Lab: SQL & Data Analysis with Claude Code

Duration: 45-60 minutes
Difficulty: Progressive (Easy → Advanced)
Tools: Claude Code CLI, SQLite3

Overview

In this lab, you'll use Claude Code to work with SQL databases—from generating queries to designing schemas, optimizing performance, validating data quality, and transforming data. This is real-world database development with AI assistance.

By the end of this lab, you'll be able to: - Generate complex SQL queries from natural language descriptions - Design normalized database schemas from business requirements - Optimize slow queries with index recommendations - Write data validation queries to ensure integrity - Transform data between different schema formats

---

Setup (5 minutes)

1. Navigate to the lab directory

```
cd /mnt/code/AIA-ClaudeCopilot/labs/sample-data/sql
```

2. Load the sample database

```
# Create the database and load schema
sqlite3 ecommerce.db < create_tables.sql

# Load seed data
sqlite3 ecommerce.db < seed_data.sql
```

3. Verify the database loaded correctly

```
sqlite3 ecommerce.db "SELECT COUNT(*) FROM customers;"
# Should return 20

sqlite3 ecommerce.db "SELECT COUNT(*) FROM products;"
# Should return 30
```

4. Start Claude Code

```
claude
```

Success Criteria: - □ `ecommerce.db` file exists - □Database contains 20 customers, 30 products - □ Claude Code session is active

---

Exercise 1: Query Generation (10 minutes)

Ask Claude Code to generate SQL queries from natural language. Start simple, build to complex.

Query 1: Simple SELECT with WHERE

Your Task:
Ask Claude Code: *"Write a SQL query to find all customers from California."*

What to Look For: - Correct table name (`customers`) - WHERE clause filtering by state - Column selection (all columns or specific ones)

Success Criteria: - Query returns multiple California customers - No syntax errors - Uses proper SQLite syntax

---

Query 2: JOIN with Aggregation

Your Task:
Ask Claude Code: *"Write a SQL query to show each customer's name and total number of orders."*

What to Look For: - JOIN between `customers` and `orders` tables - GROUP BY customer - COUNT aggregation - Proper column aliases for readability

Success Criteria: - Results show customer names and order counts - All customers appear (even those with 0 orders if using LEFT JOIN) - Query executes without errors

---

Query 3: Multi-Table JOIN with WHERE

Your Task:

Ask Claude Code: *"Find all products in the 'Electronics' category with average review rating above 4.0."*

What to Look For: - JOIN across `products`, `categories`, and `reviews` tables - WHERE clause filtering by category name - AVG aggregation on rating - HAVING clause to filter averages above 4.0

Success Criteria: - Only Electronics products appear - All have ratings > 4.0 - Query demonstrates proper multi-table JOINs

---

Query 4: Subquery with IN

Your Task:

Ask Claude Code: *"List all customers who have placed an order in the last 30 days."*

What to Look For: - Subquery selecting customer_ids from recent orders - Date filtering (CURRENT_DATE - INTERVAL or datetime functions) - IN or EXISTS clause

Success Criteria: - Only customers with recent orders appear - Date logic is correct - No duplicate customer records

---

Query 5: Window Function (Advanced)

Your Task:

Ask Claude Code: *"Show each order with its total value, customer name, and a running total of that customer's spending sorted by order date."*

What to Look For: - Window function (SUM OVER with PARTITION BY customer) - Multiple JOINs (orders, customers, order_items) - Proper ORDER BY within the window function - Calculation of order total from order_items

Success Criteria: - Running total resets for each customer - Order totals are correct - Results are sorted logically

Reflection Questions: 1. How does Claude Code's query compare to what you would have written manually? 2. Did it use techniques (CTEs, window functions) you might not have thought of? 3. Were column names and table relationships correct on the first try?

Exercise 2: Schema Design (10 minutes)

Use Claude Code to design a database schema from business requirements.

