Advanced SQL Final Project – Mini Data Warehouse (MySQL 8.0)

Mocha Madness – Final Report

Date: September 02, 2025

Group-2

Student:

Dikshali Korat, Harshkumar Damor, Rajkamal Kumaresan, Khaing Mye Naung

Course: Advanced SQL

Professor: Dr. Clement Aladi

# Table of Contents

Insert an automatic Table of Contents in Word: References → Table of Contents → Automatic.

Table of Contents

[Table of Contents 2](#_Toc207716148)

[1. Overview & How to Run 3](#_Toc207716149)

[2. OLTP Schema (ERD & Constraints) 3](#_Toc207716150)

[3. OLAP Star Schema & ETL 4](#_Toc207716151)

[4. Views & “Materialized” View 5](#_Toc207716152)

[5. Transactions & Concurrency 6](#_Toc207716153)

[6. Performance & Indexing 7](#_Toc207716154)

[7. Core Queries (10+) with Business Rationale 8](#_Toc207716155)

[1. Daily revenue & units (view) 8](#_Toc207716156)

[2. Top products (30 days) 8](#_Toc207716157)

[3. MoM revenue by category 8](#_Toc207716158)

[4. Customer LTV 8](#_Toc207716159)

[5. New vs Returning (CTE) 8](#_Toc207716160)

[6. Attachment rate 9](#_Toc207716161)

[7. Price band performance + HAVING 9](#_Toc207716162)

[8. Weekday vs weekend 9](#_Toc207716163)

[9. Pareto products (revenue share) 9](#_Toc207716164)

[10. Monthly MV usage 10](#_Toc207716165)

[8. Screenshot Map & Checklist 10](#_Toc207716166)

[9. Assumptions, Limitations, and Future Work 10](#_Toc207716167)

# 

# 1. Overview & How to Run

Goal: Build a small OLTP schema for coffee shop operations and an OLAP star schema for analytics, with views, a simulated materialized view, a transactional stored procedure, and performance indexes.

Environment: MySQL 8.0+, tested in MySQL Workbench.

Run order (from /sql):

1. 01\_create\_oltp.sql – create and load OLTP schema/data.
2. 02\_sample\_oltp\_data.sql – optional extra seed rows (idempotent).
3. 03\_triggers.sql – audit + stock decrement + payment status triggers.
4. 04\_olap\_schema.sql – create mocha\_dw and load dimensions + fact (order\_item\_id as PK).
5. 05\_views.sql – reporting views (daily sales, top products 30d).
6. 06\_materialized\_view.sql – MV table + refresh procedure (optional event).
7. 07\_indexes.sql – add three targeted indexes (capture EXPLAIN before/after).
8. 08\_procedures\_transactions.sql – transactional sp\_place\_order.
9. 09\_demo\_queries.sql – 10+ analytics queries.

# 2. OLTP Schema (ERD & Constraints)

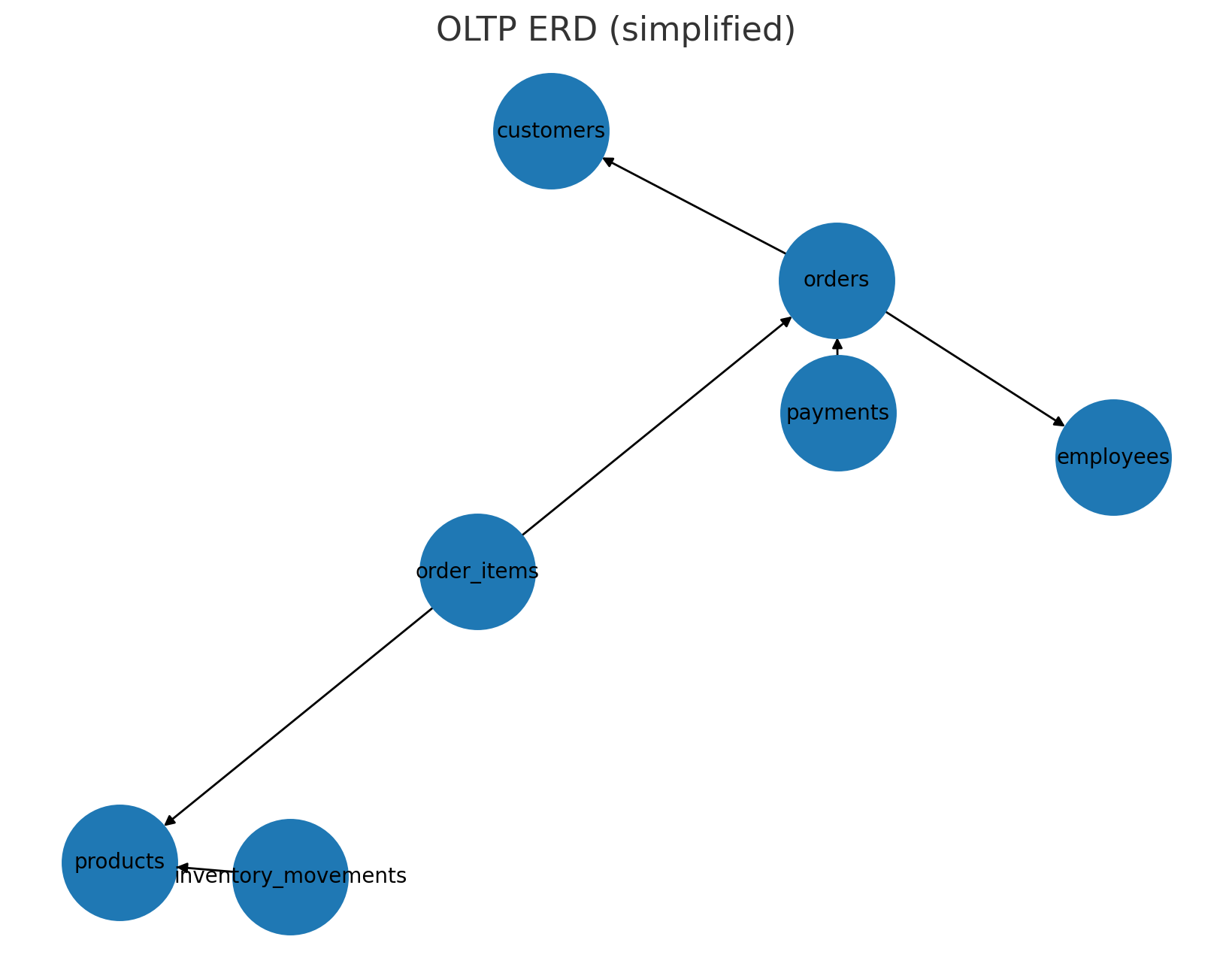
The OLTP schema models customers, orders, order\_items, products, employees, payments, and inventory\_movements. Constraints enforce data integrity: primary keys for identity, foreign keys for relationships, and ENUMs for small domains.

