WALMART

DATA ENGINEERING
INTERVIEW QUESTIONS

CTC-20+ LPA

Question 1: Identify customers who placed multiple orders with the same item within a 15-minute window.

Table Creation and Sample Data

```
-- Create the orders table
CREATE TABLE `orders` (
  `order_id` INT PRIMARY KEY AUTO_INCREMENT,
  `customer_id` INT,
  `item_id` INT,
  `order_timestamp` DATETIME
);


-- Insert sample data
-- Customer 1 places two orders for item 1 within 10 minutes (should be identified)
INSERT INTO `orders` (`customer_id`, `item_id`, `order_timestamp`) VALUES
(101, 1, '2023-01-15 10:00:00'),
(101, 1, '2023-01-15 10:05:00');


-- Customer 2 places two orders for item 2, but more than 15 minutes apart (should not be identified)
```

```
INSERT INTO `orders` (`customer_id`, `item_id`, `order_timestamp`) VALUES
(102, 2, '2023-01-15 11:00:00'),
(102, 2, '2023-01-15 11:30:00');

-- Customer 3 places two orders for different items (should not be identified)
INSERT INTO `orders` (`customer_id`, `item_id`, `order_timestamp`) VALUES
(103, 3, '2023-01-15 12:00:00'),
(103, 4, '2023-01-15 12:05:00');

-- Customer 4 places a single order (should not be identified)
INSERT INTO `orders` (`customer_id`, `item_id`, `order_timestamp`) VALUES
(104, 5, '2023-01-15 13:00:00');
```

Solution Query

This query uses a self-join to compare orders for the same customer and item. It then filters for orders where the time difference is less than or equal to 15 minutes.

```
SELECT DISTINCT
  o1.customer_id
FROM
  orders o1
JOIN
  orders o2 ON o1.customer_id = o2.customer_id
        AND o1.item_id = o2.item_id
        AND o1.order_id != o2.order_id
        -- Check if the time difference is within 15 minutes,
        -- using ABS to handle which order came first.
```

```
WHERE
  TIMESTAMPDIFF(MINUTE, o1.order_timestamp, o2.order_timestamp) BETWEEN 0 AND 15;
```

Question 2: List the most frequently purchased product categories per user in the last 30 days.

Table Creation and Sample Data

```
-- Create the products table
CREATE TABLE `products` (
  `product_id` INT PRIMARY KEY,
  `category` VARCHAR(50)
);

-- Create the orders table
CREATE TABLE `orders_q2` (
  `order_id` INT PRIMARY KEY AUTO_INCREMENT,
  `user_id` INT,
  `product_id` INT,
  `order_timestamp` DATETIME
);

-- Insert sample products
INSERT INTO `products` (`product_id`, `category`) VALUES
(1, 'Electronics'),
(2, 'Groceries'),
```

```
(3, 'Electronics'),
(4, 'Home Goods'),
(5, 'Groceries');

-- Insert sample orders (all within the last 30 days for this example)
-- User 101 purchases Electronics 3 times and Groceries once
INSERT INTO `orders_q2` (`user_id`, `product_id`, `order_timestamp`) VALUES
(101, 1, '2023-08-01 10:00:00'),
(101, 3, '2023-08-05 11:00:00'),
(101, 2, '2023-08-10 12:00:00'),
(101, 3, '2023-08-12 13:00:00');

-- User 102 purchases Home Goods 2 times and Electronics once
INSERT INTO `orders_q2` (`user_id`, `product_id`, `order_timestamp`) VALUES
(102, 4, '2023-08-02 09:00:00'),
(102, 1, '2023-08-03 10:00:00'),
(102, 4, '2023-08-15 14:00:00');

-- User 103 purchases Groceries twice
INSERT INTO `orders_q2` (`user_id`, `product_id`, `order_timestamp`) VALUES
(103, 5, '2023-08-04 15:00:00'),
(103, 5, '2023-08-06 16:00:00');
```

Solution Query

This query uses a Common Table Expression (CTE) to calculate the rank of each category for each user based on purchase frequency. It then selects only the top-ranked category.

```
WITH UserCategoryCounts AS (
    -- First, get the count of purchases for each user and category
    SELECT
        o.user_id,
        p.category,
        COUNT(o.order_id) AS purchase_count
    FROM
        orders_q2 o
    JOIN
        products p ON o.product_id = p.product_id
    -- Filter for orders placed in the last 30 days
    WHERE
        o.order_timestamp >= DATE_SUB(CURDATE(), INTERVAL 30 DAY)
    GROUP BY
        o.user_id, p.category
),
RankedCategories AS (
    -- Use a window function to rank categories per user
    SELECT
        user_id,
        category,
        purchase_count,
        -- RANK() is used to handle ties (multiple categories with the same highest count)
        RANK() OVER (PARTITION BY user_id ORDER BY purchase_count DESC) AS rnk
    FROM
        UserCategoryCounts
```

```
)
-- Select the category with the highest rank for each user
SELECT
  user_id,
  category,
  purchase_count
FROM
  RankedCategories
WHERE
  rnk = 1;
```

Question 3: Find users who upgraded to Walmart+ and placed an express delivery order within 1 hour.

Table Creation and Sample Data

```
-- Create the walmart_plus_upgrades table
CREATE TABLE `walmart_plus_upgrades` (
 `upgrade_id` INT PRIMARY KEY AUTO_INCREMENT,
 `user_id` INT,
 `upgrade_timestamp` DATETIME
);

-- Create the orders table
CREATE TABLE `orders_q3` (
 `order_id` INT PRIMARY KEY AUTO_INCREMENT,
 `user_id` INT,
```

```
  `order_timestamp` DATETIME,
  `delivery_type` VARCHAR(50)
);

-- Insert sample data
-- User 201 upgrades and places an express order 30 minutes later (should be identified)
INSERT INTO `walmart_plus_upgrades` (`user_id`, `upgrade_timestamp`) VALUES
(201, '2023-09-01 09:00:00');

INSERT INTO `orders_q3` (`user_id`, `order_timestamp`, `delivery_type`) VALUES
(201, '2023-09-01 09:30:00', 'express');

-- User 202 upgrades but places a normal delivery order 10 minutes later (should not be identified)
INSERT INTO `walmart_plus_upgrades` (`user_id`, `upgrade_timestamp`) VALUES
(202, '2023-09-02 10:00:00');

INSERT INTO `orders_q3` (`user_id`, `order_timestamp`, `delivery_type`) VALUES
(202, '2023-09-02 10:10:00', 'normal');

-- User 203 upgrades but places an express order more than 1 hour later (should not be identified)
INSERT INTO `walmart_plus_upgrades` (`user_id`, `upgrade_timestamp`) VALUES
(203, '2023-09-03 11:00:00');

INSERT INTO `orders_q3` (`user_id`, `order_timestamp`, `delivery_type`) VALUES
(203, '2023-09-03 12:05:00', 'express');
```

```
-- User 204 places an express order but never upgraded to Walmart+
INSERT INTO `orders_q3` (`user_id`, `order_timestamp`, `delivery_type`) VALUES
(204, '2023-09-04 13:00:00', 'express');
```

