

STOCK PORTFOLIO & INVESTMENT TRACKER

SQL Project Report

A comprehensive relational database project demonstrating production-level SQL for data analyst roles in the finance domain.

Domain	Finance & Investment Analytics
Database	MySQL 8.0
Tables	6 (investors, portfolios, stocks, trades, daily_prices, dividends)
Queries	12 (Tier 1: JOINS Tier 2: CTEs Tier 3: Window Functions)
Date	March 22, 2026

1. Introduction

This project presents a fully functional relational database system designed to model a real-world stock portfolio and investment tracking platform. The system manages investor data, portfolio structures, trade histories, market price data, and dividend records across a normalised six-table schema.

The primary objective is to demonstrate production-level SQL proficiency relevant to data analyst roles in the finance and fintech domain. The project covers the full analytical workflow — from schema design and data ingestion to multi-tier query development encompassing JOIN operations, subqueries, Common Table Expressions (CTEs), and advanced window functions.

2. Problem Statement

Investment platforms generate large volumes of transactional data across multiple entities — investors, portfolios, instruments, and market prices. The core analytical challenge is to derive meaningful business insights from this data at varying levels of complexity:

- Track capital deployment and position sizing across portfolios
- Calculate realised profit and loss on executed trades
- Determine dividend income earned by each investor based on shareholding history
- Identify performance patterns using sector rankings and investor tiering
- Analyse price trends using moving averages computed over time-series data

The project addresses these requirements through a structured query library organised into three tiers of increasing analytical complexity.

3. Database Schema

The schema follows a star schema pattern — a standard structure in financial databases — where high-volume transactional tables (trades, daily_prices) act as fact tables, and descriptive entities (investors, portfolios, stocks) act as dimension tables.

Table	Primary Key	Foreign Keys	Purpose
investors	investor_id	—	Platform users who own portfolios
portfolios	portfolio_id	investor_id	Accounts held by investors
stocks	stock_id	—	Tradeable instruments (equities)
trades	trade_id	portfolio_id, stock_id	Every BUY/SELL event — main fact table
daily_prices	price_id	stock_id	Historical OHLCV price data

dividends	dividend_id	stock_id	Dividend payouts with ex_date logic
-----------	-------------	----------	-------------------------------------

Table 1: Schema overview with relationships

3.1 Key Design Decisions

- **ENUM for trade_type:** Restricts values to 'BUY' or 'SELL' only, preventing invalid data entry — superior to VARCHAR which accepts arbitrary strings.
- **DATE vs DATETIME:** Date-only fields (trade_date, ex_date, price_date) use DATE type — storing time components for date-only data wastes space and causes comparison issues.
- **UNIQUE constraint on daily_prices(stock_id, price_date):** Prevents duplicate price records for the same stock on the same day.
- **Indexes on high-traffic columns:** trade_date, portfolio_id, and price_date are indexed to prevent full table scans on the largest tables.
- **Foreign key placement:** Child tables (trades, portfolios, dividends) hold foreign keys because they require parent context to have meaning. A trade without a portfolio reference is unidentifiable.

4. Data Generation Strategy

A Python script was developed to generate realistic datasets using two libraries:

- **Faker (en_IN locale):** Generated synthetic but realistic Indian investor names, emails, and dates across investors, portfolios, and trades tables.
- **yfinance:** Fetched real historical OHLCV price data from Yahoo Finance for all 15 tickers (US and NSE-listed) between January 2022 and December 2024.

This hybrid approach — synthetic transactional data combined with real market prices — makes the daily_prices analysis (moving averages, price trends) analytically meaningful while keeping investor identity data privacy-safe.

Table	Rows (approx.)	Generation Method
investors	50	Faker en_IN
portfolios	80	Faker en_IN
stocks	15	Hardcoded (real tickers)
trades	500	Faker + random
daily_prices	~11,000	yfinance (real data)
dividends	~50	Faker + random

Table 2: Dataset volumes by table

5. Query Analysis

5.1 Tier 1 — JOINS and Aggregations

Tier 1 queries establish the foundational analytical layer. They demonstrate the ability to navigate multi-table schemas and produce business-meaningful summaries using JOIN operations, GROUP BY, and aggregate functions.