Key relationships: customers→orders (1:M), orders→order\_items (1:M), products→order\_items (1:M), orders→payments (1:M), products→inventory\_movements (1:M), employees→orders (optional assignment).

Triggers:

* Customer audit (insert/update/delete) writes to customer\_audit.
* Order item AFTER INSERT decrements products.stock and logs inventory\_movements.
* Payment update sets order status to PAID when payment succeeds.

ERD:

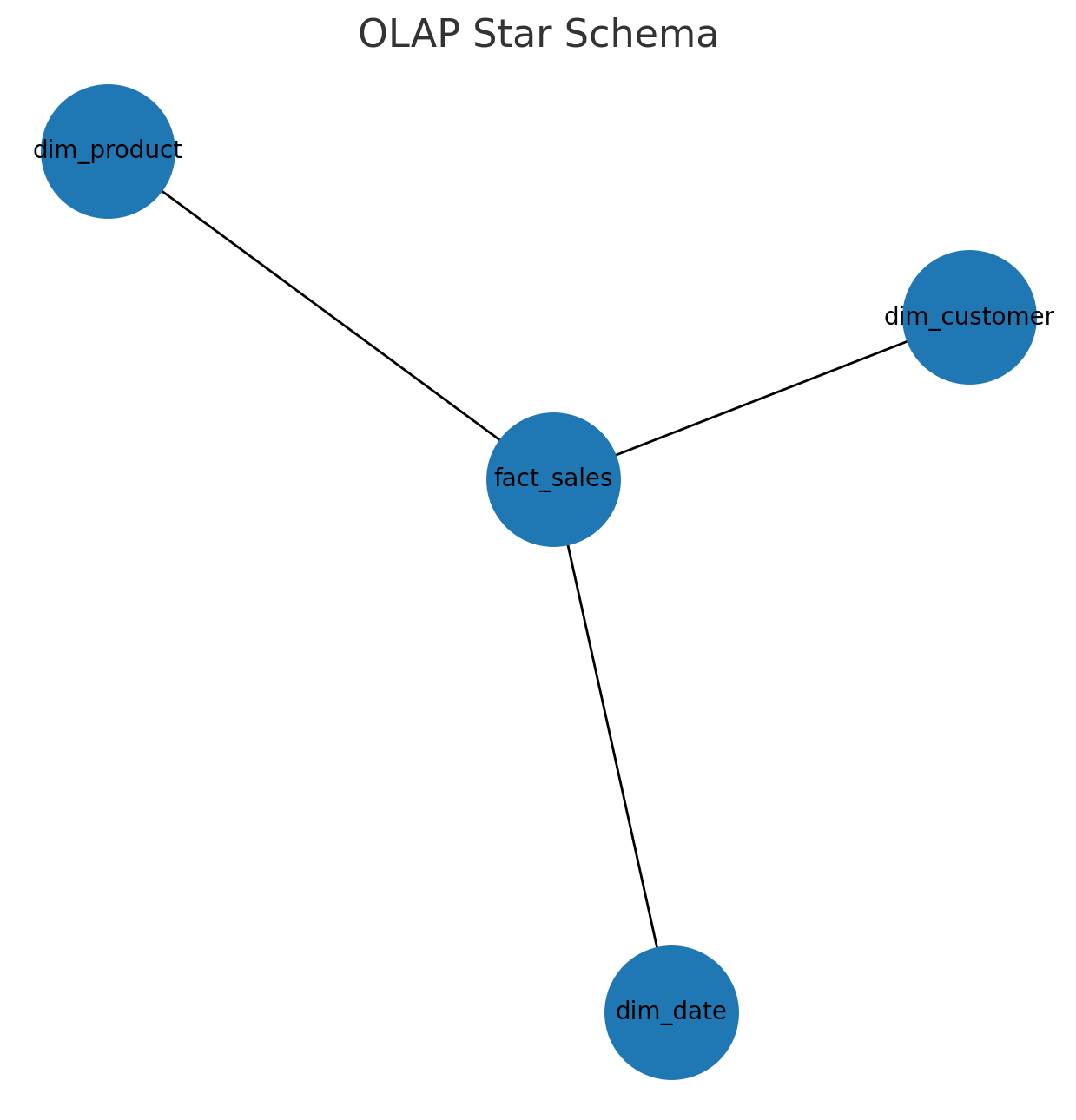


# 3. OLAP Star Schema & ETL

We separate analytics into mocha\_dw with a star schema. Grain = one row per order line (matching OLTP order\_items). Dimensions: dim\_date (calendar), dim\_product (name/category/current price), dim\_customer (profile).

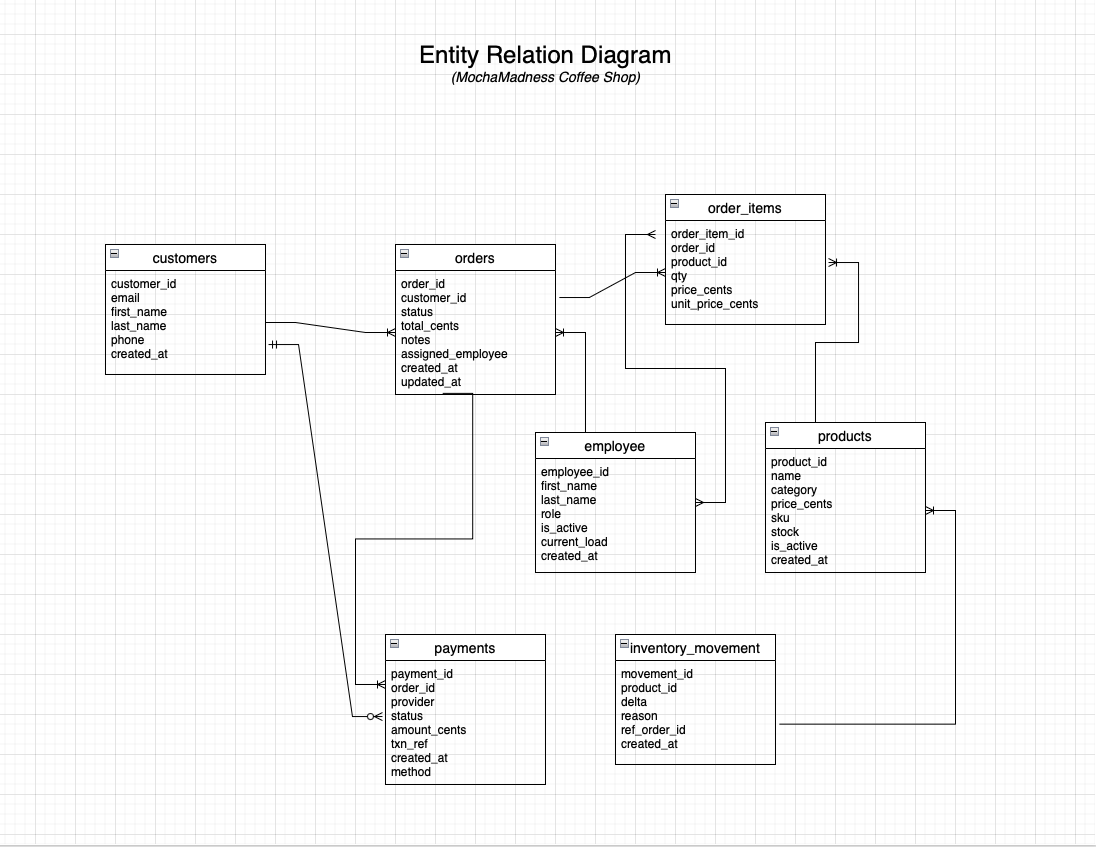
Why this grain: avoids duplicate-key issues when an order repeats a product on multiple lines; supports drill-down to line level.

ETL summary: build date spine from orders.created\_at, upsert products and customers into dims, then load fact\_sales with order\_item\_id as PK.



Indexes in OLAP: fact\_sales(date\_key), fact\_sales(product\_key), fact\_sales(customer\_key), and idx\_fs\_order(order\_id).

**Entity Relationship Diagram**



# 4. Views & “Materialized” View

Regular views centralize recurring logic; the MV simulates pre-aggregation for month-end reporting.

Views:

* vw\_daily\_sales – daily totals (revenue, units) filtered to revenue statuses.
* vw\_top\_products\_30d – top sellers in the last 30 days (revenue, units).

