IT Management & Audits

Practical Lab Manual

Financial Data Manager

Practical P06

Learning Domain

Data Management & SQL Analytics

Course Learning Outcomes

CLO06: Manage and query financial datasets using SQL and Python

Unit

Unit IV: Data Management for FinTech

Time Allocation: 3 hours

Learning Mode: Hands-on (80%) + Theory (20%)

Difficulty Level: Intermediate

Financial Data Manager

Practical P06

Quick Reference

Practical Code	P06
Practical Name	Financial Data Manager
Slot	T/P-6
Duration	3 hours
CLO Mapping	CLO06
Unit	Unit IV: Data Management for FinTech
Delivery Mode	Hands-on Lab
Target Audience	Intermediate Level
India Integration	HIGH
Screenshot Count	5 Required

Prerequisites

- Basic SQL concepts (SELECT, INSERT, JOIN, GROUP BY)
- Python programming basics (variables, functions, modules)
- Understanding of relational databases (tables, keys, relationships)
- Familiarity with command-line interface (CLI)
- Completion of P01 (Lab Setup & Infrastructure) recommended

Tools Required

Tool	**Version**	**Free**	**Notes**
Python	3.8+	✓	Required
SQLite3	Built-in	✓	Included with Python
Faker (Python library)	Latest	✓	Synthetic data generation
Rich (Python library)	Latest	✓	Terminal formatting
Click (Python library)	Latest	✓	CLI framework
pip (Package manager)	Latest	✓	Included with Python

Learning Objectives

- ✓ Design and create a relational database schema for financial data
- ✓ Seed databases with realistic synthetic financial data using Faker
- ✓ Write and execute SQL analytics queries on financial datasets
- ✓ Apply advanced SQL techniques: window functions, CTEs, subqueries
- ✓ Perform data quality checks on financial records
- ✓ Generate data quality reports for audit and compliance purposes
- ✓ Write custom SQL queries for Indian banking and FinTech scenarios

What You Will Learn

By the end of this practical, you will be able to: (1) design relational schemas for financial systems with proper keys and indexes, (2) generate synthetic data using Faker, (3) run analytical SQL queries including window functions and CTEs, (4) implement automated data quality checks across six dimensions, (5) generate HTML audit reports, and (6) write SQL for Indian banking scenarios (UPI, GST, RBI reporting).

Real-World Application

Financial data management is core to Indian banking and FinTech. Organizations like **ICICI Bank**, **HDFC Bank**, and **Paytm** process millions of daily transactions and must maintain data quality for RBI compliance. Data quality failures lead to incorrect financial statements, regulatory penalties, and lost trust. This practical simulates production data management pipelines.

Hands-On Procedure

Part A: Environment Setup

Step 1: Clone the Financial Data Manager Repository

Objective: Download and explore the financial data management tool on your local machine.

Instructions:

1. Open your terminal (bash on Linux/Mac, PowerShell on Windows)
2. Navigate to your working directory
3. Clone the repository from GitHub
4. Navigate into the project directory
5. Explore the project structure

Code/Command:

```
# Clone the repository
git clone <repository-url> financial-data-manager
cd financial-data-manager

# View project structure
ls -la       # Expected: src/, queries/, docs/, tests/
ls src/      # schema.py, seed_data.py, query_runner.py,
             # data_quality.py, reporter.py, cli.py
ls queries/ # analytics.sql, quality_checks.sql
```

Clone and Explore the Repository

Expected Output

`src/schema.py` – Table definitions with FK and indexes
`src/seed_data.py` – Faker-based synthetic data generator
`src/query_runner.py` – SQL execution and formatting
`src/data_quality.py` – Data quality check framework
`src/reporter.py` – HTML/terminal report generator
`src/cli.py` – Click-based CLI (5 commands)
`queries/` – Pre-written analytics and quality queries

If you do not have Git installed, you can download the repository as a ZIP file from the GitHub page and extract it manually.

Step 2: Set Up Python Virtual Environment and Verify CLI

Objective: Create an isolated Python environment, install dependencies, and verify the CLI tool works.

