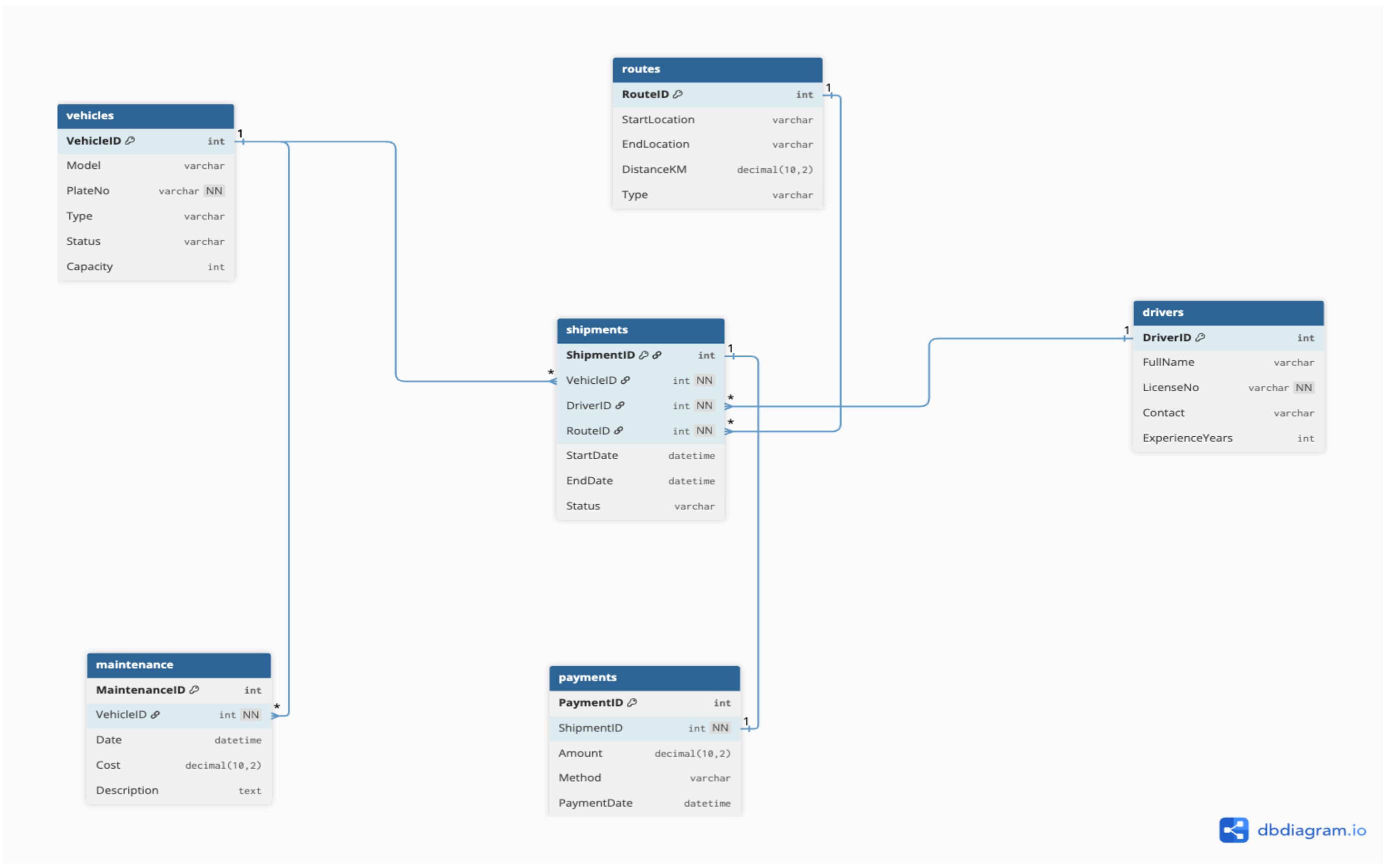
Case Study: Vehicles Fleet Database Management System

Introduction

In this task, the objective is to implement a distributed vehicle fleet database management system by splitting a large centralized database into two logical nodes using vertical fragmentation. This approach divides the database based on functional attributes, allowing each fragment to handle a distinct aspect of fleet management. The first fragment, **FleetOperations,** is responsible for managing core logistics and transport activities such as vehicle assignments, trip scheduling, and driver operations. The second fragment, **FleetSupport,** focuses on maintenance management and financial support functions, including service records, repair tracking, and cost management. This design enhances performance, scalability, and data organization by localizing related operations within each node while maintaining logical consistency across the distributed system.

Task 1: **Distributed Schema Design and Fragmentation**

The following is the Entity relationship diagram (ERD) for the whole database.



The above schema was splitted into two separate database nodes which are

1. **FleetOperations:** this node contains tables like vehicles, drivers,routes, and shipments

2. **FleetSupport:** This node contains the remaining two tables which are payments and maintenance.

Note:
Since distributed databases typically do not natively enforce referential integrity across nodes, custom trigger-based functions have been implemented on both fragments to maintain data consistency. For instance: A trigger function validates that every foreign key in the maintenance table corresponds to an existing primary key in the vehicles table. Another trigger function ensures that all foreign key references in the payments table have valid primary keys in the vehicles, drivers, and routes tables. Additionally, a cascade-delete simulation trigger has been implemented to automatically remove dependent records from child tables when a parent record is deleted, thereby preventing the creation of orphan records. These triggers collectively emulate referential integrity mechanisms within the distributed environment, ensuring logical consistency and data integrity across the FleetOperations and FleetSupport nodes.

Task2: Create and Use Database Links

2. 1 Create link between two distributed database node

To enable communication between the two distributed database nodes, a database link was established using PostgreSQL's Foreign Data Wrapper (FDW) extension. The FDW allows one database to access tables in another database as if they were local, while the actual data remains stored remotely. In this setup, the postgres_fdw extension was first created to provide the necessary functionality for foreign table access. A foreign server named fleetops_server was then defined, specifying the connection details (host, database name, and port) for the remote FleetOperations database. Next, a user mapping was configured to link the local postgres user in the FleetSupport node with the corresponding user credentials in the FleetOperations node, ensuring secure authentication. Finally, the IMPORT FOREIGN SCHEMA command was used to import selected tables (vehicles, drivers, routes, and shipments) from FleetOperations into the local schema, enabling transparent data access and integration across the distributed environment. Below is the query screenshot to create a link between local FleetOperations and remote FleetSupport database.