Materialized view (simulated): mv\_monthly\_product\_sales + sp\_refresh\_mv\_monthly\_product\_sales(). Optional nightly event at 02:00 for automatic refresh.

CREATE VIEW snippets:

USE mocha\_dw;  
CREATE OR REPLACE VIEW vw\_daily\_sales AS  
SELECT d.date\_key,  
 SUM(f.line\_amount\_cents)/100.0 AS sales\_usd,  
 SUM(f.qty) AS units  
FROM fact\_sales f  
JOIN dim\_date d ON d.date\_key = f.date\_key  
WHERE f.order\_status IN ('PAID','READY','COMPLETED')  
GROUP BY d.date\_key;

CREATE OR REPLACE VIEW vw\_top\_products\_30d AS  
SELECT p.product\_key, p.product\_name, p.category,  
 SUM(f.line\_amount\_cents)/100.0 AS revenue\_usd,  
 SUM(f.qty) AS units  
FROM fact\_sales f  
JOIN dim\_product p ON p.product\_key = f.product\_key  
WHERE f.order\_status IN ('PAID','READY','COMPLETED')  
 AND f.date\_key >= (CURRENT\_DATE - INTERVAL 30 DAY)  
GROUP BY p.product\_key, p.product\_name, p.category  
ORDER BY revenue\_usd DESC;

MV refresh call:

CALL mocha\_dw.sp\_refresh\_mv\_monthly\_product\_sales();

# 5. Transactions & Concurrency

Stored procedure sp\_place\_order ensures atomic order placement with stock checks and row locks. We stage the JSON cart into a temp table, snapshot prices, lock only needed product rows using FOR UPDATE, then either ROLLBACK on insufficient stock or COMMIT after inserting lines and updating totals.

Key guarantees: ACID, no overselling (locks), trigger-friendly (no products read in the insert firing the trigger), deterministic totals.

Procedure (excerpt):

DELIMITER $$  
CREATE PROCEDURE sp\_place\_order(IN p\_customer\_id INT, IN p\_assigned\_employee\_id INT, IN p\_items\_json JSON)  
BEGIN  
 DECLARE v\_order\_id INT; DECLARE v\_insufficient INT DEFAULT 0;  
 START TRANSACTION;  
 DROP TEMPORARY TABLE IF EXISTS \_req\_items;  
 CREATE TEMPORARY TABLE \_req\_items(product\_id INT, qty INT, price\_each\_cents INT DEFAULT 0, PRIMARY KEY(product\_id)) ENGINE=MEMORY;  
 INSERT INTO \_req\_items(product\_id, qty)  
 SELECT jt.product\_id, jt.qty FROM JSON\_TABLE(p\_items\_json, '$[\*]' COLUMNS(product\_id INT PATH '$.product\_id', qty INT PATH '$.qty')) jt;  
 UPDATE \_req\_items r JOIN products p ON p.product\_id=r.product\_id SET r.price\_each\_cents=p.price\_cents;  
 INSERT INTO orders(customer\_id,status,total\_cents,assigned\_employee\_id,notes) VALUES(p\_customer\_id,'PENDING',0,p\_assigned\_employee\_id,'Placed via sp\_place\_order');  
 SET v\_order\_id = LAST\_INSERT\_ID();  
 SELECT p.product\_id FROM products p JOIN \_req\_items r ON r.product\_id=p.product\_id FOR UPDATE;  
 SELECT COUNT(\*) INTO v\_insufficient FROM products p JOIN \_req\_items r ON r.product\_id=p.product\_id WHERE p.stock < r.qty;  
 IF v\_insufficient > 0 THEN ROLLBACK; DROP TEMPORARY TABLE IF EXISTS \_req\_items; SELECT 'FAILED\_INSUFFICIENT\_STOCK' AS result, NULL AS order\_id;  
 ELSE  
 INSERT INTO order\_items(order\_id, product\_id, qty, price\_cents)  
 SELECT v\_order\_id, r.product\_id, r.qty, (r.price\_each\_cents \* r.qty) FROM \_req\_items r;  
 UPDATE orders o JOIN (SELECT v\_order\_id AS order\_id, SUM(price\_each\_cents\*qty) AS total\_cents FROM \_req\_items) x ON x.order\_id=o.order\_id  
 SET o.total\_cents=x.total\_cents, o.status='PAID';  
 COMMIT; DROP TEMPORARY TABLE IF EXISTS \_req\_items; SELECT 'SUCCESS' AS result, v\_order\_id AS order\_id;  
 END IF;  
END $$  
DELIMITER ;

Concurrency demo: Session A holds a row lock with SELECT … FOR UPDATE; Session B's order waits until A commits, proving isolation and preventing race conditions.

# 6. Performance & Indexing

Three indexes chosen to match common joins/filters/sorts:

* order\_items(order\_id, product\_id) – speeds joins per order and aggregations per order.
* payments(status, provider) – filters success rate by provider quickly.
* orders(created\_at, status) – matches WHERE (recent+paid) + ORDER BY created\_at.

