**Use Case: Build Analytic View with Window Functions; Enforce Integrity via Transaction + Trigger**

**Overview**

You are building features for an **e-commerce analytics and integrity system** using MySQL. The system must:

1. Create **analytic views** using **window functions** like ROW\_NUMBER, LAG, SUM OVER.
2. Enforce **data integrity** by:
   * Using **transactions** during updates.
   * Creating **triggers** to log and audit data changes.

**Objectives**

| **Feature** | **Purpose** |
| --- | --- |
| Analytics View | Analyze user order patterns |
| Transactions | Ensure consistent multi-step updates |
| Triggers | Audit price changes in products |

**Schema Design**

**users**

sql

CREATE TABLE users (

user\_id INT PRIMARY KEY,

name VARCHAR(100)

);

**orders**

sql

CREATE TABLE orders (

order\_id INT PRIMARY KEY,

user\_id INT,

order\_date DATE,

FOREIGN KEY (user\_id) REFERENCES users(user\_id)

);

**products**

sql

CREATE TABLE products (

product\_id INT PRIMARY KEY,

name VARCHAR(100),

price DECIMAL(10,2)

);

**order\_items**

sql

CREATE TABLE order\_items (

order\_item\_id INT PRIMARY KEY,

order\_id INT,

product\_id INT,

quantity INT,

FOREIGN KEY (order\_id) REFERENCES orders(order\_id),

FOREIGN KEY (product\_id) REFERENCES products(product\_id)

);

**price\_audit**

sql

CREATE TABLE price\_audit (

audit\_id INT AUTO\_INCREMENT PRIMARY KEY,

product\_id INT,

old\_price DECIMAL(10,2),

new\_price DECIMAL(10,2),

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**Part 1: Create Analytic View with Window Functions**

**Goal**

Analyze customer behavior using SUM OVER, ROW\_NUMBER, LAG.

**MySQL View with Window Functions**

sql

CREATE OR REPLACE VIEW user\_order\_analytics AS

SELECT

u.user\_id,

u.name,

o.order\_id,

o.order\_date,

SUM(oi.quantity \* p.price) AS order\_total,

-- Running total spent by each user

SUM(oi.quantity \* p.price) OVER (

PARTITION BY u.user\_id

ORDER BY o.order\_date

) AS running\_total,

-- Rank orders per user

ROW\_NUMBER() OVER (

PARTITION BY u.user\_id

ORDER BY o.order\_date

) AS order\_rank,

-- Days since previous order

DATEDIFF(

o.order\_date,

LAG(o.order\_date) OVER (

PARTITION BY u.user\_id

ORDER BY o.order\_date

)

) AS days\_since\_last\_order

FROM users u

JOIN orders o ON u.user\_id = o.user\_id

JOIN order\_items oi ON o.order\_id = oi.order\_id

JOIN products p ON p.product\_id = oi.product\_id;

**Part 2: Enforce Integrity with Transaction + Trigger**

**Scenario**

When product prices are updated, we must:

* Log the change in price\_audit
* Ensure update and logging happen atomically

**Trigger to Log Price Changes**

sql

DELIMITER $$

CREATE TRIGGER trg\_price\_update

BEFORE UPDATE ON products

FOR EACH ROW

BEGIN

IF OLD.price <> NEW.price THEN

INSERT INTO price\_audit (product\_id, old\_price, new\_price)

VALUES (OLD.product\_id, OLD.price, NEW.price);

END IF;

END$$

DELIMITER ;

**Transactional Update Example**

sql

START TRANSACTION;

UPDATE products

SET price = 499.99

WHERE product\_id = 101;

-- Trigger will log change automatically

COMMIT;

**Error Handling (Manual Rollback)**

sql

START TRANSACTION;

UPDATE products

SET price = 599.99

WHERE product\_id = 102;

-- Suppose a second update fails here

UPDATE products

SET price = NULL -- illegal price

WHERE product\_id = 103;

-- Error occurs — manually rollback

ROLLBACK;

**Business Benefits**

| **Area** | **Value** |
| --- | --- |
| Analytics View | Real-time insight into customer order trends |
| Trigger Logging | Traceability and compliance for price changes |
| Transaction Safety | Avoids data corruption from partial updates |

**Next Steps**

1. Populate seed data (I can provide this if needed)
2. Execute and validate the view
3. Test update scenarios with success/failure
4. Query user\_order\_analytics to extract KPIs