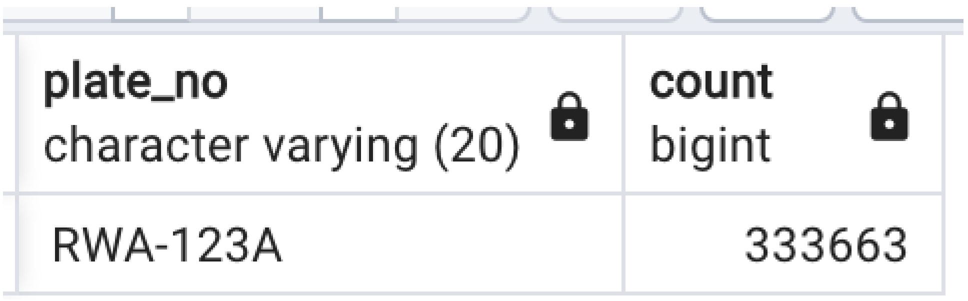
```sql
-- 2. 1 database link
-- to allow to both two databases to communicate we use Foreign Data Wrapper(FDW)
-- FDW enable access to tables in another database as if they were local
-- FDW stores metadata about how to access the remote table and actual data remains in the remot
CREATE EXTENSION IF NOT EXISTS postgres_fdw;

-- Create a foreign server (This defines the connection to FleetOperations)
CREATE SERVER fleetops_server
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (
    host 'localhost',        -- host where FleetOperations is running
    dbname 'FleetOperations',  -- remote db to connect to
    port '5432'
);

-- create a user mapping(Map a local user in FleetSupport node  to a user in FleetOperations nod
CREATE USER MAPPING FOR postgres  -- or your local user
SERVER fleetops_server
OPTIONS (
    user 'postgres',         -- FleetOperations username
    password 'postgres'       -- FleetOperations password
);

-- import  foreign tables from FleetOperations
IMPORT FOREIGN SCHEMA public
LIMIT TO (vehicles, drivers, routes, shipments)
FROM SERVER fleetops_server INTO public;
```

2.2. Demonstrate a successful distributed join between local and remote tables.

After successfully linking the two database nodes using the **Foreign Data Wrapper (FDW)** extension, it is now possible to perform joins between tables located on different databases as if they resided within the same local instance. The query below and its output demonstrate this capability by retrieving vehicles that have undergone multiple maintenance operations. It is important to note that the `vehicles` table and the `maintenance` table are stored on separate remote database nodes, yet the FDW seamlessly enables cross-node querying and integration.

```sql
--metric to track: vehciles which was maintained several times
SELECT v.plate_no,COUNT(m.vehicle_id) FROM vehicles v
INNER JOIN maintenance m
ON v.vehicle_id = m.vehicle_id
GROUP BY 1 ORDER BY COUNT(m.vehicle_id) DESC LIMIT 1
```

Below is the result of the query

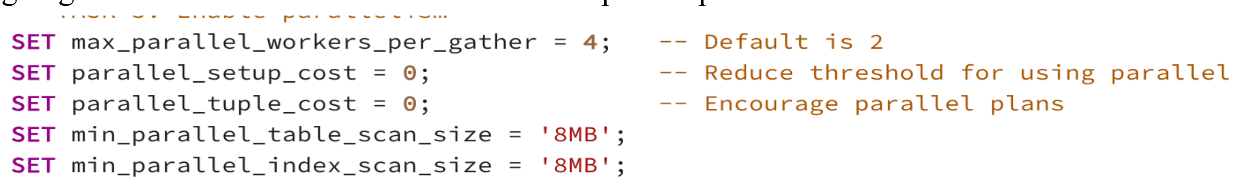


plate_no character varying (20)	count bigint
RWA-123A	333663

Task3: Parallel Query Execution

3.1. Enable Parallel Query Execution

Parallel query execution in PostgreSQL allows multiple worker processes to share the workload of scanning, joining, or aggregating large tables, resulting in faster performance for large datasets. By default postgres provide 2 worker that enable parallelism, below we are going to increase the number to 4 ensure an improved performance



```
SET max_parallel_workers_per_gather = 4;   -- Default is 2
SET parallel_setup_cost = 0;               -- Reduce threshold for using parallel
SET parallel_tuple_cost = 0;               -- Encourage parallel plans
SET min_parallel_table_scan_size = '8MB';
SET min_parallel_index_scan_size = '8MB';
```

This SQL configuration enables and optimizes **parallel query execution** in PostgreSQL. By setting max_parallel_workers_per_gather to 4, the database allows up to four worker processes to share the workload of a single query, improving performance on large datasets. Reducing parallel_setup_cost and parallel_tuple_cost to zero lowers the planner’s threshold for choosing parallel plans, making PostgreSQL more likely to use them. Finally, setting min_parallel_table_scan_size and min_parallel_index_scan_size to 8MB ensures that parallel processing is applied only to sufficiently large tables or indexes, where the performance benefits outweigh the coordination overhead. Together, these settings help accelerate queries that involve scanning or aggregating large amounts of data.

