STUDENT 26: INSURANCE CLAIM & POLICY

==

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

WHAT TO DO

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

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.

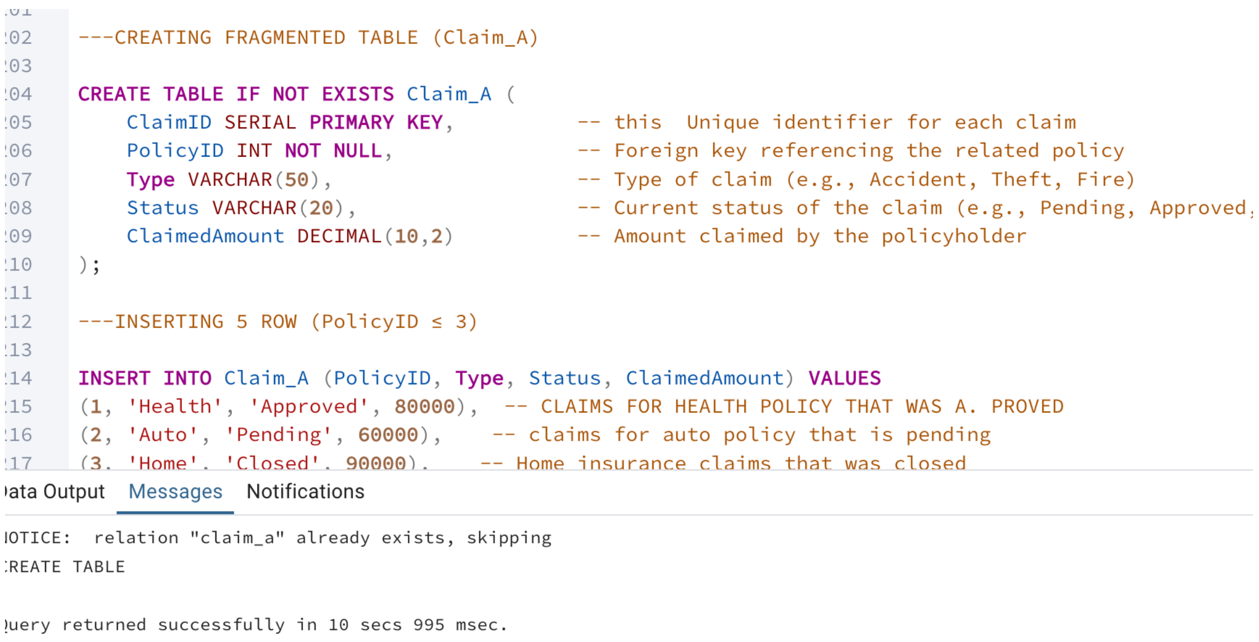
3. On Node_A, create view Claim_ALL as UNION ALL of Claim_A and Claim_B@proj_link.

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

EXPECTED OUTPUT

- ✓ - DDL for Claim_A and Claim_B; population scripts with ≤10 total committed rows.
- ✓ - CREATE DATABASE LINK proj_link ... (shown).
- ✓ - CREATE VIEW Claim_ALL … UNION ALL … (shown).
- ✓ - Matching COUNT(*) and checksum between fragments vs Claim_ALL (evidence screenshot).

A1.1a

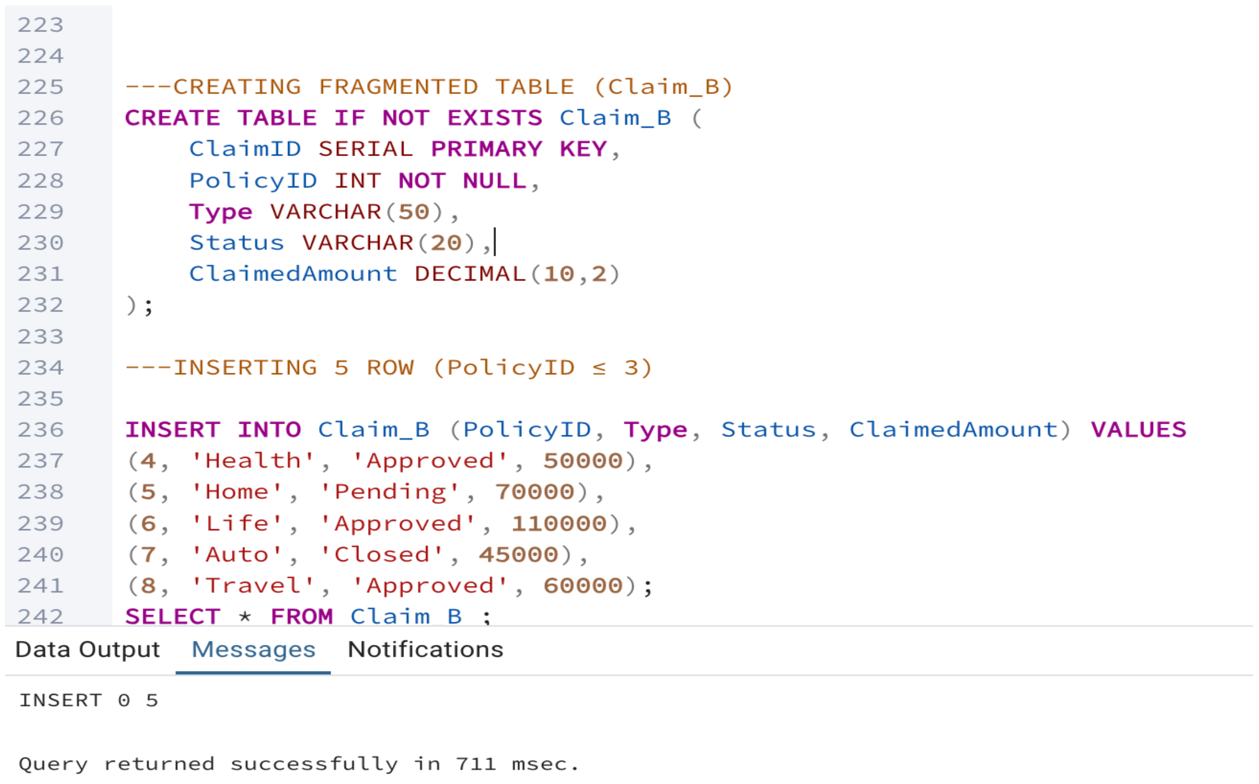


```
---CREATING FRAGMENTED TABLE (Claim_A)

CREATE TABLE IF NOT EXISTS Claim_A (
    ClaimID SERIAL PRIMARY KEY,          -- this  Unique identifier for each claim
    PolicyID INT NOT NULL,               -- Foreign key referencing the related policy
    Type VARCHAR(50),                    -- Type of claim (e.g., Accident, Theft, Fire)
    Status VARCHAR(20),                  -- Current status of the claim (e.g., Pending, Approved,
    ClaimedAmount DECIMAL(10,2)          -- Amount claimed by the policyholder
);

---INSERTING 5 ROW (PolicyID ≤ 3)

INSERT INTO Claim_A (PolicyID, Type, Status, ClaimedAmount) VALUES
(1, 'Health', 'Approved', 80000),  -- CLAIMS FOR HEALTH POLICY THAT WAS A. PROVED
(2, 'Auto', 'Pending', 60000),    -- claims for auto policy that is pending
(3, 'Home', 'Closed', 90000),    -- Home insurance claims that was closed
```

ata Output   Messages   Notifications

```
OTICE:  relation "claim_a" already exists, skipping
REATE TABLE

uery returned successfully in 10 secs 995 msec.
```