Business Requirements: Library Management System

> *We need a system to track library operations. The library has multiple branches. Each branch has books (multiple copies of the same title). Members can check out books from any branch. We need to track which member has which book, when it was checked out, and when it's due back. Members have a borrowing limit of 5 books. Books belong to genres. We need to track late fees.*

Your Task:
Ask Claude Code: *"Design a normalized database schema for the above library management system. Use PostgreSQL syntax. Include all necessary tables, relationships, constraints, and indexes."*

What to Evaluate

Tables to Expect: - `branches` (branch locations) - `books` (book titles/metadata) - `book_copies` (physical copies at branches) - `members` (library members) - `checkouts` (borrowing transactions) - `genres` (book categories) - `late_fees` (fee tracking)

Normalization (3NF) Check: - Is data duplicated unnecessarily? - Are there proper junction tables for many-to-many relationships? - Are foreign keys used correctly?

Constraints to Expect: - PRIMARY KEY on all tables - FOREIGN KEY with appropriate ON DELETE/ON UPDATE actions - NOT NULL on required fields (member name, book title, etc.) - UNIQUE on email addresses - CHECK constraints (e.g., due_date > checkout_date, borrowing_limit <= 5)

Indexes to Expect: - Foreign key columns (member_id, book_id, branch_id) - Frequently searched columns (member email, book title, checkout status)

Success Criteria: - □ All expected tables are present - □ Relationships are modeled with proper foreign keys - □ No obvious denormalization (data duplication) - □ Constraints enforce business rules - □ Indexes are suggested for common queries

What to Look For: - Did Claude Code identify all entities from the requirements? - Are junction tables used for many-to-many relationships (books ↔ genres)? - Is the distinction between `books` (titles) and `book_copies` (physical items) modeled correctly? - Are late fees tracked properly (separate table or column)?

Bonus Challenge:

Ask Claude Code: *"Generate realistic seed data for this library schema—3 branches, 20 books, 30 members, 15 active checkouts."*

Exercise 3: Query Optimization (10 minutes)

Use Claude Code to optimize slow queries. The sample database includes intentionally inefficient queries.

Load the slow queries file

```
cat slow_queries.sql
```

You'll see 3 queries with comments explaining why they're slow.

Query 1: Missing Index

Slow Query:

```
-- Find all orders placed by customers in New York
SELECT o.*
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE c.state = 'NY';
```

Your Task:

Ask Claude Code: *"Optimize this query and explain why it's slow. Recommend indexes and show the improved version."*

What to Look For: - Identifies missing index on `customers.state` - Suggests creating index: `CREATE INDEX idx_customers_state ON customers(state);` - May also suggest index on `orders.customer_id` (foreign key) - Explains that without indexes, the DB does a full table scan

Success Criteria: - Claude Code identifies the performance bottleneck - Index recommendations are specific (column names, table names) - Explanation is clear and accurate

---

Query 2: SELECT * Anti-Pattern

Slow Query:

```
-- Get customer names and their order count
SELECT *
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id;
```

Your Task:
Ask Claude Code: *"This query returns too much data. Optimize it and explain what's wrong."*

What to Look For: - Identifies that `SELECT *` retrieves unnecessary columns - Rewrites to select only needed columns (`customer_id`, `first_name`, `last_name`, COUNT) - Adds GROUP BY to aggregate orders per customer - Uses COUNT properly with LEFT JOIN (COUNT(o.order_id) to handle NULLs)

Success Criteria: - Optimized query selects specific columns only - Aggregation is correct (shows customers with 0 orders) - Explanation mentions reduced data transfer

---

Query 3: Subquery vs JOIN

Slow Query:

```
-- Find products that have been ordered at least once
SELECT *
FROM products
WHERE product_id IN (
    SELECT DISTINCT product_id
    FROM order_items
);
```

Your Task:

Ask Claude Code: *"Rewrite this using a JOIN instead of a subquery. Which is faster and why?"*

What to Look For: - Rewrites using `JOIN` or `EXISTS` - Suggests: `SELECT DISTINCT p.* FROM products p JOIN order_items oi ON p.product_id = oi.product_id` - Or: `SELECT * FROM products p WHERE EXISTS (SELECT 1 FROM order_items oi WHERE oi.product_id = p.product_id)` - Explains that JOIN allows the optimizer more flexibility - May mention that EXISTS is more efficient than IN for large subqueries

Success Criteria: - Alternative query is functionally equivalent - Explanation compares performance characteristics - Claude Code demonstrates understanding of query optimization principles

---

Reflection: 1. Did Claude Code run EXPLAIN or EXPLAIN QUERY PLAN to show the difference? 2. Were the index recommendations specific and actionable? 3. Could you apply these optimization techniques to your own queries?