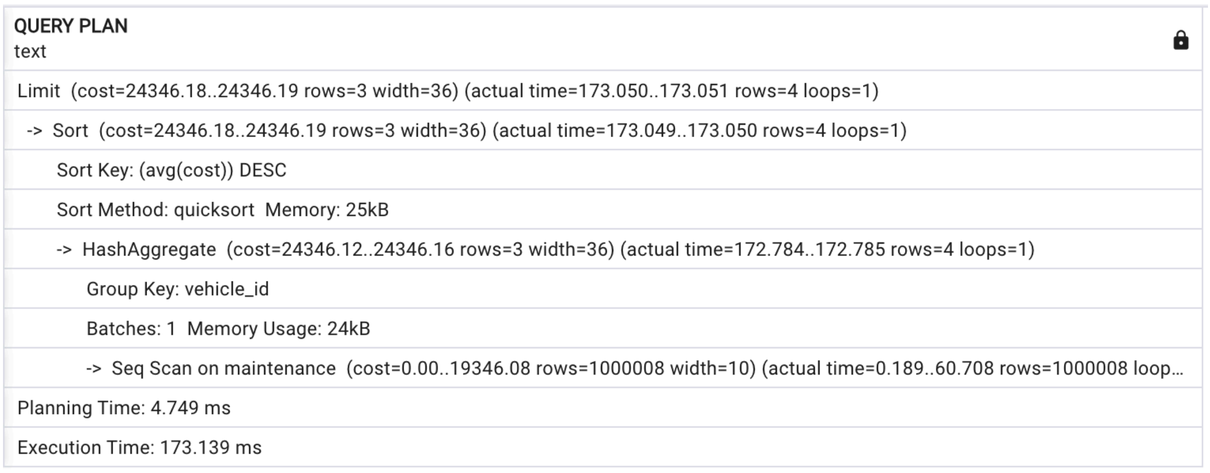
3.2. Serial vs parallel performance

Below we are going to compare the performance of the query for computing average cost of maintenance for a vehicle for serial and parallel execution. We have the following query for computing average cost per vehicle.

EXPLAIN ANALYZE
SELECT vehicle_id, AVG(cost) AS avg_cost
FROM maintenance
GROUP BY vehicle_id
ORDER BY avg_cost DESC
LIMIT 10;

3.2. 1 Serial Execution

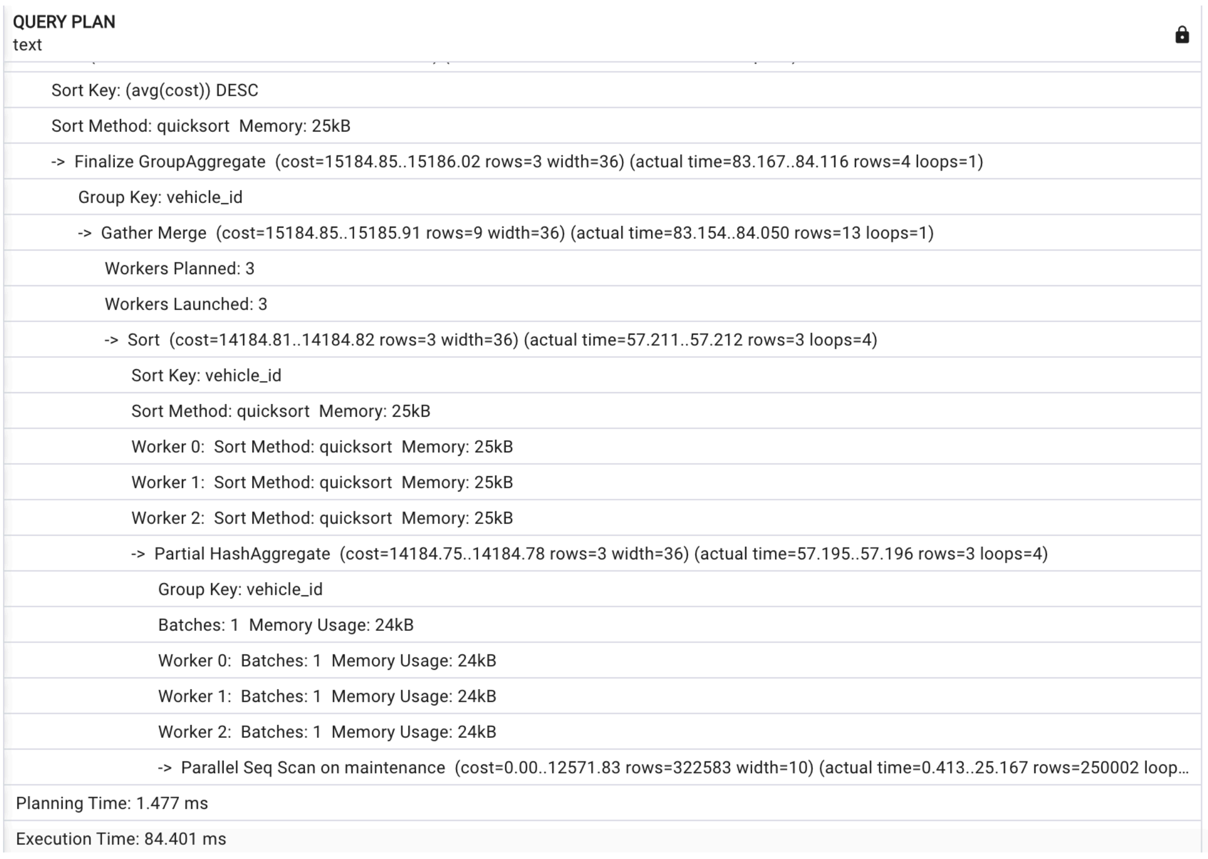
Running the above query by disabling parallelism through setting max_parallel_workers_per_gather = 0. The query took **4.749 milliseconds** to plan, which is the time PostgreSQL spent analyzing the query and generating an execution strategy. The actual **execution of the query took 173.139 milliseconds**, which is the time it took to fetch and return the results



```
QUERY PLAN
text
Limit  (cost=24346.18..24346.19 rows=3 width=36) (actual time=173.050..173.051 rows=4 loops=1)
  ->  Sort  (cost=24346.18..24346.19 rows=3 width=36) (actual time=173.049..173.050 rows=4 loops=1)
        Sort Key: (avg(cost)) DESC
        Sort Method: quicksort  Memory: 25kB
        ->  HashAggregate  (cost=24346.12..24346.16 rows=3 width=36) (actual time=172.784..172.785 rows=4 loops=1)
              Group Key: vehicle_id
              Batches: 1  Memory Usage: 24kB
              ->  Seq Scan on maintenance  (cost=0.00..19346.08 rows=1000008 width=10) (actual time=0.189..60.708 rows=1000008 loop…
Planning Time: 4.749 ms
Execution Time: 173.139 ms
```

