Reg no: 217029434
Name: Edison Ngizwenayo

Case Study: LOGISTICS FLEET & MAINTENANCE

Section A:

A1.1: Create horizontally fragmented tables Shipment_A on Node_A and Shipment_B on Node_B using a deterministic rule (HASH or RANGE on a natural key).

Since shipment depend on base tables which are vehicles,drivers,routes

```
-- 1. DDL for vehicles tables
CREATE TABLE IF NOT EXISTS vehicles (
    vehicle_id SERIAL PRIMARY KEY,  -- Unique identifier for each vehicle
    model VARCHAR(100) NOT NULL,    -- Vehicle model name
    plate_no VARCHAR(20) NOT NULL UNIQUE, -- Unique license plate number
    type VARCHAR(50) NOT NULL,      -- Vehicle type (e.g., Truck, Van)
    status VARCHAR(20) NOT NULL CHECK (status IN ('Active', 'In Transit', 'Under Maintenance'
    capacity INT NOT NULL CHECK (capacity > 0) -- Maximum load capacity
);

-- 2. Drivers Table: Stores driver details
CREATE TABLE IF NOT EXISTS drivers (
    driver_id SERIAL PRIMARY KEY,   -- Unique identifier for each driver
    full_name VARCHAR(100) NOT NULL,  -- full name of the driver
    license_no VARCHAR(50) NOT NULL UNIQUE, -- Driver's license number
    contact VARCHAR(50),            -- Contact information
    experience_years INT NOT NULL CHECK (experience_years >= 0) -- Experience in years
);
```

```
-- 3. Routes Table: Stores predefined delivery routes
CREATE TABLE IF NOT EXISTS routes (
    route_id SERIAL PRIMARY KEY,    -- Unique route identifier
    start_location VARCHAR(100) NOT NULL, -- Starting point
    end_location VARCHAR(100) NOT NULL,   -- Destination
    distance_km DECIMAL(10,2) NOT NULL CHECK (distance_km > 0), -- Distance in kilometers with precision o
    type VARCHAR(50) NOT NULL CHECK (type IN ('Urban','Rural','Long-Haul')) -- Route type classification
);
```

Horizantally Fragments the shipments table using a status column
Shipment_A contains all shipment that has status of either 'Pending','Canceled'

```
CREATE TABLE Shipment_A (
    shipment_id SERIAL PRIMARY KEY,
    vehicle_id INT NOT NULL,
    driver_id INT NOT NULL,
    route_id INT NOT NULL,
    start_date TIMESTAMP,
    end_date TIMESTAMP,
    status VARCHAR(20) NOT NULL CHECK (status IN ('Pending', 'Canceled')),
    CONSTRAINT fk_shipmentA_vehicle FOREIGN KEY (vehicle_id) REFERENCES vehicles(vehicle_id),
    CONSTRAINT fk_shipmentA_driver FOREIGN KEY (driver_id) REFERENCES drivers(driver_id),
    CONSTRAINT fk_shipmentA_route FOREIGN KEY (route_id) REFERENCES routes(route_id)
);
```

Horizantally Fragments the shipments table using a status column
Shipment_B contains all shipment that has status of either 'In Transit', 'Delivered'

```
CREATE TABLE Shipment_B (
    shipment_id SERIAL PRIMARY KEY,
    vehicle_id INT NOT NULL,
    driver_id INT NOT NULL,
    route_id INT NOT NULL,
    start_date TIMESTAMP,
    end_date TIMESTAMP,
    status VARCHAR(20) NOT NULL CHECK (status IN ('In Transit', 'Delivered')),
    CONSTRAINT fk_shipmentA_vehicle FOREIGN KEY (vehicle_id) REFERENCES vehicles(vehicle_id),
    CONSTRAINT fk_shipmentA_driver FOREIGN KEY (driver_id) REFERENCES drivers(driver_id),
    CONSTRAINT fk_shipmentA_route FOREIGN KEY (route_id) REFERENCES routes(route_id)
);
```

A1.2 Insert a TOTAL of ≤10 committed rows split across the two fragments

```
-- Let insert 5 rows in Shipment_A, ensure status either 'Pending','Canceled'
INSERT INTO Shipment_A (vehicle_id, driver_id, route_id, start_date, end_date, status)
VALUES
(1, 2, 3, '2025-10-20 08:00:00',NULL,'Pending'),
(2, 1, 1, '2025-10-21 09:30:00',NULL,'Pending'),
(3, 3, 2,  '2025-10-18 07:45:00','2025-10-19 16:20:00', 'Canceled'),
(4, 4, 4, '2025-10-22 10:15:00', NULL,'Pending'),
(5, 1, 2, '2025-10-15 06:10:00','2025-10-15 18:40:00', 'Canceled');
```

Let query the the data within the table to see if insert was successful

shipment_id [PK] integer	vehicle_id integer	driver_id integer	route_id integer	start_date timestamp without time zone	end_date timestamp without time zone	status character varying (20)
4	1	2	3	2025-10-20 08:00:00	[null]	Pending
5	2	1	1	2025-10-21 09:30:00	[null]	Pending
6	3	3	2	2025-10-18 07:45:00	2025-10-19 16:20:00	Canceled
7	4	4	4	2025-10-22 10:15:00	[null]	Pending
8	5	1	2	2025-10-15 06:10:00	2025-10-15 18:40:00	Canceled

```
-- Let insert 5 rows in Shipment_B, ensure status either 'In Transit', 'Delivered'
INSERT INTO Shipment_B (vehicle_id, driver_id, route_id, start_date, end_date, status)
VALUES
(1, 2, 3, '2025-10-18 07:00:00', NULL, 'In Transit'),
(2, 1, 1, '2025-10-14 08:45:00', '2025-10-14 20:15:00', 'Delivered'),
(3, 3, 2, '2025-10-16 09:10:00', NULL, 'In Transit'),
(4, 4, 4, '2025-10-17 07:30:00', '2025-10-17 19:00:00', 'Delivered'),
(5, 1, 2, '2025-10-23 06:50:00', NULL, 'In Transit');
```

