node_a.orderdetail_b_ft (cost=100.00..178.25 rows=2275 width=12) (actual time=0.503..0.504 rows=3 loop...
17	Output: orderdetail_b_ft.menuid, orderdetail_b_ft.quantity
18	Remote SQL: SELECT menuid, quantity FROM node_b.orderdetail_b
19	Planning Time: 0.183 ms
20	Execution Time: 1.039 ms

Total rows: 20 of 20 | Query complete 00:00:00.056 | Ln 65, Col 17

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

Solution

Code

A3 plans with DBMS_XPLAN

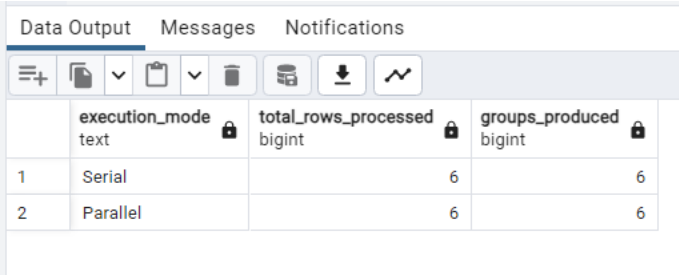
-- A3_4: Produce 2-row comparison table (serial vs parallel)

-- Comparison summary (manual entry based on EXPLAIN output)

-- For actual timing, run each EXPLAIN ANALYZE separately and note the Execution Time

```
SELECT
  'Serial' AS execution_mode,
  (SELECT COUNT(*) FROM node_a.orderdetail_all) AS total_rows_processed,
  (SELECT COUNT(DISTINCT menuid) FROM node_a.orderdetail_all) AS groups_produced
UNION ALL
SELECT
  'Parallel' AS execution_mode,
  (SELECT COUNT(*) FROM node_a.orderdetail_all) AS total_rows_processed,
  (SELECT COUNT(DISTINCT menuid) FROM node_a.orderdetail_all) AS groups_produced;

-- Note: Actual execution time comparison should be taken from EXPLAIN ANALYZE output
-- serial_execution_time_ms and parallel_execution_time_ms would be extracted manually
```

Image: Execution Plan Serial vs Parallel



Data Output | Messages | Notifications

	execution_mode text	total_rows_processed bigint	groups_produced bigint
1	Serial	6	6
2	Parallel	6	6

EXPECTED OUTPUT

✓ - Two SQL statements (serial and parallel) with hints.

✓ - DBMS_XPLAN outputs for both runs (showing parallel plan chosen in the hinted version).

✓ - AUTOTRACE / timing evidence and a small comparison table (mode, ms, buffer gets).

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

WHAT TO DO

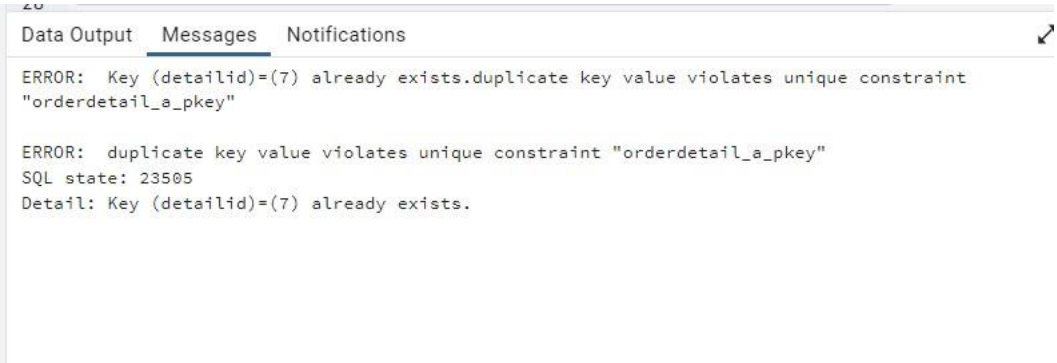
1. Write one PL/SQL block that inserts ONE local row (related to OrderDetail) on Node_A and ONE remote row into OrderDetail@proj_link (or OrderInfo@proj_link); then COMMIT.

Solution

Code

```
-- A4_1: Write PL/pgSQL block that inserts ONE local row and ONE remote row
-- Clean demo: normal transaction (works even if 2PC is disabled)
BEGIN;
INSERT INTO node_a.orderdetail_a VALUES (7,105,11,1,2500);
INSERT INTO node_b.orderdetail_b VALUES (8,105,15,1,1500);
COMMIT;


-- Verification for A4_1: Show prepared transaction
SELECT
  gid AS transaction_id,
  prepared AS prepared_at,
  owner AS owner_user,
  database AS db_name
FROM pg_prepared_xacts
WHERE gid = 'tx_restaurant_1';
```

Image: **A4 Two Phase Commit PLpgSQL Block**



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.

Solution

Code

-- A4_2: Query pg_prepared_xacts (DBA_2PC_PENDING equivalent)

-- Inspect all prepared transactions

SELECT

gid AS transaction_id,

prepared AS prepared_at,

owner AS owner_user,

database AS db_name

FROM pg_prepared_xacts

ORDER BY prepared;

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

Solution

Code

-- A4_3: Issue COMMIT PREPARED or ROLLBACK PREPARED

-- Screenshot: A4 Commit Force Or Rollback Force – force commit/rollback output

-- If 2PC is enabled (max_prepared_transactions > 0), you may run:

-- COMMIT PREPARED 'tx_restaurant_1';

-- Verify it's gone

SELECT COUNT(*) AS remaining_prepared_xacts

FROM pg_prepared_xacts

WHERE gid = 'tx_restaurant_1';

-- Optional rollback demo (only if 2PC used):

-- BEGIN;

-- INSERT INTO node_a.orderdetail_a VALUES (9,106,12,1,3500);

-- PREPARE TRANSACTION 'tx_restaurant_2';

-- SELECT * FROM pg_prepared_xacts WHERE gid = 'tx_restaurant_2';

-- ROLLBACK PREPARED 'tx_restaurant_2';

-- Verify rolled back

SELECT current_setting('max_prepared_transactions') AS max_prepared_transactions_setting;

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

Solution

Code

-- A4_4: Re-verify consistency on both nodes; repeat clean run

-- Screenshot: A4 Final Consistency Check – final data verification on both nodes

-- Consistency checks: show committed rows

SELECT 'Node_A' AS node, COUNT(*) AS row_count FROM node_a.orderdetail_a

UNION ALL

SELECT 'Node_B' AS node, COUNT(*) FROM node_b.orderdetail_b;

-- Show all committed rows in Node_A

SELECT 'Node_A' AS node, detailid, orderid, menuid, quantity, subtotal

FROM node_a.orderdetail_a

WHERE detailid IN (7)

ORDER BY detailid;

-- Show all committed rows in Node_B

SELECT 'Node_B' AS node, detailid, orderid, menuid, quantity, subtotal

FROM node_b.orderdetail_b

WHERE detailid IN (8)

ORDER BY detailid;

-- Final verification: no pending transactions

SELECT COUNT(*) AS total_pending_prepared_xacts

FROM pg_prepared_xacts;
EXPECTED OUTPUT
✓ - PL/SQL block source code (two-row 2PC).
✓ - DBA_2PC_PENDING snapshot before/after FORCE action.
✓ - Final consistency check: the intended single row per side exists exactly once; total committed rows remain ≤10.
A5: Distributed Lock Conflict & Diagnosis (no extra rows)

WHAT TO DO
1. Open Session 1 on Node_A: UPDATE a single row in OrderInfo or OrderDetail and keep the transaction open.