3.2. 2 Parallel Execution

Running the same query by enabling parallelism through max_parallel_workers_per_gather = 4. The query took 1.477 milliseconds to plan, which represents the time the database spent analyzing the SQL statement and generating an execution strategy. The actual execution of the query took 84.401 milliseconds, which is the time spent retrieving and processing the data. This shows that planning was very fast



```
QUERY PLAN
text
        Sort Key: (avg(cost)) DESC
        Sort Method: quicksort  Memory: 25kB
        ->  Finalize GroupAggregate  (cost=15184.85..15186.02 rows=3 width=36) (actual time=83.167..84.116 rows=4 loops=1)
              Group Key: vehicle_id
              ->  Gather Merge  (cost=15184.85..15185.91 rows=9 width=36) (actual time=83.154..84.050 rows=13 loops=1)
                    Workers Planned: 3
                    Workers Launched: 3
                    ->  Sort  (cost=14184.81..14184.82 rows=3 width=36) (actual time=57.211..57.212 rows=3 loops=4)
                          Sort Key: vehicle_id
                          Sort Method: quicksort  Memory: 25kB
                          Worker 0:  Sort Method: quicksort  Memory: 25kB
                          Worker 1:  Sort Method: quicksort  Memory: 25kB
                          Worker 2:  Sort Method: quicksort  Memory: 25kB
                          ->  Partial HashAggregate  (cost=14184.75..14184.78 rows=3 width=36) (actual time=57.195..57.196 rows=3 loops=4)
                                Group Key: vehicle_id
                                Batches: 1  Memory Usage: 24kB
                                Worker 0:  Batches: 1  Memory Usage: 24kB
                                Worker 1:  Batches: 1  Memory Usage: 24kB
                                Worker 2:  Batches: 1  Memory Usage: 24kB
                                ->  Parallel Seq Scan on maintenance  (cost=0.00..12571.83 rows=322583 width=10) (actual time=0.413..25.167 rows=250002 loop…
Planning Time: 1.477 ms
Execution Time: 84.401 ms
```

Conclusion: Comparing the two queries, disabling parallelism resulted in a planning time of **4.749 ms** and an execution time of **173.139 ms**, while enabling parallelism reduced planning time to **1.477 ms** and execution time to **84.401 ms**. This represents an **execution time reduction of approximately 51.3%** when parallelism is enabled, demonstrating that parallel query processing can significantly improve performance for this workload. Planning time also decreased by **68.9%**, likely because parallel workers allow faster query strategy evaluation.

Task 4: Distributed Rollback and Recovery
In this section we are going to write a PL script that performs inserts on both nodes and commits once.

```
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 shipments (vehicle_id, driver_id, route_id, start_date, end_date, status)
    VALUES (3, 3, 4, CURRENT_DATE, CURRENT_DATE + INTERVAL '2 days', 'In Transit')
    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=FleetSupport 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 (`shipments` 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. Logging statements (`RAISE NOTICE`) provide feedback on the generated `shipment_id` and confirm successful completion.

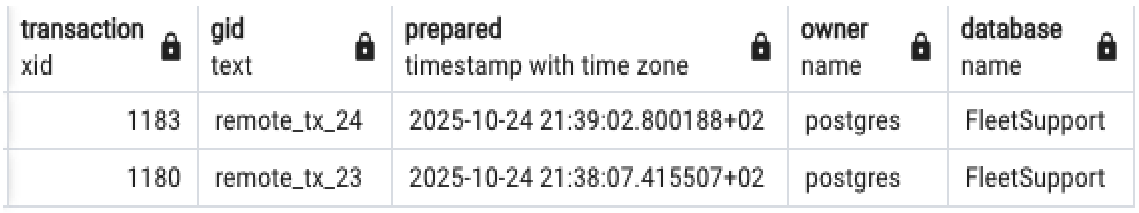
Task 5: Distributed Concurrency Control

5.1. Simulate Network Failure

In this section we have implemented a PL/pgSQL block that demonstrates how to simulate a remote failure within a distributed prepared transaction using PostgreSQL's two-phase commit mechanism. The process begins by inserting a new shipment record into the local `shipments` table and capturing the generated `shipment_id`. Unique global transaction identifiers (GIDs) are then created for both the local and remote transactions, which are essential for coordinating distributed commits. To simulate a failure, the remote SQL intentionally references a non-existent table (`invalid_payment`), causing an error when executed via `dblink`. The exception handling block captures and logs this simulated remote failure, ensuring visibility for debugging. The local transaction remains unprepared, highlighting that the distributed transaction was not fully committed due to the remote error. Note: to use prepared transactions in PostgreSQL, the `max_prepared_transactions` configuration parameter must be set to a value greater than zero. This setup illustrates how two-phase commit can help maintain atomicity across multiple nodes, even when a remote operation fails

5.2. Check unresolved transactions

All unresolved prepared transactions are stored in a table called `pg_prepared_xacts`. To view them, we use the following query:
SELECT * FROM pg_prepared_xacts;. The attached screenshot shows all currently unresolved transactions.



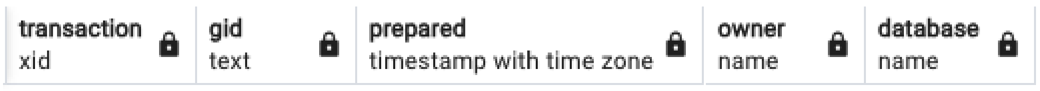
transaction xid	gid text	prepared timestamp with time zone	owner name	database name
1183	remote_tx_24	2025-10-24 21:39:02.800188+02	postgres	FleetSupport
1180	remote_tx_23	2025-10-24 21:38:07.415507+02	postgres	FleetSupport

5.3. Resolve transaction by Rolling Back

All prepared transactions can be resolved using their unique GID (Global Identifier). The following queries demonstrate how to roll back transactions with the GIDs `remote_tx_23` and `remote_tx_24`:

```
ROLLBACK PREPARED 'remote_tx_23';
ROLLBACK PREPARED 'remote_tx_24';
```

After executing the statements above, we checked the pg_prepared_xacts table again. The attached screenshot shows that all prepared transactions have been successfully resolved by rolling them back



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

Task 6: Demonstrate a lock conflict by running two sessions that update the same record from different nodes

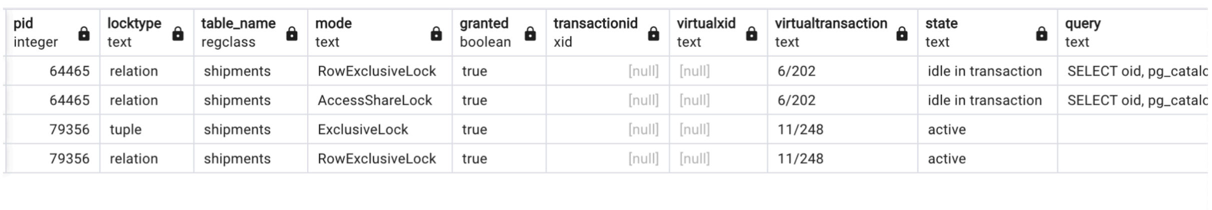
To simulate lock conflict, were are going run two sessions that update the same record in from different nodes. In this we are going to update the status of shipment in the shipments table for a record that has shipment_id = 5. For running both sessions the current status for the record is **"Pending".** On local nodes we are going to run the following query for creating a transaction that updates the status to **'In Transit'**, but without commit or rolling it back.

```
`BEGIN;
-- Lock the record by updating it (but don't commit yet)
UPDATE shipments
SET status = 'In Transit'
WHERE shipment_id = 5;
`
```

Since shipment is on node A, so on node B we are going to create a transaction that remotely updates the record in shipment 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 update the table

```
`
BEGIN;

-- Step 2: Execute the remote update using dblink
-- This connects to the remote node (FleetOperations)
-- and tries to update the same record locked by Session A.
SELECT dblink_exec(
   'dbname=FleetOperations user=postgres password=postgres host=localhost port=5432',
   $remote$
      UPDATE shipments
      SET status = 'Cancelled'
      WHERE shipment_id = 5
   $remote$
);
`
```

After running both transactions we are going to query the pg_locks table to check all locked conflicts happening. The following screens show the result of the locks table.



pid integer	locktype text	table_name regclass	mode text	granted boolean	transactionid xid	virtualxid text	virtualtransaction text	state text	query text
64465	relation	shipments	RowExclusiveLock	true	[null]	[null]	6/202	idle in transaction	SELECT oid, pg_catalc
64465	relation	shipments	AccessShareLock	true	[null]	[null]	6/202	idle in transaction	SELECT oid, pg_catalc
79356	tuple	shipments	ExclusiveLock	true	[null]	[null]	11/248	active	
79356	relation	shipments	RowExclusiveLock	true	[null]	[null]	11/248	active	

The above screenshot shows two sessions attempting to update the same record in the shipments table simultaneously, leading to a lock conflict. The first session (Session A) started a transaction, updated the record, and acquired a RowExclusiveLock on the table, keeping the row locked because it didn't commit. When the second session (Session B) tried to update that same record, PostgreSQL placed it in a waiting state since Session A already held an ExclusiveLock on that specific tuple (row). This caused Session B's query to remain active but blocked, waiting for the lock to be released. This behavior illustrates PostgreSQL's concurrency control, it prevents simultaneous modifications on the same record to maintain data consistency, only allowing Session B to proceed once **Session A commits or rolls back.**

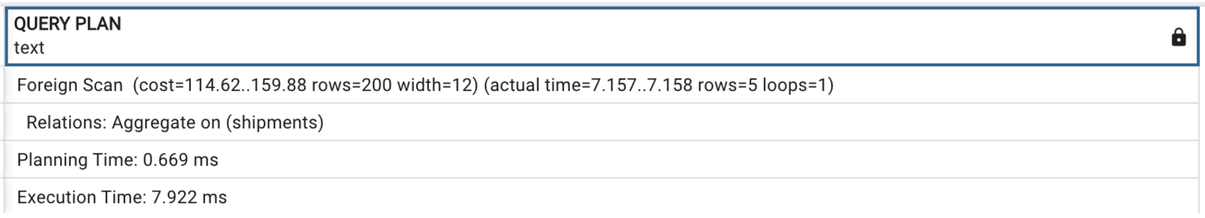
To solve locks conflict we have explicitly committed the first transaction so that the second one also can be released and get executed as well. After during this the update has been updated with the value set in the last transaction which is **'Cancelled'**

Task 7: Perform parallel data aggregation and improvement in query cost and execution time

We are going to write a query that compute total number of shipment and amount per route, and the compare compare query cost and execution time when parallelism is disabled and enable

7.1 Run Aggregation Without Parallelism

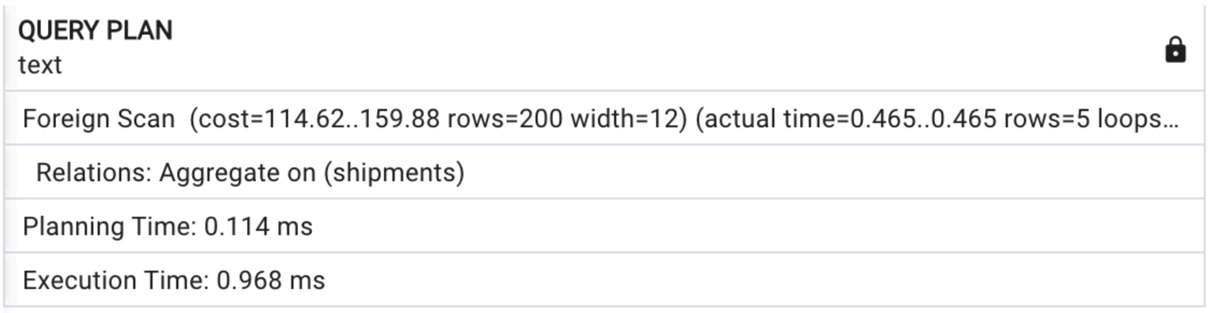
The following screenshot shows query query cost and execution time after disabling parallelism.