Exercise 4: Data Validation (10 minutes)

Use Claude Code to write queries that find data quality issues. The sample database includes intentional problems.

Known Issues in the Database

The seed data contains: - 2 orphaned order_items (referencing deleted orders) - 3 duplicate customer emails - 5 NULL values where they shouldn't be (product names, customer emails)

Your Task:

Ask Claude Code to generate validation queries for each issue type.

Validation 1: Find Orphaned Records

Your Task:

Ask Claude Code: *"Write a query to find order_items that reference non-existent orders (orphaned records)."*

What to Look For: - LEFT JOIN from `order_items` to `orders` - WHERE clause checking for NULL order_id after the join - Or: NOT EXISTS subquery

Expected Result: - Should find 2 orphaned order_items - Query demonstrates referential integrity checking

Success Criteria: - □ Query finds the orphaned records - □ Logic is correct (checks for missing foreign key references)

Validation 2: Check Referential Integrity

Your Task:
Ask Claude Code: *"Write a query to verify that all foreign key relationships are valid across the entire database."*

What to Look For: - Multiple queries checking each foreign key relationship: - orders.customer_id → customers.customer_id - order_items.order_id → orders.order_id - order_items.product_id → products.product_id - products.category_id → categories.category_id - reviews.product_id → products.product_id - reviews.customer_id → customers.customer_id

Success Criteria: - □Checks all major foreign key relationships - □Reports which relationships have orphaned records

---

Validation 3: Detect Duplicates

Your Task:
Ask Claude Code: *"Find all customers with duplicate email addresses."*

What to Look For: - GROUP BY email - HAVING COUNT(*) > 1 - Shows all columns for the duplicate records (useful for manual review)

Expected Result: - Should find 3 email addresses that appear more than once

Success Criteria: - □ Query identifies duplicate emails - □ Shows which specific records are duplicates

---

Validation 4: Find NULL Violations

Your Task:
Ask Claude Code: *"Find any records with NULL values in columns that should never be NULL (customer email, product name, order totals)."*

What to Look For: - Separate queries for each table/column - WHERE column_name IS NULL - Checks critical columns (email, name, required fields)

Expected Result: - Should find 5 NULL violations across the database

Success Criteria: - □Identifies all NULL violations - □Queries are specific to each table/column

---

Reflection: 1. How would these validation queries be used in a real project? 2. Could you run these as part of automated tests or data quality monitoring? 3. What other data quality issues might you want to check for?

Exercise 5: ETL Script (Bonus - 10 minutes)

Transform data from one schema to another using Claude Code.

Scenario

You have an old e-commerce schema with a denormalized `orders` table that includes customer info directly:

```
-- Old Schema
CREATE TABLE old_orders (
    order_id INTEGER,
    customer_name TEXT,
    customer_email TEXT,
    product_name TEXT,
    quantity INTEGER,
    price DECIMAL
);
```

You need to migrate this to the normalized schema in `ecommerce.db` (separate `customers`, `products`, `orders`, `order_items` tables).