Instructions:

1. Create a Python virtual environment
2. Activate the virtual environment
3. Install project dependencies from `requirements.txt`
4. Verify installation by running the CLI help command

Code/Command:

```
python -m venv venv
source venv/bin/activate  # Windows: venv\Scripts\activate
pip install -r requirements.txt
python src/cli.py --help
```

Set Up Environment and Verify CLI

Expected Output

```
Usage: cli.py [OPTIONS] COMMAND [ARGS]...

Financial Data Manager - SQL Analytics & Data Quality Tool

Commands:
init-db        Initialize the SQLite database with schema
seed           Seed database with synthetic financial data
query          Run SQL queries against the financial database
check-quality  Run automated data quality checks
report         Generate data quality reports (HTML/terminal)
```

Ensure you are using Python 3.8 or higher. SQLite3 is included in the Python standard library, so no separate database installation is required.

Screenshot 1

What to paste: Terminal showing the cloned repository file structure (`ls` output showing `src/`, `queries/` directories and files) and the CLI help output after running `python src/cli.py -help` displaying all 5 commands.

Paste your screenshot here

Part B: Database Creation

Step 3: Initialize the Database Schema

Objective: Create the SQLite database with the financial data schema including all tables, foreign keys, and indexes.
Instructions:

1. Run the database initialization command
2. Observe the 5 tables being created
3. Review the schema with foreign key relationships
4. Verify the database file was created

Code/Command:

```
python src/cli.py init-db
ls -la financial_data.db
sqlite3 financial_data.db ".schema"
```

Initialize the Financial Database

Expected Output

```
[INFO] Creating database: financial_data.db
[OK] Table 'customers' created (9 columns)
[OK] Table 'accounts' created (8 columns)
[OK] Table 'transactions' created (10 columns)
[OK] Table 'merchants' created (7 columns)
[OK] Table 'fraud_flags' created (6 columns)
[OK] 4 foreign key constraints applied
[OK] Indexes on: customer_id, account_id, txn_date, merchant_id
[OK] Database initialized successfully with 5 tables
```

The schema uses proper normalization (3NF) with foreign key constraints to maintain referential integrity. Indexes are added on commonly queried columns to improve performance.

Step 4: Seed the Database with Synthetic Financial Data

Objective: Populate the database with realistic synthetic data using the Faker library.
Instructions:

1. Run the seed command to generate synthetic data
2. Observe the data counts for each table
3. Review sample records from each table

4. Verify data relationships are intact

Code/Command:

```
# Seed the database with synthetic data
python src/cli.py seed

# Verify record counts
python src/cli.py query "SELECT␣'customers'␣AS␣tbl,
␣␣COUNT(*)␣AS␣cnt␣FROM␣customers␣UNION␣ALL
␣␣SELECT␣'accounts',␣COUNT(*)␣FROM␣accounts␣UNION␣ALL
␣␣SELECT␣'transactions',␣COUNT(*)␣FROM␣transactions
␣␣UNION␣ALL␣SELECT␣'merchants',␣COUNT(*)␣FROM␣merchants
␣␣UNION␣ALL␣SELECT␣'fraud_flags',␣COUNT(*)
␣␣FROM␣fraud_flags"

# View sample records
python src/cli.py query "SELECT␣id,␣name,␣city
␣␣FROM␣customers␣LIMIT␣5"
python src/cli.py query "SELECT␣id,␣amount,␣txn_type,
␣␣channel␣FROM␣transactions␣LIMIT␣5"
```

Seed Synthetic Financial Data

Expected Output

```
[INFO] Seeding database with synthetic financial data...
[OK] 500 customers | 1000 accounts | 5000 transactions
[OK] 200 merchants |  75 fraud flags
[OK] All data seeded successfully
```

The Faker library generates realistic but entirely fictional data. PAN numbers, Aadhar hashes, and account numbers are synthetic and do not correspond to any real individuals. This is critical for compliance with the DPDP Act — never use real customer data in development or testing environments.