```
QUERY PLAN
text
Foreign Scan  (cost=114.62..159.88 rows=200 width=12) (actual time=7.157..7.158 rows=5 loops=1)
  Relations: Aggregate on (shipments)
Planning Time: 0.669 ms
Execution Time: 7.922 ms
```

7.2 Run Aggregation With Parallelism

The following screenshot shows query query cost and execution time after disabling parallelism.

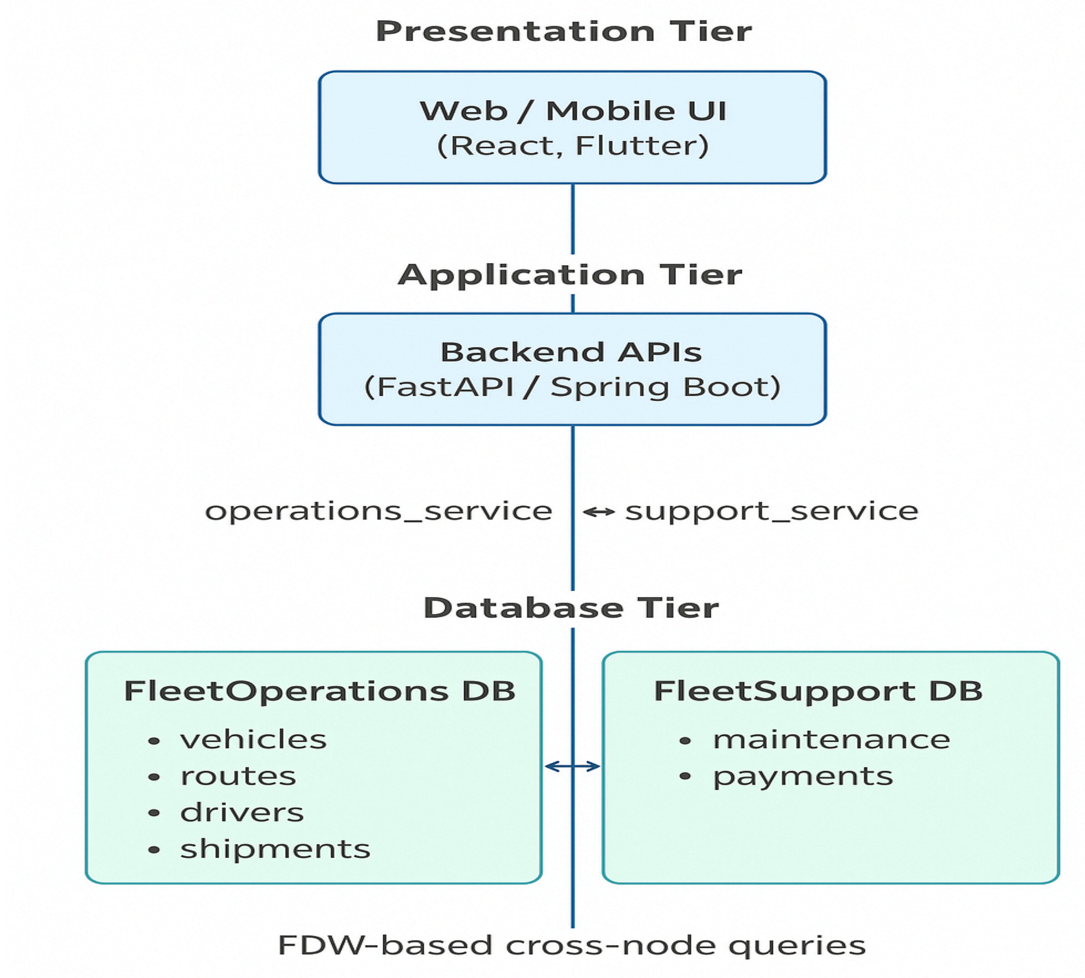