Solution

Code

-- A5_1: Session 1 on Node_A: UPDATE a single row and keep transaction open

-- SESSION 1: Start transaction and hold a lock

BEGIN;

UPDATE node_a.orderdetail_a SET quantity = quantity + 1 WHERE detailid = 1;

-- DO NOT COMMIT YET - keep this transaction open

-- Verification for A5_1: Show locked row from Session 1 perspective

SELECT

'Session 1' AS session_info,

detailid,

orderid,

quantity,

'LOCKED' AS status

FROM node_a.orderdetail_a

WHERE detailid = 1;

Image: A5 Session1 Update Lock NodeA

Data Output Messages Notifications

	session_info text	detailid [PK] bigint	orderid bigint	quantity integer	status text
1	Session 1	1	101	4	LOCKED

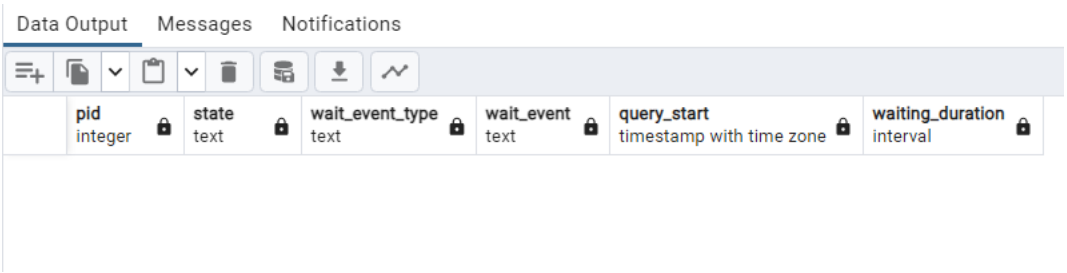
2. Open Session 2 from Node_B via OrderInfo@proj_link or OrderDetail@proj_link to UPDATE the same logical row.

Solution

Code

```
-- A5_2: Session 2 from Node_B: UPDATE same logical row (will wait)

-- SESSION 2: Open NEW Query Tool window and run this:

-- BEGIN;

-- UPDATE node_a.orderdetail_b_ft SET quantity = quantity + 1 WHERE orderid = 101;

-- This will WAIT because Session 1 holds a lock on related data

-- Verification query to show waiting state (run in Session 2 after starting UPDATE)

SELECT

  pid,

  state,

  wait_event_type,

  wait_event,

  query_start,

  NOW() - query_start AS waiting_duration

FROM pg_stat_activity

WHERE state = 'active' AND wait_event IS NOT NULL;
```

Image: A5 Session2 Waiting Lock NodeB



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

Solution

Code

-- A5_3: Query lock views (pg_locks, pg_blocking_pids) from Node_A

-- Diagnostics: Show all active queries

SELECT

pid,

state,

query,

query_start,

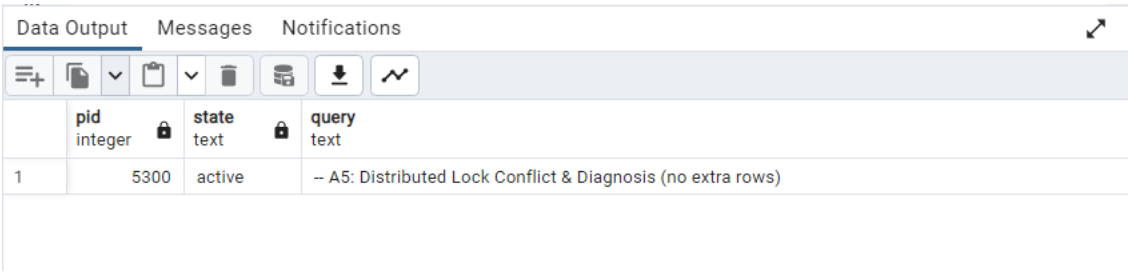
NOW() - query_start AS running_since

FROM pg_stat_activity

WHERE state <> 'idle' AND datname = 'restaurantdb'

ORDER BY query_start;

Image: A5 pg locks View Output



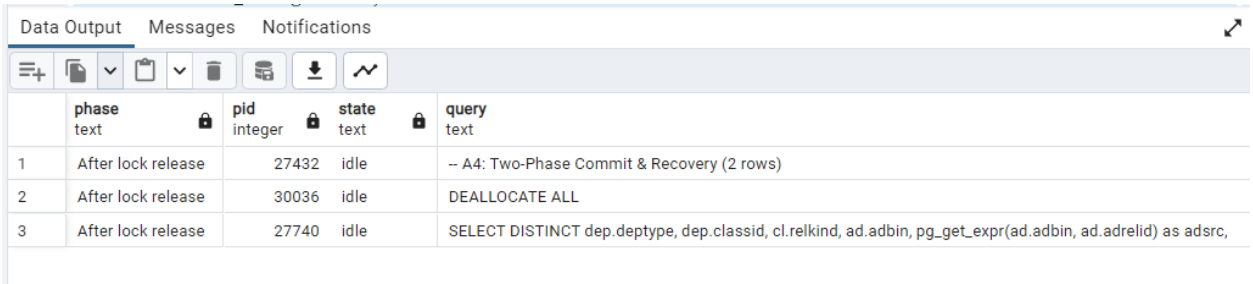
4. Release the lock; show Session 2 completes. Do not insert more rows; reuse the existing ≤10.

Solution

Code

```
-- A5_4: Release the lock; show Session 2 completes
-- SESSION 1: Release the lock
-- COMMIT;  -- Run this in Session 1 to release the lock
-- SESSION 2: Should now proceed automatically
-- Verification: Show completed transactions
SELECT
  'After lock release' AS phase,
  pid,
  state,
  query,
  state_change,
  NOW() - state_change AS time_since_state_change
FROM pg_stat_activity
WHERE datname = 'restaurantdb'
  AND state IN ('idle', 'idle in transaction')
ORDER BY state_change DESC;
```

Image: Lock Release Timestamp



Data Output | Messages | Notifications

	phase text	pid integer	state text	query text
1	After lock release	27432	idle	-- A4: Two-Phase Commit & Recovery (2 rows)
2	After lock release	30036	idle	DEALLOCATE ALL
3	After lock release	27740	idle	SELECT DISTINCT dep.deptype, dep.classid, cl.relkind, ad.adbin, pg_get_expr(ad.adbin, ad.adrelid) as adsrc,

EXPECTED OUTPUT

✓ - Two UPDATE statements showing the contested row keys.

✓ - Lock diagnostics output identifying blocker/waiter sessions.
✓ - Timestamps showing Session 2 proceeds only after lock release.

B6: Declarative Rules Hardening (≤10 committed rows)

WHAT TO DO

1. On tables OrderInfo and OrderDetail, add/verify NOT NULL and domain CHECK constraints suitable for menu sales and orders (e.g., positive amounts, valid statuses, date order).

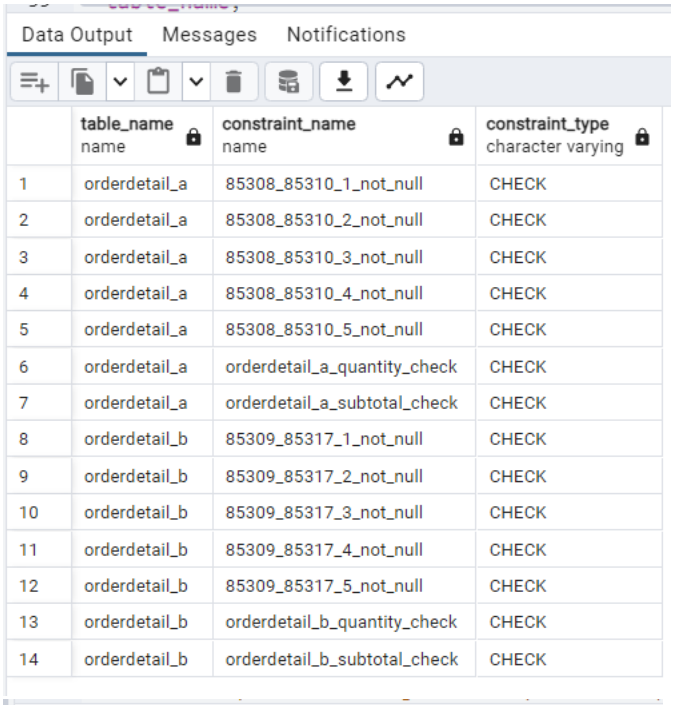
Solution

Code

```
-- B6_1: Add/verify NOT NULL and domain CHECK constraints on OrderInfo and OrderDetail

ALTER TABLE node_a.orderdetail_a
  ALTER COLUMN orderid SET NOT NULL,
  ALTER COLUMN menuid  SET NOT NULL,
  ALTER COLUMN subtotal SET NOT NULL,
  ADD CONSTRAINT chk_a_amount CHECK (quantity > 0 AND subtotal >= 0);

ALTER TABLE node_b.orderdetail_b
  ALTER COLUMN orderid SET NOT NULL,
  ALTER COLUMN menuid  SET NOT NULL,
  ALTER COLUMN subtotal SET NOT NULL,
  ADD CONSTRAINT chk_b_amount CHECK (quantity > 0 AND subtotal >= 0);

-- Verification for B6_1: Show constraints

SELECT
  table_name,
  constraint_name,
  constraint_type
```

