Reflection Report:

Design Choices and Revisions

The primary objective of this project was to construct and query a robust relational database for tracking vehicle shipments, dealerships, and inventory. The design philosophy prioritized data integrity and the logical separation of concerns, ensuring that distinct entities Vehicles, Shipments, and Dealerships remained normalized to the Third Normal Form (3NF). This structure minimizes data redundancy but requires precise query construction to join related datasets effectively. A significant portion of the design process involved adapting standard SQL queries to the specific constraints of the PostgreSQL environment. One major design revision occurred during the creation of the date-based aggregation in Query 9. My initial draft relied on the SQLite function strftime, which is not supported in PostgreSQL and resulted in syntax errors. To resolve this, I revised the design to utilize PostgreSQL's TO_CHAR(ShipDate, 'YYYY-MM') function. This choice ensured accurate grouping of shipments by month while maintaining the readability of the Common Table Expression (CTE), allowing for clear, month-over-month performance tracking of dealerships.

Furthermore, the implementation of Query 8 (The Trigger) required a fundamental architectural shift. My initial design attempted to embed the INSERT logic directly within the CREATE TRIGGER statement. While this syntax is valid in some SQL dialects like SQL Server, it is invalid in PostgreSQL. After encountering syntax errors, I revised the design to adopt the strict two-step structure required by PL/pgSQL: first defining a standalone function (log_receipt_update_func) to encapsulate the logic, and then creating a trigger to execute that function. This separation of logic and execution is a robust design choice that improves code reusability and testing, although it initially added complexity to the implementation.

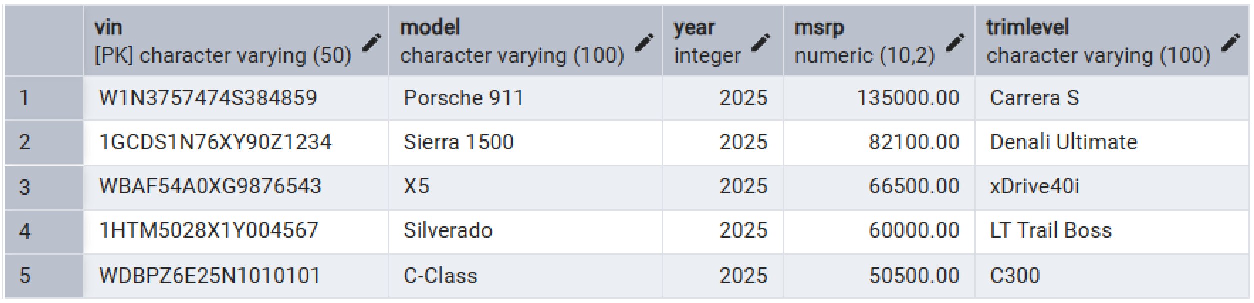
Discussion of Trade-offs

Throughout the development of the schema and queries, several technical trade-offs were balanced to optimize for both data quality and query performance. The database is highly normalized to eliminate redundancy, which necessitates a trade-off regarding query complexity. For instance, the ShipmentVehicle table acts as a junction table between Shipment and Vehicle. While this prevents storing vehicle details inside the shipment record, it necessitates complex joins to retrieve human-readable data. Query 2 is a prime example of this trade-off; retrieving a simple list of cars "En Route" required four different INNER JOIN operations. While this impacts query brevity, it guarantees that updates to a Carrier's name or a Vehicle's model are propagated instantly without update anomalies.

We also balanced read optimization against write overhead through indexing. In Query 7, we created an index IX_Dealership_Region on the Region column. Since dealerships are frequently filtered by region for logistics reporting, the index significantly speeds up SELECT queries by avoiding full table scans. However, this adds a small overhead to INSERT or UPDATE operations on the Dealership table, as the index must be updated whenever a region changes. Given that dealership address data is relatively static compared to shipment data, this was calculated to be a beneficial trade-off. Finally, we prioritized data robustness over syntax simplicity. A notable challenge arose in the trigger logic regarding NULL values. Standard comparison operators fail when comparing against NULL. To trade slight syntax simplicity for robustness, we utilized the predicate IS DISTINCT FROM. This ensures that even if a vehicle's previous status was undefined, the trigger still fires correctly when the status updates to 'Received', preventing data loss in edge cases.

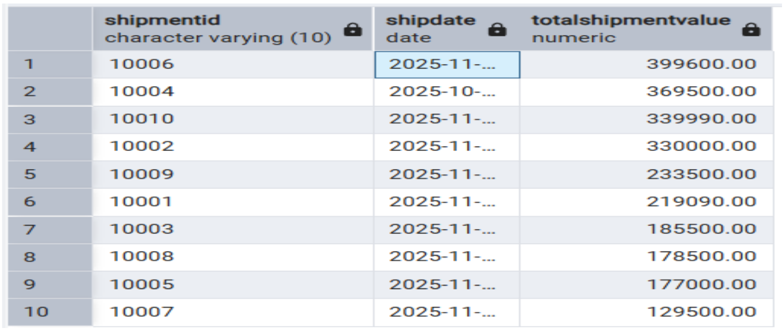
Execution Results

The attached screenshots demonstrate the successful execution of key queries, verifying that the database handles aggregations, filtering, and joins correctly. The first figure displays the output for Query 1, where the system successfully filtered for 2025 models with an MSRP over $50,000, sorting them to highlight premium inventory. The second figure illustrates the results of Query 3, which aggregated the total MSRP value for each shipment, providing essential financial data. The final figure shows the result of Query 9, validating the correct implementation of the CTE and date formatting to track monthly dealership activity.