EXPLAIN example to capture before/after screenshots:

EXPLAIN  
SELECT o.order\_id, o.created\_at, SUM(oi.price\_cents) AS line\_total  
FROM orders o  
JOIN order\_items oi ON oi.order\_id = o.order\_id  
WHERE o.created\_at >= CURRENT\_DATE - INTERVAL 60 DAY  
 AND o.status IN ('PAID','READY','COMPLETED')  
GROUP BY o.order\_id, o.created\_at  
ORDER BY o.created\_at DESC;

Expected result: reduced rows examined and indexed access on joins/filters after indexes are created.

# 7. Core Queries (10+) with Business Rationale

## 1. Daily revenue & units (view)

Business rationale: Operational trend tracking; plan staffing and purchasing.

SELECT \* FROM mocha\_dw.vw\_daily\_sales ORDER BY date\_key DESC LIMIT 14;

Insert result screenshot below (from /demo).

## 2. Top products (30 days)

Business rationale: Promotion targeting and stock focus.

SELECT \* FROM mocha\_dw.vw\_top\_products\_30d LIMIT 10;

Insert result screenshot below (from /demo).

## 3. MoM revenue by category

Business rationale: Seasonality and product-mix shifts.

SELECT d.year, d.month, p.category, ROUND(SUM(f.line\_amount\_cents)/100.0,2) AS revenue\_usd  
FROM fact\_sales f JOIN dim\_date d ON d.date\_key=f.date\_key JOIN dim\_product p ON p.product\_key=f.product\_key  
WHERE f.order\_status IN ('PAID','READY','COMPLETED')  
GROUP BY d.year, d.month, p.category  
ORDER BY d.year, d.month, revenue\_usd DESC;

Insert result screenshot below (from /demo).

## 4. Customer LTV

Business rationale: Identify high-value customers for loyalty.

SELECT c.customer\_key, c.email, ROUND(SUM(f.line\_amount\_cents)/100.0,2) AS lifetime\_value\_usd  
FROM fact\_sales f JOIN dim\_customer c ON c.customer\_key=f.customer\_key  
WHERE f.order\_status IN ('PAID','READY','COMPLETED')  
GROUP BY c.customer\_key, c.email  
ORDER BY lifetime\_value\_usd DESC LIMIT 20;

Insert result screenshot below (from /demo).

## 5. New vs Returning (CTE)

Business rationale: Balance acquisition vs retention.

WITH first\_order AS (  
 SELECT customer\_key, MIN(date\_key) AS first\_date FROM fact\_sales GROUP BY customer\_key  
)  
SELECT CASE WHEN f.date\_key = fo.first\_date THEN 'NEW' ELSE 'RETURNING' END AS cohort,  
 COUNT(DISTINCT f.order\_id) AS orders,  
 ROUND(SUM(f.line\_amount\_cents)/100.0,2) AS revenue\_usd  
FROM fact\_sales f JOIN first\_order fo ON fo.customer\_key=f.customer\_key  
WHERE f.date\_key >= CURRENT\_DATE - INTERVAL 60 DAY AND f.order\_status IN ('PAID','READY','COMPLETED')  
GROUP BY cohort;

Insert result screenshot below (from /demo).

## 6. Attachment rate

Business rationale: Cross-sell effectiveness.

WITH per\_order AS (  
 SELECT order\_id, COUNT(DISTINCT product\_key) AS distinct\_items FROM fact\_sales GROUP BY order\_id  
)  
SELECT ROUND(100.0 \* SUM(distinct\_items >= 2)/COUNT(\*), 2) AS pct\_with\_2plus FROM per\_order;

Insert result screenshot below (from /demo).

## 7. Price band performance + HAVING

Business rationale: Price sensitivity and mix health.

SELECT CASE WHEN p.current\_price\_cents < 400 THEN 'Under $4'  
 WHEN p.current\_price\_cents < 600 THEN '$4–5.99' ELSE '$6+'  
 END AS price\_band,  
 SUM(f.qty) AS units,  
 ROUND(SUM(f.line\_amount\_cents)/100.0,2) AS revenue\_usd  
FROM fact\_sales f JOIN dim\_product p ON p.product\_key=f.product\_key  
WHERE f.order\_status IN ('PAID','READY','COMPLETED')  
GROUP BY price\_band HAVING units > 0 ORDER BY revenue\_usd DESC;

Insert result screenshot below (from /demo).

## 8. Weekday vs weekend

Business rationale: Staff scheduling and hours.

SELECT CASE WHEN d.day\_of\_week IN (1,7) THEN 'Weekend' ELSE 'Weekday' END AS day\_type,  
 ROUND(SUM(f.line\_amount\_cents)/100.0,2) AS revenue\_usd  
FROM fact\_sales f JOIN dim\_date d ON d.date\_key=f.date\_key  
WHERE f.order\_status IN ('PAID','READY','COMPLETED')  
GROUP BY day\_type;

Insert result screenshot below (from /demo).

## 9. Pareto products (revenue share)

Business rationale: Focus attention on top drivers.