A1.1b

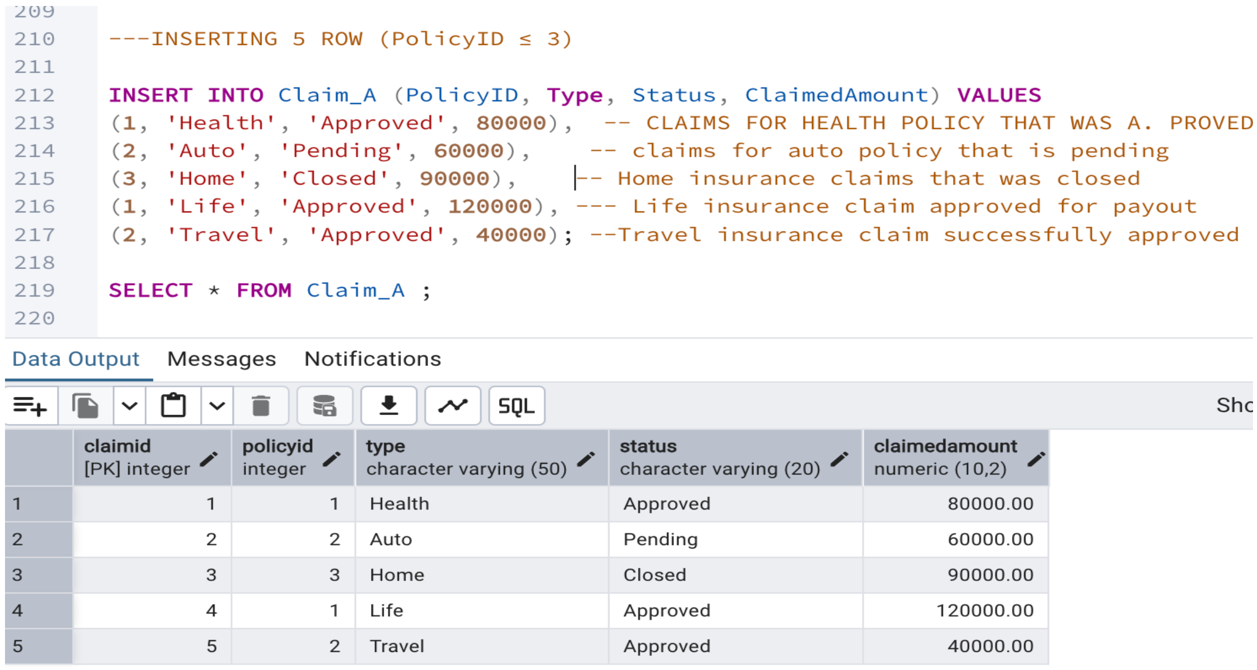


```
---CREATING FRAGMENTED TABLE (Claim_B)
CREATE TABLE IF NOT EXISTS Claim_B (
    ClaimID SERIAL PRIMARY KEY,
    PolicyID INT NOT NULL,
    Type VARCHAR(50),
    Status VARCHAR(20),
    ClaimedAmount DECIMAL(10,2)
);

---INSERTING 5 ROW (PolicyID ≤ 3)

INSERT INTO Claim_B (PolicyID, Type, Status, ClaimedAmount) VALUES
(4, 'Health', 'Approved', 50000),
(5, 'Home', 'Pending', 70000),
(6, 'Life', 'Approved', 110000),
(7, 'Auto', 'Closed', 45000),
(8, 'Travel', 'Approved', 60000);
SELECT * FROM Claim_B ;
```

Data Output   Messages   Notifications

```
INSERT 0 5

Query returned successfully in 711 msec.
```

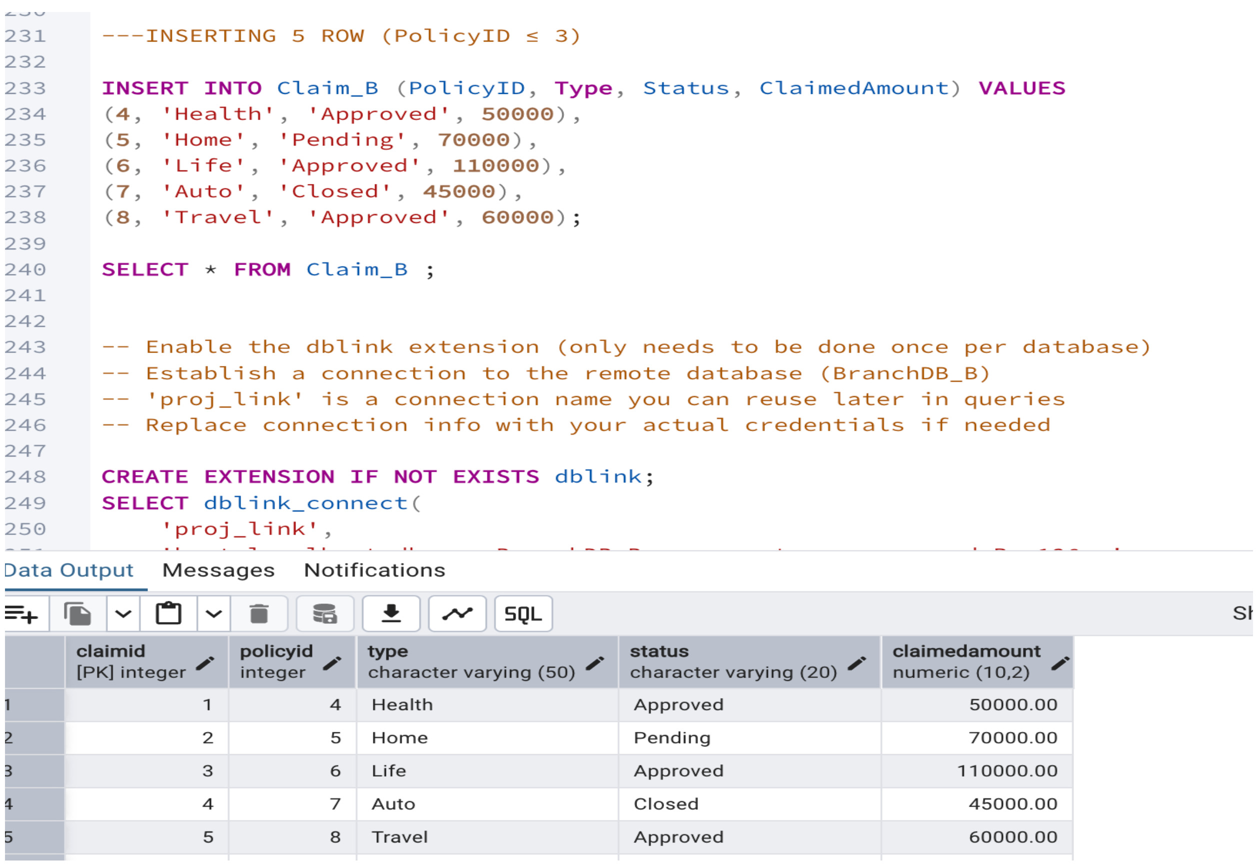
A1.2a



```sql
---INSERTING 5 ROW (PolicyID ≤ 3)

INSERT INTO Claim_A (PolicyID, Type, Status, ClaimedAmount) VALUES
(1, 'Health', 'Approved', 80000),   -- CLAIMS FOR HEALTH POLICY THAT WAS A. PROVED
(2, 'Auto', 'Pending', 60000),     -- claims for auto policy that is pending
(3, 'Home', 'Closed', 90000),     -- Home insurance claims that was closed
(1, 'Life', 'Approved', 120000), --- Life insurance claim approved for payout
(2, 'Travel', 'Approved', 40000); --Travel insurance claim successfully approved

SELECT * FROM Claim_A ;
```

	claimid [PK] integer	policyid integer	type character varying (50)	status character varying (20)	claimedamount numeric (10,2)
1	1	1	Health	Approved	80000.00
2	2	2	Auto	Pending	60000.00
3	3	3	Home	Closed	90000.00
4	4	1	Life	Approved	120000.00
5	5	2	Travel	Approved	40000.00