Let query the the data within the table to see if insert was successful

shipment_id [PK] integer	vehicle_id integer	driver_id integer	route_id integer	start_date timestamp without time zone	end_date timestamp without time zone	status character varying (20)
11	1	2	3	2025-10-18 07:00:00	[null]	In Transit
12	2	1	1	2025-10-14 08:45:00	2025-10-14 20:15:00	Delivered
13	3	3	2	2025-10-16 09:10:00	[null]	In Transit
14	4	4	4	2025-10-17 07:30:00	2025-10-17 19:00:00	Delivered
15	5	1	2	2025-10-23 06:50:00	[null]	In Transit

A1.3: On Node_A, create view Shipment_ALL as UNION ALL of Shipment_A and Shipment_B.
Since **Shipment_A** resides on the local node (**Node_A**) and **Shipment_B** on a remote node (**Node_B**), we utilize PostgreSQL's **dblink** extension to establish a connection between the two databases. This allows us to query and retrieve data from **Node_B** directly within **Node_A** as if it were local. To enable this functionality, the **dblink** extension must first be created on **Node_A** using the command:
CREATE EXTENSION IF NOT EXISTS dblink;

```
CREATE OR REPLACE VIEW Shipment_ALL AS
-- Select all rows from local fragment
SELECT *
FROM Shipment_A

UNION ALL  -- Combine with remote fragment, keeping duplicates if any

-- Fetch remote fragment Shipment_B using dblink
SELECT *
FROM dblink(
    -- Connection string to remote Node_B database
    'host=localhost port=5432 dbname=Node_B user=postgres password=postgres',

    -- SQL query to run on the remote database
    'SELECT shipment_id, vehicle_id, driver_id, route_id, start_date, end_date, status FROM Shipment_B'
) AS remote_shipments(
    -- Define column names and data types for the remote query result
    shipment_id INT,
    vehicle_id INT,
    driver_id INT,
    route_id INT,
    start_date TIMESTAMP,
    end_date TIMESTAMP,
    status VARCHAR(20)
);
```

Running the below query we get the results of the view
SELECT * FROM Shipment_ALL

shipment_id integer	vehicle_id integer	driver_id integer	route_id integer	start_date timestamp without time zone	end_date timestamp without time zone	status character varying (20)
4	1	2	3	2025-10-20 08:00:00	[null]	Pending
5	2	1	1	2025-10-21 09:30:00	[null]	Pending
6	3	3	2	2025-10-18 07:45:00	2025-10-19 16:20:00	Canceled
7	4	4	4	2025-10-22 10:15:00	[null]	Pending
8	5	1	2	2025-10-15 06:10:00	2025-10-15 18:40:00	Canceled
11	1	2	3	2025-10-18 07:00:00	[null]	In Transit
12	2	1	1	2025-10-14 08:45:00	2025-10-14 20:15:00	Delivered
13	3	3	2	2025-10-16 09:10:00	[null]	In Transit
14	4	4	4	2025-10-17 07:30:00	2025-10-17 19:00:00	Delivered
15	5	1	2	2025-10-23 06:50:00	[null]	In Transit

A1.4: Validate with COUNT(*) and a checksum on a key column (e.g., SUM(MOD(primary_key,97))) :results must match fragments vs Shipment_ALL.

Let Validate Checksum on Primary Key

```sql
-- Local fragment
SELECT SUM(MOD(shipment_id, 97)) AS checksum_local FROM Shipment_A;
```

Running the above query we get

	checksum_local bigint
1	30

```sql
-- Remote fragment via dblink
SELECT SUM(MOD(shipment_id, 97)) AS checksum_remote
FROM dblink(
    'host=localhost port=5432 dbname=Node_B user=postgres password=postgres',
    'SELECT shipment_id FROM Shipment_B'
) AS remote_shipments(shipment_id INT);
```

Running the above query we get

	checksum_remote bigint
1	65

```sql
-- Checksum for global view
SELECT SUM(MOD(shipment_id, 97)) AS checksum_global FROM Shipment_ALL;
```

Running the above we get

	checksum_global bigint
1	95

Conclusion: from the above we can see that **Validation Rule** is valid since cnt_global(for view) = cnt_local + cnt_remote

A2 :Database Link & Cross-Node

A2.1. From Node_A, create a database link to Node_B.

To allow to both two databases to communicate we use Foreign Data Wrapper(FDW)
This enable access to tables in another database as if they were a local database. Below is the query to create connect between Node_A and Node_B

```
CREATE EXTENSION IF NOT EXISTS postgres_fdw;
-- Create a foreign server
CREATE SERVER NodeB_connect
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (
    host 'localhost',       -- host where FleetOperations is running
    dbname 'Node_B',  -- remote db to connect to
    port '5432'
);
-- create a user mapping(Map a local user in FleetSupport node  to a user in FleetOperations node)
CREATE USER MAPPING FOR postgres  -- or your local user
SERVER NodeB_connect
OPTIONS (
    user 'postgres',         -- FleetOperations username
    password 'postgres'       -- FleetOperations password
);
-- import  foreign tables from FleetOperations
IMPORT FOREIGN SCHEMA public
LIMIT TO (Shipment_B)
FROM SERVER NodeB_connect INTO public;
```

A2.2. Run remote SELECT on Shipment_B showing up to 5 sample rows.

```
SELECT *  FROM Shipment_B
LIMIT 5
```

Below is the results of the query

	shipment_id integer	vehicle_id integer	driver_id integer	route_id integer	start_date timestamp without time zone	end_date timestamp without time zone	status character varying (20)
1	11	1	2	3	2025-10-18 07:00:00	[null]	In Transit
2	12	2	1	1	2025-10-14 08:45:00	2025-10-14 20:15:00	Delivered
3	13	3	3	2	2025-10-16 09:10:00	[null]	In Transit
4	14	4	4	4	2025-10-17 07:30:00	2025-10-17 19:00:00	Delivered
5	15	5	1	2	2025-10-23 06:50:00	[null]	In Transit

A2. 3. Run a distributed join: remote Shipment_B (or base Shipment) joined with a local vehicle.