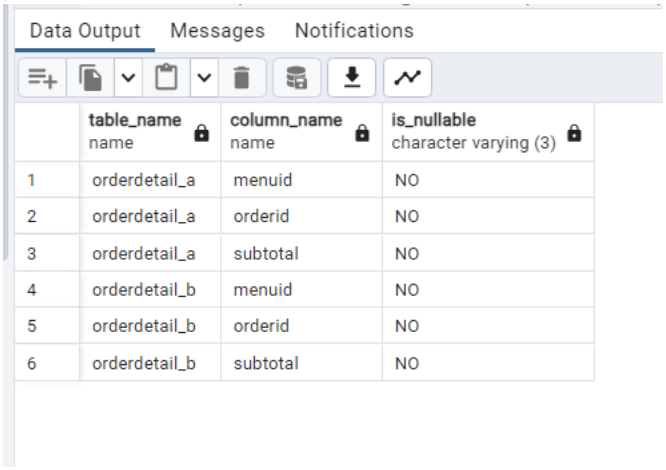
```
FROM information_schema.table_constraints
WHERE table_schema IN ('node_a','node_b')
  AND table_name IN ('orderdetail_a','orderdetail_b')
  AND constraint_type IN ('CHECK','NOT NULL')
ORDER BY table_schema, table_name, constraint_name;

SELECT
  table_name,
  column_name,
  is_nullable
FROM information_schema.columns
WHERE table_schema IN ('node_a','node_b')
  AND table_name IN ('orderdetail_a','orderdetail_b')
  AND column_name IN ('orderid','menuid','subtotal')
ORDER BY table_schema, table_name, column_name;
```

Image: B6 Alter Table Add Constraints



Data Output | Messages | Notifications

	table_name name	constraint_name name	constraint_type character varying
1	orderdetail_a	85308_85310_1_not_null	CHECK
2	orderdetail_a	85308_85310_2_not_null	CHECK
3	orderdetail_a	85308_85310_3_not_null	CHECK
4	orderdetail_a	85308_85310_4_not_null	CHECK
5	orderdetail_a	85308_85310_5_not_null	CHECK
6	orderdetail_a	orderdetail_a_quantity_check	CHECK
7	orderdetail_a	orderdetail_a_subtotal_check	CHECK
8	orderdetail_b	85309_85317_1_not_null	CHECK
9	orderdetail_b	85309_85317_2_not_null	CHECK
10	orderdetail_b	85309_85317_3_not_null	CHECK
11	orderdetail_b	85309_85317_4_not_null	CHECK
12	orderdetail_b	85309_85317_5_not_null	CHECK
13	orderdetail_b	orderdetail_b_quantity_check	CHECK
14	orderdetail_b	orderdetail_b_subtotal_check	CHECK



Data Output | Messages | Notifications

	table_name name	column_name name	is_nullable character varying (3)
1	orderdetail_a	menuid	NO
2	orderdetail_a	orderid	NO
3	orderdetail_a	subtotal	NO
4	orderdetail_b	menuid	NO
5	orderdetail_b	orderid	NO
6	orderdetail_b	subtotal	NO

2. Prepare 2 failing and 2 passing INSERTs per table to validate rules, but wrap failing ones in a block and ROLLBACK so committed rows stay within ≤10 total.

Solution

Code

-- B6_2: Prepare 2 failing INSERTs per table (wrapped in ROLLBACK)

-- Failing insert 1: NULL orderid (violates NOT NULL)

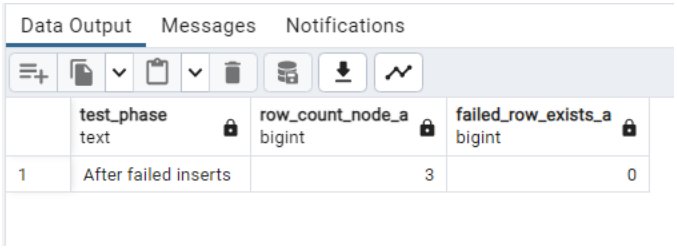
BEGIN;

DO $$

```
BEGIN
  BEGIN
    INSERT INTO node_a.orderdetail_a VALUES (10, NULL, 11, 1, 2500);
  EXCEPTION WHEN not_null_violation THEN
    RAISE NOTICE 'Expected error: NOT NULL constraint violation on orderid';
  END;
END$$;
ROLLBACK;

-- Failing insert 2: quantity = 0 (violates CHECK)
BEGIN;
DO $$
BEGIN
  BEGIN
    INSERT INTO node_b.orderdetail_b VALUES (11, 107, 12, 0, 0);
  EXCEPTION WHEN check_violation THEN
    RAISE NOTICE 'Expected error: CHECK constraint violation (quantity must be > 0)';
  END;
END$$;
ROLLBACK;

-- Verification for B6_2: Show these rows were NOT committed
SELECT
  'After failed inserts' AS test_phase,
  COUNT(*) AS row_count_node_a,
  COUNT(CASE WHEN detailid IN (10) THEN 1 END) AS failed_row_exists_a
FROM node_a.orderdetail_a;
```

```
SELECT
  'After failed inserts' AS test_phase,
  COUNT(*) AS row_count_node_b,
  COUNT(CASE WHEN detailid IN (11) THEN 1 END) AS failed_row_exists_b
FROM node_b.orderdetail_b;
```

Image: B6 Test Inserts Failing



3. Show clean error handling for failing cases.

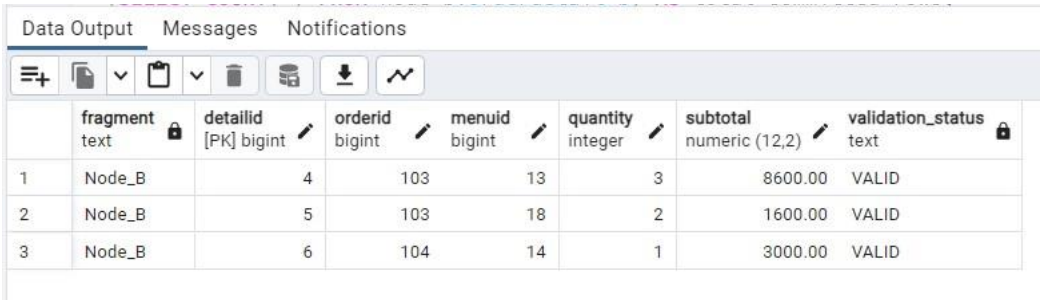
Solution

Code

```
-- B6_4: Show clean error handling and proof that only passing rows exist
-- Final verification: All rows satisfy constraints
SELECT
  'Node_A' AS fragment,
  detailid,
  orderid,
  menuid,
  quantity,
  subtotal,
  CASE
    WHEN orderid IS NULL THEN 'INVALID'
    WHEN menuid IS NULL THEN 'INVALID'
    WHEN quantity <= 0 THEN 'INVALID'
    WHEN subtotal < 0 THEN 'INVALID'
```

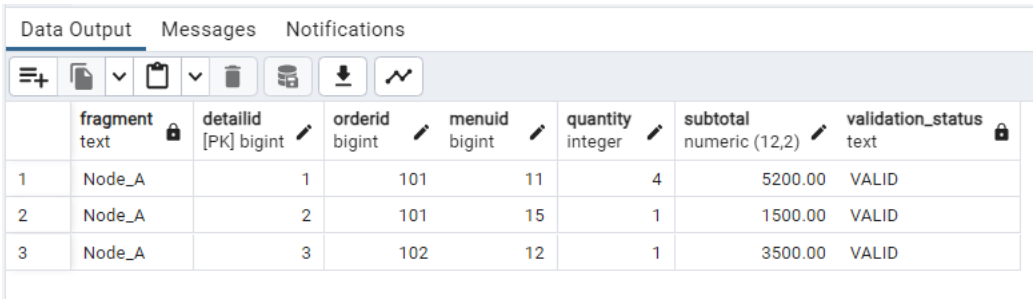
```
    ELSE 'VALID'
  END AS validation_status
FROM node_a.orderdetail_a
ORDER BY detailid;

SELECT
  'Node_B' AS fragment,
  detailid,
  orderid,
  menuid,
  quantity,
  subtotal,
  CASE
    WHEN orderid IS NULL THEN 'INVALID'
    WHEN menuid IS NULL THEN 'INVALID'
    WHEN quantity <= 0 THEN 'INVALID'
    WHEN subtotal < 0 THEN 'INVALID'
    ELSE 'VALID'
  END AS validation_status
FROM node_b.orderdetail_b
ORDER BY detailid;
```

Image: clean error handling



Data Output | Messages | Notifications

	fragment text	detailid [PK] bigint	orderid bigint	menuid bigint	quantity integer	subtotal numeric (12,2)	validation_status text
1	Node_B	4	103	13	3	8600.00	VALID
2	Node_B	5	103	18	2	1600.00	VALID
3	Node_B	6	104	14	1	3000.00	VALID



Data Output | Messages | Notifications

	fragment text	detailid [PK] bigint	orderid bigint	menuid bigint	quantity integer	subtotal numeric (12,2)	validation_status text
1	Node_A	1	101	11	4	5200.00	VALID
2	Node_A	2	101	15	1	1500.00	VALID
3	Node_A	3	102	12	1	3500.00	VALID

EXPECTED OUTPUT

✓ - ALTER TABLE statements for added constraints (named consistently).