Solution Query

This query joins the upgrades table with the orders table and filters for express delivery orders where the timestamp difference is between 0 and 60 minutes.

```
SELECT DISTINCT
  w.user_id
FROM
  walmart_plus_upgrades w
JOIN
  orders_q3 o ON w.user_id = o.user_id
WHERE
  -- Filter for express delivery orders
  o.delivery_type = 'express'
  -- Ensure the order was placed after the upgrade
  AND o.order_timestamp > w.upgrade_timestamp
  -- Check if the time difference is within 60 minutes
  AND TIMESTAMPDIFF(MINUTE, w.upgrade_timestamp, o.order_timestamp) <= 60;
```


Question 4: Generate a daily timeline of online orders and left join with inventory availability logs.

Table Creation and Sample Data

```
-- Create the online_orders table
CREATE TABLE `online_orders` (
  `order_id` INT PRIMARY KEY AUTO_INCREMENT,
  `item_id` INT,
  `order_date` DATE
);

-- Create the inventory_logs table
CREATE TABLE `inventory_logs` (
  `log_id` INT PRIMARY KEY AUTO_INCREMENT,
  `item_id` INT,
  `log_date` DATE,
  `availability_status` VARCHAR(50)
);

-- Insert sample orders
INSERT INTO `online_orders` (`item_id`, `order_date`) VALUES
(101, '2023-09-01'),
(102, '2023-09-01'),
(103, '2023-09-02'),
(104, '2023-09-04'),
(105, '2023-09-04');

-- Insert sample inventory logs. Note there is no log for '2023-09-02'
INSERT INTO `inventory_logs` (`item_id`, `log_date`, `availability_status`) VALUES
(101, '2023-09-01', 'In Stock'),
```

```
(102, '2023-09-01', 'Low Stock'),
(103, '2023-09-03', 'In Stock'),
(104, '2023-09-04', 'In Stock');
```

Solution Query

This query first creates a timeline of all unique order dates. It then LEFT JOINs the inventory_logs table on the date, so that every order date is shown, even if there is no corresponding inventory log entry.

```
SELECT
  o.order_date,
  COUNT(o.order_id) AS total_orders,
  -- Use a CASE statement to show status or 'No Log'
  CASE
    WHEN COUNT(l.log_id) > 0 THEN 'Log Exists'
    ELSE 'No Log'
  END AS inventory_log_status
FROM
  online_orders o
LEFT JOIN
  inventory_logs l ON o.order_date = l.log_date
GROUP BY
  o.order_date
ORDER BY
  o.order_date;
```

Question 5: Rank fulfillment centers based on average order dispatch latency per day.

Table Creation and Sample Data

```
-- Create the orders table

CREATE TABLE `orders_q5` (

  `order_id` INT PRIMARY KEY AUTO_INCREMENT,

  `fulfillment_center_id` INT,

  `order_placed_timestamp` DATETIME,

  `dispatch_timestamp` DATETIME

);


-- Insert sample data

-- Day 1

INSERT INTO `orders_q5` (`fulfillment_center_id`, `order_placed_timestamp`, `dispatch_timestamp`) VALUES

(1, '2023-09-01 08:00:00', '2023-09-01 08:15:00'), -- Latency: 15 mins

(1, '2023-09-01 09:00:00', '2023-09-01 09:10:00'), -- Latency: 10 mins

(2, '2023-09-01 10:00:00', '2023-09-01 10:30:00'), -- Latency: 30 mins

(2, '2023-09-01 11:00:00', '2023-09-01 11:20:00'), -- Latency: 20 mins

(3, '2023-09-01 12:00:00', '2023-09-01 12:05:00'), -- Latency: 5 mins

(3, '2023-09-01 13:00:00', '2023-09-01 13:05:00'); -- Latency: 5 mins

-- Day 2

INSERT INTO `orders_q5` (`fulfillment_center_id`, `order_placed_timestamp`, `dispatch_timestamp`) VALUES

(1, '2023-09-02 08:00:00', '2023-09-02 08:10:00'), -- Latency: 10 mins

(2, '2023-09-02 09:00:00', '2023-09-02 09:10:00'); -- Latency: 10 mins
```

Solution Query

This query uses a Common Table Expression (CTE) to first calculate the average dispatch latency per day for each fulfillment center. It then uses the RANK() window function to rank the centers, with the lowest average latency receiving a rank of 1.

```
WITH DailyLatency AS (
  SELECT
    DATE(order_placed_timestamp) AS order_date,
    fulfillment_center_id,
    -- Calculate average latency in minutes
    AVG(TIMESTAMPDIFF(MINUTE, order_placed_timestamp, dispatch_timestamp)) AS avg_latency_minutes
  FROM
    orders_q5
  GROUP BY
    order_date, fulfillment_center_id
)
SELECT
  order_date,
  fulfillment_center_id,
  avg_latency_minutes,
  -- Rank the centers per day based on average latency (lower is better)
  RANK() OVER (PARTITION BY order_date ORDER BY avg_latency_minutes ASC) AS daily_rank
FROM
  DailyLatency
ORDER BY
```

order_date, daily_rank;

Question 6: Detect users who changed their payment method multiple times in a single checkout session.