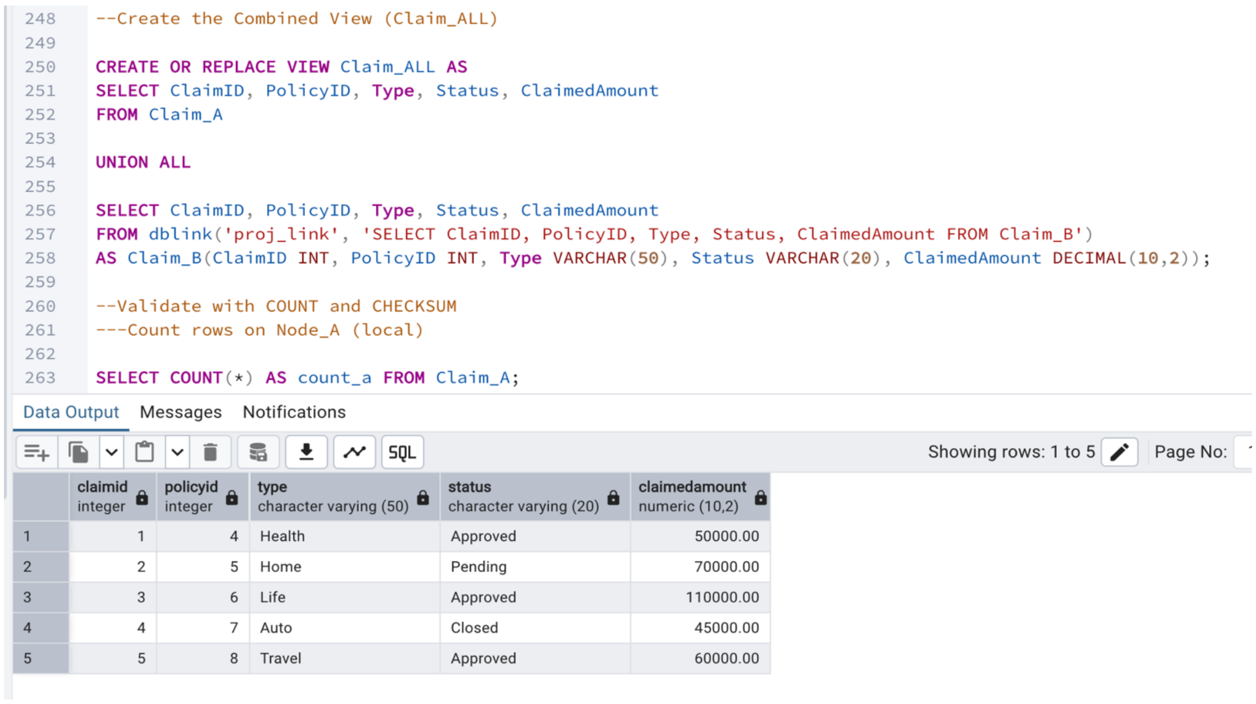
A1.2b



```sql
---INSERTING 5 ROW (PolicyID ≤ 3)

INSERT INTO Claim_B (PolicyID, Type, Status, ClaimedAmount) VALUES
(4, 'Health', 'Approved', 50000),
(5, 'Home', 'Pending', 70000),
(6, 'Life', 'Approved', 110000),
(7, 'Auto', 'Closed', 45000),
(8, 'Travel', 'Approved', 60000);

SELECT * FROM Claim_B ;


-- Enable the dblink extension (only needs to be done once per database)
-- Establish a connection to the remote database (BranchDB_B)
-- 'proj_link' is a connection name you can reuse later in queries
-- Replace connection info with your actual credentials if needed

CREATE EXTENSION IF NOT EXISTS dblink;
SELECT dblink_connect(
    'proj_link',
```

	claimid [PK] integer	policyid integer	type character varying (50)	status character varying (20)	claimedamount numeric (10,2)
1	1	4	Health	Approved	50000.00
2	2	5	Home	Pending	70000.00
3	3	6	Life	Approved	110000.00
4	4	7	Auto	Closed	45000.00
5	5	8	Travel	Approved	60000.00

A1.3



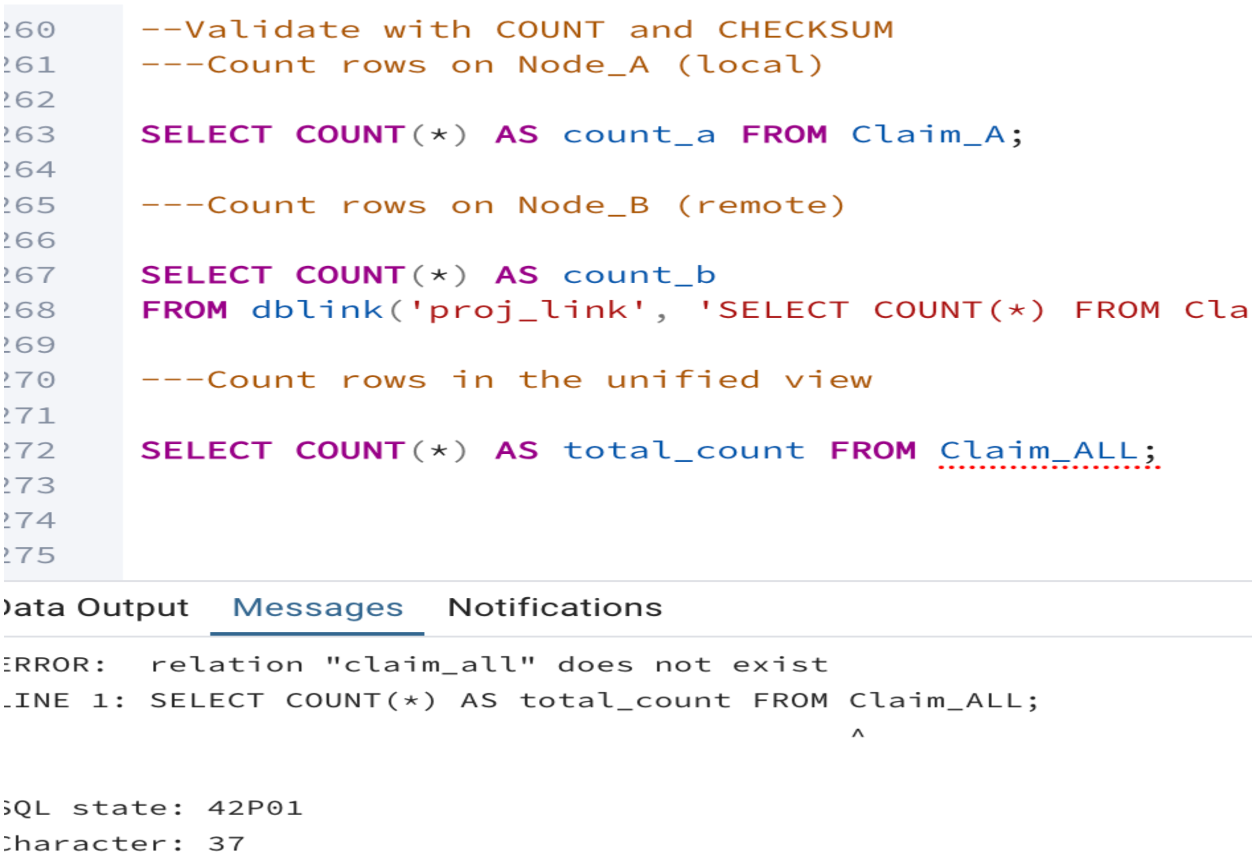
```
--Create the Combined View (Claim_ALL)

CREATE OR REPLACE VIEW Claim_ALL AS
SELECT ClaimID, PolicyID, Type, Status, ClaimedAmount
FROM Claim_A

UNION ALL

SELECT ClaimID, PolicyID, Type, Status, ClaimedAmount
FROM dblink('proj_link', 'SELECT ClaimID, PolicyID, Type, Status, ClaimedAmount FROM Claim_B')
AS Claim_B(ClaimID INT, PolicyID INT, Type VARCHAR(50), Status VARCHAR(20), ClaimedAmount DECIMAL(10,2));

--Validate with COUNT and CHECKSUM
---Count rows on Node_A (local)

SELECT COUNT(*) AS count_a FROM Claim_A;
```

Data Output Messages Notifications

Showing rows: 1 to 5 Page No:

	claimid integer	policyid integer	type character varying (50)	status character varying (20)	claimedamount numeric (10,2)
1	1	4	Health	Approved	50000.00
2	2	5	Home	Pending	70000.00
3	3	6	Life	Approved	110000.00
4	4	7	Auto	Closed	45000.00
5	5	8	Travel	Approved	60000.00

A1.4



```
--Validate with COUNT and CHECKSUM
---Count rows on Node_A (local)

SELECT COUNT(*) AS count_a FROM Claim_A;

---Count rows on Node_B (remote)

SELECT COUNT(*) AS count_b
FROM dblink('proj_link', 'SELECT COUNT(*) FROM Cla

---Count rows in the unified view

SELECT COUNT(*) AS total_count FROM Claim_ALL;
```

ata Output Messages Notifications

```
RROR:  relation "claim_all" does not exist
INE 1: SELECT COUNT(*) AS total_count FROM Claim_ALL;
                                           ^


QL state: 42P01
haracter: 37
```

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.

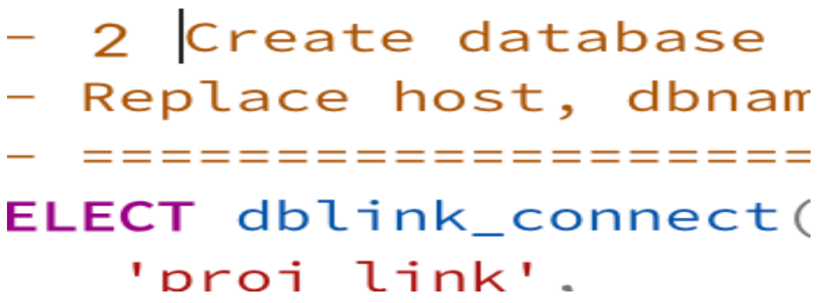
2. Run remote SELECT on Policy@proj_link showing up to 5 sample rows.