Screenshot 2

What to paste: Terminal showing the database initialization output (5 tables created with foreign keys and indexes) and the seeded data record counts for all tables. Include the sample customer and transaction records.

Paste your screenshot here

Part C: SQL Analytics

Step 5: Run Financial Analytics Queries

Objective: Execute SQL queries to extract meaningful business insights from the financial dataset.
Instructions:

1. Run queries for top customers by transaction volume
2. Analyze monthly transaction trends
3. Perform merchant category analysis
4. Calculate average transaction amounts by account type
5. Identify suspicious large transactions

Code/Command:

```
# Query 1: Top 10 customers by transaction volume
python src/cli.py query "SELECT␣c.name,␣c.city,
␣␣COUNT(t.id)␣AS␣txn_count,␣SUM(t.amount)␣AS␣total_amount
␣␣FROM␣customers␣c␣JOIN␣accounts␣a␣ON␣c.id␣=␣a.customer_id
␣␣JOIN␣transactions␣t␣ON␣a.id␣=␣t.account_id
␣␣GROUP␣BY␣c.id␣ORDER␣BY␣total_amount␣DESC␣LIMIT␣10"

# Query 2: Monthly transaction trends
python src/cli.py query "SELECT␣strftime('%Y-%m',␣txn_date)
␣␣AS␣month,␣COUNT(*)␣AS␣txn_count,
␣␣ROUND(SUM(amount),2)␣AS␣total_volume
␣␣FROM␣transactions␣GROUP␣BY␣month␣ORDER␣BY␣month"

# Query 3: Merchant category analysis
python src/cli.py query "SELECT␣m.category,
␣␣COUNT(t.id)␣AS␣txn_count,
␣␣ROUND(SUM(t.amount),2)␣AS␣total_revenue
␣␣FROM␣merchants␣m␣JOIN␣transactions␣t␣ON␣m.id=t.merchant_id
␣␣GROUP␣BY␣m.category␣ORDER␣BY␣total_revenue␣DESC"

# Query 4: Average transaction by account type
python src/cli.py query "SELECT␣a.account_type,
␣␣COUNT(t.id)␣AS␣txn_count,␣ROUND(AVG(t.amount),2)␣AS␣avg
␣␣FROM␣accounts␣a␣JOIN␣transactions␣t␣ON␣a.id=t.account_id
␣␣GROUP␣BY␣a.account_type␣ORDER␣BY␣avg␣DESC"

# Query 5: Suspicious large transactions (> 2x average)
python src/cli.py query "SELECT␣t.id,␣c.name,␣t.amount,
␣␣t.channel,␣t.txn_date␣FROM␣transactions␣t
␣␣JOIN␣accounts␣a␣ON␣t.account_id=a.id
␣␣JOIN␣customers␣c␣ON␣a.customer_id=c.id
␣␣WHERE␣t.amount␣>␣(SELECT␣AVG(amount)*3␣FROM␣transactions)
␣␣ORDER␣BY␣t.amount␣DESC␣LIMIT␣10"
```

Financial Analytics Queries

Expected Output

Each query returns a Rich-formatted table. Query 1 shows top customers ranked by total transaction amount with city and count. Query 2 shows monthly trends. Query 3 breaks down merchant categories by revenue. Query 4 compares account types. Query 5 flags statistically unusual large transactions.

The Rich library formats query results as beautiful tables in the terminal. You can also export results to CSV by appending `-format csv` to the query command if the CLI supports it.

Step 6: Run Advanced SQL Analytics

Objective: Execute advanced SQL queries using window functions, CTEs, and complex joins for deeper financial analysis.
Instructions:

1. Run the pre-written advanced analytics queries from `queries/analytics.sql`
2. Understand window functions for running totals and rankings
3. Use CTEs for account balance reconciliation
4. Detect fraud patterns using SQL analysis