Table Creation and Sample Data

```
-- Create the payment_events table
CREATE TABLE `payment_events` (
  `event_id` INT PRIMARY KEY AUTO_INCREMENT,
  `user_id` INT,
  `session_id` VARCHAR(50),
  `payment_method` VARCHAR(50),
  `event_timestamp` DATETIME
);


-- Insert sample data
-- User 301 changes payment method from 'Visa' to 'Mastercard' in session 'S101' (should be identified)
INSERT INTO `payment_events` (`user_id`, `session_id`, `payment_method`, `event_timestamp`) VALUES
(301, 'S101', 'Visa', '2023-09-05 10:00:00'),
(301, 'S101', 'Mastercard', '2023-09-05 10:05:00');


-- User 302 only uses one payment method in session 'S102' (should not be identified)
INSERT INTO `payment_events` (`user_id`, `session_id`, `payment_method`, `event_timestamp`) VALUES
(302, 'S102', 'Mastercard', '2023-09-05 11:00:00');
```

```
-- User 303 uses different methods across different sessions (should not be identified for a single session)
INSERT INTO `payment_events` (`user_id`, `session_id`, `payment_method`, `event_timestamp`) VALUES
(303, 'S103', 'Visa', '2023-09-05 12:00:00'),
(303, 'S104', 'Amex', '2023-09-05 13:00:00');
```

Solution Query

This query groups the payment_events by user and checkout session. It then uses the COUNT(DISTINCT payment_method) aggregation to count the number of unique payment methods used in each session. The HAVING clause filters for sessions where this count is greater than 1.

```
SELECT
  user_id,
  session_id,
  COUNT(DISTINCT payment_method) AS distinct_payment_methods
FROM
  payment_events
GROUP BY
  user_id, session_id
-- Filter for sessions with more than one distinct payment method
HAVING
  COUNT(DISTINCT payment_method) > 1;
```


Question 7: Calculate the average number of items per order per user over the past month.

Table Creation and Sample Data

```
-- Create the orders_q7 table, with a new order item for each item in an order
CREATE TABLE `orders_q7` (
  `order_item_id` INT PRIMARY KEY AUTO_INCREMENT,
  `order_id` INT,
  `user_id` INT,
  `item_id` INT,
  `order_date` DATE
);


-- Insert sample data for the past month (for this example, we'll use dates in the last 30 days)
-- User 101: 3 orders.
-- Order 1: 2 items
INSERT INTO `orders_q7` (`order_id`, `user_id`, `item_id`, `order_date`) VALUES
(1, 101, 1, '2023-10-10'),
(1, 101, 2, '2023-10-10');
-- Order 2: 3 items
INSERT INTO `orders_q7` (`order_id`, `user_id`, `item_id`, `order_date`) VALUES
(2, 101, 3, '2023-10-15'),
(2, 101, 4, '2023-10-15'),
(2, 101, 5, '2023-10-15');
-- Order 3: 1 item
INSERT INTO `orders_q7` (`order_id`, `user_id`, `item_id`, `order_date`) VALUES
(3, 101, 6, '2023-10-20');
```

```
-- User 102: 2 orders.
-- Order 4: 4 items
INSERT INTO `orders_q7` (`order_id`, `user_id`, `item_id`, `order_date`) VALUES
(4, 102, 7, '2023-10-12'),
(4, 102, 8, '2023-10-12'),
(4, 102, 9, '2023-10-12'),
(4, 102, 10, '2023-10-12');
-- Order 5: 2 items
INSERT INTO `orders_q7` (`order_id`, `user_id`, `item_id`, `order_date`) VALUES
(5, 102, 11, '2023-10-25'),
(5, 102, 12, '2023-10-25');

-- User 103: 1 order outside the last month (should be excluded)
INSERT INTO `orders_q7` (`order_id`, `user_id`, `item_id`, `order_date`) VALUES
(6, 103, 13, '2023-09-01');
```

Solution Query

This query uses a subquery to first calculate the number of items for each order within the last month. The main query then takes the result and calculates the average number of items per user.

```
SELECT
  user_id,
  -- Calculate the average of the counts from the subquery
  AVG(items_per_order) AS avg_items_per_order
FROM (
  -- Subquery to find the number of items for each order in the last month
  SELECT
```

```
    user_id,
    order_id,
    COUNT(item_id) AS items_per_order
  FROM
    orders_q7
  WHERE
    order_date >= DATE_SUB(CURDATE(), INTERVAL 1 MONTH)
  GROUP BY
    user_id, order_id
) AS order_counts
GROUP BY
  user_id
ORDER BY
  user_id;
```

Question 8: Identify top 5 products contributing to 80% of total revenue.

Table Creation and Sample Data

```
-- Create the products table
CREATE TABLE `products_q8` (
  `product_id` INT PRIMARY KEY,
  `product_name` VARCHAR(100),
  `price` DECIMAL(10, 2)
);


-- Create the orders table
```

```
CREATE TABLE `orders_q8` (
  `order_id` INT PRIMARY KEY AUTO_INCREMENT,
  `product_id` INT,
  `quantity` INT
);

-- Insert sample products with varying prices
INSERT INTO `products_q8` (`product_id`, `product_name`, `price`) VALUES
(1, 'Laptop', 1200.00),
(2, 'TV', 800.00),
(3, 'Headphones', 150.00),
(4, 'Mouse', 30.00),
(5, 'Keyboard', 75.00),
(6, 'Webcam', 50.00),
(7, 'Monitor', 250.00);

-- Insert sample orders to generate revenue
INSERT INTO `orders_q8` (`product_id`, `quantity`) VALUES
(1, 1),  -- Laptop: 1200
(1, 1),  -- Laptop: 1200
(2, 2),  -- TV: 1600
(3, 5),  -- Headphones: 750
(4, 10), -- Mouse: 300
(5, 3),  -- Keyboard: 225
(6, 4),  -- Webcam: 200
(7, 2),  -- Monitor: 500
```

```
(1, 1),  -- Laptop: 1200
(2, 1);  -- TV: 800
-- Total Revenue = 1200+1200+1600+750+300+225+200+500+1200+800 = 7975
-- 80% of Total Revenue = 6380
```