Q1. Total Capital Invested per Portfolio

Calculates Assets Under Management (AUM) per portfolio by summing trade value including fees.

```
SELECT p.portfolio_id, p.portfolio_name,
ROUND(SUM(t.quantity * t.price_per_share) + SUM(t.fees), 2) AS total_invested
FROM portfolios p
JOIN trades t ON p.portfolio_id = t.portfolio_id
WHERE t.trade_type = 'BUY'
GROUP BY p.portfolio_id;
```

- **Techniques:** SUM() aggregate, JOIN, GROUP BY, WHERE filter on ENUM column

Q2. Trade Count per Investor (BUY/SELL Split)

Identifies active vs passive investors. Uses conditional aggregation to split BUY and SELL counts into separate columns on the same row — not achievable with a simple GROUP BY trade_type.

```
SELECT i.full_name,
COUNT(CASE WHEN t.trade_type = 'BUY' THEN 1 END) AS buy_trades,
COUNT(CASE WHEN t.trade_type = 'SELL' THEN 1 END) AS sell_trades
FROM investors i
JOIN portfolios p ON i.investor_id = p.investor_id
JOIN trades t ON p.portfolio_id = t.portfolio_id
GROUP BY i.investor_id, i.full_name;
```

- **Techniques:** 3-table JOIN chain, COUNT(CASE WHEN) conditional aggregation

Q3. Most Traded Stocks by Total Volume

Ranks instruments by total shares traded across all portfolios. Useful for identifying popular instruments on the platform.

```
SELECT s.ticker, s.company_name,
SUM(t.quantity) AS total_volume
FROM stocks s
JOIN trades t ON s.stock_id = t.stock_id
GROUP BY s.stock_id
ORDER BY total_volume DESC
LIMIT 5;
```

- **Techniques:** SUM aggregate, ORDER BY with LIMIT for top-N pattern

Q4. Current Net Shares Held per Portfolio per Stock

Computes live position sizing by treating BUY as positive and SELL as negative quantity. HAVING filters out fully exited positions.

```
SELECT p.portfolio_id, s.ticker,  
SUM(CASE WHEN t.trade_type = 'BUY' THEN t.quantity  
WHEN t.trade_type = 'SELL' THEN -t.quantity END) AS shares_held  
FROM trades t  
JOIN portfolios p ON p.portfolio_id = t.portfolio_id  
JOIN stocks s ON s.stock_id = t.stock_id  
GROUP BY p.portfolio_id, s.stock_id  
HAVING shares_held > 0;
```

- **Techniques:** SUM(CASE WHEN) with positive/negative logic, HAVING vs WHERE distinction

Q5. Total Fees Paid per Investor

Fee revenue report from the platform perspective. HAVING excludes zero-fee investors.

```
SELECT i.full_name,  
ROUND(SUM(t.fees), 2) AS total_fees  
FROM investors i  
JOIN portfolios p ON i.investor_id = p.investor_id  
JOIN trades t ON p.portfolio_id = t.portfolio_id  
GROUP BY i.investor_id  
HAVING SUM(t.fees) > 0  
ORDER BY total_fees DESC;
```

- **Techniques:** 3-table JOIN, HAVING for post-aggregation filtering

5.2 Tier 2 — Subqueries and Common Table Expressions (CTEs)

Tier 2 queries address multi-step analytical problems. CTEs (WITH clauses) are used to decompose complex logic into named, readable intermediate steps — the professional standard in production SQL environments. A CTE exists only for the duration of a single query, making it ideal for intermediate calculations that are not needed permanently.

Q6. Portfolios Above Platform Average Investment

Identifies high-value portfolios for priority service. The nested subquery first computes total investment per portfolio, then averages those totals — producing a single platform-wide benchmark figure for HAVING comparison.

```
SELECT p.portfolio_id,
ROUND(SUM(t.quantity * t.price_per_share), 2) AS total_invested
FROM portfolios p
JOIN trades t ON p.portfolio_id = t.portfolio_id
GROUP BY p.portfolio_id
HAVING total_invested > (
SELECT AVG(portfolio_total) FROM (
SELECT SUM(quantity * price_per_share) AS portfolio_total
FROM trades GROUP BY portfolio_id
) AS sub
);
```

- **Techniques:** Nested subquery, HAVING with aggregate comparison, two-level aggregation

Q7. Realised P&L; on Every SELL Trade

Computes trade-level profit and loss. The CTE calculates the average cost basis (weighted average buy price) per (portfolio, stock) combination. The main query joins SELL trades against this to compute: (sell price - avg buy price) x quantity - fees. Results are labelled PROFIT, LOSS, or BREAK EVEN.

```
WITH avg_buy_cost AS (
SELECT portfolio_id, stock_id,
SUM(quantity*price_per_share)/SUM(quantity) AS avg_buy_price
FROM trades WHERE trade_type = 'BUY'
GROUP BY portfolio_id, stock_id
)
SELECT t.portfolio_id, s.ticker,
ROUND((t.price_per_share - a.avg_buy_price)
* t.quantity - t.fees, 2) AS net_pnl,
CASE WHEN ... THEN 'PROFIT' ... END AS result
FROM trades t
JOIN avg_buy_cost a
ON a.portfolio_id = t.portfolio_id
AND a.stock_id = t.stock_id
WHERE t.trade_type = 'SELL';
```