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

EXPECTED OUTPUT

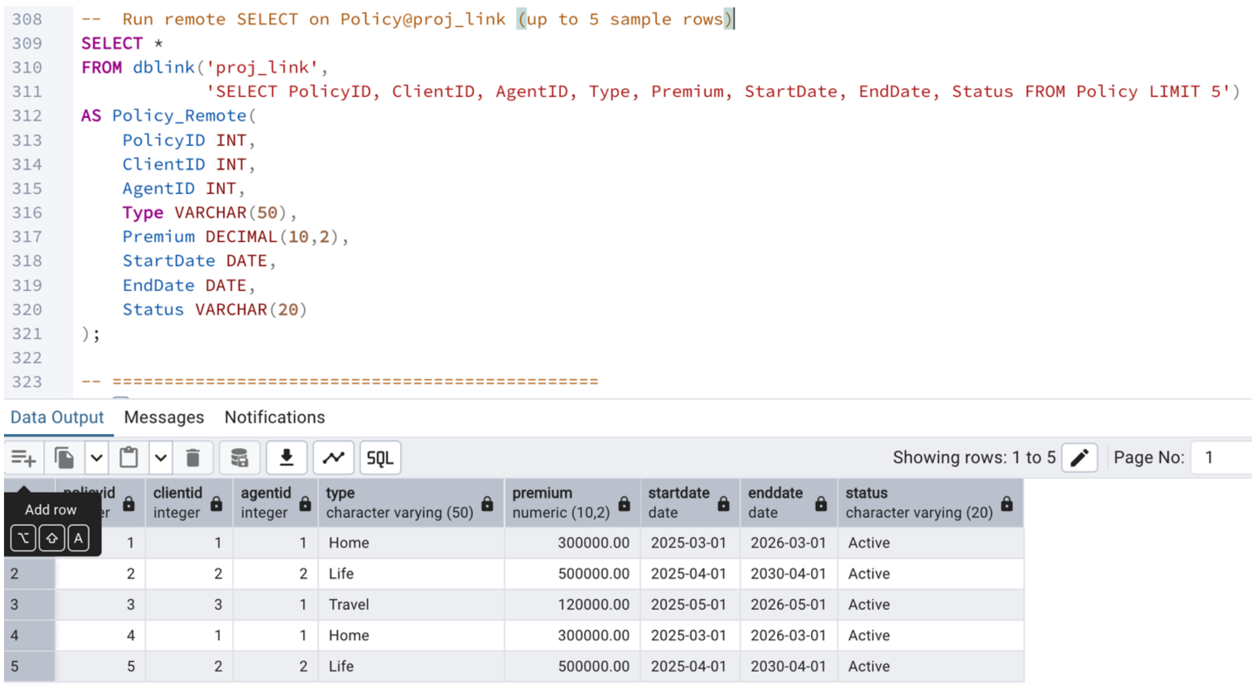
- ✓ - CREATE DATABASE LINK proj_link with connection details.
- ✓ - Screenshot of SELECT * FROM Policy@proj_link FETCH FIRST 5 ROWS ONLY.
- ✓ - Screenshot of distributed join on Claim ⋈ Agent@proj_link returning 3–10 rows.

A2.1



```
- 2 Create database
- Replace host, dbnam
- ====================
ELECT dblink_connect(
    'proj link'.
```

A2.2.



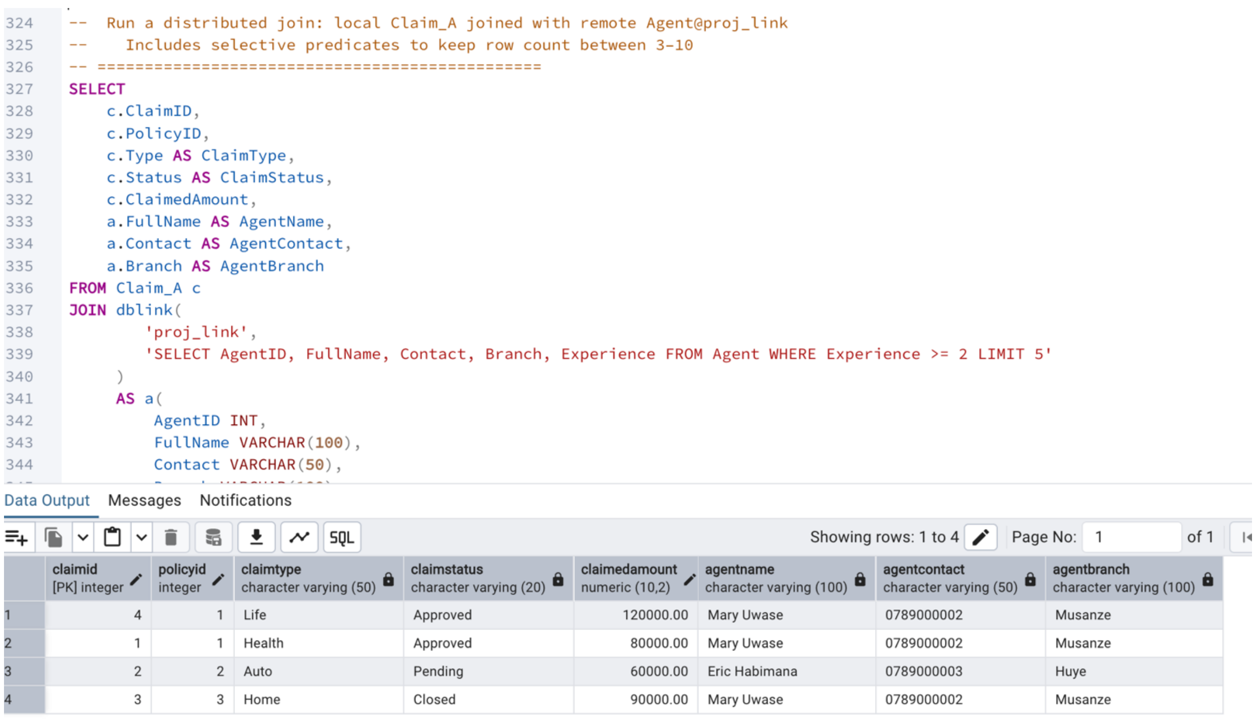
```sql
--  Run remote SELECT on Policy@proj_link (up to 5 sample rows)
SELECT *
FROM dblink('proj_link',
            'SELECT PolicyID, ClientID, AgentID, Type, Premium, StartDate, EndDate, Status FROM Policy LIMIT 5')
AS Policy_Remote(
    PolicyID INT,
    ClientID INT,
    AgentID INT,
    Type VARCHAR(50),
    Premium DECIMAL(10,2),
    StartDate DATE,
    EndDate DATE,
    Status VARCHAR(20)
);

-- ==============================================
```

Data Output Messages Notifications

Showing rows: 1 to 5 Page No: 1

	policyid integer	clientid integer	agentid integer	type character varying (50)	premium numeric (10,2)	startdate date	enddate date	status character varying (20)
	1	1	1	Home	300000.00	2025-03-01	2026-03-01	Active
2	2	2	2	Life	500000.00	2025-04-01	2030-04-01	Active
3	3	3	1	Travel	120000.00	2025-05-01	2026-05-01	Active
4	4	1	1	Home	300000.00	2025-03-01	2026-03-01	Active
5	5	2	2	Life	500000.00	2025-04-01	2030-04-01	Active

A2.3



```sql
--  Run a distributed join: local Claim_A joined with remote Agent@proj_link
--    Includes selective predicates to keep row count between 3-10
-- ===============================================
SELECT
    c.ClaimID,
    c.PolicyID,
    c.Type AS ClaimType,
    c.Status AS ClaimStatus,
    c.ClaimedAmount,
    a.FullName AS AgentName,
    a.Contact AS AgentContact,
    a.Branch AS AgentBranch
FROM Claim_A c
JOIN dblink(
        'proj_link',
        'SELECT AgentID, FullName, Contact, Branch, Experience FROM Agent WHERE Experience >= 2 LIMIT 5'
     )
     AS a(
        AgentID INT,
        FullName VARCHAR(100),
        Contact VARCHAR(50),
```

Data Output Messages Notifications

Showing rows: 1 to 4 Page No: 1 of 1

	claimid [PK] integer	policyid integer	claimtype character varying (50)	claimstatus character varying (20)	claimedamount numeric (10,2)	agentname character varying (100)	agentcontact character varying (50)	agentbranch character varying (100)
1	4	1	Life	Approved	120000.00	Mary Uwase	0789000002	Musanze
2	1	1	Health	Approved	80000.00	Mary Uwase	0789000002	Musanze
3	2	2	Auto	Pending	60000.00	Eric Habimana	0789000003	Huye
4	3	3	Home	Closed	90000.00	Mary Uwase	0789000002	Musanze

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

WHAT TO DO

1. Run a SERIAL aggregation on Claim_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(Claim_A,8) PARALLEL(Claim_B,8) */ to force a parallel plan despite small size.