Solution Query

This query uses a Common Table Expression (CTE) to calculate the revenue per product and the running total of revenue. It then selects the top 5 products whose cumulative revenue is less than or equal to 80% of the total.

```
WITH ProductRevenue AS (
  -- Calculate total revenue for each product
  SELECT
    p.product_id,
    p.product_name,
    SUM(o.quantity * p.price) AS total_revenue
  FROM
    products_q8 p
  JOIN
    orders_q8 o ON p.product_id = o.product_id
  GROUP BY
    p.product_id, p.product_name
),
RankedProductRevenue AS (
  -- Calculate a running total of revenue and rank products by revenue
  SELECT
    product_id,
    product_name,
```

```
    total_revenue,
    -- Use SUM() over a window to create the cumulative sum
    SUM(total_revenue) OVER (ORDER BY total_revenue DESC) AS cumulative_revenue,
    SUM(total_revenue) OVER () AS overall_total_revenue
  FROM
    ProductRevenue
)
-- Select the products that fall within the top 80% of revenue
SELECT
  product_id,
  product_name,
  total_revenue,
  cumulative_revenue
FROM
  RankedProductRevenue
WHERE
  -- The condition to check if the cumulative revenue is within the top 80%
  cumulative_revenue <= overall_total_revenue * 0.8
ORDER BY
  total_revenue DESC
LIMIT 5;
```


Question 9: Compare daily active users across membership types (Guest, Regular, Walmart+).

Table Creation and Sample Data

```
-- Create the users table
```

```
CREATE TABLE `users_q9` (
  `user_id` INT PRIMARY KEY,
  `membership_type` ENUM('Guest', 'Regular', 'Walmart+')
);

-- Create the orders table
CREATE TABLE `orders_q9` (
  `order_id` INT PRIMARY KEY AUTO_INCREMENT,
  `user_id` INT,
  `order_date` DATE
);

-- Insert sample users
INSERT INTO `users_q9` (`user_id`, `membership_type`) VALUES
(101, 'Guest'),
(102, 'Guest'),
(103, 'Regular'),
(104, 'Regular'),
(105, 'Walmart+'),
(106, 'Walmart+');

-- Insert sample orders
-- Day 1
INSERT INTO `orders_q9` (`user_id`, `order_date`) VALUES
(101, '2023-11-01'), -- Guest
(102, '2023-11-01'), -- Guest
```

```
(103, '2023-11-01'); -- Regular

-- Day 2

INSERT INTO `orders_q9` (`user_id`, `order_date`) VALUES

(101, '2023-11-02'), -- Guest

(104, '2023-11-02'), -- Regular

(105, '2023-11-02'), -- Walmart+

(106, '2023-11-02'); -- Walmart+

-- Day 3

INSERT INTO `orders_q9` (`user_id`, `order_date`) VALUES

(103, '2023-11-03'), -- Regular

(105, '2023-11-03'); -- Walmart+
```

Solution Query

This query joins the orders_q9 and users_q9 tables. It then pivots the data using SUM and CASE statements to count the distinct daily active users for each membership type, presenting the results side-by-side for easy comparison.

```
SELECT

  o.order_date,

  SUM(CASE WHEN u.membership_type = 'Guest' THEN 1 ELSE 0 END) AS
guest_daily_active_users,

  SUM(CASE WHEN u.membership_type = 'Regular' THEN 1 ELSE 0 END) AS
regular_daily_active_users,

  SUM(CASE WHEN u.membership_type = 'Walmart+' THEN 1 ELSE 0 END) AS
walmart_plus_daily_active_users

FROM

  orders_q9 o

JOIN
```

```
  users_q9 u ON o.user_id = u.user_id
GROUP BY
  o.order_date
ORDER BY
  o.order_date;
```

Question 10: Find users whose longest shopping session occurred outside store operating hours.

Table Creation and Sample Data

```
-- Create the user_sessions table
CREATE TABLE `user_sessions` (
 `session_id` INT PRIMARY KEY AUTO_INCREMENT,
 `user_id` INT,
 `session_start` DATETIME,
 `session_end` DATETIME
);

-- Insert sample data
-- User 101 has their longest session outside operating hours (should be identified)
INSERT INTO `user_sessions` (`user_id`, `session_start`, `session_end`) VALUES
(101, '2023-12-01 09:00:00', '2023-12-01 09:30:00'), -- 30 mins, in hours
(101, '2023-12-01 22:30:00', '2023-12-02 00:00:00'); -- 90 mins, out of hours

-- User 102 has a long session, but it is within operating hours (should not be identified)
INSERT INTO `user_sessions` (`user_id`, `session_start`, `session_end`) VALUES
```

```
(102, '2023-12-02 10:00:00', '2023-12-02 11:30:00'), -- 90 mins, in hours
(102, '2023-12-02 18:00:00', '2023-12-02 18:15:00'); -- 15 mins, in hours

-- User 103's only session is a short one outside operating hours (should be identified)
INSERT INTO `user_sessions` (`user_id`, `session_start`, `session_end`) VALUES
(103, '2023-12-03 01:00:00', '2023-12-03 01:10:00'); -- 10 mins, out of hours
```

Solution Query

This query uses a Common Table Expression (CTE) to find the longest session for each user. It then joins this result back to the user_sessions table to check if that specific longest session occurred outside of the defined operating hours.

```
WITH UserLongestSession AS (
  -- Calculate the duration of each session and find the max duration for each user
  SELECT
    user_id,
    MAX(TIMESTAMPDIFF(MINUTE, session_start, session_end)) AS max_duration_minutes
  FROM
    user_sessions
  GROUP BY
    user_id
)
SELECT DISTINCT
  s.user_id
FROM
  user_sessions s
JOIN
  UserLongestSession u ON s.user_id = u.user_id
```

```
WHERE
  -- The session must be the user's longest
  TIMESTAMPDIFF(MINUTE, s.session_start, s.session_end) = u.max_duration_minutes
  -- Check if the session's start OR end time is outside the 8 AM to 10 PM window
  AND (
    TIME(s.session_start) NOT BETWEEN '08:00:00' AND '22:00:00'
    OR TIME(s.session_end) NOT BETWEEN '08:00:00' AND '22:00:00'
  );
```

Question 11: Count how many users exceeded their return limit more than 3 times in a week.

