Day 3 SQL Learning Notes

Date: October 30, 2025

Topic: JOINs - Combining Data from Multiple Tables

Project: Bank Account Management System - Multi-Table Queries

---

What I Learned Today

Understanding JOINs - The Core Concept

- JOINs connect data from multiple tables using a common column (like customer_id)
- Think of it like matching information from different notebooks using a reference number
- Tables have relationships: one customer → many accounts → many transactions
- Used table aliases (c, a, t) to make queries shorter and clearer

INNER JOIN - Most Common Join

- Shows only rows that have matches in BOTH tables
- Syntax: `FROM table1 t1 INNER JOIN table2 t2 ON t1.id = t2.id`
- Example: Show customers WITH accounts only (excludes customers without accounts)
- Used to connect customers → accounts → transactions
- Can join 3+ tables by chaining multiple JOINs
- Always specify which table columns come from using alias (c.name, a.balance)

LEFT JOIN - Including Unmatched Rows

- Shows ALL rows from left table + matching rows from right table
- Non-matching rows show NULL for right table columns
- Syntax: `FROM table1 t1 LEFT JOIN table2 t2 ON t1.id = t2.id`
- Critical for finding missing data: customers WITHOUT accounts
- Combined with `WHERE right_table.id IS NULL` to find unmatched records
- Used with COALESCE to replace NULLs with default values (0 for no balance)
- Essential for "show everyone, even if no data" scenarios

RIGHT JOIN - Reverse of LEFT JOIN

- Shows ALL rows from right table + matching rows from left table

- Rarely used - can usually rearrange query to use LEFT JOIN instead
- Useful for finding orphaned records (accounts without customers)

FULL OUTER JOIN - Everything from Both Tables

- Shows ALL rows from BOTH tables whether they match or not
- NULLs appear where no match exists
- Used for data quality audits and reconciliation
- Helps find mismatches between systems

CROSS JOIN - Every Combination (Cartesian Product)

- Creates every possible combination of rows from both tables
- No ON clause needed
- Example: 3 customer types × 4 account types = 12 combinations
- Used for creating reference matrices and report templates
- WARNING: Can create huge result sets (100 × 100 = 10,000 rows)
- Use carefully and only when intentionally needed

SELF JOIN - Table Joins Itself

- Same table used twice with different aliases
- Used for hierarchical data (employee-manager, customer referrals)
- Example: Show customers and who referred them
- Syntax: `FROM customers c1 JOIN customers c2 ON c1.referred_by = c2.customer_id`
- c1 = customer, c2 = their referrer (same table, different perspective)
- Great for finding relationships within same entity

JOIN Best Practices I Learned

- Always use table aliases (makes queries readable and shorter)
- Specify which table each column comes from (c.name, a.balance)
- Use DISTINCT in COUNT when joining to avoid duplicate counts
- Filter with WHERE after JOIN to narrow results
- LEFT JOIN + WHERE IS NULL finds missing relationships
- Start with INNER JOIN, switch to LEFT JOIN if need unmatched rows
- Use explicit JOIN syntax, not old comma style (clearer intent)

Key Patterns Practiced

- **Customer Portfolio:** Customer + all their accounts (one customer, multiple rows if multiple accounts)
- **Transaction History:** Customer + accounts + transactions (3-table join)

- **Missing Data:** LEFT JOIN + WHERE IS NULL (find customers without accounts)
- **Aggregation:** GROUP BY after JOIN to summarize (one row per customer with totals)
- **Referral Chain:** SELF JOIN to show who referred whom

Combining JOINs with Other Concepts

- **JOINs + WHERE:** Filter after joining (only Premium customers)
- **JOINs + CASE:** Categorize joined data (High/Medium/Low value)
- **JOINs + COALESCE:** Handle NULLs from LEFT JOINs (show 0 instead of NULL)
- **JOINs + GROUP BY:** Summarize data (count accounts per customer)
- **JOINs + Aggregates:** SUM balances, COUNT transactions per customer
- **JOINs + ORDER BY:** Sort combined results

Real Banking Scenarios Practiced

- Customer portfolio view with all accounts and balances
- Transaction history across all accounts
- Finding customers who registered but never opened accounts
- Referral program tracking (who referred whom)
- Customer activity report with deposits/withdrawals summary
- Identifying inactive accounts (no transactions in 90+ days)
- Account reconciliation (finding orphaned records)

Common Mistakes I Learned to Avoid

- Forgetting ON clause creates accidental CROSS JOIN (huge result set)
- Using wrong JOIN type (INNER when need LEFT)
- Not using table aliases causes confusion with duplicate column names
- Forgetting DISTINCT in COUNT after JOIN (counts duplicates)
- Not handling NULLs from LEFT JOIN (use COALESCE)
- Joining too many tables without filtering (performance issues)

Performance Tips

- JOIN on indexed columns (PRIMARY KEY, FOREIGN KEY)
- Filter data before JOIN when possible (reduces rows to join)
- Use INNER JOIN when don't need unmatched rows (faster than LEFT)
- Avoid CROSS JOIN unless specifically needed
- Use EXPLAIN to check query performance

Key Formulas

Basic INNER JOIN:

```
FROM table1 t1
INNER JOIN table2 t2 ON t1.common_id = t2.common_id
```

LEFT JOIN for unmatched:

```
FROM table1 t1
LEFT JOIN table2 t2 ON t1.id = t2.id
WHERE t2.id IS NULL
```

3-table chain:

```
FROM customers c
JOIN accounts a ON c.customer_id = a.customer_id
JOIN transactions t ON a.account_id = t.account_id
```

SELF JOIN:

```
FROM table t1
JOIN table t2 ON t1.parent_id = t2.id
```

Understand:

- Table relationships (one-to-many, many-to-many)
- When to use INNER vs LEFT vs FULL JOIN
- How GROUP BY works after JOINs
- Why table aliases are essential
- CROSS JOIN risks and use cases

Key Takeaways

1. **INNER JOIN = only matches | LEFT JOIN = all from left + matches**
2. **Always use ON clause** - without it creates accidental CROSS JOIN
3. **Table aliases are mandatory** for multi-table queries
4. **LEFT JOIN + WHERE IS NULL** finds missing relationships
5. **SELF JOIN = same table twice** for hierarchical data
6. **GROUP BY after JOIN** summarizes related data
7. **COALESCE handles NULLs** from LEFT JOINs