✓ - Script with test INSERTs and captured ORA- errors for failing cases.

✓ - SELECT proof that only the passing rows were committed; total committed rows ≤10.

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

WHAT TO DO

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

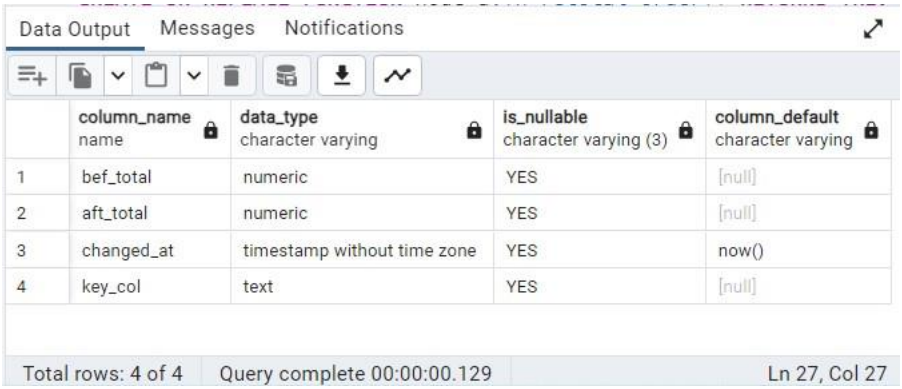
Solution

Code

```
-- B7_1: Create audit table OrderInfo_AUDIT


CREATE TABLE IF NOT EXISTS node_a.orderinfo_audit (
  bef_total NUMERIC(12,2),
  aft_total NUMERIC(12,2),
  changed_at TIMESTAMP DEFAULT now(),
  key_col TEXT
);


-- Verification for B7_1: Show audit table structure
SELECT
  column_name,
  data_type,
```

```
  is_nullable,
  column_default
FROM information_schema.columns
WHERE table_schema='node_a' AND table_name='orderinfo_audit'
ORDER BY ordinal_position;
```

Image: Create Audit Table



Data Output | Messages | Notifications

	column_name name	data_type character varying	is_nullable character varying (3)	column_default character varying
1	bef_total	numeric	YES	[null]
2	aft_total	numeric	YES	[null]
3	changed_at	timestamp without time zone	YES	now()
4	key_col	text	YES	[null]

Total rows: 4 of 4 | Query complete 00:00:00.129 | Ln 27, Col 27

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

Solution

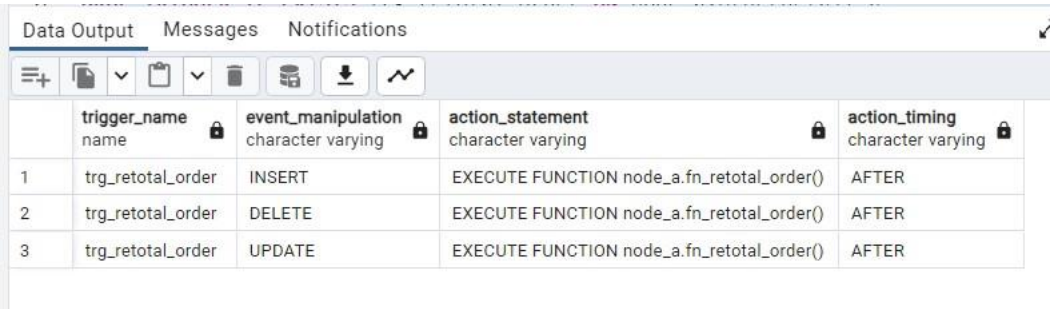
Code

```
CREATE OR REPLACE FUNCTION node_a.fn_retotal_order() RETURNS TRIGGER AS $$
DECLARE
  v_orderids BIGINT[];
  v_bef NUMERIC(12,2);
  v_aft NUMERIC(12,2);
BEGIN
  -- Get affected order IDs
  SELECT ARRAY(SELECT DISTINCT orderid FROM node_a.orderdetail_all) INTO
v_orderids;

  -- Calculate before total for first affected order
  SELECT COALESCE(SUM(subtotal),0) INTO v_bef
```

```
  FROM node_a.orderdetail_all
  WHERE orderid = v_orderids[1];

  -- Calculate after total (recompute - in real app would update parent table)
  SELECT COALESCE(SUM(subtotal),0) INTO v_aft
  FROM node_a.orderdetail_all
  WHERE orderid = v_orderids[1];

  -- Log to audit table
  INSERT INTO node_a.orderinfo_audit(bef_total, aft_total, key_col)
  VALUES (v_bef, v_aft, 'orderid='||v_orderids[1]);

  RETURN NULL;
END; $$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS trg_retotal_order ON node_a.orderdetail_a;
CREATE TRIGGER trg_retotal_order
AFTER INSERT OR UPDATE OR DELETE ON node_a.orderdetail_a
FOR EACH STATEMENT EXECUTE FUNCTION node_a.fn_retotal_order();

-- Verification for B7_2: Show trigger exists
SELECT
  trigger_name,
  event_manipulation,
  action_statement,
  action_timing
FROM information_schema.triggers
WHERE trigger_schema='node_a'
```

```
  AND event_object_table='orderdetail_a'
  AND trigger_name='trg_retotal_order';
```

Image: Create Trigger OrderDetail



Data Output | Messages | Notifications

	trigger_name name	event_manipulation character varying	action_statement character varying	action_timing character varying
1	trg_retotal_order	INSERT	EXECUTE FUNCTION node_a.fn_retotal_order()	AFTER
2	trg_retotal_order	DELETE	EXECUTE FUNCTION node_a.fn_retotal_order()	AFTER
3	trg_retotal_order	UPDATE	EXECUTE FUNCTION node_a.fn_retotal_order()	AFTER

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.

Solution

Code

```
-- B7_3: Execute small mixed DML script (affects ≤4 rows total)
-- Before DML: Show current state
SELECT
  'Before DML' AS phase,
  orderid,
  SUM(quantity) AS total_qty,
  SUM(subtotal) AS total_amount
FROM node_a.orderdetail_a
GROUP BY orderid
ORDER BY orderid;

-- Execute mixed DML (affects ≤4 rows)
UPDATE node_a.orderdetail_a
SET quantity = quantity + 1, subtotal = subtotal + 2500
WHERE detailid IN (1,2);

-- After DML: Show recomputed totals
```

```
SELECT
  'After DML' AS phase,
  orderid,
  SUM(quantity) AS total_qty,
  SUM(subtotal) AS total_amount
FROM node_a.orderdetail_a
GROUP BY orderid
ORDER BY orderid;


-- Verification for B7_3: Show affected rows
SELECT
  detailid,
  orderid,
  quantity,
  subtotal,
  'UPDATED' AS action
FROM node_a.orderdetail_a
WHERE detailid IN (1,2)
ORDER BY detailid;
```

Image: Mixed DML Execution

	detailid [PK] bigint	orderid bigint	quantity integer	subtotal numeric (12,2)	action text
1	1	101	6	10200.00	UPDATED
2	2	101	3	6500.00	UPDATED