Table Creation and Sample Data

```
-- Create the product_returns table
CREATE TABLE `product_returns` (
 `return_id` INT PRIMARY KEY AUTO_INCREMENT,
 `user_id` INT,
 `return_date` DATE
);

-- Insert sample data
-- User 201 returns 4 items in a single week (should be counted)
INSERT INTO `product_returns` (`user_id`, `return_date`) VALUES
(201, '2023-12-01'),
(201, '2023-12-02'),
(201, '2023-12-03'),
(201, '2023-12-04');
```

```
-- User 202 returns 3 items in a single week (does not exceed the limit)
INSERT INTO `product_returns` (`user_id`, `return_date`) VALUES
(202, '2023-12-01'),
(202, '2023-12-02'),
(202, '2023-12-03');

-- User 203 returns 4 items, but spread across two different weeks (does not exceed the limit in any single week)
INSERT INTO `product_returns` (`user_id`, `return_date`) VALUES
(203, '2023-12-01'),
(203, '2023-12-02'),
(203, '2023-12-10'),
(203, '2023-12-11');
```

Solution Query

This query groups returns by user and week, counting the total returns. The HAVING clause then filters for groups where the count is greater than 3, and the final COUNT(DISTINCT user_id) provides the number of users who met this criteria.

```
SELECT
  COUNT(DISTINCT user_id) AS users_who_exceeded_return_limit
FROM (
  -- Subquery to find users who made more than 3 returns in a single week
  SELECT
    user_id,
    YEARWEEK(return_date) AS return_week,
    COUNT(return_id) AS weekly_returns_count
```

```
  FROM
    product_returns
  GROUP BY
    user_id, return_week
  HAVING
    COUNT(return_id) > 3
) AS weekly_return_counts;
```

Question 12: Analyze drop-offs between account creation and first successful purchase.

Table Creation and Sample Data

```
-- Create the users table
CREATE TABLE `users_q12` (
  `user_id` INT PRIMARY KEY,
  `account_creation_date` DATE
);

-- Create the orders table
CREATE TABLE `orders_q12` (
  `order_id` INT PRIMARY KEY AUTO_INCREMENT,
  `user_id` INT,
  `order_date` DATE,
  `status` ENUM('SUCCESS', 'FAILED')
);

-- Insert sample users
```

```
INSERT INTO `users_q12` (`user_id`, `account_creation_date`) VALUES
(101, '2023-12-01'),
(102, '2023-12-05'),
(103, '2023-12-10');

-- Insert sample orders
-- User 101: Creates account and makes a successful purchase later (not a drop-off)
INSERT INTO `orders_q12` (`user_id`, `order_date`, `status`) VALUES
(101, '2023-12-03', 'SUCCESS'),
(101, '2023-12-04', 'SUCCESS');

-- User 102: Creates account, but has only failed orders (is a drop-off)
INSERT INTO `orders_q12` (`user_id`, `order_date`, `status`) VALUES
(102, '2023-12-06', 'FAILED');

-- User 103: Creates account, but has no orders (is a drop-off)
-- No orders inserted for this user
```

Solution Query

This query uses a LEFT JOIN to link all users to their orders. The MIN() aggregate function finds the date of the first successful purchase. By using LEFT JOIN, we ensure that users with no successful purchases are included, with a NULL value for the purchase date, allowing us to identify the drop-offs.

```
SELECT
  u.user_id,
  u.account_creation_date,
  MIN(o.order_date) AS first_successful_purchase_date,
```

```
    -- Calculate the number of days until the first purchase
    DATEDIFF(MIN(o.order_date), u.account_creation_date) AS days_to_first_purchase
FROM
    users_q12 u
LEFT JOIN
    orders_q12 o ON u.user_id = o.user_id AND o.status = 'SUCCESS'
GROUP BY
    u.user_id, u.account_creation_date
ORDER BY
    u.user_id;
```

Question 13: Track users who encountered 2 or more consecutive payment failures in a single day.

Table Creation and Sample Data

```
-- Create the payment_transactions table
CREATE TABLE `payment_transactions` (
  `transaction_id` INT PRIMARY KEY AUTO_INCREMENT,
  `user_id` INT,
  `transaction_timestamp` DATETIME,
  `status` ENUM('SUCCESS', 'FAILED')
);

-- Insert sample data
-- User 101: 2 consecutive failures on the same day (should be identified)
```

```
INSERT INTO `payment_transactions` (`user_id`, `transaction_timestamp`, `status`) VALUES
(101, '2023-12-05 10:00:00', 'SUCCESS'),
(101, '2023-12-05 10:05:00', 'FAILED'),
(101, '2023-12-05 10:08:00', 'FAILED'),
(101, '2023-12-05 10:15:00', 'SUCCESS');

-- User 102: Failures are not consecutive on the same day (should not be identified)
INSERT INTO `payment_transactions` (`user_id`, `transaction_timestamp`, `status`) VALUES
(102, '2023-12-05 11:00:00', 'FAILED'),
(102, '2023-12-05 11:05:00', 'SUCCESS'),
(102, '2023-12-05 11:10:00', 'FAILED');

-- User 103: Two failures, but on different days (should not be identified)
INSERT INTO `payment_transactions` (`user_id`, `transaction_timestamp`, `status`) VALUES
(103, '2023-12-06 09:00:00', 'FAILED'),
(103, '2023-12-07 09:00:00', 'FAILED');
```

Solution Query

This query uses a Common Table Expression (CTE) and the LAG() window function to look at the previous transaction's status. It then filters for users who had a 'FAILED' transaction immediately following another 'FAILED' one on the same day.

```
WITH RankedTransactions AS (
  -- Assign a row number to each transaction for each user per day
  SELECT
```

```
    user_id,
    transaction_timestamp,
    status,
    LAG(status, 1, 'N/A') OVER (PARTITION BY user_id, DATE(transaction_timestamp)
ORDER BY transaction_timestamp) AS previous_status
  FROM
    payment_transactions
)
SELECT DISTINCT
  user_id
FROM
  RankedTransactions
WHERE
  status = 'FAILED'
  AND previous_status = 'FAILED';
```

Question 14: Check if a user made at least one order every week over the past 6 weeks.