The following query generate number of vehicle per shipment status

```
-- A2.3. Run a distributed join: remote Shipment_B (or base Shipment) joined with a local vehicle
SELECT s.status,COUNT(v.vehicle_id) FROM Shipment_B s
INNER JOIN vehicles v ON v.vehicle_id = s.vehicle_id
GROUP BY 1
```

'

Running the query we get the following result

status character varying (20)	count bigint
Delivered	2
In Transit	3

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

A3. 1. Run a SERIAL aggregation on Shipment_ALL over the small dataset (e.g., totals by a domain. Below is the query for computing total shipments by status

```
SET max_parallel_workers_per_gather = 0;    -- Disable parallelism
SELECT -- Aggregate total shipments by status
    status,
    COUNT(*) AS total_shipments
FROM Shipment_ALL
GROUP BY status
ORDER BY status;
```

Running the above query we get the following results

status character varying (20)	total_shipments bigint
Canceled	2
Delivered	2
In Transit	3
Pending	3

A3.2. Run the same aggregation with /*+ PARALLEL(Shipment_A,8) PARALLEL(Shipment_B,8) */ to force a parallel plan despite small size.

To enable parallelism in postgres, we have to change the value of max_parallel_workers_per_gather to match the number of workers recommended. In this section we are setting max_parallel_workers_per_gather to 4

```
SET max_parallel_workers_per_gather = 8;    -- Enable parallelism  and use 8 workers as per instruction

SELECT -- Aggregate total shipments by status
    status,
    COUNT(*) AS total_shipments
FROM Shipment_ALL
GROUP BY status
ORDER BY status;
```

Below is the result of the query

status character varying (20)	total_shipments bigint
Canceled	2
Delivered	2
In Transit	3
Pending	3

A3.3. Capture execution plans with EXPLAIN and show AUTOTRACE statistics; timings may be similar due to small data.

Serial query plan

QUERY PLAN text
Sort (cost=53.45..53.95 rows=200 width=66) (actual time=63.860..63.860 rows=4 loops=1)
Sort Key: shipment_a.status
Sort Method: quicksort Memory: 25kB
-> HashAggregate (cost=43.80..45.80 rows=200 width=66) (actual time=63.819..63.821 rows=4 loops=1)
Group Key: shipment_a.status
Batches: 1 Memory Usage: 40kB
-> Append (cost=0.00..35.35 rows=1690 width=58) (actual time=0.041..63.802 rows=10 loops=1)
-> Seq Scan on shipment_a (cost=0.00..16.90 rows=690 width=58) (actual time=0.041..0.042 rows=5 loops=1)
-> Function Scan on dblink remote_shipments (cost=0.00..10.00 rows=1000 width=58) (actual time=63.758..63.759 rows=5 loop…
Planning Time: 0.255 ms
Execution Time: 63.985 ms

Parallel Query Plan

QUERY PLAN text
Sort (cost=53.45..53.95 rows=200 width=66) (actual time=38.989..38.989 rows=4 loops=1)
Sort Key: shipment_a.status
Sort Method: quicksort Memory: 25kB
-> HashAggregate (cost=43.80..45.80 rows=200 width=66) (actual time=38.981..38.983 rows=4 loops=1)
Group Key: shipment_a.status
Batches: 1 Memory Usage: 40kB
-> Append (cost=0.00..35.35 rows=1690 width=58) (actual time=0.012..38.968 rows=10 loops=1)
-> Seq Scan on shipment_a (cost=0.00..16.90 rows=690 width=58) (actual time=0.012..0.013 rows=5 loops=1)
-> Function Scan on dblink remote_shipments (cost=0.00..10.00 rows=1000 width=58) (actual time=38.953..38.953 rows=5 loop…
Planning Time: 0.122 ms
Execution Time: 39.031 ms

A3.4. Produce a 2-row comparison table (serial vs parallel) with plan notes.

Execution Type	Planning Time (ms)	Execution Time (ms)	Plan Notes
Serial	**0.225**	**62.985**	Query executed without parallel workers; a single process scanned and aggregated all data sequentially
Parallel	0.122	39.031	Query executed with parallel workers (8 per fragment); planning slightly faster, execution reduced due to concurrent scanning and aggregation across fragments

A4 :Two-Phase Commit & Recovery (2 rows)
A4.1. Write one PL/SQL block that inserts ONE local row (related Shipment) on Node_A and ONE remote row into payments; then COMMIT.

```
DO $$
DECLARE
    -- Define variable to store the shipment_id generated by the local insert
    new_shipment_id INT;

    -- Define variable to hold the remote SQL statement for dblink execution
    remote_sql TEXT;
BEGIN
    -- Insert a new shipment into the 'shipments' table at Node 1(local DB)
    INSERT INTO Shipment_A (vehicle_id, driver_id, route_id, start_date, end_date, status)
    VALUES (3, 3, 4, CURRENT_DATE, CURRENT_DATE + INTERVAL '2 days', 'Pending')
    RETURNING shipment_id INTO new_shipment_id; -- captures the generated shipment_id into a PL/pgSQL variable

    -- Log the newly generated shipment_id for debugging
    RAISE NOTICE 'New shipment_id = %', new_shipment_id;

    -- Prepare the remote SQL for inserting a payment into Node 2
    -- Use format() with %L to safely quote the shipment_id literal
    remote_sql := format($sql$
        INSERT INTO payments (shipment_id, amount, method, payment_date)
        VALUES (%L, 10000, 'Mobile Money', CURRENT_DATE);
    $sql$, new_shipment_id);

    -- Execute the remote insert using dblink_exec
    PERFORM dblink_exec(
        'dbname=Node_B user=postgres password=postgres host=localhost port=5432',
        remote_sql
    );
```

```
    -- Confirm both inserts succeeded
    RAISE NOTICE 'Data inserted successfully on both nodes.';

EXCEPTION
    -- Exception handling: log the error and re-raise it for further debugging
    WHEN OTHERS THEN
        RAISE NOTICE 'Transaction failed: %', SQLERRM;
        RAISE;
END;
$$;
```