Code/Command:

```
# Run all analytics queries from file
python src/cli.py query --file queries/analytics.sql

# Window function: Running total per customer
python src/cli.py query "WITH ranked AS (
  SELECT c.name, t.txn_date, t.amount,
    SUM(t.amount) OVER (PARTITION BY c.id
      ORDER BY t.txn_date) AS running_total,
    ROW_NUMBER() OVER (PARTITION BY c.id
      ORDER BY t.amount DESC) AS rank
  FROM customers c JOIN accounts a ON c.id=a.customer_id
  JOIN transactions t ON a.id=t.account_id)
SELECT name, txn_date, amount, running_total
FROM ranked WHERE rank<=3 LIMIT 15"

# CTE: Account balance reconciliation
python src/cli.py query "WITH activity AS (
  SELECT a.account_number, a.balance,
    SUM(CASE WHEN t.txn_type='credit' THEN t.amount
      ELSE 0 END) AS credits,
    SUM(CASE WHEN t.txn_type='debit' THEN t.amount
      ELSE 0 END) AS debits
  FROM accounts a LEFT JOIN transactions t
    ON a.id=t.account_id GROUP BY a.id)
```

```
SELECT␣account_number,␣balance,␣credits,␣debits,
␣␣ROUND(credits-debits,2)␣AS␣calc,
␣␣CASE␣WHEN␣ABS(balance-(credits-debits))<0.01
␣␣␣␣THEN␣'RECONCILED'␣ELSE␣'MISMATCH'␣END␣AS␣status
FROM␣activity␣LIMIT␣10"

# Fraud pattern: Rapid successive large transactions
python src/cli.py query "SELECT␣c.name,
␣␣t1.amount␣AS␣amt1,␣t2.amount␣AS␣amt2,␣t1.channel
FROM␣transactions␣t1␣JOIN␣transactions␣t2
␣␣ON␣t1.account_id=t2.account_id␣AND␣t1.id<t2.id
␣␣AND␣ABS(julianday(t2.txn_date)-
␣␣␣␣julianday(t1.txn_date))<0.01
␣␣AND␣t1.amount>50000␣AND␣t2.amount>50000
JOIN␣accounts␣a␣ON␣t1.account_id=a.id
JOIN␣customers␣c␣ON␣a.customer_id=c.id␣LIMIT␣10"
```

Advanced SQL Analytics

Expected Output

`Reconciliation:` Shows each account with current balance, total credits, total debits, calculated balance, and RECONCILED/MISMATCH status.

`Fraud Patterns:` Lists pairs of large transactions (>50K) occurring within minutes on the same account, with customer name and channel.

Window functions like `SUM() OVER (PARTITION BY ...)` and `ROW_NUMBER()` are essential for financial analytics. They allow you to compute running totals, rankings, and comparisons without subqueries. SQLite supports window functions from version 3.25.0 onwards.

Screenshot 3

What to paste: Terminal showing formatted SQL query results including at least two analytics queries — for example, the top customers by transaction volume table and the merchant category analysis table. Show the Rich-formatted output with borders and alignment.

Paste your screenshot here

Part D: Data Quality and Reporting

Step 7: Run Data Quality Checks

Objective: Execute automated data quality checks across all tables to identify data issues.

Instructions:

1. Run the data quality check command
2. Review the 10 quality checks executed
3. Understand pass/fail criteria for each check
4. Analyze the overall data quality score

Code/Command:

```
python src/cli.py check-quality
```

Run Data Quality Checks

Expected Output

```
=== Financial Data Quality Report ===
Check 1:  Null Detection ..........  [PASS] 0 nulls in required
fields
Check 2:  Duplicate Records .......  [PASS] 0 duplicates in unique
columns
Check 3:  Referential Integrity ....  [PASS] All FK references valid
Check 4:  Date Range Validation ....  [PASS] All dates within range
Check 5:  Amount Outliers .........  [WARN] 12 txns > 3 std devs
Check 6:  Format Validation .......  [PASS] PAN, IFSC, email formats
OK
Check 7:  Completeness ...........  [OK] 99.4% complete
Check 8:  Consistency ............  [PASS] Valid enum values
Check 9:  Timeliness .............  [PASS] Data is current
Check 10:  Uniqueness ............  [OK] 100% PK uniqueness

=== Overall Data Quality Score:  96.5/100 ===
PASS: 18 | WARN: 2 | FAIL: 0
```

The six dimensions of data quality are: **Completeness** (no missing values), **Accuracy** (correct values), **Consistency** (no contradictions), **Timeliness** (up-to-date), **Validity** (conforms to format rules), and **Uniqueness** (no unintended duplicates). Financial regulators expect all six dimensions to be monitored continuously.