Table Creation and Sample Data

```
-- Create the orders table
CREATE TABLE `orders_q14` (
 `order_id` INT PRIMARY KEY AUTO_INCREMENT,
 `user_id` INT,
 `order_date` DATE
);
```

```
-- Insert sample data (current date assumed to be '2023-12-05')
-- User 201: Orders in each of the last 6 weeks (should be identified)
INSERT INTO `orders_q14` (`user_id`, `order_date`) VALUES
(201, '2023-11-29'), -- Week 1
(201, '2023-11-22'), -- Week 2
(201, '2023-11-15'), -- Week 3
(201, '2023-11-08'), -- Week 4
(201, '2023-11-01'), -- Week 5
(201, '2023-10-25'); -- Week 6

-- User 202: Misses a week (should not be identified)
INSERT INTO `orders_q14` (`user_id`, `order_date`) VALUES
(202, '2023-11-29'), -- Week 1
(202, '2023-11-22'), -- Week 2
-- Missing order for Week 3
(202, '2023-11-08'), -- Week 4
(202, '2023-11-01'), -- Week 5
(202, '2023-10-25'); -- Week 6
```

Solution Query

This query uses YEARWEEK() to group orders by week. It then counts the number of distinct weeks each user had an order and uses the HAVING clause to filter for users who have a count of exactly 6.

```
SELECT
  user_id
FROM
```

```
  orders_q14
WHERE
  -- Filter orders to only the last 6 weeks
  order_date >= DATE_SUB(CURDATE(), INTERVAL 6 WEEK)
GROUP BY
  user_id
-- Check if the number of distinct weeks with an order is 6
HAVING
  COUNT(DISTINCT YEARWEEK(order_date)) = 6;
```

Question 15: Identify inactive products that haven't been purchased in over 60 days.

Table Creation and Sample Data

```
-- Create the products table
CREATE TABLE `products` (
 `product_id` INT PRIMARY KEY,
 `product_name` VARCHAR(100)
);

-- Create the orders table
CREATE TABLE `orders_q15` (
 `order_id` INT PRIMARY KEY AUTO_INCREMENT,
 `product_id` INT,
 `order_date` DATE
);
```

```
-- Insert sample products
INSERT INTO `products` (`product_id`, `product_name`) VALUES
(101, 'Active Product A'),
(102, 'Inactive Product B'),
(103, 'Never Purchased Product C');

-- Insert sample orders (current date assumed to be '2023-12-05')
-- Product 101: Purchased recently (should not be identified)
INSERT INTO `orders_q15` (`product_id`, `order_date`) VALUES
(101, '2023-11-10');

-- Product 102: Purchased over 60 days ago (should be identified)
INSERT INTO `orders_q15` (`product_id`, `order_date`) VALUES
(102, '2023-09-01');

-- Product 103: Never purchased (should be identified via a NULL max date)
-- No orders inserted for product 103
```

Solution Query

This query uses a LEFT JOIN from products to orders_q15 to include all products, even those with no orders. It then groups by product to find the MAX(order_date). The HAVING clause filters for products whose last purchase date is either older than 60 days or is NULL (meaning no purchases were ever made).

```
SELECT
  p.product_id,
  p.product_name,
  MAX(o.order_date) AS last_purchase_date
```

```
FROM
  products p
LEFT JOIN
  orders_q15 o ON p.product_id = o.product_id
GROUP BY
  p.product_id, p.product_name
HAVING
  -- Check if the product has no purchases, or if the last purchase was over 60 days ago
  MAX(o.order_date) IS NULL OR DATEDIFF(CURDATE(), MAX(o.order_date)) > 60
ORDER BY
  p.product_id;
```

Question 16: Create sessionized shopping logs with a session break at 20 minutes of inactivity.

Table Creation and Sample Data

We'll use a user_activity table to log user events with precise timestamps. The key to this problem is using a window function to compare the timestamp of the current event to the previous one for the same user.

```
-- Drop table if it already exists
DROP TABLE IF EXISTS `user_activity`;


-- Create the user_activity table
CREATE TABLE `user_activity` (
 `event_id` INT PRIMARY KEY AUTO_INCREMENT,
 `user_id` INT,
```

```
  `event_timestamp` DATETIME,
  `event_type` VARCHAR(50)
);

-- Insert sample data
-- User 101: 3 events in a single session, all within a 20-minute window
INSERT INTO `user_activity` (`user_id`, `event_timestamp`, `event_type`) VALUES
(101, '2023-12-05 10:00:00', 'page_view'),
(101, '2023-12-05 10:05:00', 'add_to_cart'),
(101, '2023-12-05 10:15:00', 'checkout');

-- User 102: Two sessions. The second event is more than 20 minutes after the first.
INSERT INTO `user_activity` (`user_id`, `event_timestamp`, `event_type`) VALUES
(102, '2023-12-05 11:00:00', 'page_view'),
(102, '2023-12-05 11:25:00', 'add_to_cart'); -- Inactivity > 20 mins, new session starts

-- User 103: A single session with no gaps
INSERT INTO `user_activity` (`user_id`, `event_timestamp`, `event_type`) VALUES
(103, '2023-12-05 12:00:00', 'page_view'),
(103, '2023-12-05 12:02:00', 'add_to_cart');
```

Solution Query

This query uses a Common Table Expression (CTE) to create a session_start_flag. It then uses a window function to create a running sum of these flags, effectively assigning a unique session ID to each group of consecutive events.

```
WITH SessionStarts AS (
    -- Identify the start of a new session based on inactivity
```

```
  SELECT
    user_id,
    event_timestamp,
    event_type,
    -- Use LAG() to get the previous event's timestamp
    TIMESTAMPDIFF(MINUTE, LAG(event_timestamp) OVER (PARTITION BY user_id ORDER BY event_timestamp), event_timestamp) AS minutes_since_last_event,
    CASE
      -- If it's the first event for a user or the gap is > 20 mins, it's a new session
      WHEN LAG(event_timestamp) OVER (PARTITION BY user_id ORDER BY event_timestamp) IS NULL OR TIMESTAMPDIFF(MINUTE, LAG(event_timestamp) OVER (PARTITION BY user_id ORDER BY event_timestamp), event_timestamp) > 20
      THEN 1
      ELSE 0
    END AS session_start_flag
  FROM
    user_activity
),
SessionIDs AS (
  -- Assign a unique session ID to each group of events
  SELECT
    user_id,
    event_timestamp,
    event_type,
    SUM(session_start_flag) OVER (PARTITION BY user_id ORDER BY event_timestamp) AS session_id
  FROM
```

```
    SessionStarts
)
-- Final output: a table with sessionized logs
SELECT
  user_id,
  session_id,
  event_timestamp,
  event_type
FROM
  SessionIDs
ORDER BY
  user_id, session_id, event_timestamp;
```

Question 17: Find users whose monthly order volume increased continuously for 3 months.