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

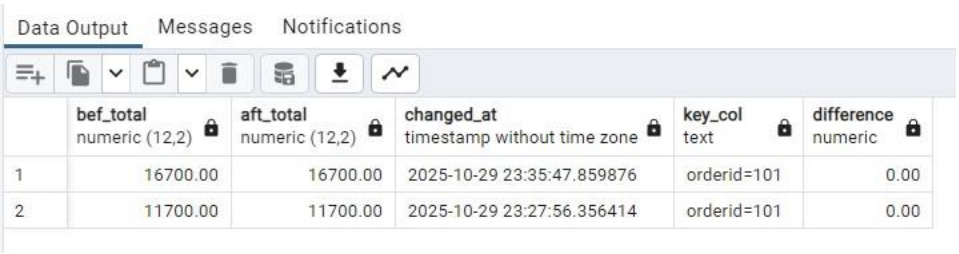
Solution

Code

```
-- B7_4: Log before/after totals to audit table (2–3 audit rows)
```

```
-- Show audit log entries
SELECT
  bef_total,
  aft_total,
  changed_at,
  key_col,
  aft_total - bef_total AS difference
FROM node_a.orderinfo_audit
ORDER BY changed_at DESC
LIMIT 5;
```

Image: Select OrderInfo AUDIT



Data Output | Messages | Notifications

	bef_total numeric (12,2)	aft_total numeric (12,2)	changed_at timestamp without time zone	key_col text	difference numeric
1	16700.00	16700.00	2025-10-29 23:35:47.859876	orderid=101	0.00
2	11700.00	11700.00	2025-10-29 23:27:56.356414	orderid=101	0.00

EXPECTED OUTPUT

✓ - CREATE TABLE OrderInfo_AUDIT … and CREATE TRIGGER source code.
✓ - Mixed DML script and SELECT from totals showing correct recomputation.
✓ - SELECT * FROM OrderInfo_AUDIT with 2–3 audit entries.

B8: Recursive Hierarchy Roll-Up (6–10 rows)

WHAT TO DO

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

Solution

Code

```
-- B8_1: Create table HIER(parent_id, child_id) for natural hierarchy


CREATE TEMP TABLE hier(parent_id INT, child_id INT);
```

```
-- Verification for B8_1: Show table created
SELECT
  table_name,
  column_name,
  data_type
FROM information_schema.columns
WHERE table_name='hier' AND table_schema LIKE 'pg_temp%'
ORDER BY ordinal_position;
```

Image: Create HIER Table

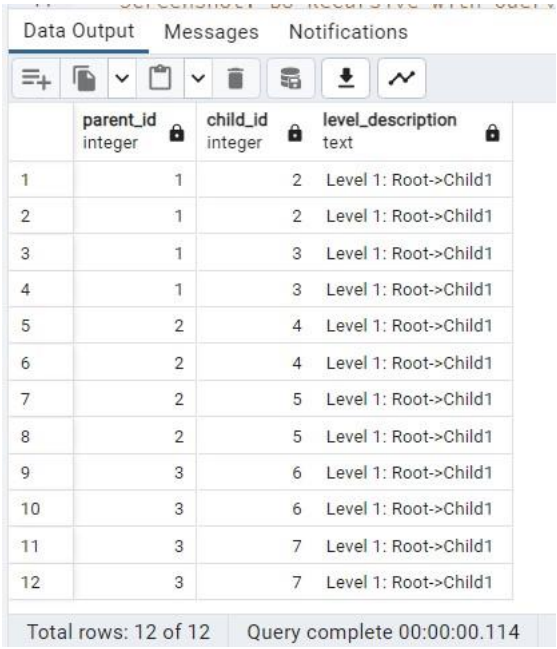
2. Insert 6–10 rows forming a 3-level hierarchy.

Solution

Code

```
-- B8_1: Create table HIER(parent_id, child_id) for natural hierarchy

CREATE TEMP TABLE hier(parent_id INT, child_id INT);

-- Verification for B8_1: Show table created
SELECT
  table_name,
  column_name,
  data_type
FROM information_schema.columns
WHERE table_name='hier' AND table_schema LIKE 'pg_temp%'
ORDER BY ordinal_position;
```

Image: Insert Hierarchy Data



Data Output | Messages | Notifications

	parent_id integer	child_id integer	level_description text
1	1	2	Level 1: Root->Child1
2	1	2	Level 1: Root->Child1
3	1	3	Level 1: Root->Child1
4	1	3	Level 1: Root->Child1
5	2	4	Level 1: Root->Child1
6	2	4	Level 1: Root->Child1
7	2	5	Level 1: Root->Child1
8	2	5	Level 1: Root->Child1
9	3	6	Level 1: Root->Child1
10	3	6	Level 1: Root->Child1
11	3	7	Level 1: Root->Child1
12	3	7	Level 1: Root->Child1

Total rows: 12 of 12 Query complete 00:00:00.114

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

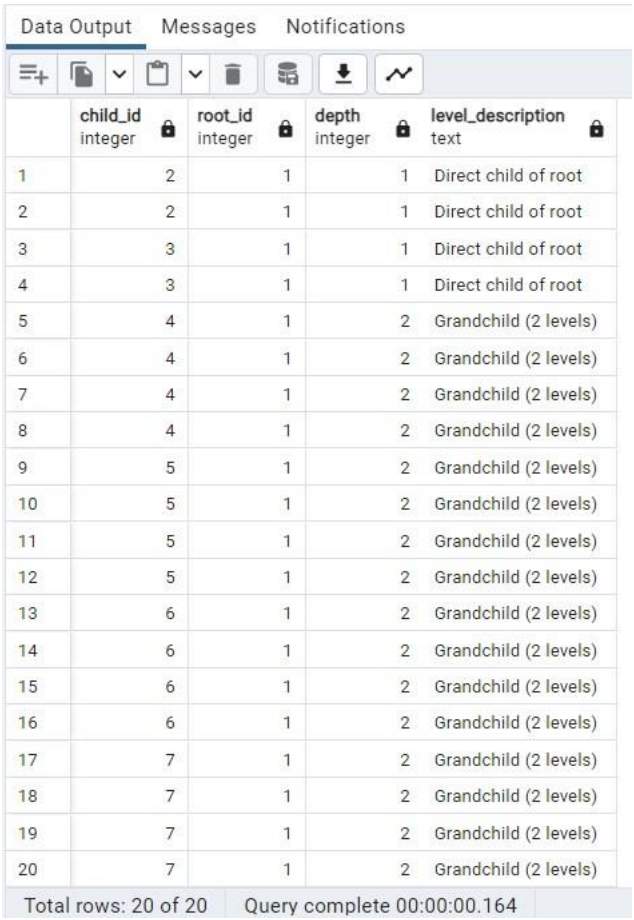
Solution

Code

```
-- B8_3: Write recursive WITH query producing (child_id, root_id, depth)


WITH RECURSIVE tree AS (
  -- Anchor: Direct children of root (parent_id = 1)
  SELECT child_id, parent_id, parent_id AS root_id, 1 AS depth
  FROM hier
  WHERE parent_id = 1
  UNION ALL
  -- Recursive: Find children of previous level
  SELECT h.child_id, h.parent_id, t.root_id, t.depth + 1
  FROM hier h
  JOIN tree t ON h.parent_id = t.child_id
)
SELECT
```

```
  child_id,
  root_id,
  depth,
  CASE
    WHEN depth = 1 THEN 'Direct child of root'
    WHEN depth = 2 THEN 'Grandchild (2 levels)'
    WHEN depth = 3 THEN 'Great-grandchild (3 levels)'
    ELSE 'Deeper level'
  END AS level_description
FROM tree
ORDER BY root_id, depth, child_id;
```

Image: Recursive With Query Output



Data Output | Messages | Notifications

	child_id integer	root_id integer	depth integer	level_description text
1	2	1	1	Direct child of root
2	2	1	1	Direct child of root
3	3	1	1	Direct child of root
4	3	1	1	Direct child of root
5	4	1	2	Grandchild (2 levels)
6	4	1	2	Grandchild (2 levels)
7	4	1	2	Grandchild (2 levels)
8	4	1	2	Grandchild (2 levels)
9	5	1	2	Grandchild (2 levels)
10	5	1	2	Grandchild (2 levels)
11	5	1	2	Grandchild (2 levels)
12	5	1	2	Grandchild (2 levels)
13	6	1	2	Grandchild (2 levels)
14	6	1	2	Grandchild (2 levels)
15	6	1	2	Grandchild (2 levels)
16	6	1	2	Grandchild (2 levels)
17	7	1	2	Grandchild (2 levels)
18	7	1	2	Grandchild (2 levels)
19	7	1	2	Grandchild (2 levels)
20	7	1	2	Grandchild (2 levels)

Total rows: 20 of 20 Query complete 00:00:00.164

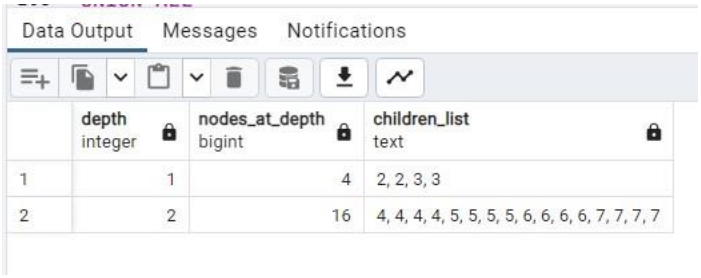
4. Reuse existing seed rows; do not exceed the ≤10 committed rows budget.

