TechMart SQL Queries – (PostgreSQL)

Covers Challenge 1–4 with clean, production-ready SQL matching the TechMart schema.

Challenge 1: Inventory Management

1) Identify products that need reordering (network-wide)

Products whose total stock across all warehouses is below their reorder_level:

```
WITH stock AS (
  SELECT i.product_id, SUM(i.quantity) AS total_on_hand
  FROM inventory i
  GROUP BY i.product_id
)
SELECT
  p.product_id,
  p.product_name,
  COALESCE(s.total_on_hand, 0) AS total_on_hand,
  p.reorder_level,
  GREATEST(p.reorder_level - COALESCE(s.total_on_hand, 0), 0) AS reorder_qty
FROM products p
LEFT JOIN stock s ON s.product_id = p.product_id
WHERE p.is_active = TRUE
  AND COALESCE(s.total_on_hand, 0) < p.reorder_level
ORDER BY reorder_qty DESC, p.product_name;
```

2) Cost of restocking all products below reorder level

```
WITH stock AS (
  SELECT product_id, SUM(quantity) AS total_on_hand
  FROM inventory
  GROUP BY product_id
), needs AS (
  SELECT
    p.product_id, p.product_name, p.cost_price, p.reorder_level,
    COALESCE(s.total_on_hand, 0) AS total_on_hand,
    GREATEST(p.reorder_level - COALESCE(s.total_on_hand, 0), 0) AS reorder_qty
  FROM products p
  LEFT JOIN stock s ON s.product_id = p.product_id
  WHERE p.is_active = TRUE
)
SELECT
  SUM(reorder_qty * cost_price) AS total_restock_cost,
  COUNT(*) FILTER (WHERE reorder_qty > 0) AS product_count_needing_restock
FROM needs
WHERE reorder_qty > 0;
```

3) Recommend warehouse transfers to balance inventory

Balance by moving units from warehouses above average to those below average for each product:

```
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  -- donor has surplus
 AND d2.diff_from_avg < 0  -- receiver 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) Cohort size by registration month (simple)

```
SELECT
  DATE_TRUNC('month', registration_date) AS cohort_month,
  COUNT(*) AS total_customers
FROM customers
GROUP BY DATE_TRUNC('month', registration_date)
ORDER BY cohort_month;
```

2) Cohort retention by months since signup

```
WITH customers_cohort AS (
  SELECT c.customer_id, DATE_TRUNC('month', c.registration_date::timestamp) AS
cohort_month
```

```
  FROM customers c
), activity AS (
  SELECT o.customer_id, DATE_TRUNC('month', o.order_date) AS activity_month
  FROM orders o
  WHERE o.order_status <> 'Cancelled'
  GROUP BY o.customer_id, DATE_TRUNC('month', o.order_date)
), cohort_size AS (
  SELECT cohort_month, COUNT(*) AS customers_in_cohort
  FROM customers_cohort
  GROUP BY cohort_month
), retention AS (
  SELECT
    cc.cohort_month, a.activity_month,
    (EXTRACT(YEAR FROM a.activity_month) * 12 + EXTRACT(MONTH FROM
a.activity_month))
    - (EXTRACT(YEAR FROM cc.cohort_month) * 12 + EXTRACT(MONTH FROM
cc.cohort_month)) AS months_since_signup,
    COUNT(DISTINCT a.customer_id) AS active_customers
  FROM customers_cohort cc
  JOIN activity a ON a.customer_id = cc.customer_id
  GROUP BY cc.cohort_month, a.activity_month
)
SELECT
  r.cohort_month, r.months_since_signup, r.active_customers,
cs.customers_in_cohort,
  ROUND((r.active_customers::numeric / cs.customers_in_cohort), 4) AS
retention_rate
FROM retention r
JOIN cohort_size cs USING (cohort_month)
WHERE r.months_since_signup >= 0
ORDER BY r.cohort_month, r.months_since_signup;
```

3) Monthly churn rate (simple)

```
WITH active_months AS (
  SELECT customer_id, DATE_TRUNC('month', order_date) AS activity_month
  FROM orders
  WHERE order_status <> 'Cancelled'
  GROUP BY customer_id, DATE_TRUNC('month', order_date)
)
SELECT
  curr.activity_month AS month,
  COUNT(DISTINCT prev.customer_id) AS active_previous_month,
  COUNT(DISTINCT curr.customer_id) AS active_current_month,
  COUNT(DISTINCT prev.customer_id) - COUNT(DISTINCT curr.customer_id) AS
churn_count
FROM active_months curr
LEFT JOIN active_months prev
  ON prev.customer_id = curr.customer_id
 AND prev.activity_month = curr.activity_month - INTERVAL '1 month'
GROUP BY curr.activity_month
ORDER BY curr.activity_month;
```

4) Customers likely to upgrade loyalty tiers (simple thresholds)

```
SELECT
  customer_id, first_name, last_name, loyalty_tier, total_spent,
  CASE
    WHEN loyalty_tier = 'Bronze' THEN 500 - total_spent
    WHEN loyalty_tier = 'Silver' THEN 2000 - total_spent
    WHEN loyalty_tier = 'Gold'   THEN 5000 - total_spent
    ELSE NULL
  END AS amount_needed_for_upgrade
FROM customers
WHERE loyalty_tier <> 'Platinum'
ORDER BY amount_needed_for_upgrade;
```

Challenge 3: Revenue Optimization

1) Most frequent 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) Discount effectiveness on revenue (simple)

```
SELECT
  CASE WHEN discount > 0 THEN 'Discount Applied' ELSE 'No Discount' END AS
discount_type,
  SUM(subtotal) AS total_revenue
FROM order_items
GROUP BY discount_type;
```

3) Revenue per warehouse (via shipments)

```
SELECT
  w.warehouse_name,
  SUM(oi.subtotal) AS revenue
FROM shipments s
JOIN order_items oi ON oi.order_id = s.order_id
JOIN warehouses w ON w.warehouse_id = s.warehouse_id
GROUP BY w.warehouse_name
ORDER BY revenue DESC;
```

Challenge 4: Performance Tuning

1) Use EXPLAIN to inspect a query

```
EXPLAIN ANALYZE
SELECT * FROM orders WHERE customer_id = 5;
```

2) Create useful indexes

```
CREATE INDEX idx_orders_customer ON orders(customer_id);
CREATE INDEX idx_order_items_product ON order_items(product_id);
CREATE INDEX idx_inventory_product ON inventory(product_id);
```

3) Rewrite a subquery using JOIN

Before (slower):

```
SELECT *
FROM products
WHERE product_id IN (SELECT product_id FROM order_items);
```

After (faster):

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