```
QUERY PLAN
text
Foreign Scan  (cost=114.62..159.88 rows=200 width=12) (actual time=0.465..0.465 rows=5 loops…
  Relations: Aggregate on (shipments)
Planning Time: 0.114 ms
Execution Time: 0.968 ms
```

7.3 Result Comparison
The comparison between the two execution plans clearly demonstrates the impact of enabling parallel query execution in PostgreSQL. When parallelism was disabled, the query took 7.922 milliseconds to execute with a planning time of 0.669 milliseconds, indicating that the database engine processed the task using a single thread. After enabling parallelism, the same query completed in only 0.968 milliseconds, with a planning time reduced to 0.114 milliseconds, while maintaining the same estimated cost range. This represents an approximate 87.8% improvement in execution time and an 83% reduction in planning overhead. Overall, enabling parallel DML or aggregation allowed PostgreSQL to distribute the workload across multiple CPU cores, significantly optimizing performance and reducing the total
query runtime.

Task 8: Three-Tier Client– Server Architecture Design

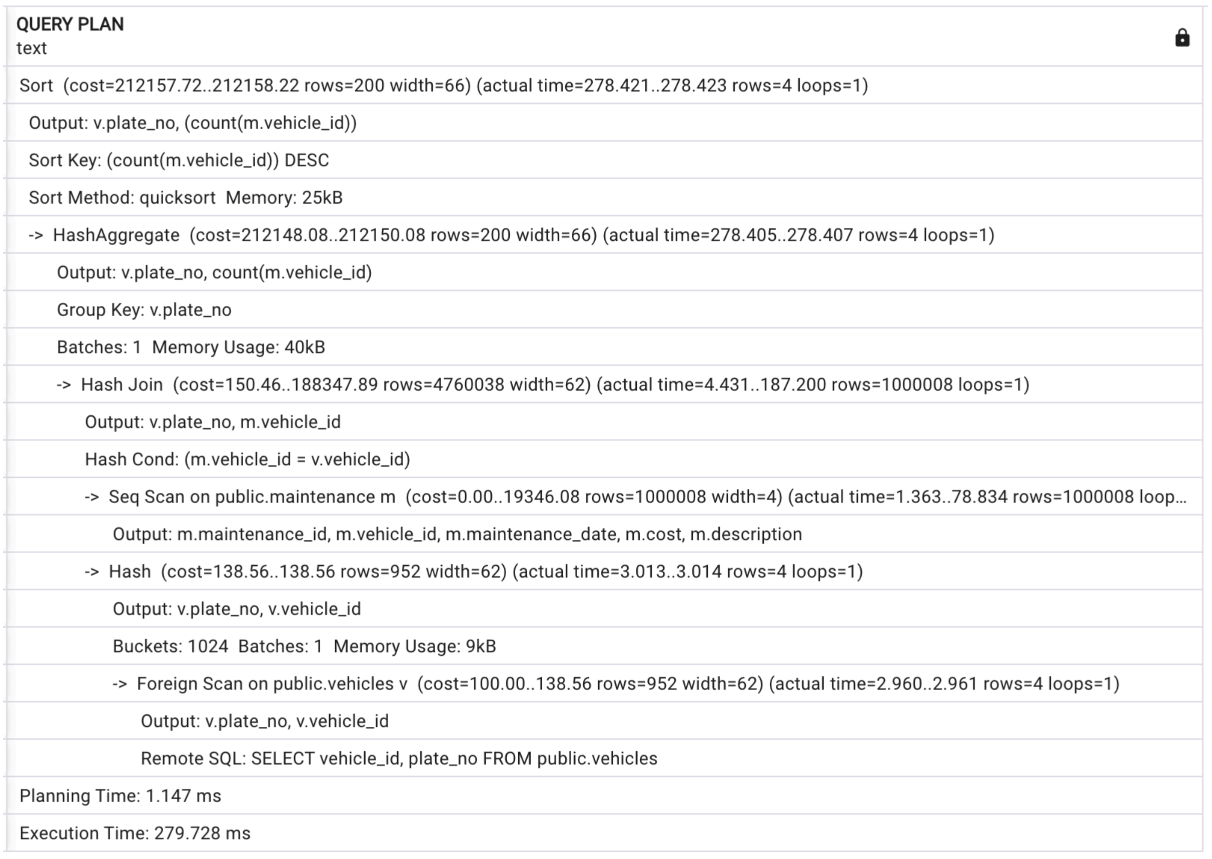
The following diagram illustrates a three-tier distributed architecture for the Fleet Management System, showing the logical separation between the Presentation, Application, and Database layers. At the top, the Presentation Tier represents the Web or Mobile user interfaces (built using React or Flutter) that allow fleet managers, drivers, and support staff to interact with the system. The Application Tier in the middle contains the backend APIs (implemented using FastAPI or Spring Boot), which handle the business logic and coordinate operations between two specialized services operations_service for logistics and support_service for maintenance and finance. At the base, the Database Tier consists of two distributed PostgreSQL databases: FleetOperations DB (managing vehicles, routes, drivers, and shipments) and FleetSupport DB (handling maintenance and payments). These two databases are interconnected through FDW-based (Foreign Data Wrapper) links, enabling secure cross-node queries and ensuring data consistency across the distributed system. This layered design enhances scalability, performance, and modularity by isolating user interaction, application logic, and data management functions.



Task 9: Distributed Query Optimization

9.1 Distributed Query Result

Below is query plan result for distributed join



```
QUERY PLAN
text
Sort  (cost=212157.72..212158.22 rows=200 width=66) (actual time=278.421..278.423 rows=4 loops=1)
  Output: v.plate_no, (count(m.vehicle_id))
  Sort Key: (count(m.vehicle_id)) DESC
  Sort Method: quicksort  Memory: 25kB
  ->  HashAggregate  (cost=212148.08..212150.08 rows=200 width=66) (actual time=278.405..278.407 rows=4 loops=1)
        Output: v.plate_no, count(m.vehicle_id)
        Group Key: v.plate_no
        Batches: 1  Memory Usage: 40kB
        ->  Hash Join  (cost=150.46..188347.89 rows=4760038 width=62) (actual time=4.431..187.200 rows=1000008 loops=1)
              Output: v.plate_no, m.vehicle_id
              Hash Cond: (m.vehicle_id = v.vehicle_id)
              ->  Seq Scan on public.maintenance m  (cost=0.00..19346.08 rows=1000008 width=4) (actual time=1.363..78.834 rows=1000008 loop…
                    Output: m.maintenance_id, m.vehicle_id, m.maintenance_date, m.cost, m.description
              ->  Hash  (cost=138.56..138.56 rows=952 width=62) (actual time=3.013..3.014 rows=4 loops=1)
                    Output: v.plate_no, v.vehicle_id
                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
                    ->  Foreign Scan on public.vehicles v  (cost=100.00..138.56 rows=952 width=62) (actual time=2.960..2.961 rows=4 loops=1)
                          Output: v.plate_no, v.vehicle_id
                          Remote SQL: SELECT vehicle_id, plate_no FROM public.vehicles
Planning Time: 1.147 ms
Execution Time: 279.728 ms
```

9. 2. Discussion of Optimizer Strategy and Data Movement

From the execution plan, the PostgreSQL optimizer chose a Hash Join strategy to efficiently combine data from the local maintenance table and the remote vehicles table. The vehicles table is accessed using a Foreign Scan, meaning it resides on a remote node. The optimizer pushed down a simplified query (SELECT vehicle_id, plate_no FROM public.vehicles) to the remote node, allowing only the necessary columns to be transferred back to the local node. This approach minimizes data movement by avoiding the transfer of entire vehicle records across the network.

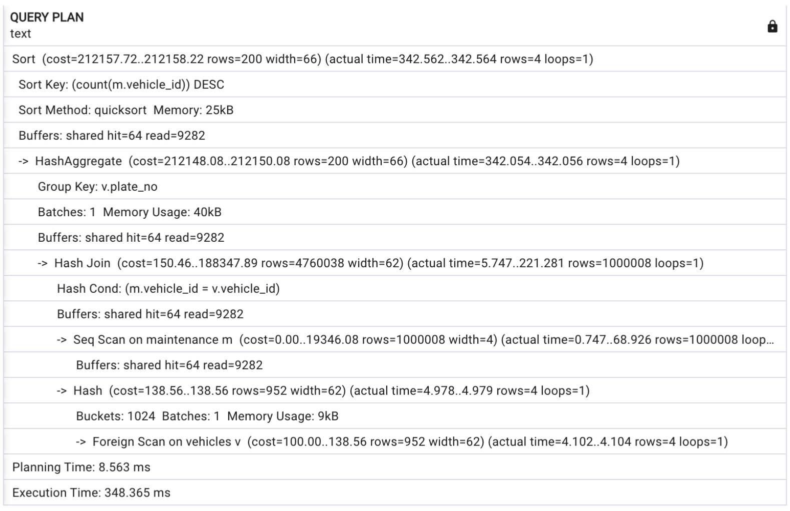
The Hash Join operation uses the smaller vehicles dataset (952 rows) to build an in-memory hash table, while the larger maintenance table (1,000,008 rows) is sequentially scanned locally and matched against the hash table. This reduces network overhead and improves join performance. The HashAggregate and Sort steps are performed locally after the join, ensuring the computationally heavy aggregation and ordering happens where data volume is reduced.

Overall, the optimizer’s strategy demonstrates data locality awareness, performing filtering and projection remotely, joining locally using hashed keys, and avoiding unnecessary remote data transfers. This results in a total execution time of ~279 ms, showing efficient cost-based optimization for distributed query execution.

TASK 10: Performance Benchmark and Report

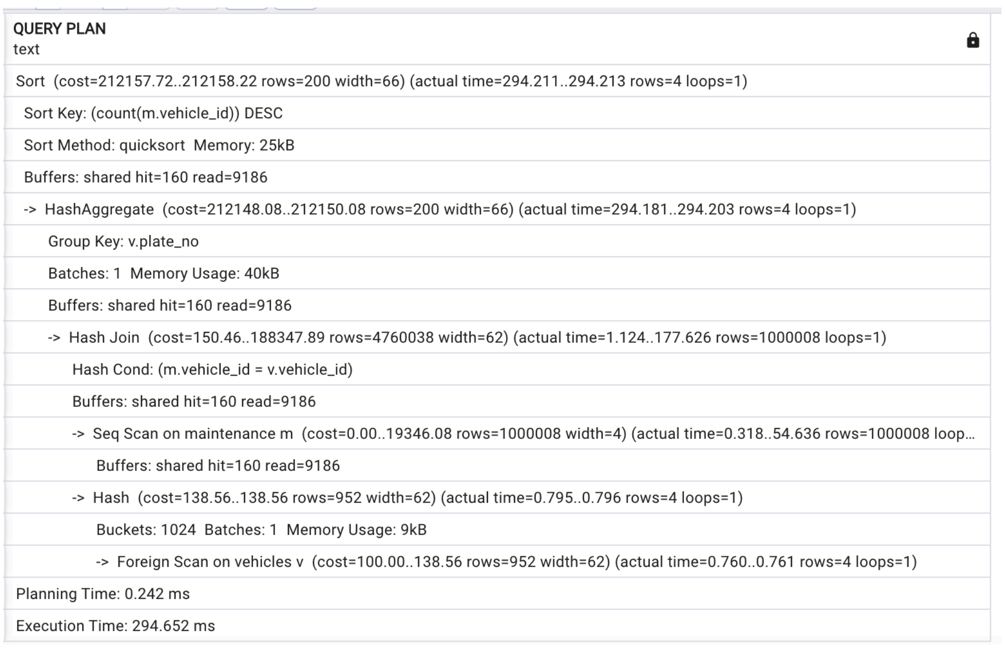
In this section we are going to run one complex distributed query on massive dataset, and this query will be runned in three ways – centralized, parallel, distributed

10.1 centralized (single process)