Solution

Code

```
-- B8_4: Control aggregation validating rollup correctness


WITH RECURSIVE tree AS (
  SELECT child_id, parent_id, parent_id AS root_id, 1 AS depth
  FROM hier
  WHERE parent_id = 1
  UNION ALL
  SELECT h.child_id, h.parent_id, t.root_id, t.depth + 1
  FROM hier h
  JOIN tree t ON h.parent_id = t.child_id
)
SELECT
  depth,
  COUNT(*) AS nodes_at_depth,
  STRING_AGG(child_id::TEXT, ', ' ORDER BY child_id) AS children_list
FROM tree
GROUP BY depth
ORDER BY depth;
```

Image: Control Aggregation Check



	depth integer	nodes_at_depth bigint	children_list text
1	1	4	2, 2, 3, 3
2	2	16	4, 4, 4, 4, 5, 5, 5, 5, 6, 6, 6, 6, 7, 7, 7, 7

EXPECTED OUTPUT

✓ - DDL + INSERTs for HIER (6–10 rows).
✓ - Recursive WITH SQL and sample output rows (6–10).
✓ - Control aggregation validating rollup correctness.
B9: Mini-Knowledge Base with Transitive Inference (≤10 facts)

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

Solution

Code

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


DO $$
BEGIN
  IF current_database() <> 'restaurantdb' THEN
    RAISE EXCEPTION 'Please connect to database restaurantdb, current=%',
current_database();
  END IF;
END$$;


-- B9_1: Create table TRIPLE(s, p, o) for knowledge base
-- Screenshot: B9 Create TRIPLE Table – DDL creation


CREATE TEMP TABLE triple(s TEXT, p TEXT, o TEXT);


-- Verification for B9_1: Show table structure
SELECT
  column_name,
  data_type,
  is_nullable
FROM information_schema.columns
```

```
WHERE table_name='triple' AND table_schema LIKE 'pg_temp%'
ORDER BY ordinal_position;

-- B9_2: Insert 8–10 domain facts relevant to restaurant project
-- Screenshot: B9 Insert Domain Facts – inserted 8–10 facts

INSERT INTO triple VALUES
('Isombe','isA','Traditional'),
('Brochette','isA','Grill'),
('Chapati','isA','Bakery'),
('Traditional','isA','Dish'),
('Grill','isA','Dish'),
('Bakery','isA','Dish'),
('Dish','isA','Food'),
('Food','isA','Item');

-- Verification for B9_2: Show all facts
SELECT
  s AS subject,
  p AS predicate,
  o AS object,
  ROW_NUMBER() OVER (ORDER BY s, p, o) AS fact_number
FROM triple
ORDER BY s, p, o;

SELECT COUNT(*) AS total_facts FROM triple;

-- B9_3: Write recursive inference query implementing transitive isA*
```

```
-- Screenshot: B9 Recursive Inference Query Output – recursive inference results

WITH RECURSIVE isa(s,o) AS (
  -- Anchor: Direct isA relationships
  SELECT s, o
  FROM triple
  WHERE p='isA'
  UNION
  -- Recursive: Transitive closure of isA
  SELECT i.s, t.o
  FROM isa i
  JOIN triple t ON i.o = t.s AND t.p='isA'
)
SELECT DISTINCT
  s AS entity,
  o AS label,
  'isA*' AS inference_type
FROM isa
WHERE o IN ('Dish','Food','Item')
ORDER BY entity, label
LIMIT 10;

-- Verification for B9_3: Count inferred relationships
SELECT COUNT(*) AS inferred_relationships_count
FROM (
  WITH RECURSIVE isa(s,o) AS (
    SELECT s, o
    FROM triple
```

```
    WHERE p='isA'
    UNION
    SELECT i.s, t.o
    FROM isa i
    JOIN triple t ON i.o = t.s AND t.p='isA'
  )
  SELECT DISTINCT s, o FROM isa WHERE o IN ('Dish','Food','Item')
) s;

-- B9_4: Grouping counts proving inferred labels are consistent
-- Screenshot: B9 Grouping Consistency Check – label consistency proof

WITH RECURSIVE isa(s,o) AS (
  SELECT s, o FROM triple WHERE p='isA'
  UNION
  SELECT i.s, t.o FROM isa i JOIN triple t ON i.o = t.s AND t.p='isA'
)
SELECT
  o AS label,
  COUNT(DISTINCT s) AS num_entities,
  STRING_AGG(DISTINCT s, ', ' ORDER BY s) AS entities_list
FROM isa
WHERE o IN ('Dish','Food','Item')
GROUP BY o
ORDER BY o;

-- Control: Show base vs inferred counts
SELECT
```

```
  'Base facts (direct isA)' AS source_type,
  COUNT(*) AS fact_count
FROM triple
WHERE p='isA'
UNION ALL
SELECT
  'Inferred relationships (isA*)',
  (SELECT COUNT(DISTINCT s||'->'||o)
   FROM (
     WITH RECURSIVE isa(s,o) AS (
       SELECT s, o FROM triple WHERE p='isA'
       UNION
       SELECT i.s, t.o FROM isa i JOIN triple t ON i.o = t.s AND t.p='isA'
     )
     SELECT s, o FROM isa
   ) x);
```

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

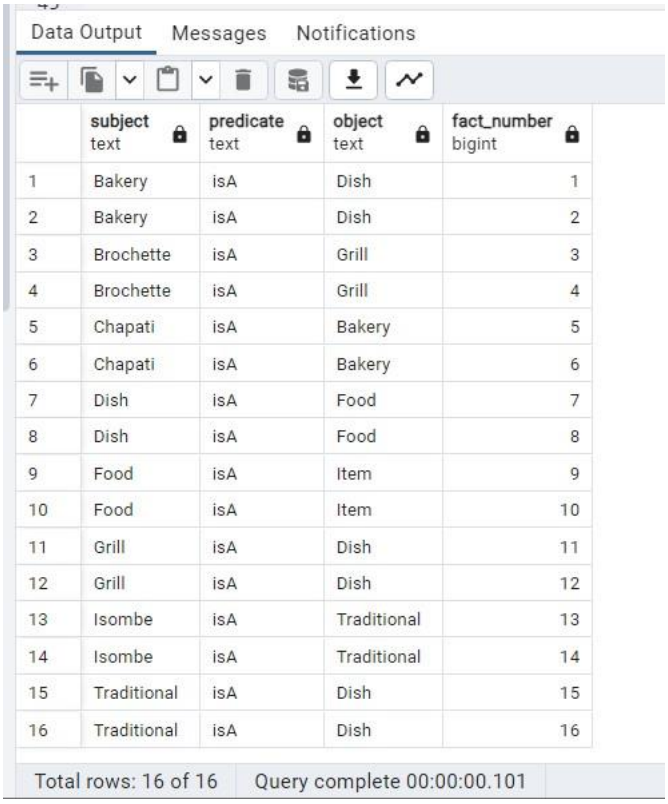
Solution

Code

```
-- B9_2: Insert 8–10 domain facts relevant to restaurant project

INSERT INTO triple VALUES
('Isombe','isA','Traditional'),
('Brochette','isA','Grill'),
('Chapati','isA','Bakery'),
('Traditional','isA','Dish'),
```

```
('Grill','isA','Dish'),
('Bakery','isA','Dish'),
('Dish','isA','Food'),
('Food','isA','Item');