This PL/pgSQL block simulates a **two-phase commit** across two database nodes by performing inserts on both the local (`Shipment_A` table) and remote (`payments` table via dblink) databases within a single atomic operation. First, it inserts a new shipment into the local `shipments` table and captures the generated `shipment_id`. It then constructs a remote SQL statement using that `shipment_id` to insert a corresponding payment record into the remote database. The entire operation is **atomic**, meaning that if any part of the process fails—either the local or remote insert the **entire transaction will be rolled back**, ensuring no partial updates occur. Conversely, if both inserts succeed without issues, the **transaction is committed**, guaranteeing consistent data across both nodes.

A4.2. Induce a failure in a second run (e.g., disable the link between inserts) to create an in-doubt transaction.
In this step, we intentionally simulate a connection failure to Node_B by providing incorrect authentication details. Specifically, we attempt to connect to the remote database using the

username postgres and the password post, which are invalid credentials. This deliberate misconfiguration helps us observe how the system handles authentication errors and connection failures when communication with Node_B is disrupted. Below is the PL code that simulate the failure.

```
DO $$
DECLARE
    local_shipment_id INT;   -- Variable to hold the newly inserted shipment ID
    remote_sql TEXT;         -- Variable to hold dynamically generated remote SQL command
    remote_gid TEXT;         -- Global transaction ID (GID) for the remote database transaction
    local_gid TEXT;          -- Global transaction ID (GID) for the local database transaction
BEGIN
    -- Step 1: Insert a new shipment record locally in the 'shipments' table.
    -- The generated shipment_id is captured into the 'local_shipment_id' variable.
    INSERT INTO Shipment_A (vehicle_id, driver_id, route_id, start_date, end_date, status)
    VALUES (3, 3, 4, CURRENT_DATE, CURRENT_DATE + INTERVAL '2 days', 'Canceled')
    RETURNING shipment_id INTO local_shipment_id;

    -- Step 2: Generate unique transaction identifiers (GIDs) for both local and remote transactions.
    -- These identifiers are important for managing distributed transactions using two-phase commit.
    local_gid := format('local_tx_%s', local_shipment_id);
    remote_gid := format('remote_tx_%s', local_shipment_id);

    -- Step 3: Log the generated identifiers for tracking.
    RAISE NOTICE 'Local shipment_id = %, local_gid = %, remote_gid = %',
                 local_shipment_id, local_gid, remote_gid;

    -- Step 4: Simulate a remote failure, by connecting using wrong credentials
    -- to trigger an error, representing a remote database failure scenario.
    remote_sql := format($remote$
        BEGIN;
        INSERT INTO payments (shipment_id, amount, method, payment_date)
        VALUES (%L, 10000, 'Mobile Money', CURRENT_DATE);
        PREPARE TRANSACTION %L;
    $remote$, local_shipment_id, remote_gid);

    BEGIN
        -- Step 5: Attempt to execute the remote transaction using dblink.
        -- If the remote SQL fails (due to the invalid table), the exception block will capture it.
        PERFORM dblink_exec(
            'dbname=Node_B user=postgres password=post host=localhost port=5432',
            remote_sql
        );
    EXCEPTION
        WHEN OTHERS THEN
            -- Step 6: Log the simulated failure for debugging and visibility.
            -- The error message (SQLERRM) will describe the cause of failure.
            RAISE NOTICE '⚠Remote transaction failed: %', SQLERRM;
    END;

    -- Step 7: The local transaction remains unprepared.
    -- It can later be manually prepared or rolled back using its GID if needed.
END;
$$;
```

```
-- manually prepare the transaction
-- this is achieved by runningh PREPARE TRANSACTION outside of DO block and
-- assign a unique GID(Global identifier)
BEGIN;
PREPARE TRANSACTION 'remote_tx_1';
```

Running the following PL code we get the following output, since we have customized the error message.


```
-- manually prepare the transaction
-- this is achieved by runningh PREPARE TRANSACTION outside of DO block and
-- assign a unique GID(Global identifier)
BEGIN;
PREPARE TRANSACTION 'remote_tx_1';
```

A4.3. Query pg_prepared_xacts; then issue COMMIT FORCE or ROLLBACK FORCE; re-verify consistency on both nodes.
Below is the query to all unprepared transactions

```
SELECT * FROM pg_prepared_xacts;
```

Running the above query we get the following unresolved transactions

transaction xid	gid text	prepared timestamp with time zone	owner name	database name
1311	remote_tx_14	2025-10-28 21:57:12.165031+02	postgres	Node_A

Run the following query to resolved by ROLLBACK

```
ROLLBACK PREPARED 'remote_tx_14';
```

A4.4. Repeat a clean run to show there are no pending transactions.
After running the following query to resolve all transaction

```
SELECT * FROM pg_prepared_xacts;
```

Below is the output of the query, and it is clear all transaction are resolved.

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

A5: Distributed Lock Conflict & Diagnosis (no extra rows)
A5.1.1. Open Session 1 on Node_A: UPDATE a single row in Maintenance and keep the transaction open.