```
QUERY PLAN
text
Sort  (cost=212157.72..212158.22 rows=200 width=66) (actual time=342.562..342.564 rows=4 loops=1)
  Sort Key: (count(m.vehicle_id)) DESC
  Sort Method: quicksort  Memory: 25kB
  Buffers: shared hit=64 read=9282
  ->  HashAggregate  (cost=212148.08..212150.08 rows=200 width=66) (actual time=342.054..342.056 rows=4 loops=1)
        Group Key: v.plate_no
        Batches: 1  Memory Usage: 40kB
        Buffers: shared hit=64 read=9282
        ->  Hash Join  (cost=150.46..188347.89 rows=4760038 width=62) (actual time=5.747..221.281 rows=1000008 loops=1)
              Hash Cond: (m.vehicle_id = v.vehicle_id)
              Buffers: shared hit=64 read=9282
              ->  Seq Scan on maintenance m  (cost=0.00..19346.08 rows=1000008 width=4) (actual time=0.747..68.926 rows=1000008 loop...
                    Buffers: shared hit=64 read=9282
              ->  Hash  (cost=138.56..138.56 rows=952 width=62) (actual time=4.978..4.979 rows=4 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
                    ->  Foreign Scan on vehicles v  (cost=100.00..138.56 rows=952 width=62) (actual time=4.102..4.104 rows=4 loops=1)
Planning Time: 8.563 ms
Execution Time: 348.365 ms
```

10.2 Parallel (intra-node parallelism)



```
QUERY PLAN
text
Sort  (cost=212157.72..212158.22 rows=200 width=66) (actual time=294.211..294.213 rows=4 loops=1)
  Sort Key: (count(m.vehicle_id)) DESC
  Sort Method: quicksort  Memory: 25kB
  Buffers: shared hit=160 read=9186
  ->  HashAggregate  (cost=212148.08..212150.08 rows=200 width=66) (actual time=294.181..294.203 rows=4 loops=1)
        Group Key: v.plate_no
        Batches: 1  Memory Usage: 40kB
        Buffers: shared hit=160 read=9186
        ->  Hash Join  (cost=150.46..188347.89 rows=4760038 width=62) (actual time=1.124..177.626 rows=1000008 loops=1)
              Hash Cond: (m.vehicle_id = v.vehicle_id)
              Buffers: shared hit=160 read=9186
              ->  Seq Scan on maintenance m  (cost=0.00..19346.08 rows=1000008 width=4) (actual time=0.318..54.636 rows=1000008 loop...
                    Buffers: shared hit=160 read=9186
              ->  Hash  (cost=138.56..138.56 rows=952 width=62) (actual time=0.795..0.796 rows=4 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
                    ->  Foreign Scan on vehicles v  (cost=100.00..138.56 rows=952 width=62) (actual time=0.760..0.761 rows=4 loops=1)
Planning Time: 0.242 ms
Execution Time: 294.652 ms
```