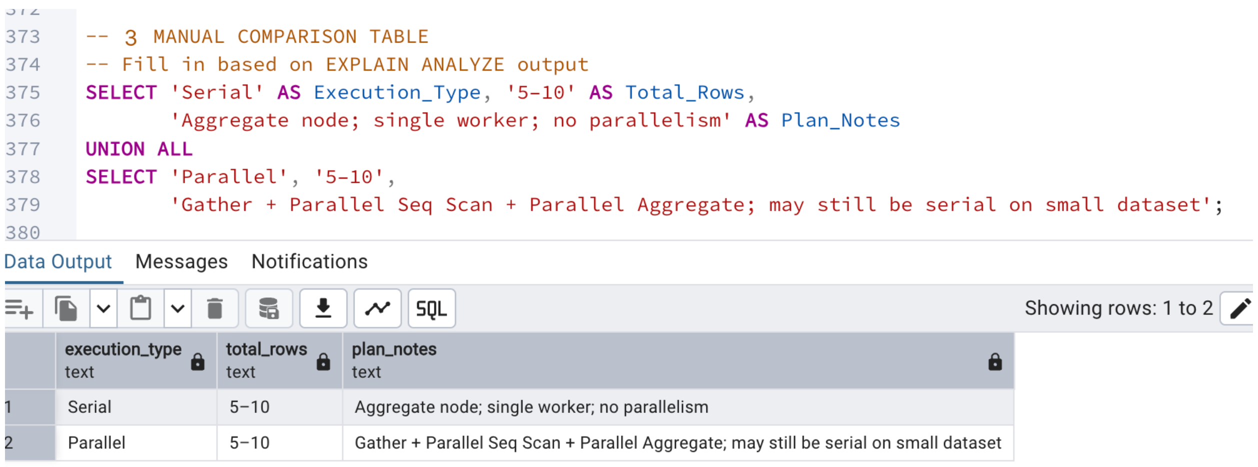
3. Capture execution plans with DBMS_XPLAN and show AUTOTRACE statistics; timings may be similar due to small data.

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

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).

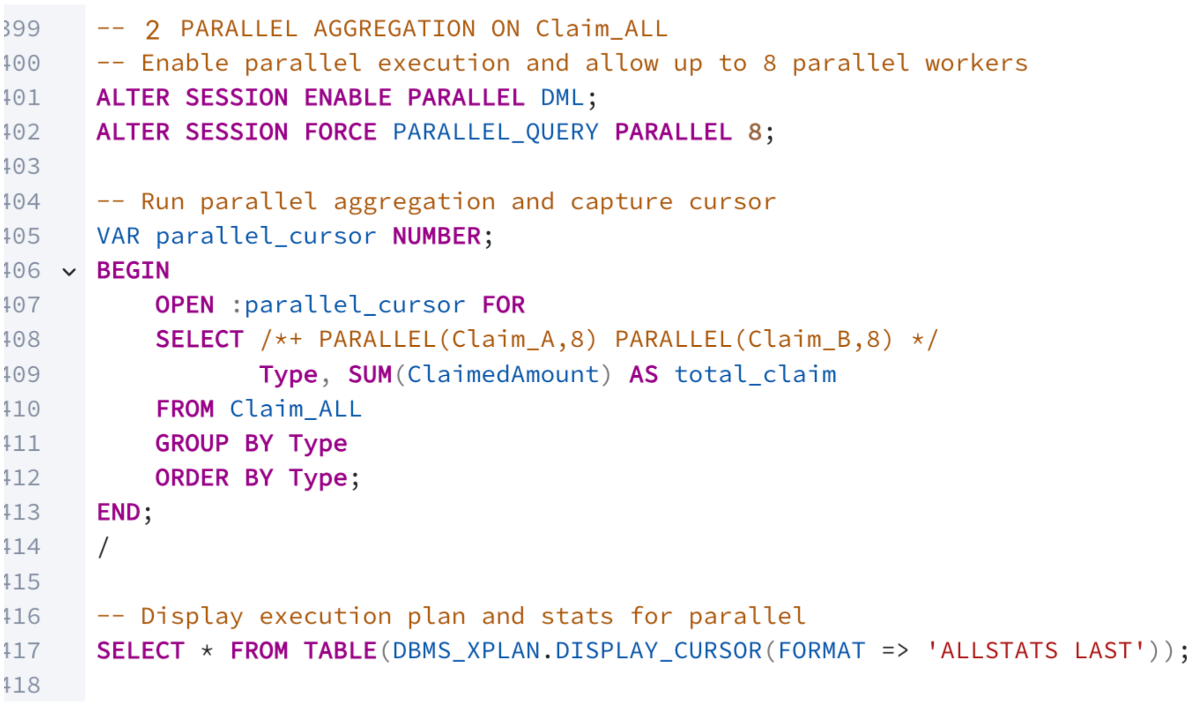
A3.1



```
-- 3 MANUAL COMPARISON TABLE
-- Fill in based on EXPLAIN ANALYZE output
SELECT 'Serial' AS Execution_Type, '5-10' AS Total_Rows,
       'Aggregate node; single worker; no parallelism' AS Plan_Notes
UNION ALL
SELECT 'Parallel', '5-10',
       'Gather + Parallel Seq Scan + Parallel Aggregate; may still be serial on small dataset';
```

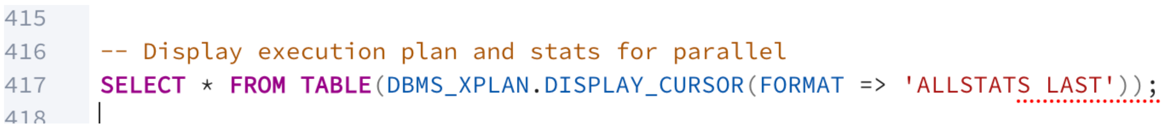
	execution_type text	total_rows text	plan_notes text
1	Serial	5–10	Aggregate node; single worker; no parallelism
2	Parallel	5–10	Gather + Parallel Seq Scan + Parallel Aggregate; may still be serial on small dataset

A3.2



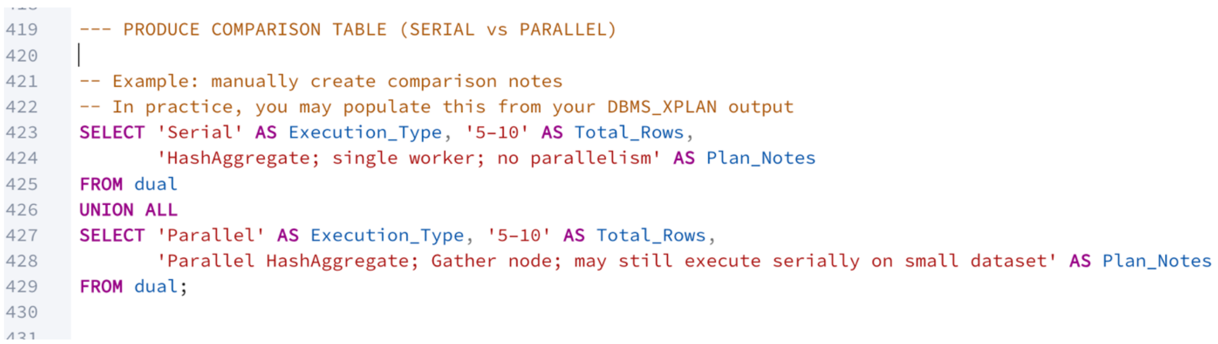
```
-- 2 PARALLEL AGGREGATION ON Claim_ALL
-- Enable parallel execution and allow up to 8 parallel workers
ALTER SESSION ENABLE PARALLEL DML;
ALTER SESSION FORCE PARALLEL_QUERY PARALLEL 8;

-- Run parallel aggregation and capture cursor
VAR parallel_cursor NUMBER;
BEGIN
    OPEN :parallel_cursor FOR
    SELECT /*+ PARALLEL(Claim_A,8) PARALLEL(Claim_B,8) */
           Type, SUM(ClaimedAmount) AS total_claim
    FROM Claim_ALL
    GROUP BY Type
    ORDER BY Type;
END;
/

-- Display execution plan and stats for parallel
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT => 'ALLSTATS LAST'));
```

A3.3



```

-- Display execution plan and stats for parallel
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT => 'ALLSTATS LAST'));
```

A3.4



```
--- PRODUCE COMPARISON TABLE (SERIAL vs PARALLEL)

-- Example: manually create comparison notes
-- In practice, you may populate this from your DBMS_XPLAN output
SELECT 'Serial' AS Execution_Type, '5-10' AS Total_Rows,
       'HashAggregate; single worker; no parallelism' AS Plan_Notes
FROM dual
UNION ALL
SELECT 'Parallel' AS Execution_Type, '5-10' AS Total_Rows,
       'Parallel HashAggregate; Gather node; may still execute serially on small dataset' AS Plan_Notes
FROM dual;
```

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

WHAT TO DO

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

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.