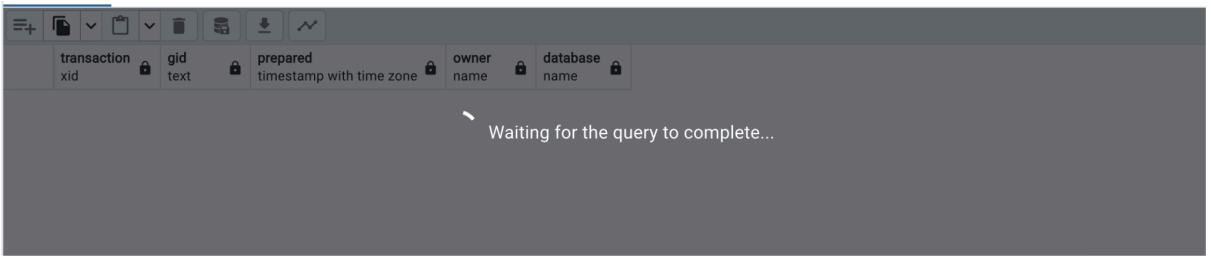
On Node_A, we are initiating a transaction for updating maintenance cost of a for vehicle_id = 5,

```
BEGIN;
-- Lock the record by updating it (but don't commit yet)
UPDATE maintenance
SET cost = 10000
WHERE vehicle_id = 3;
```

A5.2. Open Session 2 from Node_B to update the same logical row.
Since we are going to update remote table maintenance which is on node A, so on node B we are going to create a transaction that remotely updates the record in maintenance table, and since this record being updated has been locked by the first transaction the second transaction will be hanged until the first one is committed or rolled back. The following is the query that remotely updates the table.

```
BEGIN;
SELECT dblink_exec( -- Execute the remote update using dblink
    'dbname=Node_A user=postgres password=postgres host=localhost port=5432',
    $remote$
        UPDATE maintenance
        SET cost = 10000
        WHERE vehicle_id = 3;
    $remote$
);
```

Running the above table we can see that the operation is hanged until the first transactions that locked the rows is either committed all rolledback



A5.3 Query lock views from Node_A to show the waiting session.

The following query lock view

```
-- PostgreSQL internal lock view
SELECT
    pid,
    locktype,
    relation::regclass AS table_name,
    page,
    tuple,
    virtualtransaction,
    mode,
    granted
FROM pg_locks l
JOIN pg_class c ON l.relation = c.oid
WHERE c.relname = 'maintenance';
```

After running the following query below are the waiting sessions

pid integer	locktype text	table_name regclass	page integer	tuple smallint	virtualtransaction text	mode text	granted boolean
71203	relation	maintenance	[null]	[null]	11/6180	RowExclusiveLock	true
69756	relation	maintenance	[null]	[null]	9/2414	RowExclusiveLock	true
71203	tuple	maintenance	0	1	11/6180	ExclusiveLock	true

A5.4. Release the lock; show Session 2 completes.

This is done by running the following query which rollback the operation

RollBACK;

Below is the screenshot that show session 2 is complete

B6 :Declarative Rules Hardening (≤10 committed rows)

B6.1 On tables Vehicle and Maintenance, add/verify NOT NULL and domain CHECK constraints suitable for revenue and downtime (e.g., positive amounts, valid statuses, date order).

In this section we implanted Declarative Rules (constraints) that validating the data being inserted, this is crucial to improved accuracy and ensure the database is reliable. NO NULL constraints prevent the column to accept null or missing value CHECK these constraints ensure values being inserted or updated meet predefined conditions . Below is the query for achieving this.

```
-- vehicle table defintion with constraints requested
CREATE TABLE IF NOT EXISTS vehicles (
    vehicle_id SERIAL PRIMARY KEY,  -- Unique identifier for each vehicle
    model VARCHAR(100) NOT NULL,    -- Vehicle model name
    plate_no VARCHAR(20) NOT NULL UNIQUE, -- Unique license plate number
    type VARCHAR(50) NOT NULL,      -- Vehicle type (e.g., Truck, Van)
    status VARCHAR(20) NOT NULL CHECK (status IN ('Active', 'In Transit', 'Under Maintenance','Retired')),
    capacity INT NOT NULL CHECK (capacity > 0) -- Maximum load capacity
);

-- maintenance table defintion with constraints requested
CREATE TABLE IF NOT EXISTS maintenance (
    maintenance_id SERIAL PRIMARY KEY, -- Unique maintenance record
    vehicle_id INT NOT NULL,           -- Vehicle undergoing maintenance
    maintenance_date TIMESTAMP NOT NULL,
    cost DECIMAL(10,2) NOT NULL CHECK (cost >= 0), -- Maintenance cost
    description TEXT                -- Optional details about maintenance
);
```

B6.2. 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.

```
-- PASSING INSERTS FOR vehicles TABLE
INSERT INTO vehicles (model, plate_no, type, status, capacity)
VALUES
('Toyota Hilux', 'RAD123A', 'Truck', 'Active', 3000),
('Isuzu NPR', 'RAD456B', 'Van', 'In Transit', 2500);


-- FAILING INSERTS FOR vehicles TABLE (Wrapped in a ROLLBACK Block)

BEGIN;

-- Fails due to invalid status (not in allowed list)
INSERT INTO vehicles (model, plate_no, type, status, capacity)
VALUES ('Mitsubishi Fuso', 'RAD789C', 'Truck', 'Broken', 4000);

-- Fails due to negative capacity (violates CHECK constraint)
INSERT INTO vehicles (model, plate_no, type, status, capacity)
VALUES ('Nissan Caravan', 'RAD999D', 'Van', 'Active', -500);

ROLLBACK;  -- Undo the failing inserts so only valid rows remain
```

Run select query to check if changes were committed in both tables.