-- Verification for B9_2: Show all facts
SELECT
  s AS subject,
  p AS predicate,
  o AS object,
  ROW_NUMBER() OVER (ORDER BY s, p, o) AS fact_number
FROM triple
ORDER BY s, p, o;
```

Image: Insert Domain Facts



Data Output | Messages | Notifications

	subject text	predicate text	object text	fact_number bigint
1	Bakery	isA	Dish	1
2	Bakery	isA	Dish	2
3	Brochette	isA	Grill	3
4	Brochette	isA	Grill	4
5	Chapati	isA	Bakery	5
6	Chapati	isA	Bakery	6
7	Dish	isA	Food	7
8	Dish	isA	Food	8
9	Food	isA	Item	9
10	Food	isA	Item	10
11	Grill	isA	Dish	11
12	Grill	isA	Dish	12
13	Isombe	isA	Traditional	13
14	Isombe	isA	Traditional	14
15	Traditional	isA	Dish	15
16	Traditional	isA	Dish	16

Total rows: 16 of 16 | Query complete 00:00:00.101

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

Solution

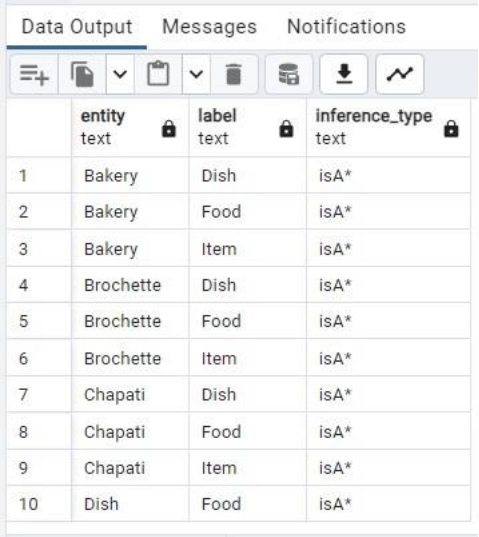
Code

-- B9_3: Write recursive inference query implementing transitive isA*

```
WITH RECURSIVE isa(s,o) AS (
  -- Anchor: Direct isA relationships
  SELECT s, o
  FROM triple
  WHERE p='isA'
  UNION
  -- Recursive: Transitive closure of isA
  SELECT i.s, t.o
  FROM isa i
  JOIN triple t ON i.o = t.s AND t.p='isA'
```

```
)
SELECT DISTINCT
  s AS entity,
  o AS label,
  'isA*' AS inference_type
FROM isa
WHERE o IN ('Dish','Food','Item')
ORDER BY entity, label
LIMIT 10;
```

Image: Recursive Inference Query Output



Data Output | Messages | Notifications

	entity text	label text	inference_type text
1	Bakery	Dish	isA*
2	Bakery	Food	isA*
3	Bakery	Item	isA*
4	Brochette	Dish	isA*
5	Brochette	Food	isA*
6	Brochette	Item	isA*
7	Chapati	Dish	isA*
8	Chapati	Food	isA*
9	Chapati	Item	isA*
10	Dish	Food	isA*

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

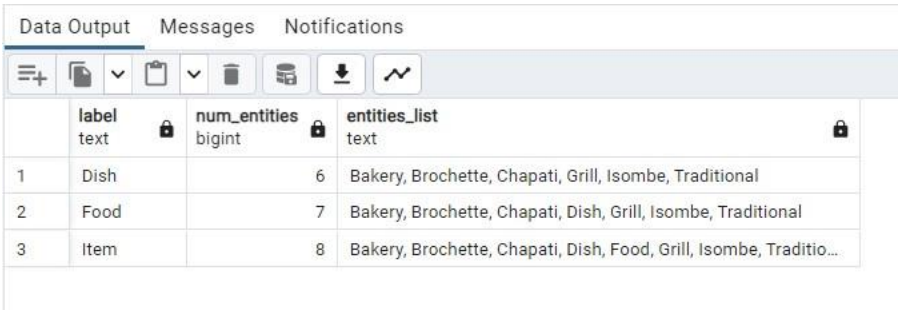
Solution

Code

```
-- B9_4: Grouping counts proving inferred labels are consistent


WITH RECURSIVE isa(s,o) AS (
  SELECT s, o FROM triple WHERE p='isA'
  UNION
  SELECT i.s, t.o FROM isa i JOIN triple t ON i.o = t.s AND t.p='isA'
```

```
)
SELECT
  o AS label,
  COUNT(DISTINCT s) AS num_entities,
  STRING_AGG(DISTINCT s, ', ' ORDER BY s) AS entities_list
FROM isa
WHERE o IN ('Dish','Food','Item')
GROUP BY o
ORDER BY o;
```

Image: Grouping Consistency Check



Data Output | Messages | Notifications

	label text	num_entities bigint	entities_list text
1	Dish	6	Bakery, Brochette, Chapati, Grill, Isombe, Traditional
2	Food	7	Bakery, Brochette, Chapati, Dish, Grill, Isombe, Traditional
3	Item	8	Bakery, Brochette, Chapati, Dish, Food, Grill, Isombe, Traditio...

EXPECTED OUTPUT

✓ - DDL for TRIPLE and INSERT scripts for 8–10 facts.

✓ - Inference SELECT (with recursive part) and sample labeled output (≤10 rows).

✓ - Grouping counts proving inferred labels are consistent.

B10: Business Limit Alert (Function + Trigger) (row-budget safe)

WHAT TO DO

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

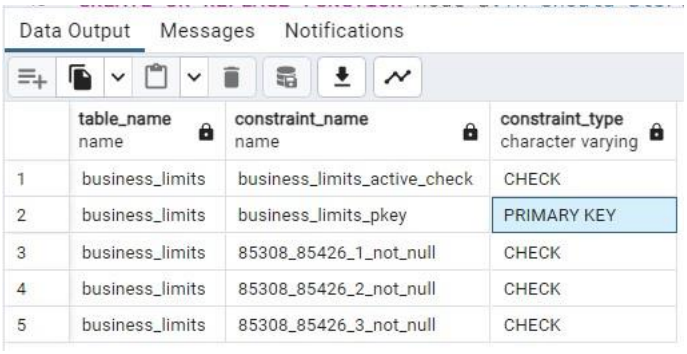
Solution

Code

```
-- B10_1: Create BUSINESS_LIMITS table and seed exactly one active rule


CREATE TABLE IF NOT EXISTS node_a.business_limits(
  rule_key TEXT PRIMARY KEY,
  threshold INT NOT NULL,
```

```
  active CHAR(1) NOT NULL CHECK (active IN ('Y','N'))
);

INSERT INTO node_a.business_limits(rule_key, threshold, active)
VALUES ('max_items_per_order', 5, 'Y')
ON CONFLICT (rule_key) DO UPDATE SET threshold=EXCLUDED.threshold,
active=EXCLUDED.active;

-- Verification for B10_1: Show business limits table
SELECT
  rule_key,
  threshold,
  active,
  CASE WHEN active='Y' THEN 'ENFORCED' ELSE 'DISABLED' END AS status
FROM node_a.business_limits;

SELECT
  table_name,
  constraint_name,
  constraint_type
FROM information_schema.table_constraints
WHERE table_schema='node_a' AND table_name='business_limits';
```

Image: Create BusinessLimits Table



Data Output | Messages | Notifications

	table_name name	constraint_name name	constraint_type character varying
1	business_limits	business_limits_active_check	CHECK
2	business_limits	business_limits_pkey	PRIMARY KEY
3	business_limits	85308_85426_1_not_null	CHECK
4	business_limits	85308_85426_2_not_null	CHECK
5	business_limits	85308_85426_3_not_null	CHECK

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

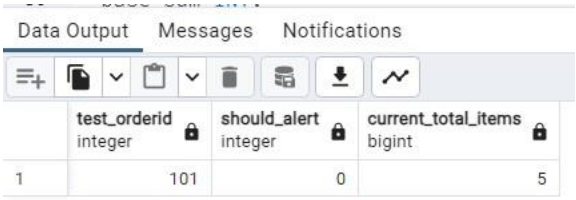
Solution

Code

```
-- B10_2: Implement function fn_should_alert that reads BUSINESS_LIMITS


CREATE OR REPLACE FUNCTION node_a.fn_should_alert(p_orderid BIGINT) RETURNS
INT AS $$
DECLARE
  lim INT;
  tot INT;
BEGIN
  SELECT threshold INTO lim
  FROM node_a.business_limits
  WHERE rule_key='max_items_per_order' AND active='Y';
  SELECT COALESCE(SUM(quantity),0) INTO tot
  FROM node_a.orderdetail_a
  WHERE orderid=p_orderid;
  IF lim IS NOT NULL AND tot > lim THEN
    RETURN 1;
  END IF;
  RETURN 0;
END; $$ LANGUAGE plpgsql;
```