WITH per\_product AS (  
 SELECT p.product\_key, p.product\_name, SUM(f.line\_amount\_cents) AS rev  
 FROM fact\_sales f JOIN dim\_product p ON p.product\_key=f.product\_key  
 WHERE f.order\_status IN ('PAID','READY','COMPLETED')  
 GROUP BY p.product\_key, p.product\_name  
), tot AS (SELECT SUM(rev) AS total\_rev FROM per\_product)  
SELECT pp.product\_key, pp.product\_name, ROUND(pp.rev/t.total\_rev\*100.0,2) AS pct\_of\_total  
FROM per\_product pp CROSS JOIN tot t  
ORDER BY pct\_of\_total DESC LIMIT 10;

Insert result screenshot below (from /demo).

## 10. Monthly MV usage

Business rationale: Fast month-end reporting leveraging pre-aggregation.

CALL mocha\_dw.sp\_refresh\_mv\_monthly\_product\_sales();  
SELECT \* FROM mocha\_dw.mv\_monthly\_product\_sales ORDER BY year DESC, month DESC, revenue\_usd DESC LIMIT 20;

Insert result screenshot below (from /demo).

# 8. Screenshot Map & Checklist

Include the following in /demo and paste into this document:

* Schema pane (both schemas).
* SHOW CREATE TABLE orders, order\_items, products, payments (PKs/FKs/ENUMs).
* Row counts across major tables.
* Views: vw\_daily\_sales (grid), vw\_top\_products\_30d (grid).
* MV: CALL refresh (Action Output), SELECT from mv table (grid).
* Procedure: SHOW CREATE PROCEDURE sp\_place\_order.
* Transactions: before stock → SUCCESS → after (stock/movement/order).
* Rollback: FAILED\_INSUFFICIENT\_STOCK result, no changes.
* Concurrency: Tab A lock, Tab B waiting, Tab B success after commit.
* EXPLAIN before and after creating indexes.

# 9. Assumptions, Limitations, and Future Work

Assumptions: MySQL 8.0+ with JSON\_TABLE; event scheduler may be OFF on lab machines; payment success is simulated.

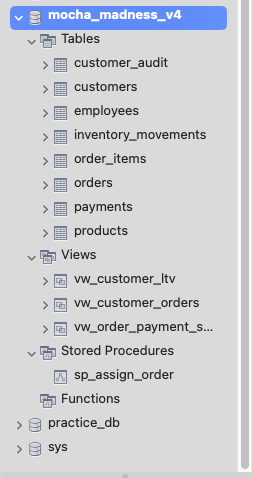
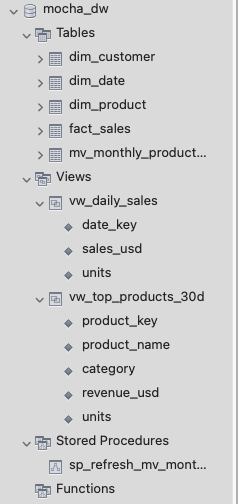
Limitations: No returns/refunds fact; product pricing treated as SCD1 (overwrite).

Future work: add SCD2 for price history, returns/refunds handling, dim\_employee for staffing analysis, and BI dashboard layer.

Annexes

# Screenshot Map & Checklist

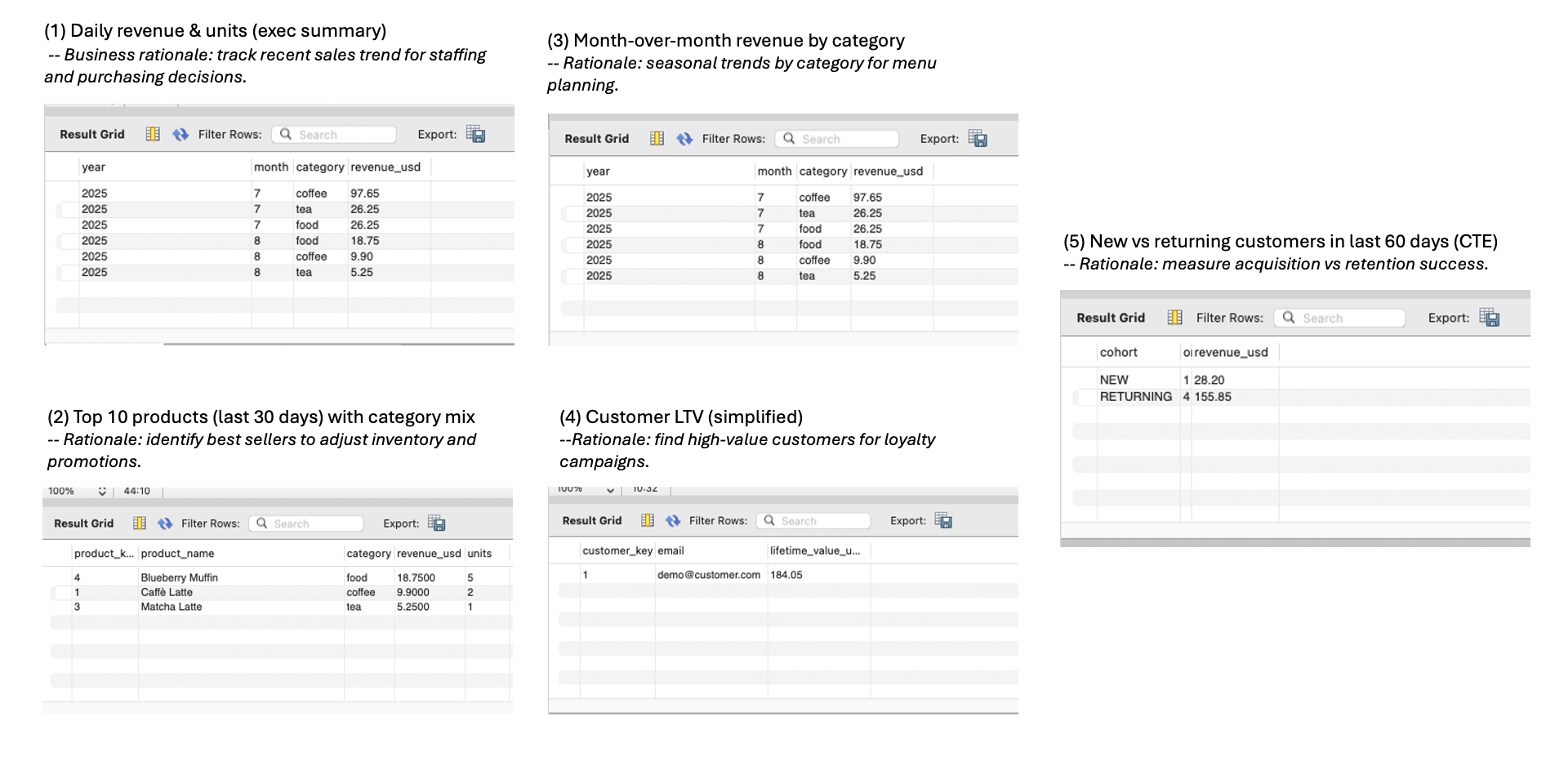
* Schema pane (both schemas).