1. vehicles

vehicle_id [PK] integer	model character varying (100)	plate_no character varying (20)	type character varying (50)	status character varying (20)	capacity integer
1	Mitsubishi Fuso	RWA-654E	Truck	Retired	15000
2	Mercedes-Benz Actros	RWA-789C	Truck	Under Maintenance	20000
3	Toyota Hiace	RWA-456B	Van	Active	1500
4	Volvo FH16	RWA-123A	Truck	Active	18000
5	Isuzu NPR	RWA-321D	Truck	Active	12000
6	Toyota Hilux	RAD123A	Truck	Active	3000
7	Isuzu NPR	RAD456B	Van	In Transit	2500

2. maintenace

maintenance_id [PK] integer	vehicle_id integer	maintenance_date timestamp without time zone	cost numeric (10,2)	description text
1	1	2025-10-20 18:09:02.619706	50000.00	Engine service
2	2	2025-10-27 18:09:02.619706	25000.00	Brake pad replacement

```
-- Failing rows (should violate constraints)
BEGIN;

-- Negative quantity
INSERT INTO ProductionOrder (product_id, batch_number, quantity, start_date, end_date, status)
VALUES (106, 'BATCH006', -10, '2025-10-25 08:00:00', '2025-10-25 16:00:00', 'Planned');

-- End date before start date
INSERT INTO ProductionOrder (product_id, batch_number, quantity, start_date, end_date, status)
VALUES (107, 'BATCH007', 100, '2025-10-26 10:00:00', '2025-10-26 08:00:00', 'Completed');

ROLLBACK;  -- Undo failing inserts
```

B6. 3. Show clean error handling for failing cases.

Below is the error message for failed cases in both tables.

1. vehicles

```
ERROR:  Failing row contains (8, Mitsubishi Fuso, RAD789C, Truck, Broken, 4000).new row for relation "vehicles" violates check constraint
"vehicles_status_check"

ERROR:  new row for relation "vehicles" violates check constraint "vehicles_status_check"
SQL state: 23514
Detail: Failing row contains (8, Mitsubishi Fuso, RAD789C, Truck, Broken, 4000).
```

2. maintenance.

```
ERROR:  Failing row contains (3, 1, 2025-10-30 18:11:04.789403, -10000.00, Oil change).new row for relation "maintenance" violates check
constraint "maintenance_cost_check"

ERROR:  new row for relation "maintenance" violates check constraint "maintenance_cost_check"
SQL state: 23514
Detail: Failing row contains (3, 1, 2025-10-30 18:11:04.789403, -10000.00, Oil change).
```

B7: E–C–A Trigger for Denormalized Totals (small DML set)
B7.1. Create an audit table Vehicle_AUDIT(bef_total NUMBER, aft_total NUMBER, changed_at TIMESTAMP, key_col VARCHAR2(64)).

```sql
-- Vehicle_AUDIT Table Definition

CREATE TABLE IF NOT EXISTS vehicle_audit (
    bef_total NUMERIC,             -- Total before the change
    aft_total NUMERIC,             -- Total after the change
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  -- When the change occurred
    key_col VARCHAR(64)            -- Key or identifier of the affected record
);
```

B7. 2. Implement a statement-level AFTER INSERT/UPDATE/DELETE trigger on Maintenance that recomputes denormalized totals in Vehicle once per statement.

First of all we need to modify the vehicles to include total_maintenance_cost needed for the computation, below is DDL statement to achieve this

ALTER TABLE vehicles
ADD COLUMN total_maintenance_cost NUMERIC(10,2) DEFAULT 0;

Below is triggered function that will get executed AFTER INSERT/UPDATE/DELETE is made on maintenance table

```
CREATE OR REPLACE FUNCTION recompute_vehicle_totals()
RETURNS TRIGGER AS $$
BEGIN
    -- Recompute and update each vehicle's total maintenance cost
    -- The subquery calculates the SUM of all maintenance costs per vehicle.
    -- COALESCE ensures that if a vehicle has no maintenance records,
    -- its total_maintenance_cost is set to 0 instead of NULL.
    UPDATE vehicles v
    SET total_maintenance_cost = COALESCE((
        SELECT SUM(m.cost)
        FROM maintenance m
        WHERE m.vehicle_id = v.vehicle_id
    ), 0);

    -- Since this function will be used in a statement-level trigger,
    -- it does not process individual rows — returning NULL is required.
    ----------------------------------------------------------------------
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

-- Invoke procedure defined below AFTER INSERT, UPDATE, or DELETE

CREATE TRIGGER trg_recompute_vehicle_totals
AFTER INSERT OR UPDATE OR DELETE ON maintenance
FOR EACH STATEMENT
EXECUTE FUNCTION recompute_vehicle_totals();
```

Below is the screenshot that shows total_maintenance_cost in vehicles after insert is made on the maintenance table.

vehicle_id [PK] integer	model character varying (100)	plate_no character varying (20)	type character varying (50)	status character varying (20)	capacity integer	total_maintenance_cost numeric (10,2)
1	Mitsubishi Fuso	RWA-654E	Truck	Retired	15000	50000.00
2	Mercedes-Benz Actros	RWA-789C	Truck	Under Maintenance	20000	25000.00
3	Toyota Hiace	RWA-456B	Van	Active	1500	150000.00
4	Volvo FH16	RWA-123A	Truck	Active	18000	35000.00
5	Isuzu NPR	RWA-321D	Truck	Active	12000	0.00

B7. 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.

Below is the query for implementing a DML transaction for achieving this

```
-- MIXED DML transaction: Controlled maintenance record modifications
BEGIN;

-- Insert 2 new maintenance records (simulate new services)
INSERT INTO maintenance (vehicle_id, maintenance_date, cost, description)
VALUES
    (1, CURRENT_TIMESTAMP - INTERVAL '2 days', 450.00, 'Engine oil replacement'),
    (2, CURRENT_TIMESTAMP - INTERVAL '1 day', 300.00, 'Brake pad replacement');

-- Update 1 existing maintenance record (adjust cost)
UPDATE maintenance
SET cost = cost + 50.00, description = 'Cost adjusted after inspection'
WHERE maintenance_id = 3;

-- Delete 1 old or invalid record (simulate data cleanup)
DELETE FROM maintenance
WHERE maintenance_id = 4;

-- Commit the transaction to apply the net 4 changes
COMMIT;
```

Let the query maintenance table to check if net committed rows across the project remain ≤10. Using the following query

```
SELECT COUNT(*) AS total_committed_maintenance_records
FROM maintenance;
```