Screenshot 4

What to paste: Terminal showing the complete data quality check results with all 10 checks displayed, including PASS/WARN/FAIL status for each check and the overall data quality score at the bottom.

Paste your screenshot here

Step 8: Generate Data Quality Report

Objective: Generate a comprehensive HTML data quality report suitable for audit and compliance submissions.

Instructions:

1. Generate the HTML report using the report command
2. Open the generated report in a web browser
3. Review the report sections: summary, detailed checks, recommendations
4. Note the audit trail information included in the report

Code/Command:

```
# Generate HTML report
python src/cli.py report --format html \
  --output dq_report.html

# Verify and open report
ls -la dq_report.html
# Linux: xdg-open dq_report.html
# Mac: open dq_report.html
# Windows: start dq_report.html
```

Generate HTML Data Quality Report

Expected Output

```
[OK] Report saved to:  dq_report.html (24.5 KB)
```

Report contains: Executive Summary, Quality Scores by Dimension, Detailed Check Results, Data Profiling, Recommendations, and Audit Trail.

The HTML report includes visual charts and color-coded indicators that make it easy for auditors and management to understand data quality status at a glance. Reports like these are commonly required during RBI inspections and internal audit cycles.

Screenshot 5

What to paste: The HTML data quality report opened in a web browser, showing the executive summary section with the overall quality score, the quality score breakdown by dimension, and the color-coded check results.

Paste your screenshot here

Part E: Custom Queries for Indian Banking

Step 9: Write Custom SQL Queries for Indian FinTech Scenarios

Objective: Design and execute your own SQL queries addressing real Indian banking and FinTech requirements including UPI analytics, GST analysis, and RBI reporting.
Instructions:

1. Write a query analyzing UPI transaction patterns
2. Write a query for GST-related merchant analysis
3. Write a query aligned with RBI reporting requirements
4. Document your queries and interpret the results

Code/Command:

```
# Query A: UPI Transaction Analysis by city
python src/cli.py query "SELECT␣c.city,
␣␣COUNT(t.id)␣AS␣upi_count,
␣␣ROUND(SUM(t.amount),2)␣AS␣upi_volume,
␣␣ROUND(AVG(t.amount),2)␣AS␣avg_upi
FROM␣transactions␣t␣JOIN␣accounts␣a␣ON␣t.account_id=a.id
JOIN␣customers␣c␣ON␣a.customer_id=c.id
WHERE␣t.channel='UPI'␣GROUP␣BY␣c.city
ORDER␣BY␣upi_volume␣DESC␣LIMIT␣10"

# Query B: GST Merchant Analysis
python src/cli.py query "SELECT␣m.name,␣m.gst_number,
␣␣COUNT(t.id)␣AS␣txn_count,
␣␣ROUND(SUM(t.amount),2)␣AS␣revenue,
␣␣ROUND(SUM(t.amount)*0.18,2)␣AS␣est_gst
FROM␣merchants␣m␣JOIN␣transactions␣t␣ON␣m.id=t.merchant_id
WHERE␣m.gst_number␣IS␣NOT␣NULL␣GROUP␣BY␣m.id
HAVING␣revenue>100000␣ORDER␣BY␣revenue␣DESC␣LIMIT␣10"

# Query C: RBI Large Value Transactions (>= 10 lakh)
python src/cli.py query "SELECT␣t.id,␣c.name,
␣␣c.pan_number,␣t.amount,␣t.channel,␣t.txn_date
FROM␣transactions␣t␣JOIN␣accounts␣a␣ON␣t.account_id=a.id
JOIN␣customers␣c␣ON␣a.customer_id=c.id
WHERE␣t.amount>=1000000␣ORDER␣BY␣t.txn_date␣DESC"

# Query D: Customer Risk Profiling
python src/cli.py query "SELECT␣c.name,␣c.city,
␣␣COUNT(DISTINCT␣t.id)␣AS␣txns,
␣␣COUNT(DISTINCT␣f.id)␣AS␣flags,
␣␣ROUND(COUNT(DISTINCT␣f.id)*100.0/
␣␣␣␣NULLIF(COUNT(DISTINCT␣t.id),0),2)␣AS␣fraud_pct
FROM␣customers␣c␣JOIN␣accounts␣a␣ON␣c.id=a.customer_id
JOIN␣transactions␣t␣ON␣a.id=t.account_id
LEFT␣JOIN␣fraud_flags␣f␣ON␣t.id=f.transaction_id
GROUP␣BY␣c.id␣HAVING␣flags>0
ORDER␣BY␣fraud_pct␣DESC␣LIMIT␣10"
```