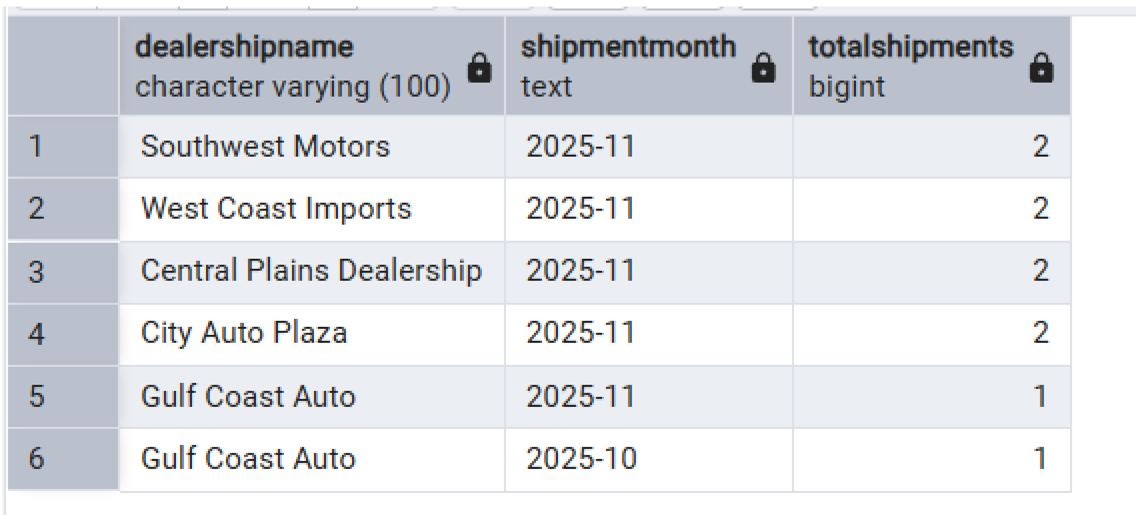
	vin [PK] character varying (50)	model character varying (100)	year integer	msrp numeric (10,2)	trimlevel character varying (100)
1	W1N3757474S384859	Porsche 911	2025	135000.00	Carrera S
2	1GCDS1N76XY90Z1234	Sierra 1500	2025	82100.00	Denali Ultimate
3	WBAF54A0XG9876543	X5	2025	66500.00	xDrive40i
4	1HTM5028X1Y004567	Silverado	2025	60000.00	LT Trail Boss
5	WDBPZ6E25N1010101	C-Class	2025	50500.00	C300

Figure 1: Results for Query 1, displaying high-value 2025 vehicles sorted by MSRP.



	shipmentid character varying (10)	shipdate date	totalshipmentvalue numeric
1	10006	2025-11-...	399600.00
2	10004	2025-10-...	369500.00
3	10010	2025-11-...	339990.00
4	10002	2025-11-...	330000.00
5	10009	2025-11-...	233500.00
6	10001	2025-11-...	219090.00
7	10003	2025-11-...	185500.00
8	10008	2025-11-...	178500.00
9	10005	2025-11-...	177000.00
10	10007	2025-11-...	129500.00

Figure 2: Results for Query 3, showing the total calculated MSRP value for each shipment.



	dealershipname character varying (100)	shipmentmonth text	totalshipments bigint
1	Southwest Motors	2025-11	2
2	West Coast Imports	2025-11	2
3	Central Plains Dealership	2025-11	2
4	City Auto Plaza	2025-11	2
5	Gulf Coast Auto	2025-11	1
6	Gulf Coast Auto	2025-10	1

Figure 3: Results for Query 9, using a CTE to aggregate total shipments per dealership per month.

Reflection and Evolution of Thinking

The most challenging aspect of this project was navigating the specific procedural requirements of PL/pgSQL triggers. My initial mental model was based on T-SQL (SQL Server), where triggers are defined as single, self-contained blocks. When I repeatedly encountered syntax errors regarding the OLD variable and INSERT statements, it was difficult to diagnose because the logic itself seemed sound. My thinking evolved from viewing the trigger as a simple "rule" to viewing it as a "switch" mechanism. I realized that in PostgreSQL, the trigger is merely a listener that flips a switch, while the function is the machinery that performs the actual work. This realization helped me solve the "Transaction Aborted" errors by strictly separating the components.

Additionally, handling the relation already exists errors taught me the importance of idempotency in SQL scripts. Early drafts of my code failed when run a second time because the tables or views already existed. By evolving the scripts to include DROP VIEW IF EXISTS and DROP TRIGGER IF EXISTS before creation, I ensured the code was robust and repeatable. This shifted my perspective from writing "one-off" queries to writing professional deployment scripts that can be executed safely in any environment.