Below is the result of the query and it show the total rows remain <=10

total_committed_maintenance_records bigint
5

B7. 4. Log before/after totals to the audit table (2–3 audit rows).

```
-- step 1. Log before totals
INSERT INTO vehicle_audit (bef_total, aft_total, key_col)
SELECT
    total_maintenance_cost AS bef_total,
    NULL AS aft_total,          -- No after value yet
    vehicle_id AS key_col
FROM vehicles
WHERE vehicle_id IN (1,2,3);

-- Step 2: Make updates
UPDATE vehicles
SET total_maintenance_cost = total_maintenance_cost * 1.1
WHERE vehicle_id IN (1,2,3);
```

Below is the result from vehicle_audit table

bef_total numeric	aft_total numeric	changed_at timestamp without time zone	key_col character varying (64)
50450.00	[null]	2025-10-30 19:17:47.503522	1
25300.00	[null]	2025-10-30 19:17:47.503522	2
0.00	[null]	2025-10-30 19:17:47.503522	3

```
--  Log after totals
UPDATE vehicle_audit a
SET aft_total = v.total_maintenance_cost
FROM vehicles v
WHERE CAST(a.key_col AS INTEGER) = v.vehicle_id
  AND a.aft_total IS NULL;
```

Below data from vehicle_audit after executing the above query

bef_total numeric	aft_total numeric	changed_at timestamp without time zone	key_col character varying (64)
50450.00	55495.00	2025-10-30 19:17:47.503522	1
25300.00	27830.00	2025-10-30 19:17:47.503522	2
0.00	0.00	2025-10-30 19:17:47.503522	3

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

B8.1 1. Create table HIER(parent_id, child_id) for a natural hierarchy (domain-specific).

```
-- below is HIER Table table definition
CREATE TABLE HIER (
    parent_id INT NOT NULL,
    child_id INT NOT NULL,
    PRIMARY KEY (parent_id, child_id)
);
```

B8.1 2.Insert 6–10 rows forming a 3-level hierarchy.

Below is the query to achieve the above

```
-- 2 Insert 6–10 Rows Forming a 3-Level Hierarchy
INSERT INTO HIER (parent_id, child_id) VALUES
(1, 3),   -- Root 1 → Sub 3
(1, 4),   -- Root 1 → Sub 4
(2, 5),   -- Root 2 → Sub 5
(3, 6),   -- Sub 3 → Product 6
(3, 7),   -- Sub 3 → Product 7
(4, 8);   -- Sub 4 → Product 8
```

B8. 3. Write a recursive WITH query to produce (child_id, root_id, depth) and join to Product or its parent to compute rollups; return 6–10 rows total.

```
WITH RECURSIVE hier_rollup AS (
    -- Base case: immediate parent-child links
    SELECT
        child_id,
        parent_id AS root_id,
        1 AS depth
    FROM HIER

    UNION ALL

    -- Recursive step: climb up the hierarchy
    SELECT
        h.child_id,
```

```
        r.root_id,
        r.depth + 1
    FROM HIER h
    JOIN hier_rollup r
      ON h.parent_id = r.child_id
)
SELECT child_id, root_id, depth
FROM hier_rollup
ORDER BY root_id, depth;
```

Below is the output of the query

child_id integer	root_id integer	depth integer
3	1	1
4	1	1
8	1	2
6	1	2
7	1	2
5	2	1
6	3	1
7	3	1
8	4	1

B9 :Mini-Knowledge Base with Transitive Inference (≤10 facts)
B9. 1 Create table TRIPLE(s VARCHAR2(64), p VARCHAR2(64), o VARCHAR2(64)).

--- DDL defintion to create TRIPLE Table

```
CREATE TABLE TRIPLE (
   s VARCHAR(64) NOT NULL,  -- Subject
   p VARCHAR(64) NOT NULL,  -- Predicate
   o VARCHAR(64) NOT NULL,  -- Object
   PRIMARY KEY (s, p, o)
);
```

B9. 2. Insert 8–10 domain facts relevant to your project (e.g., simple type hierarchy or rule implications).

Below is DML query for inserting data into TRIPLE

```
INSERT INTO TRIPLE (s, p, o) VALUES
('Laptop', 'isA', 'Computer'),
('Desktop', 'isA', 'Computer'),
('Computer', 'isA', 'Electronics'),
('Tablet', 'isA', 'Electronics'),
('Electronics', 'isA', 'Device'),
('Mouse', 'isA', 'Peripheral'),
```

('Keyboard', 'isA', 'Peripheral'),
('Peripheral', 'isA', 'Device');

Below is the query result after inserting

s [PK] character varying (64)	p [PK] character varying (64)	o [PK] character varying (64)
Laptop	isA	Computer
Desktop	isA	Computer
Computer	isA	Electronics
Tablet	isA	Electronics
Electronics	isA	Device
Mouse	isA	Peripheral
Keyboard	isA	Peripheral
Peripheral	isA	Device

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

Below is the query for implementing transitive isA

```
WITH RECURSIVE isa_inference AS (
   -- Base case: direct isA relationships
   SELECT s, o, 1 AS depth
   FROM TRIPLE
   WHERE p = 'isA'

   UNION ALL

   -- Recursive step: infer transitive isA
   SELECT i.s, t.o, i.depth + 1
   FROM isa_inference i
   JOIN TRIPLE t
     ON i.o = t.s
   WHERE t.p = 'isA'
)
SELECT s AS subject, o AS inferred_object, depth
FROM isa_inference
ORDER BY s, depth
LIMIT 10;  -- limit output to ≤10 rows
```

Below is the query result

subject character varying (64)	inferred_object character varying (64)	depth integer
Computer	Electronics	1
Computer	Device	2
Desktop	Computer	1
Desktop	Electronics	2
Desktop	Device	3
Electronics	Device	1
Keyboard	Peripheral	1
Keyboard	Device	2
Laptop	Computer	1
Laptop	Electronics	2