Indian FinTech SQL Queries

Expected Output

Each query returns a Rich-formatted table with results. UPI analysis shows city-wise volumes; GST query shows merchant revenues with estimated tax; RBI query lists high-value transactions with PAN and channel details; Risk profiling shows customers with fraud flag percentages.

Under RBI guidelines, all cash transactions above INR 10 lakh and all suspicious transactions must be reported to the Financial Intelligence Unit (FIU-IND). Banks must also file Currency Transaction Reports (CTRs) and Suspicious Transaction Reports (STRs) as part of Anti-Money Laundering (AML) compliance.

Conceptual Background

Relational Database Fundamentals

Relational databases organize data into structured tables with defined relationships (E.F. Codd, 1970). Core concepts: **Tables** (rows and columns representing entities), **Primary Keys** (unique row identifiers), **Foreign Keys** (references enforcing referential integrity), **Indexes** (speed up queries on frequently searched columns), and **Normalization** (reducing redundancy: 1NF, 2NF, 3NF, BCNF).

Entity-Relationship Diagram

The 5-table schema follows these One-to-Many relationships: customers → accounts (via customer_id), accounts → transactions (via account_id), merchants → transactions (via merchant_id), transactions → fraud_flags (via transaction_id). Tables store KYC data, account details, transaction events, merchant profiles, and fraud audit trails respectively.

SQL Query Types

Basic operations: SELECT (retrieve), WHERE (filter), JOIN (combine tables), GROUP BY (aggregate), HAVING (filter groups), ORDER BY (sort).

Advanced techniques:

- **Subqueries:** Nested queries in WHERE/FROM for complex filtering
- **CTEs:** Named temporary result sets via `WITH` clause for readability
- **Window Functions:** `SUM() OVER()`, `ROW_NUMBER()`, `RANK()`, `LAG()`, `LEAD()` — compute across rows without collapsing
- **CASE:** Conditional logic for classification within queries

Data Quality Dimensions

Data quality is measured across six dimensions: **Completeness** (no missing PAN numbers), **Accuracy** (amounts match payments), **Consistency** (balance = sum of transactions), **Timeliness** (recorded within minutes), **Validity** (PAN follows XXXXX9999X), and **Uniqueness** (each account number appears once).

India-Specific Context

RBI Reporting Requirements

RBI mandates: (1) Currency Transaction Reports (CTR) for cash transactions above INR 10 lakh to FIU-IND, (2) Suspicious Transaction Reports (STR) for AML patterns, (3) accurate KYC data with periodic updates, (4) 5-year minimum data retention after account closure, and (5) complete audit trails for all data modifications.

Financial Data Standards in India

Key formats: **PAN** (XXXXX9999X, 10 chars), **IFSC** (XXXX0XXXXXX, 11 chars), **UPI** (10B+ monthly transactions), **GST Number** (15-digit merchant ID), **Aadhar** (12-digit, must be stored as hashed values per DPDP Act).

GST and UPI Data Management

GST: Merchants must maintain taxable transaction records, file monthly GSTR-1/GSTR-3B returns, ensure invoice-level data matching, and classify using HSN/SAC codes.

