Lab 02 Deliverable – SQL Fundamentals with Real Datasets

Student Name: Maryam
Course: Database Systems Lab
Instructor: Muhammad Usama Afridi
Date Submitted: 2026-02-25
GitHub Repo: https://github.com/maryam/database-labs

Section 1 Performance Report (20 points)

EXPLAIN ANALYZE Output for Query 5

```sql
EXPLAIN ANALYZE
SELECT order_id, customer_id, total_amount, status
FROM orders
ORDER BY total_amount DESC
LIMIT 10;
```

Full Output:
```
Limit  (cost=28.65..28.67 rows=10 width=102) (actual time=0.026..0.028 rows=10 loops=1)
  ->  Sort  (cost=28.65..30.12 rows=590 width=102) (actual time=0.026..0.027 rows=10 loops=1)
        Sort Key: total_amount DESC
        Sort Method: top-N heapsort  Memory: 26kB
        ->  Seq Scan on orders  (cost=0.00..15.90 rows=590 width=102) (actual time=0.004..0.008 rows=30 loops=1)
Planning Time: 0.026 ms
Execution Time: 0.044 ms
```

Analysis

Scan Type: Seq Scan (Sequential Scan)

The database performed a **Seq Scan**, meaning it read every row in the orders table (30 rows). This is expected because:
1. There is no index on the `total_amount` column
2. The table is very small (30 rows), so a full scan is actually

efficient

Execution Time: 0.044 milliseconds

If the table had 5 million rows:
- The Seq Scan would become extremely slow (potentially several seconds or minutes)
- PostgreSQL would likely still use Seq Scan for ORDER BY without LIMIT
- With LIMIT 10, it might use a "top-N heapsort" optimization (as shown) to avoid sorting all rows
- An index on `total amount` would dramatically improve performance, potentially reducing query time from minutes to milliseconds
- The "Memory: 26kB" for heapsort would increase significantly

Key Insight: On small tables, Seq Scan is fine. On large tables, proper indexing is critical for performance.

Section 3 Reflection (10 points)

See attached file: `reflection.md`

Key Points:
- **Surprised by:** SQL execution order (WHERE before SELECT) and NULL behavior
- **What clicked:** CASE WHEN as SQL's if/else, EXPLAIN ANALYZE interpretation
- **Still confusing:** Index usage patterns, top-N heapsort optimization
- **Want to learn:** JOINs, aggregate functions with GROUP BY

Word Count: ~280 words (exceeds 150-word minimum)

Submission Checklist

- [x] ecommerce_setup.sql committed to GitHub
- [x] queries.sql committed to GitHub
- [x] All 10 queries with screenshots and explanations
- [x] EXPLAIN ANALYZE output with interpretation
- [x] AI Learning Log with 3 genuine entries

- [x] Reflection (150+ words)
- [x] File named: Lab02 Maryam [RollNumber].pdf
- [x] GitHub repo URL included

GitHub Repository: https://github.com/maryam/database-labs

Files Submitted:
1. `lab2/ecommerce_setup.sql` - Database schema and seed data
2. `lab2/queries.sql` - All 10 SQL queries
3. `lab2/ai_learning_log.md` - AI interaction documentation
4. `lab2/reflection.md` - Personal reflection
5. `lab2/Lab02 Maryam [RollNumber].pdf` - This deliverable

AI Learning Log - Lab 02
Student: Maryam
Lab: SQL Fundamentals with Real Datasets

AI INTERACTION #1

Why helpful: The explanation clarified that NULL is not a value but the absence of a value. The AI explained that nothing equals NULL, not even NULL itself, which is why we must use IS NULL.

KEY LEARNINGS
- NULL represents missing/unknown data, not a actual value
- Using `= NULL` is silently wrong (returns 0 rows, no error)
- The correct syntax is `IS NULL` or `IS NOT NULL`
- This is a common beginner mistake in SQL

HOW I VERIFIED
I ran both queries in psql:
```sql
-- Wrong approach - returns 0 rows
SELECT * FROM orders WHERE shipped_date = NULL;

-- Correct approach - returns 7 unshipped orders
SELECT * FROM orders WHERE shipped_date IS NULL;
```
The results confirmed the AI's explanation.