- **Techniques:** CTE for cost basis calculation, multi-column JOIN condition, CASE WHEN classification

Q8. Dividend Income per Investor

Calculates dividend earnings respecting the ex_date rule: an investor only receives a dividend if they held shares before the ex_date. The CTE computes net shares held at each ex_date using BUY-SELL logic filtered by trade_date < ex_date. COALESCE handles NULL SELL values for buy-and-hold positions.

```
WITH holdings AS (  
  SELECT t.portfolio_id, t.stock_id, d.dividend_id,  
    SUM(CASE WHEN t.trade_type='BUY' THEN t.quantity ELSE 0 END)  
    - SUM(CASE WHEN t.trade_type='SELL' THEN t.quantity ELSE 0 END)  
  AS shares_held  
  FROM trades t  
  JOIN dividends d ON d.stock_id = t.stock_id  
  WHERE t.trade_date < d.ex_date  
  GROUP BY t.portfolio_id, t.stock_id, d.dividend_id  
  HAVING shares_held > 0  
)  
SELECT i.full_name,  
  ROUND(SUM(h.shares_held * d.amount_per_share), 2) AS dividend_income  
FROM holdings h  
JOIN portfolios p ON p.portfolio_id = h.portfolio_id  
JOIN investors i ON i.investor_id = p.investor_id  
JOIN dividends d ON d.dividend_id = h.dividend_id  
GROUP BY i.investor_id;
```

- **Techniques:** CTE with ex_date date logic, BUY-SELL netting inside CTE, 4-table JOIN in main query

5.3 Tier 3 — Window Functions

Window functions perform calculations across a set of rows related to the current row without collapsing them — unlike GROUP BY which reduces rows to one per group. The syntax pattern is FUNCTION() OVER (PARTITION BY ... ORDER BY ... ROWS BETWEEN ...). PARTITION BY defines the group scope; ORDER BY defines the row sequence within that group; ROWS BETWEEN defines the frame size for rolling calculations.

Function	Query	Business Use
RANK()	Stocks by volume within sector	Sector performance leaderboard
SUM() OVER	Running total per portfolio	Capital deployment timeline
AVG() OVER + ROWS	7-day moving average price	Price trend smoothing
NTILE(4)	Investor quartile tiering	Customer segmentation (Standard to Platinum)

Table 3: Window functions used and their business applications

Q9. Rank Stocks by Trade Volume Within Sector

Produces a sector-level leaderboard. RANK() resets to 1 for each sector partition, with ties receiving the same rank and the next rank skipping accordingly (1,1,3 pattern).

```
WITH total_vol AS (  
  SELECT s.stock_id, s.sector, s.ticker,  
         SUM(t.quantity) AS total_volume  
  FROM trades t JOIN stocks s ON s.stock_id = t.stock_id  
  GROUP BY s.stock_id  
)  
SELECT sector, ticker, total_volume,  
       RANK() OVER (  
  PARTITION BY sector  
  ORDER BY total_volume DESC  
) AS sector_rank  
FROM total_vol  
ORDER BY sector, sector_rank;
```

- **Techniques:** RANK() window function, PARTITION BY for per-sector ranking, CTE pre-aggregation

Q10. Running Total Invested per Portfolio Over Time

Shows cumulative capital deployment after each trade, ordered chronologically within each portfolio. ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW is the explicit frame definition for a cumulative sum.

```
SELECT portfolio_id, trade_date,
ROUND(quantity * price_per_share, 2) AS trade_value,
ROUND(SUM(quantity * price_per_share) OVER (
PARTITION BY portfolio_id
ORDER BY trade_date
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
), 2) AS running_total
FROM trades
WHERE trade_type = 'BUY'
ORDER BY portfolio_id, trade_date;
```

- **Techniques:** SUM() OVER cumulative pattern, ROWS BETWEEN frame clause

Q11. 7-Day Moving Average of Stock Closing Price

Smooths daily price noise to reveal underlying trends — used in virtually every trading dashboard. ROWS BETWEEN 6 PRECEDING AND CURRENT ROW defines a 7-row window. When fewer than 7 rows exist (start of data), MySQL averages available rows without error.

```
SELECT s.ticker, dp.price_date, dp.close_price,
ROUND(AVG(dp.close_price) OVER (
PARTITION BY dp.stock_id
ORDER BY dp.price_date
ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
), 2) AS moving_avg_7day
FROM daily_prices dp
JOIN stocks s ON s.stock_id = dp.stock_id
ORDER BY s.ticker, dp.price_date;
```