UPI: India's UPI processes 10+ billion monthly transactions. Key considerations include real-time settlement, VPA-to-account mapping, per-transaction limits (INR 1–2 lakh), and T+5 day dispute resolution per NPCI guidelines.

Data Privacy: DPDP Act Implications

The Digital Personal Data Protection Act requires purpose limitation, data minimization, consent management, breach notification within 72 hours, and imposes penalties up to INR 250 crore. Financial data must balance DPDP requirements with RBI retention mandates.

Real-World: ICICI/HDFC Data Pipelines

ICICI Bank processes 15+ million daily transactions using centralized data warehouses with real-time ETL and automated quality checks. **HDFC Bank** maintains MDM across 7000+ branches with SQL-based fraud detection. Both use a layered architecture: OLTP databases for real-time operations, OLAP warehouses for analytics, and data lakes for ML.

Assessment & Deliverables

Deliverables Checklist

Item	Description	Type	Status
Screenshot 1	Repo structure & CLI help	Paste	☐
Screenshot 2	Database schema & seeded data counts	Paste	☐
Screenshot 3	SQL query results (formatted table)	Paste	☐
Screenshot 4	Data quality check results	Paste	☐
Screenshot 5	HTML data quality report in browser	Paste	☐
Custom Queries	Indian banking SQL queries	Paste	☐
DQ Report	HTML report file	File	☐
Query Results	Exported analytics results	Text	☐

Verification Checklist

Complete all items below before submitting:

- ☐ Repository cloned and virtual environment set up
- ☐ CLI tool runs successfully with `-help` showing 5 commands
- ☐ Database initialized with 5 tables, foreign keys, and indexes
- ☐ Database seeded with 500 customers, 1000 accounts, 5000 transactions
- ☐ At least 5 analytics queries executed with formatted results
- ☐ Advanced queries (window functions, CTEs) executed successfully
- ☐ All 10 data quality checks run with pass/fail results
- ☐ HTML data quality report generated and opened in browser
- ☐ Custom SQL queries written for Indian banking scenarios
- ☐ All 5 required screenshots captured and pasted

Grading Rubric

Criteria	**Description**	**Points**	**Score**
Setup	Repo cloned, environment ready, CLI verified	10	___/10
Schema Creation	Database initialized with 5 tables	10	___/10
Data Seeding	Synthetic data generated correctly	10	___/10
Basic Queries	5+ analytics queries executed	15	___/15
Advanced SQL	Window functions, CTEs, reconciliation	15	___/15
Data Quality	10 quality checks run with results	15	___/15
Report Generation	HTML report created and reviewed	10	___/10
Custom Queries	Indian banking scenario queries	10	___/10
Documentation	Answers & explanations complete	5	___/5
	TOTAL	**100**	___/100

Assessment Questions

Answer the following questions in your submission:

Q1. What are the six dimensions of data quality? Explain each with a financial data example.

Q2. What is normalization in relational databases? Why is it important for financial data? Explain with an example from the schema used in this practical.

Q3. Explain the difference between a subquery and a CTE (Common Table Expression). When would you prefer one over the other?

Q4. What is referential integrity? What would happen if the `fraud_flags` table referenced a non-existent `transaction_id`?

Q5. Describe three SQL queries you would write to detect money laundering patterns in transaction data.

Q6. What are the RBI's requirements for financial data retention and reporting? How does the CTR/STR framework work?

Q7. How does the DPDP Act affect how banks store and process customer data? What changes would you make to this database schema to comply?

Q8. Compare OLTP and OLAP systems. Why do banks like ICICI and HDFC maintain both types? Which type does this practical simulate?