B10 :Business Limit Alert (Function + Trigger) (row-budget safe)
10. 1. Create BUSINESS_LIMITS(rule_key VARCHAR(64), threshold NUMBER, active CHAR(1)CHECK(active IN('Y','N'))) and seed exactly one active rule**.**

```
-- Step 1: DDL table definition for BUSINESS_LIMITS
CREATE TABLE BUSINESS_LIMITS (
    rule_key VARCHAR(64) PRIMARY KEY,
    threshold NUMERIC NOT NULL,
    active CHAR(1) NOT NULL CHECK (active IN ('Y','N'))
);

-- Step 2: Seed exactly one active rule
INSERT INTO BUSINESS_LIMITS (rule_key, threshold, active)
VALUES ('LIMIT_001', 100000, 'Y');

– step 3: Verify the insert
SELECT * FROM BUSINESS_LIMITS;
```

Running the above query we get the following result

rule_key [PK] character varying (64)	threshold numeric	active character
LIMIT_001	100000	Y

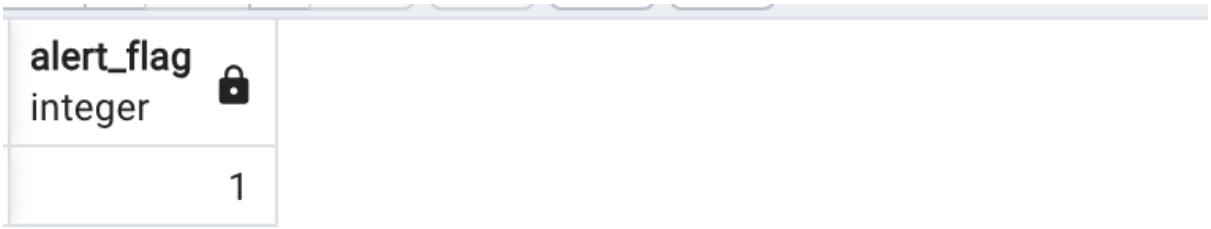
2. Implement function fn_should_alert(...) that reads BUSINESS_LIMITS and inspects current data
in QualityCheck or ProductionOrder to decide a violation (return 1/0).

Below is the procedure function that reads BUSINESS_LIMITS and performs the inspection to check whether the maintenance cost exceeds the threshold. If the result is 1 then maintenance cost exceed the threshold, otherwise not

```
CREATE OR REPLACE FUNCTION fn_should_alert(p_vehicle_id INTEGER)
RETURNS INTEGER AS $$
DECLARE
   v_threshold NUMERIC;
   v_metric NUMERIC;
BEGIN
   -- Step 1: Read active threshold
   SELECT threshold
     INTO v_threshold
     FROM business_limits
    WHERE active = 'Y'
    LIMIT 1;  -- In case there are multiple active rules

   -- Step 2: Calculate metric (total maintenance cost)
   SELECT COALESCE(SUM(cost), 0)
     INTO v_metric
     FROM maintenance
    WHERE vehicle_id = p_vehicle_id;

   -- Step 3: Compare with threshold
   IF v_metric > v_threshold THEN
      RETURN 1; -- Violation
   ELSE
      RETURN 0; -- No violation
   END IF;

EXCEPTION
   WHEN NO_DATA_FOUND THEN
```

```
        RETURN 0;  -- No active rule, assume no alert
    WHEN OTHERS THEN
        RETURN 0;  -- Other errors
END;
$$ LANGUAGE plpgsql;

-- invoke the function and pass foreign key value of the vehicle
SELECT fn_should_alert(1) AS alert_flag;
```

Below is the result of the query



alert_flag integer
1

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

```
-- Define a procedural function
CREATE OR REPLACE FUNCTION trg_check_business_limit()
RETURNS TRIGGER AS $$
DECLARE
    v_alert INTEGER;
BEGIN
    -- Step 1: Call fn_should_alert with the vehicle being modified
    v_alert := fn_should_alert(NEW.vehicle_id);

    -- Step 2: If a violation occurs, raise an error
    IF v_alert = 1 THEN
        RAISE EXCEPTION
            'Business limit violation: total maintenance cost for vehicle % exceeds threshold.',
            NEW.vehicle_id
            USING ERRCODE = 'P0001';  -- Custom application error
    END IF;

    -- Step 3: Allow insert/update if within limit
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Create a trigger to invoke the procedure function defined below
```

```
CREATE TRIGGER check_business_limit
BEFORE INSERT OR UPDATE
ON maintenance
FOR EACH ROW
EXECUTE FUNCTION trg_check_business_limit();
```

Let write a query to test if the above implementation is working fine

```
-- Insert a maintenance record that causes the total to exceed the limit
INSERT INTO maintenance (vehicle_id, maintenance_date, cost, description)
VALUES (1, NOW(), 12000000, 'Oil change');
```

Below is the result of the query

```
ERROR:  Business limit violation: total maintenance cost for vehicle 1 exceeds threshold.
CONTEXT:  PL/pgSQL function trg_check_business_limit() line 10 at RAISE
```

The above error message was raised because the maintenance cost for this vehicle is 12,000,00 and it exceed the threshold which is 1,000,000

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

```
-- Two passing cases on maintenance table ensure cost <=100000
INSERT INTO maintenance (vehicle_id, maintenance_date, cost, description)
VALUES (1, NOW(), 1000, 'Oil change'),
(1, NOW(), 200, 'Oil change');
```

```
-- Two failing DML cases (cost exceed threshold)
BEGIN;
INSERT INTO maintenance (vehicle_id, maintenance_date, cost, description)
VALUES (1, NOW(), 5000000, 'Oil change'),
(1, NOW(),4000000, 'Oil change');
ROLLBACK;  -- rollback failing case
```

Below is the result for failed DML since cost exceed threshold

```
ERROR:  Business limit violation: total maintenance cost for vehicle 1 exceeds threshold.
CONTEXT:  PL/pgSQL function trg_check_business_limit() line 10 at RAISE
```