

# Phase 7: Practice Challenges

---

This section contains structured PostgreSQL queries (pgAdmin-ready) for the practice challenges. All MySQL-specific syntax has been removed (e.g., YEAR(), DATE\_FORMAT(), SHOW INDEX).

## Challenge 1: Inventory Management

### 1) Identify products that need reordering across all warehouses

```
SELECT
    p.product_id,
    p.product_name,
    p.reorder_level,
    COALESCE(SUM(i.quantity), 0) AS total_stock_all_warehouses,
    (p.reorder_level - COALESCE(SUM(i.quantity), 0)) AS units_short
FROM products p
LEFT JOIN inventory i ON i.product_id = p.product_id
WHERE p.is_active = TRUE
GROUP BY p.product_id, p.product_name, p.reorder_level
HAVING COALESCE(SUM(i.quantity), 0) < p.reorder_level
ORDER BY units_short DESC;
```

### 2) Calculate the cost of restocking all products below reorder level

```
SELECT
    ROUND(SUM(restock_cost), 2) AS total_restock_cost
FROM (
    SELECT
        GREATEST(p.reorder_level - COALESCE(SUM(i.quantity), 0), 0) *
        p.cost_price AS restock_cost
    FROM products p
    LEFT JOIN inventory i ON i.product_id = p.product_id
    WHERE p.is_active = TRUE
    GROUP BY p.product_id, p.reorder_level, p.cost_price
    HAVING COALESCE(SUM(i.quantity), 0) < p.reorder_level
) x;
```

### 3) Recommend warehouse transfers to balance inventory

```
WITH per_wh AS (
    SELECT i.product_id, i.warehouse_id, i.quantity
    FROM inventory i
),
avg_qty AS (
    SELECT product_id, AVG(quantity)::numeric(10,2) AS avg_per_warehouse
```

```

        FROM per_wh
        GROUP BY product_id
    ),
diff AS (
    SELECT
        p.product_id,
        p.product_name,
        w.warehouse_id,
        w.warehouse_name,
        pw.quantity,
        a.avg_per_warehouse,
        (pw.quantity - a.avg_per_warehouse) AS diff_from_avg
    FROM per_wh pw
    JOIN avg_qty a ON a.product_id = pw.product_id
    JOIN products p ON p.product_id = pw.product_id
    JOIN warehouses w ON w.warehouse_id = pw.warehouse_id
)
SELECT
    d1.product_id,
    d1.product_name,
    d1.warehouse_name AS from_warehouse,
    d2.warehouse_name AS to_warehouse,
    FLOOR(LEAST(d1.diff_from_avg, -d2.diff_from_avg))::int AS
    suggested_transfer_units
FROM diff d1
JOIN diff d2
    ON d1.product_id = d2.product_id
    AND d1.diff_from_avg > 0      -- has extra
    AND d2.diff_from_avg < 0      -- needs stock
WHERE FLOOR(LEAST(d1.diff_from_avg, -d2.diff_from_avg))::int > 0
ORDER BY suggested_transfer_units DESC, d1.product_id;

```

## Challenge 2: Customer Analytics

### 1) Create a customer cohort analysis by registration month

```

SELECT
    to_char(date_trunc('month', registration_date), 'YYYY-MM') AS
    cohort_month,
    COUNT(*) AS customers_registered
FROM customers
GROUP BY date_trunc('month', registration_date)
ORDER BY cohort_month;

```

### 2) Calculate customer churn rate (no orders in the last 90 days)

```

WITH last_order AS (
    SELECT

```

```

        c.customer_id,
        MAX(o.order_date) AS last_order_date
    FROM customers c
    LEFT JOIN orders o
        ON o.customer_id = c.customer_id
        AND o.order_status <> 'Cancelled'
    GROUP BY c.customer_id
),
flags AS (
    SELECT
        customer_id,
        last_order_date,
        CASE
            WHEN last_order_date IS NULL THEN 1
            WHEN last_order_date < (CURRENT_DATE - INTERVAL '90 days') THEN 1
            ELSE 0
        END AS is_churned
    FROM last_order
)
SELECT
    COUNT(*) AS total_customers,
    SUM(is_churned) AS churned_customers,
    ROUND(SUM(is_churned) * 100.0 / NULLIF(COUNT(*), 0), 2) AS churn_rate_pct
FROM flags;