- **Techniques:** AVG() OVER with ROWS BETWEEN, time-series window on real market data

Q12. Investor Quartile Tiering with NTILE

Segments all investors into four equal performance buckets by total investment. NTILE(4) assigns bucket 1 (lowest 25%) through 4 (top 25%). Combined with CASE WHEN to map bucket numbers to business tier labels.

```
WITH investor_totals AS (  
  SELECT i.investor_id, i.full_name,  
         ROUND(SUM(t.quantity * t.price_per_share), 2) AS total_invested  
  FROM trades t  
  JOIN portfolios p ON p.portfolio_id = t.portfolio_id  
  JOIN investors i ON i.investor_id = p.investor_id  
  GROUP BY i.investor_id  
)  
SELECT full_name, total_invested,  
       NTILE(4) OVER (ORDER BY total_invested) AS quartile,  
       CASE NTILE(4) OVER (ORDER BY total_invested)  
         WHEN 1 THEN 'Standard'  
         WHEN 2 THEN 'Silver'  
         WHEN 3 THEN 'Gold'  
         WHEN 4 THEN 'Platinum'  
       END AS investor_tier  
FROM investor_totals  
ORDER BY total_invested DESC;
```

- **Techniques:** NTILE(4) for equal-bucket segmentation, CASE WHEN for label mapping, CTE pre-aggregation

6. Key Concepts and Reasoning

HAVING vs WHERE

WHERE filters individual rows before GROUP BY executes — aggregate functions do not exist yet at that point. HAVING filters after GROUP BY, operating on aggregated results. Any filter involving SUM(), COUNT(), or AVG() must use HAVING.

GROUP BY vs Window Functions

GROUP BY collapses multiple rows into one row per group, losing individual row detail. Window functions compute across a group while preserving every row — enabling calculations like running totals, rankings, and moving averages that require both individual and group context simultaneously.

Cost Basis Calculation

The weighted average buy price formula — $\text{SUM}(\text{quantity} \times \text{price}) / \text{SUM}(\text{quantity})$ — accounts for investors who made multiple purchases of the same stock at different prices. Simple averaging of prices would be mathematically incorrect.

Ex-Date Logic in Dividends

A dividend is only paid to investors who held shares before the ex_date. This is enforced via `trade_date < ex_date` in the WHERE clause of the holdings CTE, followed by BUY-SELL netting to determine actual share count at that point in time.

Foreign Key Direction

Child tables hold foreign keys because they require parent context to be meaningful. A trade without a portfolio_id is unidentifiable. A portfolio without an investor_id has no owner. The parent table (stocks, investors) is self-contained and complete on its own.

Star Schema Design

trades and daily_prices are fact tables — high-volume, numeric event records. investors, portfolios, and stocks are dimension tables — descriptive context. This mirrors real financial database architecture and is the correct framing for interviews.

7. SQL Errors Encountered and Resolved

The following MySQL errors were encountered and resolved during query development. Understanding error codes is an important signal of SQL maturity in interviews.

Error Code	Name	Cause	Resolution
1054	Unknown Column	Typo in column or table name	Check spelling; verify column exists in referenced table

1052	Ambiguous Column	Column name exists in multiple joined tables	Prefix with table name: table.column
1055	ONLY_FULL_GROUP_BY	SELECT column not in GROUP BY or aggregate	Add column to GROUP BY or wrap in aggregate function
1064	Syntax Error	Missing keyword, comma, or closing parenthesis	Check query structure; CTE must be followed by SELECT
1066	Not Unique Table Alias	Same table joined twice without aliasing	Use table aliases; combine conditions in single JOIN
1109	Unknown Table in Field	CTE or subquery alias not in scope	Ensure CTE precedes the SELECT that references it
1111	Invalid Group Function	Nested aggregate: MAX(SUM())	Use subquery or CTE to pre-aggregate before outer aggregate
1140	Non-Aggregated Column	Mixing non-aggregated columns without GROUP BY	Add GROUP BY or use window function instead

Table 4: MySQL error codes encountered and their resolutions

8. Conclusion

This project demonstrates a complete SQL analytical workflow applied to a realistic finance domain — from normalised schema design through multi-tier query development. The three-tier query structure mirrors the progression expected of a data analyst: foundational aggregation, business-logic CTEs, and advanced window function analytics.

Key analytical capabilities demonstrated include cost basis calculation for P&L; reporting, ex_date-aware dividend income analysis, time-series moving averages on real market data, and investor segmentation using NTILE — all patterns directly applicable to roles in fintech, investment banking, and financial data analytics.

The project is published on GitHub and serves as a portfolio artefact demonstrating production-level SQL proficiency in a domain-relevant context.