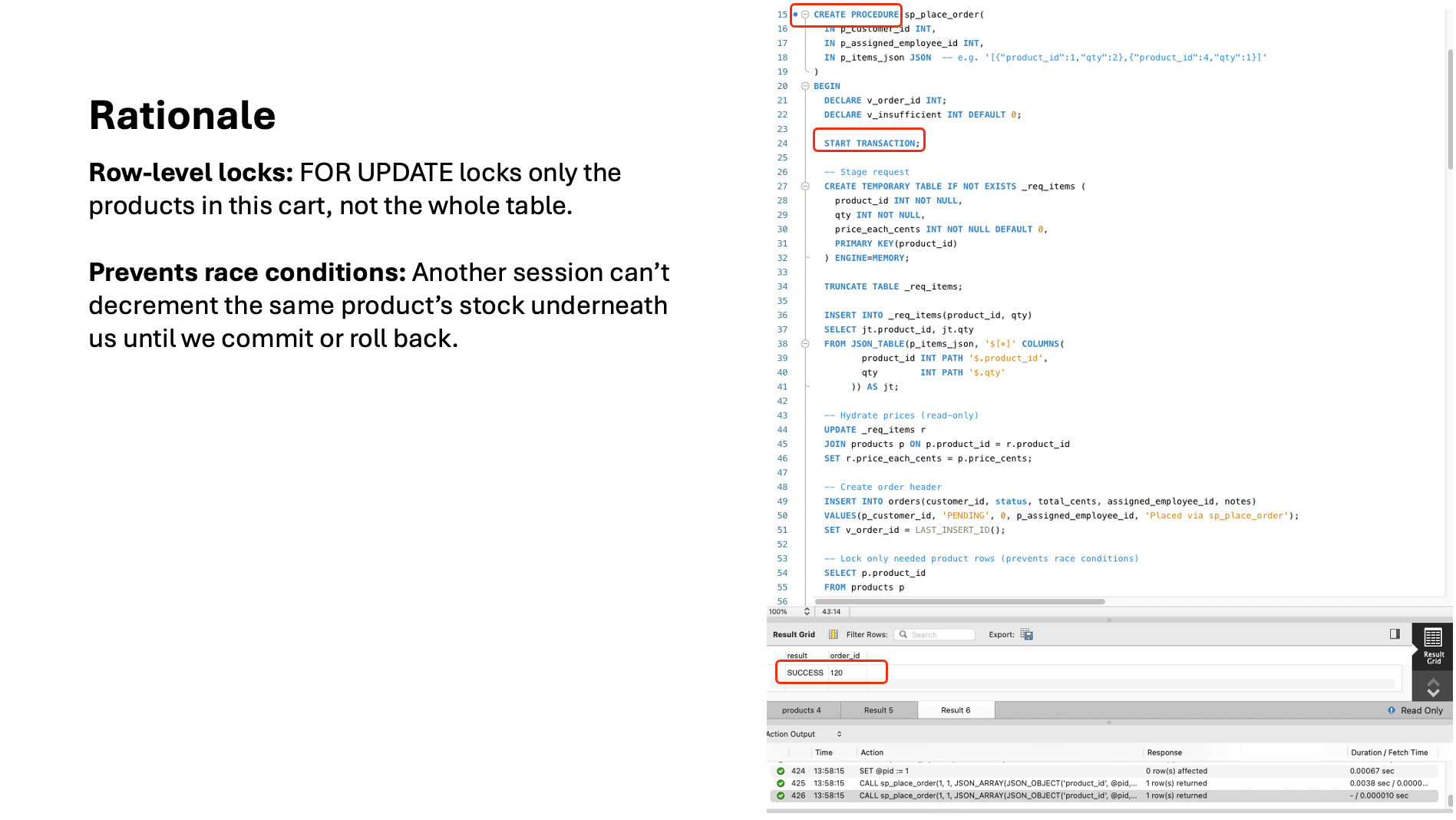
* SHOW CREATE TABLE orders, order\_items, products, payments (PKs/FKs/ENUMs).