Table Creation and Sample Data

```
-- Create the orders_q17 table
CREATE TABLE `orders_q17` (
 `order_id` INT PRIMARY KEY AUTO_INCREMENT,
 `user_id` INT,
 `order_date` DATE
);

-- Insert sample data
```

```
-- User 201: Monthly order volume increases continuously for 3 months (should be identified)
-- Month 1 (Oct): 2 orders
INSERT INTO `orders_q17` (`user_id`, `order_date`) VALUES
(201, '2023-10-10'),
(201, '2023-10-20');
-- Month 2 (Nov): 3 orders
INSERT INTO `orders_q17` (`user_id`, `order_date`) VALUES
(201, '2023-11-05'),
(201, '2023-11-15'),
(201, '2023-11-25');
-- Month 3 (Dec): 4 orders
INSERT INTO `orders_q17` (`user_id`, `order_date`) VALUES
(201, '2023-12-05'),
(201, '2023-12-15'),
(201, '2023-12-25'),
(201, '2023-12-28');

-- User 202: Monthly order volume decreases (should not be identified)
-- Month 1 (Oct): 4 orders
INSERT INTO `orders_q17` (`user_id`, `order_date`) VALUES
(202, '2023-10-01'), (202, '2023-10-10'), (202, '2023-10-20'), (202, '2023-10-30');
-- Month 2 (Nov): 3 orders
INSERT INTO `orders_q17` (`user_id`, `order_date`) VALUES
(202, '2023-11-05'), (202, '2023-11-15'), (202, '2023-11-25');
-- Month 3 (Dec): 2 orders
```

```
INSERT INTO `orders_q17` (`user_id`, `order_date`) VALUES
(202, '2023-12-05'), (202, '2023-12-15');
```

Solution Query

This query uses a CTE to count monthly orders for each user. It then uses the LAG() window function twice to check if the current month's count is greater than the previous month's and the previous month's count is greater than the month before that.

```
WITH MonthlyOrders AS (
  -- Count orders per user per month
  SELECT
    user_id,
    DATE_FORMAT(order_date, '%Y-%m-01') AS order_month,
    COUNT(order_id) AS monthly_order_count
  FROM
    orders_q17
  GROUP BY
    user_id, order_month
),
RankedMonthlyOrders AS (
  -- Use LAG to get the counts for the previous two months
  SELECT
    user_id,
    order_month,
    monthly_order_count,
    LAG(monthly_order_count, 1) OVER (PARTITION BY user_id ORDER BY order_month) AS prev_month_count,
```

```
    LAG(monthly_order_count, 2) OVER (PARTITION BY user_id ORDER BY order_month) AS prev_2_month_count
  FROM
    MonthlyOrders
)
-- Select users where the monthly order count increased continuously
SELECT DISTINCT
  user_id
FROM
  RankedMonthlyOrders
WHERE
  monthly_order_count > prev_month_count
  AND prev_month_count > prev_2_month_count;
```

Question 18: Extract product categories frequently bought together in a single session.

Table Creation and Sample Data

```
-- Create the products table
CREATE TABLE `products_q18` (
 `product_id` INT PRIMARY KEY,
 `category` VARCHAR(50)
);

-- Create the session_items table
CREATE TABLE `session_items` (
```

```
  `session_id` INT,
  `product_id` INT
);

-- Insert sample products
INSERT INTO `products_q18` (`product_id`, `category`) VALUES
(1, 'Electronics'),
(2, 'Accessories'),
(3, 'Electronics'),
(4, 'Groceries'),
(5, 'Groceries'),
(6, 'Home Goods');

-- Insert sample sessions
-- Session 101: Electronics and Accessories bought together
INSERT INTO `session_items` (`session_id`, `product_id`) VALUES
(101, 1), -- Electronics
(101, 2); -- Accessories
-- Session 102: Groceries and Groceries (but different products) bought together
INSERT INTO `session_items` (`session_id`, `product_id`) VALUES
(102, 4), -- Groceries
(102, 5); -- Groceries
-- Session 103: Home Goods and Electronics bought together
INSERT INTO `session_items` (`session_id`, `product_id`) VALUES
(103, 6), -- Home Goods
(103, 3); -- Electronics
```

```
-- Session 104: Electronics and Accessories bought together again
INSERT INTO `session_items` (`session_id`, `product_id`) VALUES
(104, 3), -- Electronics
(104, 2); -- Accessories
```

Solution Query

This query uses a self-join to find pairs of categories purchased within the same session. It uses JOINs to get the category names for each product. The WHERE clause ensures we don't count a category with itself or double-count pairs (e.g., 'A, B' and 'B, A').

```
SELECT
  p1.category AS category_1,
  p2.category AS category_2,
  COUNT(t1.session_id) AS co_purchase_count
FROM
  session_items t1
JOIN
  session_items t2 ON t1.session_id = t2.session_id
-- Join to get the category for the first item
JOIN
  products_q18 p1 ON t1.product_id = p1.product_id
-- Join to get the category for the second item
JOIN
  products_q18 p2 ON t2.product_id = p2.product_id
WHERE
  -- Ensure we're not pairing the same product with itself
  t1.product_id < t2.product_id
  -- Ensure we're not pairing a category with itself
```

```
    AND p1.category != p2.category
GROUP BY
    p1.category, p2.category
ORDER BY
    co_purchase_count DESC;
```

Question 19: Estimate user lifetime spend and number of orders based on pricing tiers.

For this problem, "pricing tiers" will be interpreted as the individual prices of products, which contribute to the total spend. We will calculate the total spend and total number of orders for each user across all their purchases.