Appendix A: SQL Quick Reference

Statement	Syntax
SELECT	`SELECT col1, col2 FROM table WHERE condition`
JOIN	`FROM t1 JOIN t2 ON t1.id = t2.fk_id`
GROUP BY	`SELECT col, COUNT(*) FROM t GROUP BY col`
HAVING	`GROUP BY col HAVING SUM(amt) > 1000`
Subquery	`WHERE col IN (SELECT col FROM t2)`
CTE	`WITH cte AS (SELECT ...) SELECT * FROM cte`
Window	`SUM(amt) OVER (PARTITION BY col ORDER BY dt)`
ROW_NUMBER	`ROW_NUMBER() OVER (PARTITION BY col ORDER BY amt)`
CASE	`CASE WHEN cond THEN val ELSE default END`
strftime	`strftime('%Y-%m', date_col)` (SQLite)

Aggregates: `COUNT(*)`, `SUM()`, `AVG()`, `MIN()`, `MAX()`, `GROUP_CONCAT()`. All ignore NULLs except `COUNT(*)`.

Appendix B: Data Quality Framework

No.	Check	SQL Approach
1	Null Detection	`WHERE col IS NULL`
2	Duplicates	`GROUP BY col HAVING COUNT(*) > 1`
3	Referential Integrity	`LEFT JOIN ... WHERE pk IS NULL`
4	Date Validation	`WHERE date NOT BETWEEN ... AND ...`
5	Outliers	`WHERE val > AVG + 3*STDDEV`
6	Format Validation	`WHERE col NOT LIKE pattern`
7	Completeness	`COUNT(col)/COUNT(*) * 100`
8	Consistency	`WHERE col NOT IN ('valid_values')`
9	Timeliness	`WHERE date < datetime('now','-1 day')`
10	Uniqueness	`COUNT(DISTINCT id) = COUNT(*)`

Scoring: Weighted average across six dimensions — Completeness (20%), Accuracy (20%), Consistency (20%), Timeliness (15%), Validity (15%), Uniqueness (10%).

Appendix C: Schema Reference

customers (id PK, name, email, phone, pan_number, aadhar_hash, city, state, created_at) → **accounts** (id PK, customer_id FK, account_type, account_number, ifsc_code, balance, status, opened_at) → **transactions** (id PK, account_id FK, merchant_id FK, txn_type, amount, currency, txn_date, description, channel, status) ← **merchants** (id PK, name, category, mcc_code, city, gst_number, is_active). **fraud_flags** (id PK, transaction_id FK, flag_type, severity, detected_at, resolved) links to transactions. All relationships are One-to-Many.

Appendix D: Troubleshooting

Problem: `sqlite3.OperationalError` or permission denied. **Solutions:** Check write permissions; ensure no other process locks the DB file; delete existing `financial_data.db` and retry.

Problem: `ModuleNotFoundError` for faker, rich, or click. **Solutions:** Ensure venv is activated (check `(venv)` prefix); run `pip install -r requirements.txt`; verify with `pip list`.

Problem: Query runs but no rows returned. **Solutions:** Verify DB is seeded: `python src/cli.py query "SELECT COUNT(*) FROM transactions"`; check table/column names; relax WHERE clauses.

Problem: Browser shows blank page or file not found. **Solutions:** Check file exists: `ls -la dq_report.html`; open manually via File → Open; check terminal for errors.

Appendix E: Additional Resources

Documentation

→ SQLite: `https://www.sqlite.org/docs.html`

→ Python sqlite3: `https://docs.python.org/3/library/sqlite3.html`

→ Faker: `https://faker.readthedocs.io/`

→ Rich: `https://rich.readthedocs.io/`

→ Click: `https://click.palletsprojects.com/`

→ RBI: `https://www.rbi.org.in`

Books

- "SQL for Data Scientists" – Renee Teate
- "Database System Concepts" – Silberschatz, Korth, Sudarshan
- "Designing Data-Intensive Applications" – Martin Kleppmann

Tools Used

All tools are free and open-source: **Python 3.8+** (runtime), **SQLite3** (embedded database), **Faker** (synthetic data), **Rich** (terminal formatting), **Click** (CLI framework), **pip** (package manager), **Git** (version control), **Web Browser** (HTML reports).

—END OF LAB MANUAL—

Document Version: 1.0

IT Management & Audits – Practical Lab Series