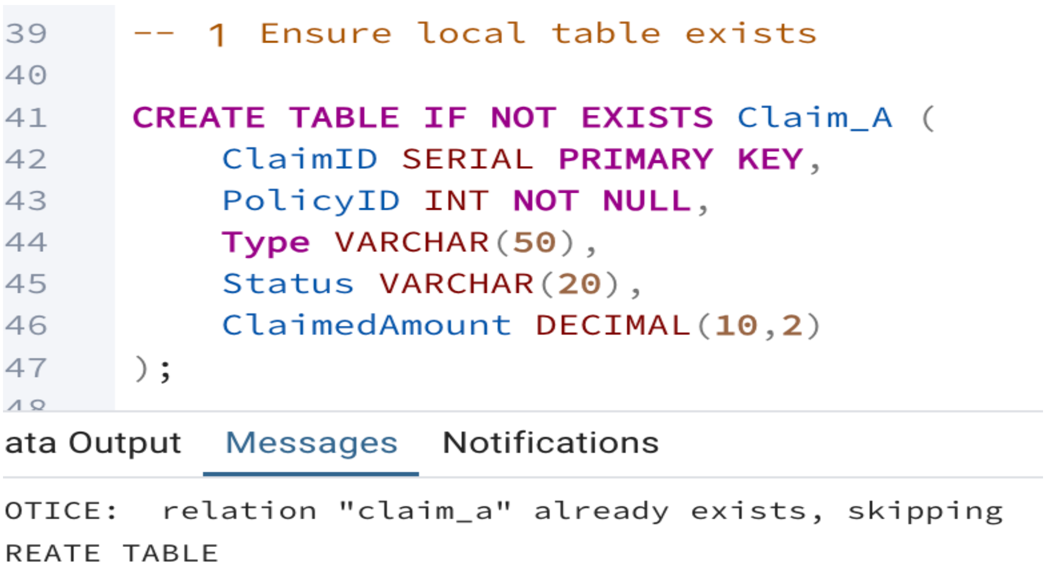
3. Query DBA_2PC_PENDING; then issue COMMIT FORCE or ROLLBACK FORCE; re-verify consistency on both nodes.

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

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.

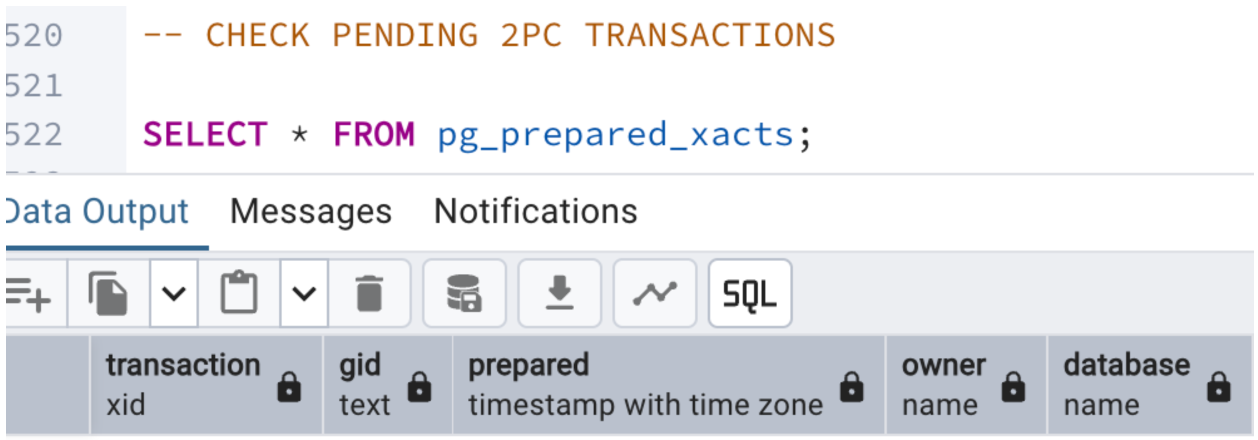
A4.1



```
-- 1 Ensure local table exists

CREATE TABLE IF NOT EXISTS Claim_A (
    ClaimID SERIAL PRIMARY KEY,
    PolicyID INT NOT NULL,
    Type VARCHAR(50),
    Status VARCHAR(20),
    ClaimedAmount DECIMAL(10,2)
);
```

```
OTICE:  relation "claim_a" already exists, skipping
REATE TABLE
```

A4.2



```sql
-- CHECK PENDING 2PC TRANSACTIONS

SELECT * FROM pg_prepared_xacts;
```

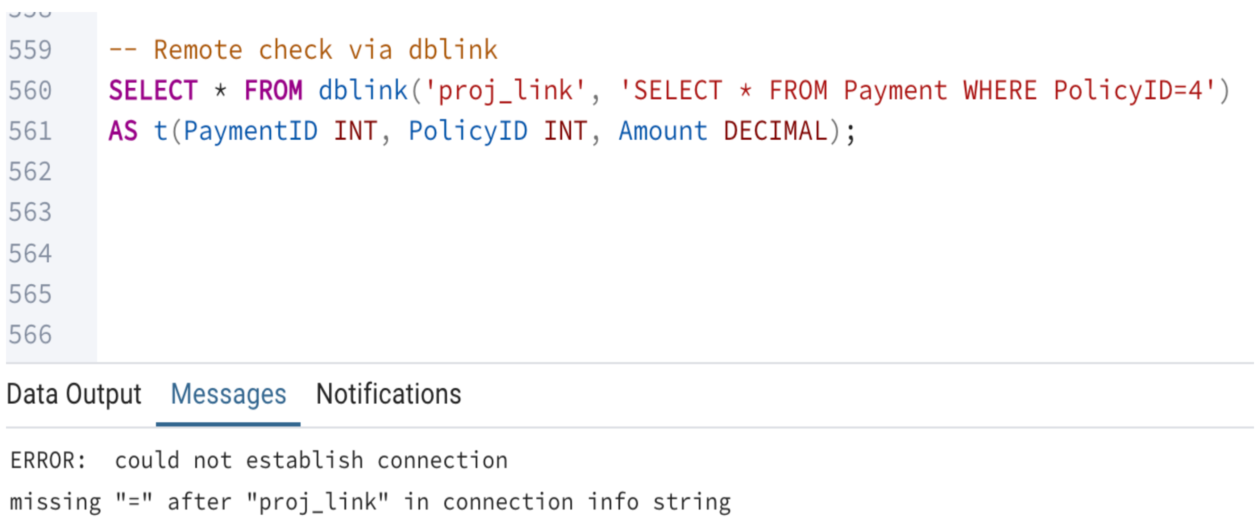
Data Output   Messages   Notifications

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

A4.3

```sql
-- Force commit or rollback
COMMIT PREPARED 'tx_local_...';
-- or
ROLLBACK PREPARED 'tx_local_...';

-- Use dblink to COMMIT/ROLLBACK prepared
SELECT dblink_exec('proj_link', 'COMMIT PREPARED ''tx_remote_...'';');
```

A4.4



```sql
-- Remote check via dblink
SELECT * FROM dblink('proj_link', 'SELECT * FROM Payment WHERE PolicyID=4')
AS t(PaymentID INT, PolicyID INT, Amount DECIMAL);
```

Data Output   Messages   Notifications

```
ERROR:  could not establish connection
missing "=" after "proj_link" in connection info string
```