WHAT I MODIFIED
I added comments in my queries.sql file to remind myself about this:
```sql
-- Query 8: NULL handling
-- IMPORTANT: Never use = NULL, always use IS NULL
WHERE shipped_date IS NULL
```

AI INTERACTION #2

Why helpful: The AI explained the SQL execution order (FROM ~~SELECT LIMIT) and showed that WHERE runs before SELECT, so the~~

alias doesn't exist yet.

KEY LEARNINGS
- SQL clauses execute in a specific order, not the order they're written
- Execution order: FROM SELECT LIMIT
- WHERE runs BEFORE SELECT, so aliases defined in SELECT don't exist yet
- Must repeat the expression in WHERE or use a subquery/CTE

HOW I VERIFIED
I tested both versions:
```sql
-- Wrong - alias doesn't exist in WHERE
SELECT product_name, price * 0.85 AS discounted_price
FROM products WHERE discounted_price < 3000;

-- Correct - repeat the expression
SELECT product_name, price * 0.85 AS discounted_price
FROM products WHERE price * 0.85 < 3000;
```

WHAT I MODIFIED
I now understand why my Query 9 works correctly - I compute the values in SELECT only, without trying to filter on them.

AI INTERACTION #3
—
Why helpful: Clear explanation of CASE WHEN as SQL's if/else statement. Showed how to add multiple WHEN clauses.

KEY LEARNINGS
- CASE WHEN is like if/else in programming
- Multiple WHEN clauses are checked in order
- First matching condition wins
- ELSE is the default if nothing matches
- Great for categorizing data (CRITICAL/URGENT/NORMAL)

HOW I VERIFIED
I extended my Query 10 with three tiers:
```sql
CASE
    WHEN total_amount > 20000 THEN 'CRITICAL'
```

~~   WHEN total_amount > 10000 THEN 'URGENT'~~
~~   ELSE 'NORMAL'~~
~~END AS priority~~
~~```~~
~~The query returned correct priority labels based on order amounts.~~

~~### WHAT I MODIFIED~~
~~Updated Query 10 in queries.sql to include all three priority tiers (CRITICAL, URGENT, NORMAL) as required by the lab.~~

~~---~~

~~## Summary~~
~~**Total AI Interactions:** 3~~
~~**Main Topics Covered:** NULL handling, SQL execution order, CASE WHEN statements~~
~~**Overall Learning:** AI tools are helpful for understanding SQL concepts when you ask specific questions and verify the results yourself.~~

Reflection - Lab 02: SQL Fundamentals

Student: Maryam
Date: 2026-02-25

What Surprised Me About How SQL Works

The most surprising thing I learned was that SQL clauses do not execute in the order we write them. I always assumed the database reads queries from top to bottom, but learning that WHERE runs before SELECT explained so many errors I've made. The fact that you cannot use a column alias in WHERE because it "doesn't exist yet" was a revelation - it's not a bug, it's just how SQL is designed.

Another surprise was how NULL works. I expected `= NULL` to either work or throw an error, but it silently returns zero rows. This is dangerous because you might think your query is correct when it's actually giving wrong results. The distinction that NULL is "absence of a value" rather than a value itself is fundamental but not intuitive.

What Clicked

The CASE WHEN statement finally made sense when I thought of it as SQL's version of if/else. I've used conditional logic in Python, but couldn't figure out how to do the same in SQL. Now I understand that CASE WHEN is how you categorize data, create priority levels, or handle different business rules directly in your queries.

The EXPLAIN ANALYZE output was also illuminating. Seeing "Seq Scan" and understanding that it means reading every row helped me understand why indexes matter. On our small 30-row table it took 0.044ms, but I can now reason about what would happen with millions of rows.

What Is Still Confusing

I'm still not entirely clear on when PostgreSQL will use an Index Scan versus a Seq Scan. The manual mentions that LIKE patterns starting with % cannot use indexes, but I don't fully understand what other factors influence this decision. I also want to understand more about the "top-N heapsort" optimization I saw in the EXPLAIN output.

What I Want to Understand Better Before Week 3

I want to practice more with JOINs since we only did basic single-table queries. The order_items table references orders and products, but I haven't yet written queries that combine them. I also want to understand aggregate functions (COUNT, SUM, AVG) with GROUP BY better, as these seem essential for data analysis but we didn't cover them deeply in this lab.

Word Count: ~280 words