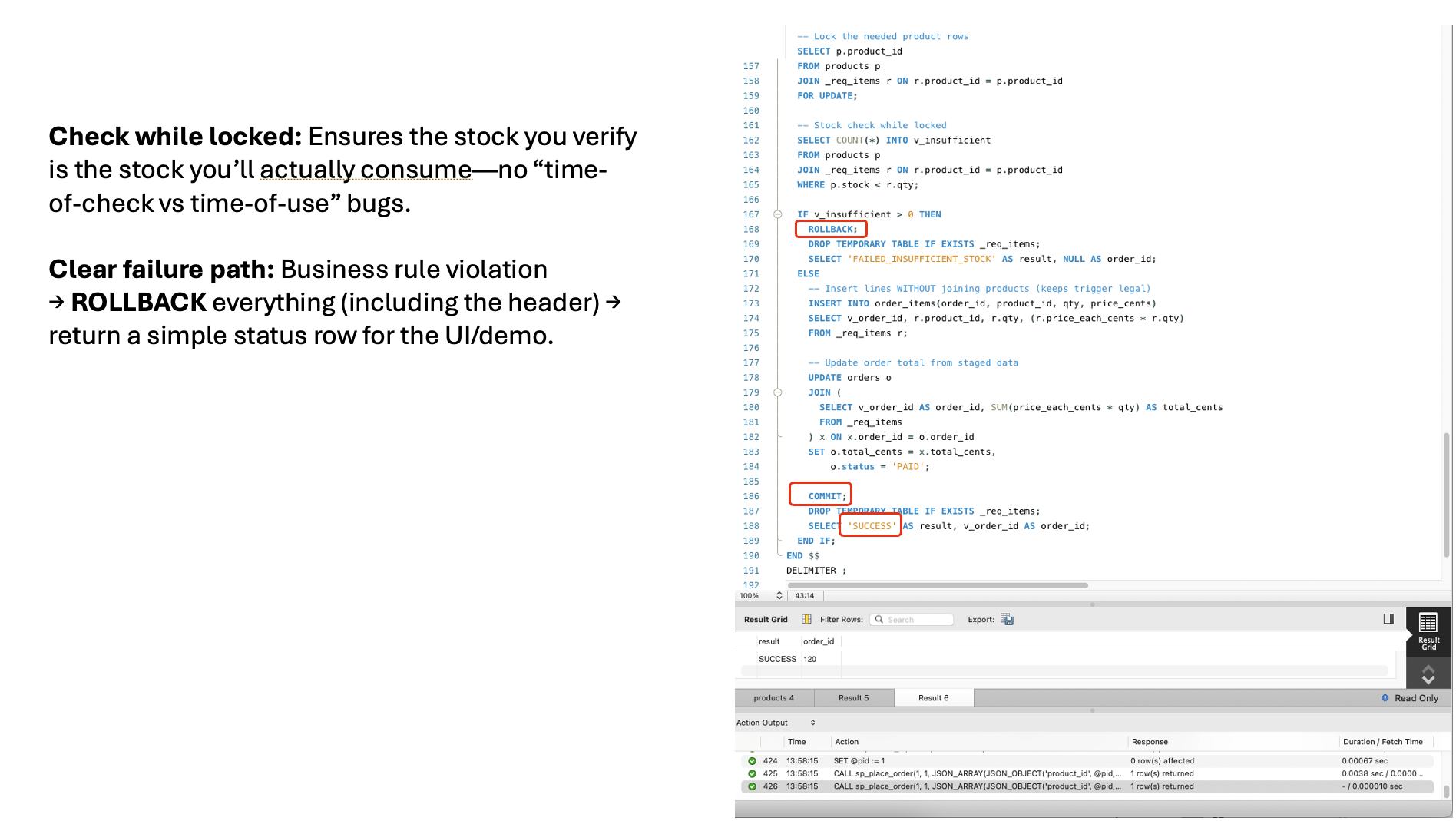
* Row counts across major tables.
* Views: vw\_daily\_sales (grid), vw\_top\_products\_30d (grid).



* MV: CALL refresh (Action Output), SELECT from mv table (grid).
* Procedure: SHOW CREATE PROCEDURE sp\_place\_order.



* Transactions: before stock → SUCCESS → after (stock/movement/order).
* Rollback: FAILED\_INSUFFICIENT\_STOCK result, no changes.
* Concurrency: Tab A lock, Tab B waiting, Tab B success after commit.



* EXPLAIN before and after creating indexes.