```

### 3) Identify customers likely to upgrade loyalty tiers (within 40% of next tier)

```

WITH thresholds AS (
    SELECT
        1000::numeric AS bronze_to_silver,
        5000::numeric AS silver_to_gold,
        10000::numeric AS gold_to_platinum
),
next_target AS (
    SELECT
        c.customer_id,
        c.first_name,
        c.last_name,
        c.email,
        c.loyalty_tier,
        c.total_spent,
        CASE
            WHEN c.loyalty_tier = 'Bronze' THEN t.bronze_to_silver
            WHEN c.loyalty_tier = 'Silver' THEN t.silver_to_gold
            WHEN c.loyalty_tier = 'Gold' THEN t.gold_to_platinum
            ELSE NULL
        END AS next_tier_target
    FROM customers c
        JOIN thresholds t
            ON c.loyalty_tier = t.threshold
            AND c.total_spent >= t.threshold
            AND c.total_spent < t.threshold * 1.4
)
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    c.email,
    c.loyalty_tier,
    c.total_spent,
    nt.next_tier_target
FROM customers c
    JOIN next_target nt
        ON c.customer_id = nt.customer_id
        AND nt.next_tier_target IS NOT NULL
        AND nt.next_tier_target > c.loyalty_tier
ORDER BY c.loyalty_tier;

```

```

        CROSS JOIN thresholds t
    )
SELECT
    customer_id,
    first_name,
    last_name,
    email,
    loyalty_tier,
    total_spent,
    next_tier_target,
    ROUND((next_tier_target - total_spent), 2) AS amount_needed
FROM next_target
WHERE next_tier_target IS NOT NULL
    AND total_spent < next_tier_target
    AND total_spent >= next_tier_target * 0.60
ORDER BY amount_needed ASC;

```

## Challenge 3: Revenue Optimization

### 1) Find the most profitable product combinations (pairs bought together)

```

SELECT
    LEAST(oi1.product_id, oi2.product_id) AS product_a,
    GREATEST(oi1.product_id, oi2.product_id) AS product_b,
    COUNT(*) AS times_bought_together
FROM order_items oi1
JOIN order_items oi2
    ON oi1.order_id = oi2.order_id
    AND oi1.product_id < oi2.product_id
GROUP BY product_a, product_b
ORDER BY times_bought_together DESC
LIMIT 20;

```

### 2) Analyze discount effectiveness on revenue

```

SELECT
    CASE
        WHEN COALESCE(oi.discount, 0) = 0 THEN 'No Discount'
        ELSE 'Discount Applied'
    END AS discount_effectiveness
    -- Add other columns from the query here

```

```

        END AS discount_group,
        COUNT(*) AS line_items,
        ROUND(SUM(oi.subtotal), 2) AS revenue,
        ROUND(AVG(oi.subtotal), 2) AS avg_line_revenue,
        ROUND(AVG(COALESCE(oi.discount, 0)), 2) AS avg_discount
    FROM order_items oi
    GROUP BY 1
    ORDER BY 1;

```

### 3) Calculate revenue per warehouse

```

SELECT
    w.warehouse_id,
    w.warehouse_name,
    COUNT(DISTINCT o.order_id) AS total_orders,
    ROUND(SUM(o.total_amount), 2) AS revenue
FROM shipments s
JOIN warehouses w ON w.warehouse_id = s.warehouse_id
JOIN orders o ON o.order_id = s.order_id
WHERE o.order_status <> 'Cancelled'
GROUP BY w.warehouse_id, w.warehouse_name
ORDER BY revenue DESC;

```

## Challenge 4: Performance Tuning

### 1) Optimize the slowest queries using EXPLAIN

```

EXPLAIN (ANALYZE, BUFFERS)
SELECT *
FROM orders
WHERE order_date >= TIMESTAMP '2024-01-01 00:00:00'
    AND order_date <  TIMESTAMP '2025-01-01 00:00:00';

```

### 2) Create appropriate indexes

```

-- Orders: filter by customer + date
CREATE INDEX IF NOT EXISTS idx_orders_customer_date
ON orders (customer_id, order_date);

-- Orders: filter by date
CREATE INDEX IF NOT EXISTS idx_orders_order_date
ON orders (order_date);

-- Order items: join helpers
CREATE INDEX IF NOT EXISTS idx_order_items_order
ON order_items (order_id);

CREATE INDEX IF NOT EXISTS idx_order_items_product

```

```
ON order_items (product_id);

-- Inventory: product + warehouse lookup
CREATE INDEX IF NOT EXISTS idx_inventory_product_wh
ON inventory (product_id, warehouse_id);

-- Shipments: order to warehouse
CREATE INDEX IF NOT EXISTS idx_shipments_order
ON shipments (order_id);

CREATE INDEX IF NOT EXISTS idx_shipments_wh
ON shipments (warehouse_id);

-- Refresh planner statistics
ANALYZE orders;
ANALYZE order_items;
ANALYZE inventory;
ANALYZE shipments;
```

### 3) Rewrite a subquery using JOINs for better performance

```
SELECT DISTINCT
    p.product_id,
    p.product_name
FROM products p
JOIN order_items oi ON oi.product_id = p.product_id;
```