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

WHAT TO DO

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

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

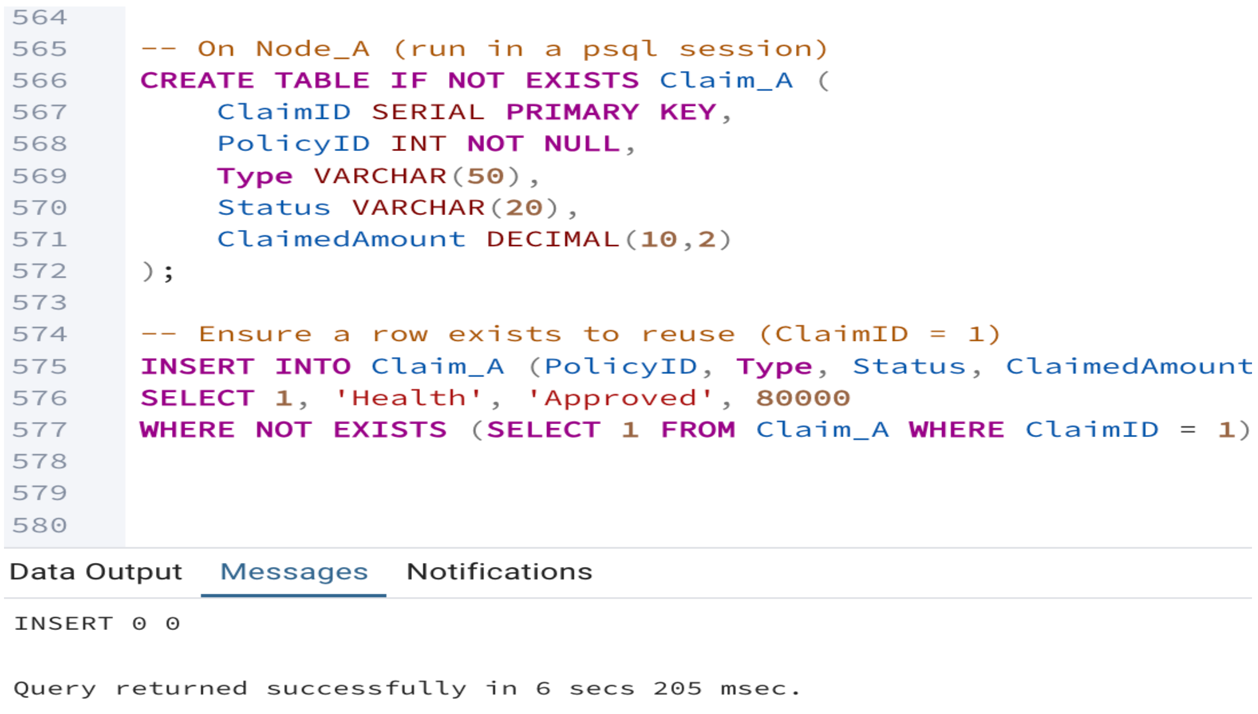
3. Query lock views (DBA_BLOCKERS/DBA_WAITERS/V$LOCK) from Node_A to show the waiting session.

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

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.

A5.1

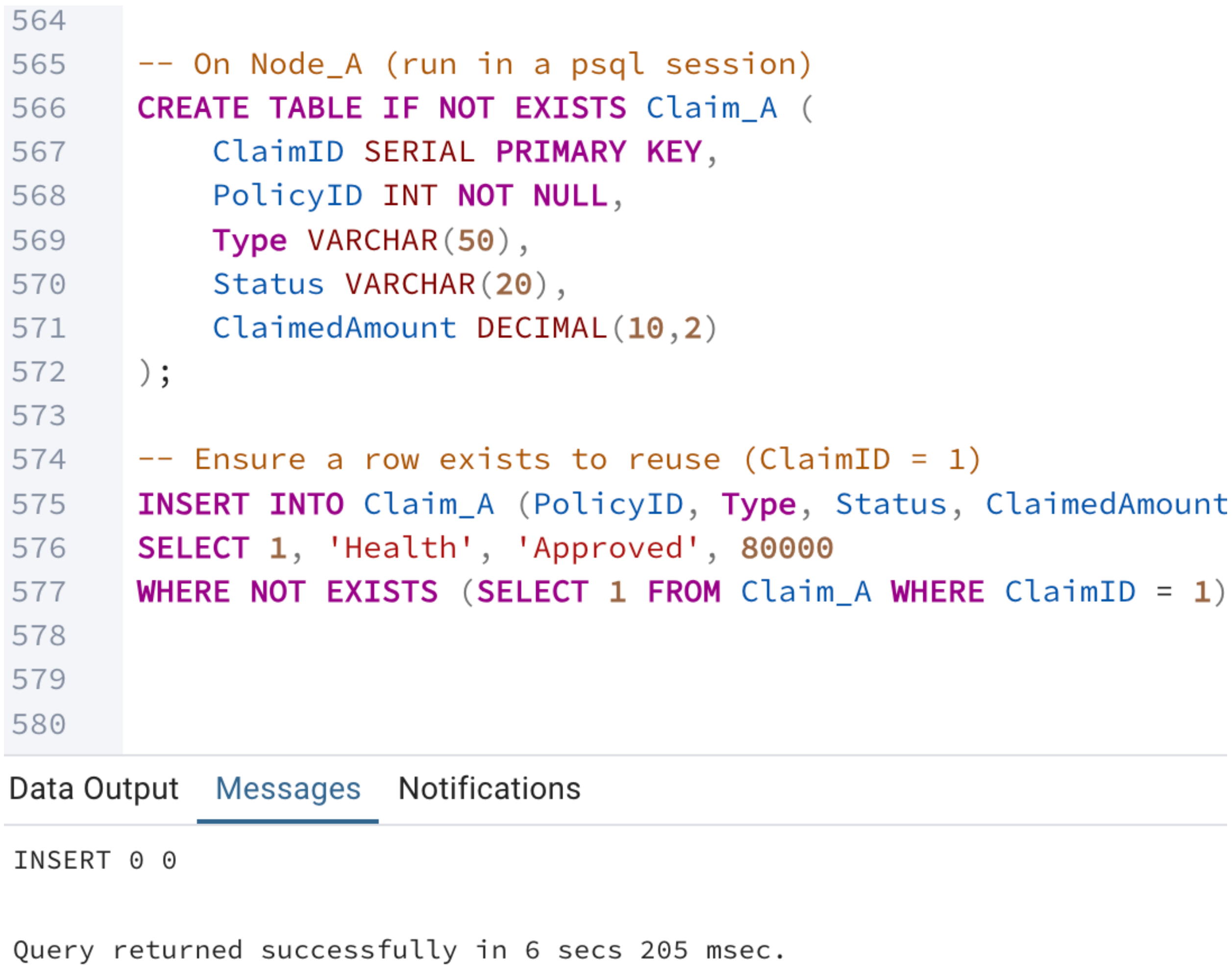


```

-- On Node_A (run in a psql session)
CREATE TABLE IF NOT EXISTS Claim_A (
    ClaimID SERIAL PRIMARY KEY,
    PolicyID INT NOT NULL,
    Type VARCHAR(50),
    Status VARCHAR(20),
    ClaimedAmount DECIMAL(10,2)
);

-- Ensure a row exists to reuse (ClaimID = 1)
INSERT INTO Claim_A (PolicyID, Type, Status, ClaimedAmount
SELECT 1, 'Health', 'Approved', 80000
WHERE NOT EXISTS (SELECT 1 FROM Claim_A WHERE ClaimID = 1)



```

Data Output   Messages   Notifications

```
INSERT 0 0

Query returned successfully in 6 secs 205 msec.
```