Your Task:

Ask Claude Code: *"Write a SQL migration script to transform data from the old denormalized orders table into the normalized schema (customers, products, orders, order_items). Handle duplicates correctly."*

What to Look For:

1. **Extract Customers** `sql INSERT INTO customers (first_name, last_name, email) SELECT DISTINCT substr(customer_name, 1, instr(customer_name, ' ')-1) AS first_name, substr(customer_name, instr(customer_name, ' ')+1) AS last_name, customer_email FROM old_orders WHERE customer_email NOT IN (SELECT email FROM customers);`
2. **Extract Products** `sql INSERT INTO products (name, price) SELECT DISTINCT product_name, price FROM old_orders WHERE product_name NOT IN (SELECT name FROM products);`
3. **Insert Orders** `sql INSERT INTO orders (customer_id, order_date) SELECT c.customer_id, CURRENT_DATE FROM old_orders o JOIN customers c ON o.customer_email = c.email GROUP BY o.order_id, c.customer_id;`
4. **Insert Order Items** `sql INSERT INTO order_items (order_id, product_id, quantity, unit_price) SELECT o.order_id, p.product_id, old.quantity, old.price FROM old_orders old JOIN customers c ON old.customer_email = c.email JOIN orders o ON o.customer_id = c.customer_id JOIN products p ON old.product_name = p.name;`

Success Criteria: - □ Script is idempotent (safe to run multiple times) - □ Handles duplicates correctly (INSERT WHERE NOT EXISTS or IGNORE) - □Preserves all data from old schema - □ Maintains referential integrity in new schema - □ Includes validation queries to verify migration success

Validation Queries to Ask For: - "Write queries to verify the migration preserved all data." - "Check that the number of unique customers matches between old and new schemas." - "Verify that total order quantities match before and after migration."

Wrap-Up & Reflection

What You Accomplished

- □Generated 5 progressively complex SQL queries using natural language
- □Designed a normalized database schema from business requirements

- □Optimized 3 slow queries with index recommendations
- □Wrote 4 data validation queries to detect quality issues
- □(Bonus) Created an ETL migration script

Key Takeaways

1. **Natural Language → SQL**
   Claude Code understands intent and generates syntactically correct, semantically accurate queries. Complex joins, subqueries, and window functions are handled naturally.
2. **Schema Design**
   Claude Code applies normalization principles, creates proper relationships, and enforces constraints automatically. It catches design anti-patterns you might miss.
3. **Query Optimization**
   Claude Code identifies performance bottlenecks, recommends specific indexes, and rewrites inefficient queries. It can explain *why* one approach is faster than another.
4. **Data Quality**
   Validation queries are critical for maintaining database integrity. Claude Code generates comprehensive checks for common issues (orphaned records, duplicates, NULLs).
5. **ETL & Transformation**
   Migrating between schemas is complex and error-prone. Claude Code generates safe, idempotent transformation scripts with built-in validation.

Real-World Applications

- **Backend Development:** Generate database queries from API requirements
- **Data Analysis:** Create complex analytical queries for reporting dashboards
- **Database Administration:** Optimize slow queries, validate data integrity
- **Data Engineering:** Build ETL pipelines, migrate legacy systems
- **Quality Assurance:** Write automated data validation tests

Next Steps

- Try these techniques on your own projects
- Create CLAUDE.md files with your database conventions (naming, normalization standards)
- Build a library of validation queries for your critical tables
- Use Claude Code to document your existing database schema
- Generate seed data for testing environments

Troubleshooting

Issue: Query returns no results
Solution: Check table names and column names. Use `.schema tablename` in SQLite to see structure.

Issue: Syntax error in generated query
Solution: Make sure you specified the database engine (SQLite, PostgreSQL, MySQL). Syntax varies between engines.

Issue: Claude Code doesn't understand the schema
Solution: Use `@file` mentions to explicitly include the schema file. Example: `claude @create_tables.sql "write a query to find..."`

Issue: Optimization recommendations don't apply to SQLite
Solution: SQLite has limited index types. Specify "using SQLite" in your prompt for appropriate recommendations.

End of Lab
Great work! You've completed SQL & Data Analysis with Claude Code. Move on to Lab 2 (Real Development Workflow) when ready.

Generated from Day2_Lab_SQL_Data_Analysis.md on Day2_Lab_SQL_Data_Analysis.pdf

© 2026 AIA Copilot Training — Claude Code Fundamentals