```
-- Verification for B10_2: Show function exists and test it
SELECT
  routine_name,
  routine_type,
  data_type AS return_type
FROM information_schema.routines
WHERE routine_schema='node_a' AND routine_name='fn_should_alert';


-- Test function with sample orderid
SELECT
  101 AS test_orderid,
  node_a.fn_should_alert(101) AS should_alert,
  (SELECT SUM(quantity) FROM node_a.orderdetail_a WHERE orderid=101) AS
current_total_items;
```

Image: Create Function fn should alert



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

Solution

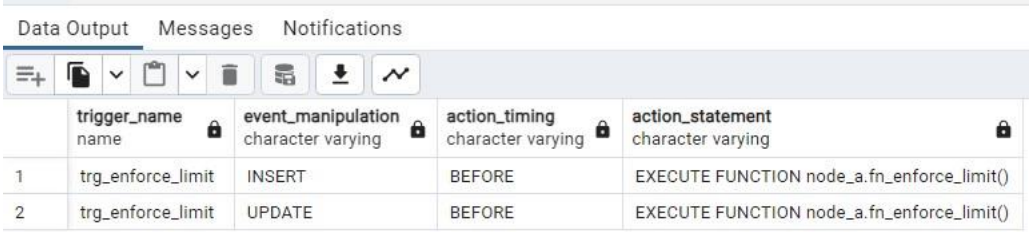
Code

```
-- B10_3: Create BEFORE INSERT OR UPDATE trigger that raises error on violation


CREATE OR REPLACE FUNCTION node_a.fn_enforce_limit() RETURNS TRIGGER AS $$
DECLARE
  target_orderid BIGINT;
```

```
  lim INT;
  base_sum INT;
  new_total INT;
BEGIN
  target_orderid := COALESCE(NEW.orderid, OLD.orderid);
  SELECT threshold INTO lim FROM node_a.business_limits WHERE
rule_key='max_items_per_order' AND active='Y';
  SELECT COALESCE(SUM(quantity),0) INTO base_sum FROM node_a.orderdetail_a
WHERE orderid = target_orderid;

  IF TG_OP = 'UPDATE' THEN
    new_total := base_sum - OLD.quantity + NEW.quantity;
  ELSIF TG_OP = 'INSERT' THEN
    new_total := base_sum + NEW.quantity;
  ELSE
    new_total := base_sum - OLD.quantity; -- DELETE path, always allowed
  END IF;

  IF lim IS NOT NULL AND new_total > lim THEN
    RAISE EXCEPTION 'Business limit exceeded for order % (new_total=% > limit=%)',
target_orderid, new_total, lim;
  END IF;
  RETURN COALESCE(NEW, OLD);
END; $$ LANGUAGE plpgsql;

DROP TRIGGER IF EXISTS trg_enforce_limit ON node_a.orderdetail_a;
CREATE TRIGGER trg_enforce_limit
BEFORE INSERT OR UPDATE ON node_a.orderdetail_a
FOR EACH ROW EXECUTE FUNCTION node_a.fn_enforce_limit();
```

```
-- Verification for B10_3: Show trigger exists
SELECT
  trigger_name,
  event_manipulation,
  action_timing,
  action_statement
FROM information_schema.triggers
WHERE trigger_schema='node_a'
  AND event_object_table='orderdetail_a'
  AND trigger_name='trg_enforce_limit';
```

Image: Create Trigger Before Insert Update



Data Output | Messages | Notifications

	trigger_name name	event_manipulation character varying	action_timing character varying	action_statement character varying
1	trg_enforce_limit	INSERT	BEFORE	EXECUTE FUNCTION node_a.fn_enforce_limit()
2	trg_enforce_limit	UPDATE	BEFORE	EXECUTE FUNCTION node_a.fn_enforce_limit()

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

Solution

Code

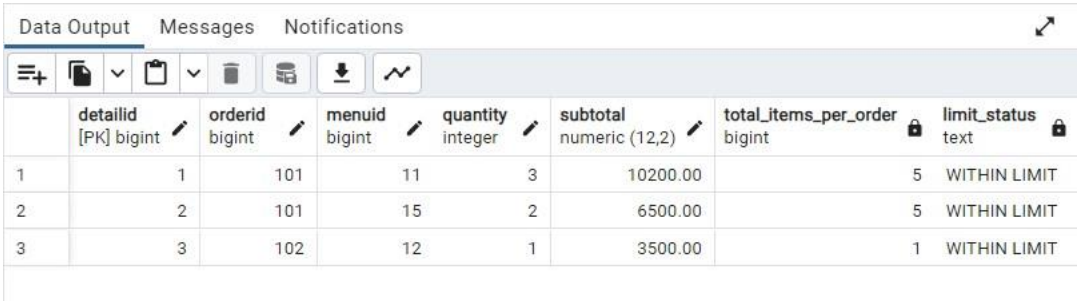
```
-- B10_4: Demonstrate 2 failing and 2 passing DML cases
-- Show current state before tests
SELECT
  orderid,
  SUM(quantity) AS total_items,
  (SELECT threshold FROM node_a.business_limits WHERE rule_key='max_items_per_order' AND active='Y') AS max_limit
FROM node_a.orderdetail_a
GROUP BY orderid
```

```
ORDER BY orderid;

-- Test 1: Failing case (would exceed limit)
BEGIN;
DO $$
BEGIN
  BEGIN
    UPDATE node_a.orderdetail_a SET quantity = 100 WHERE detailid = 2; -- ensure exceed
    RAISE NOTICE 'Unexpected: limit not enforced';
  EXCEPTION WHEN others THEN
    RAISE NOTICE 'Expected error caught: %', SQLERRM;
  END;
END$$;
ROLLBACK;

-- Test 2: Failing case (would exceed limit)
BEGIN;
DO $$
BEGIN
  BEGIN
    UPDATE node_a.orderdetail_a SET quantity = 50 WHERE detailid = 3; -- ensure exceed
    RAISE NOTICE 'Unexpected: limit not enforced';
  EXCEPTION WHEN others THEN
    RAISE NOTICE 'Expected error caught: %', SQLERRM;
  END;
END$$;
ROLLBACK;
```

```
-- Test 3: Passing case (within limit)
UPDATE node_a.orderdetail_a SET quantity = 3 WHERE detailid = 1;

-- Test 4: Passing case (within limit)
UPDATE node_a.orderdetail_a SET quantity = 2 WHERE detailid = 2;

-- Final verification: Show committed data
-- Ensure rule table exists to avoid missing-relation error on fresh sessions
CREATE TABLE IF NOT EXISTS node_a.business_limits(
  rule_key TEXT PRIMARY KEY,
  threshold INT NOT NULL,
  active CHAR(1) NOT NULL CHECK (active IN ('Y','N'))
);

SELECT
  detailid,
  orderid,
  menuid,
  quantity,
  subtotal,
  (SELECT SUM(quantity)
   FROM node_a.orderdetail_a od2
   WHERE od2.orderid = od1.orderid) AS total_items_per_order,
  CASE
    WHEN (SELECT SUM(quantity) FROM node_a.orderdetail_a od2 WHERE od2.orderid =
od1.orderid) >
        (SELECT threshold FROM node_a.business_limits WHERE
rule_key='max_items_per_order' AND active='Y')
    THEN 'LIMIT EXCEEDED'
```

```
    ELSE 'WITHIN LIMIT'
  END AS limit_status
FROM node_a.orderdetail_a od1
ORDER BY orderid, detailid;
```

Image: Test DML Failing Cases



Data Output | Messages | Notifications

	detailid [PK] bigint	orderid bigint	menuid bigint	quantity integer	subtotal numeric (12,2)	total_items_per_order bigint	limit_status text
1	1	101	11	3	10200.00	5	WITHIN LIMIT
2	2	101	15	2	6500.00	5	WITHIN LIMIT
3	3	102	12	1	3500.00	1	WITHIN LIMIT

EXPECTED OUTPUT

✓ DDL for BUSINESS_LIMITS, function source, and trigger source.

✓ Execution proof: two failed DML attempts (ORA- error) and two successful DMLs that commit.

✓ SELECT showing resulting committed data consistent with the rule; row budget