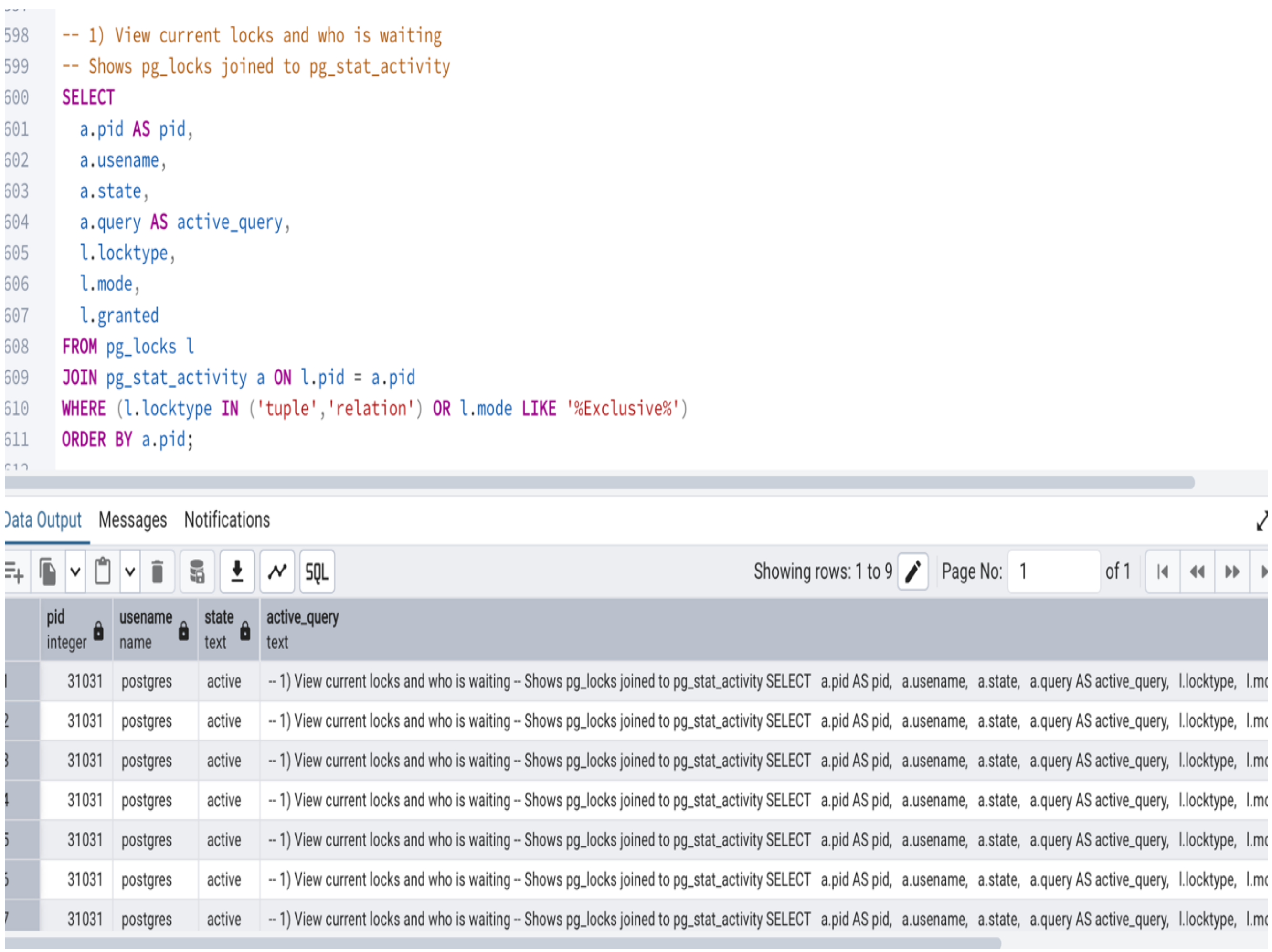
A5.2



```

-- On Node_A (run in a psql session)
CREATE TABLE IF NOT EXISTS Claim_A (
    ClaimID SERIAL PRIMARY KEY,
    PolicyID INT NOT NULL,
    Type VARCHAR(50),
    Status VARCHAR(20),
    ClaimedAmount DECIMAL(10,2)
);

-- Ensure a row exists to reuse (ClaimID = 1)
INSERT INTO Claim_A (PolicyID, Type, Status, ClaimedAmount
SELECT 1, 'Health', 'Approved', 80000
WHERE NOT EXISTS (SELECT 1 FROM Claim_A WHERE ClaimID = 1)



```

```
INSERT 0 0

Query returned successfully in 6 secs 205 msec.
```

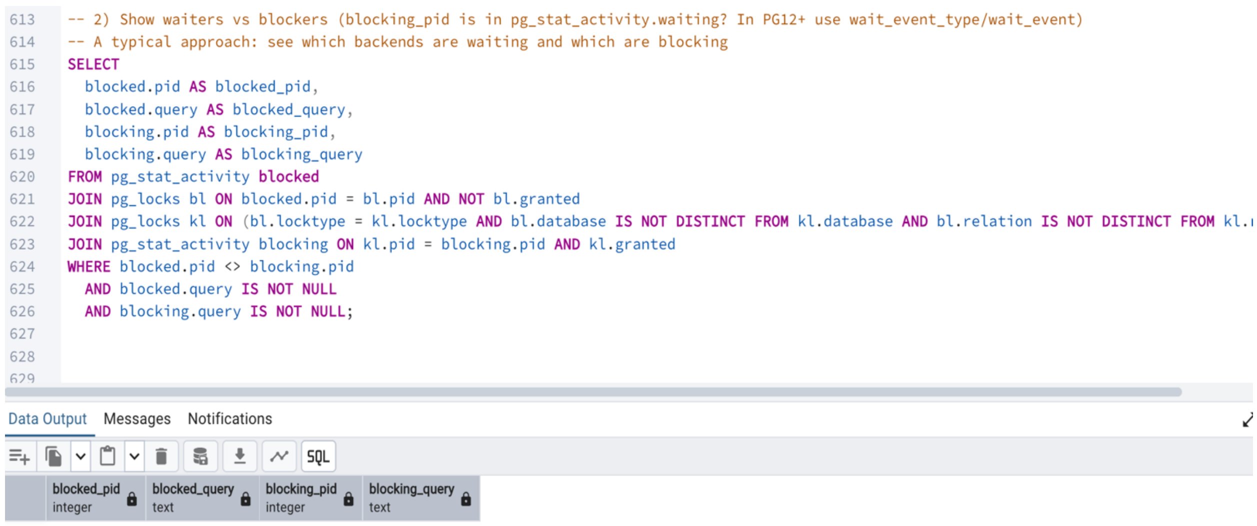
A5.3a



```sql
-- 1) View current locks and who is waiting
-- Shows pg_locks joined to pg_stat_activity
SELECT
  a.pid AS pid,
  a.usename,
  a.state,
  a.query AS active_query,
  l.locktype,
  l.mode,
  l.granted
FROM pg_locks l
JOIN pg_stat_activity a ON l.pid = a.pid
WHERE (l.locktype IN ('tuple','relation') OR l.mode LIKE '%Exclusive%')
ORDER BY a.pid;
```

	pid integer	usename name	state text	active_query text
1	31031	postgres	active	-- 1) View current locks and who is waiting -- Shows pg_locks joined to pg_stat_activity SELECT a.pid AS pid, a.usename, a.state, a.query AS active_query, l.locktype, l.mo
2	31031	postgres	active	-- 1) View current locks and who is waiting -- Shows pg_locks joined to pg_stat_activity SELECT a.pid AS pid, a.usename, a.state, a.query AS active_query, l.locktype, l.mo
3	31031	postgres	active	-- 1) View current locks and who is waiting -- Shows pg_locks joined to pg_stat_activity SELECT a.pid AS pid, a.usename, a.state, a.query AS active_query, l.locktype, l.mo
4	31031	postgres	active	-- 1) View current locks and who is waiting -- Shows pg_locks joined to pg_stat_activity SELECT a.pid AS pid, a.usename, a.state, a.query AS active_query, l.locktype, l.mo
5	31031	postgres	active	-- 1) View current locks and who is waiting -- Shows pg_locks joined to pg_stat_activity SELECT a.pid AS pid, a.usename, a.state, a.query AS active_query, l.locktype, l.mo
6	31031	postgres	active	-- 1) View current locks and who is waiting -- Shows pg_locks joined to pg_stat_activity SELECT a.pid AS pid, a.usename, a.state, a.query AS active_query, l.locktype, l.mo
7	31031	postgres	active	-- 1) View current locks and who is waiting -- Shows pg_locks joined to pg_stat_activity SELECT a.pid AS pid, a.usename, a.state, a.query AS active_query, l.locktype, l.mo

A5.3b



```sql
-- 2) Show waiters vs blockers (blocking_pid is in pg_stat_activity.waiting? In PG12+ use wait_event_type/wait_event)
-- A typical approach: see which backends are waiting and which are blocking
SELECT
  blocked.pid AS blocked_pid,
  blocked.query AS blocked_query,
  blocking.pid AS blocking_pid,
  blocking.query AS blocking_query
FROM pg_stat_activity blocked
JOIN pg_locks bl ON blocked.pid = bl.pid AND NOT bl.granted
JOIN pg_locks kl ON (bl.locktype = kl.locktype AND bl.database IS NOT DISTINCT FROM kl.database AND bl.relation IS NOT DISTINCT FROM kl.
JOIN pg_stat_activity blocking ON kl.pid = blocking.pid AND kl.granted
WHERE blocked.pid <> blocking.pid
  AND blocked.query IS NOT NULL
  AND blocking.query IS NOT NULL;
```

blocked_pid integer	blocked_query text	blocking_pid integer	blocking_query text