Table Creation and Sample Data

```
-- Create the users table
CREATE TABLE `users_q19` (
  `user_id` INT PRIMARY KEY,
  `username` VARCHAR(50)
);

-- Create the products table with prices
CREATE TABLE `products_q19` (
  `product_id` INT PRIMARY KEY,
  `product_name` VARCHAR(100),
  `price` DECIMAL(10, 2)
);
```

```
-- Create the orders table
CREATE TABLE `orders_q19` (
  `order_id` INT PRIMARY KEY AUTO_INCREMENT,
  `user_id` INT,
  `order_date` DATE
);

-- Create the order_items table to link orders to products and quantities
CREATE TABLE `order_items` (
  `order_item_id` INT PRIMARY KEY AUTO_INCREMENT,
  `order_id` INT,
  `product_id` INT,
  `quantity` INT
);

-- Insert sample users
INSERT INTO `users_q19` (`user_id`, `username`) VALUES
(1, 'Alice'),
(2, 'Bob'),
(3, 'Charlie');

-- Insert sample products with different prices (representing tiers)
INSERT INTO `products_q19` (`product_id`, `product_name`, `price`) VALUES
(101, 'Basic T-Shirt', 15.00),
(102, 'Premium Jeans', 50.00),
(103, 'Luxury Watch', 500.00),
```

```
(104, 'Budget Headphones', 25.00),
(105, 'High-End Laptop', 1200.00);

-- Insert sample orders
-- Alice's orders
INSERT INTO `orders_q19` (`order_id`, `user_id`, `order_date`) VALUES
(1001, 1, '2023-01-10'),
(1002, 1, '2023-02-15');
-- Bob's orders
INSERT INTO `orders_q19` (`order_id`, `user_id`, `order_date`) VALUES
(1003, 2, '2023-03-01');
-- Charlie (no orders yet)

-- Insert order items
-- Order 1001 (Alice): T-Shirt (15*2=30), Jeans (50*1=50) -> Total: 80
INSERT INTO `order_items` (`order_id`, `product_id`, `quantity`) VALUES
(1001, 101, 2),
(1001, 102, 1);
-- Order 1002 (Alice): Watch (500*1=500), Laptop (1200*1=1200) -> Total: 1700
INSERT INTO `order_items` (`order_id`, `product_id`, `quantity`) VALUES
(1002, 103, 1),
(1002, 105, 1);
-- Order 1003 (Bob): Headphones (25*3=75) -> Total: 75
INSERT INTO `order_items` (`order_id`, `product_id`, `quantity`) VALUES
(1003, 104, 3);
```

Solution Query

This query joins users, orders, order items, and products to calculate the total spend and total number of orders for each user.

```
SELECT
  u.user_id,
  u.username,
  -- Count distinct orders for each user
  COUNT(DISTINCT o.order_id) AS total_orders_lifetime,
  -- Calculate total spend by summing (quantity * price) for all items
  SUM(oi.quantity * p.price) AS total_spend_lifetime
FROM
  users_q19 u
LEFT JOIN
  orders_q19 o ON u.user_id = o.user_id
LEFT JOIN
  order_items oi ON o.order_id = oi.order_id
LEFT JOIN
  products_q19 p ON oi.product_id = p.product_id
GROUP BY
  u.user_id, u.username
ORDER BY
  u.user_id;
```


Question 20: Identify regions where average delivery time is consistently higher than national average.

For "consistently higher," we'll focus on the current average delivery time for simplicity, comparing each region's average to the overall national average. If "consistently" implies a trend over time, it would require more complex time-series analysis.

Table Creation and Sample Data

```
-- Create the orders_q20 table
CREATE TABLE `orders_q20` (
  `order_id` INT PRIMARY KEY AUTO_INCREMENT,
  `region` VARCHAR(50),
  `order_timestamp` DATETIME,
  `delivery_timestamp` DATETIME
);

-- Insert sample data
-- Region 'East': Generally faster delivery
INSERT INTO `orders_q20` (`region`, `order_timestamp`, `delivery_timestamp`) VALUES
('East', '2023-04-01 10:00:00', '2023-04-01 10:30:00'), -- 30 mins
('East', '2023-04-01 11:00:00', '2023-04-01 11:40:00'); -- 40 mins
-- Average East: (30+40)/2 = 35 mins

-- Region 'West': Generally slower delivery
INSERT INTO `orders_q20` (`region`, `order_timestamp`, `delivery_timestamp`) VALUES
('West', '2023-04-01 12:00:00', '2023-04-01 13:00:00'), -- 60 mins
('West', '2023-04-01 13:00:00', '2023-04-01 14:15:00'); -- 75 mins
-- Average West: (60+75)/2 = 67.5 mins

-- Region 'South': Mixed delivery times
```

```
INSERT INTO `orders_q20` (`region`, `order_timestamp`, `delivery_timestamp`) VALUES
('South', '2023-04-01 14:00:00', '2023-04-01 14:45:00'), -- 45 mins
('South', '2023-04-01 15:00:00', '2023-04-01 16:00:00'); -- 60 mins
-- Average South: (45+60)/2 = 52.5 mins


-- Overall National Average: (30+40+60+75+45+60) / 6 = 310 / 6 = 51.67 mins
```

Solution Query

This query uses a Common Table Expression (CTE) to calculate the average delivery time for each region. It then joins this with a subquery that calculates the overall national average delivery time, filtering for regions whose average is higher.

```
WITH RegionalAvgDelivery AS (
  -- Calculate average delivery time per region in minutes
  SELECT
    region,
    AVG(TIMESTAMPDIFF(MINUTE, order_timestamp, delivery_timestamp)) AS
avg_delivery_minutes
  FROM
    orders_q20
  GROUP BY
    region
),
NationalAvgDelivery AS (
  -- Calculate the national average delivery time
  SELECT
    AVG(TIMESTAMPDIFF(MINUTE, order_timestamp, delivery_timestamp)) AS
national_avg_minutes
```

```
    FROM
        orders_q20
)
SELECT
    r.region,
    r.avg_delivery_minutes,
    n.national_avg_minutes
FROM
    RegionalAvgDelivery r,
    NationalAvgDelivery n
WHERE
    r.avg_delivery_minutes > n.national_avg_minutes
ORDER BY
    r.avg_delivery_minutes DESC;
```