⚡ Day 4: Advanced SQL for Data Engineering - Complete Guide

📚 What You'll Learn Today

- **Advanced Window Functions** for complex analytics
- **Complex JOINs and Subqueries** for multi-table analysis
- **Common Table Expressions (CTEs)** for readable, modular queries
- **Query Performance Optimization** for production systems
- **Real-world Advanced Analytics** with multiple Kaggle datasets

---

🎯 Learning Objectives

By the end of Day 4, you will:

1. Master advanced window functions for sophisticated analytics
2. Build complex multi-table queries with optimal performance
3. Use CTEs for hierarchical and recursive data processing
4. Optimize query performance for large datasets
5. Create production-ready analytical queries

📊 Real Kaggle Datasets for Day 4

Primary Dataset: Sample Superstore Dataset - Practice Your Data Analysis Skills as a Superstore Data Analyst

- **Kaggle Link**: kaggle.com/datasets/bravehart101/sample-supermarket-dataset
- **Size**: 9,426 records
- **Use Case**: Advanced analytics, complex JOINs, performance optimization

Secondary Dataset: E-Commerce Transactions Dataset

- **Kaggle Link**: kaggle.com/datasets/smayanj/e-commerce-transactions-dataset
- **Size**: 50,000+ records
- **Use Case:** Large dataset performance testing, complex analytics

Download Instructions:

```bash
# Download both datasets
kaggle datasets download -d bravehart101/sample-supermarket-dataset --unzip
kaggle datasets download -d smayanj/e-commerce-transactions-dataset --unzip
```

🔥 Advanced Window Functions

1. Ranking and Dense Ranking

Window Functions allow calculations across a set of table rows that are related to the current row. Unlike traditional aggregate functions, which collapse the result set into a single value per group, window functions return a value for every row in the result set.

```
sql
```

```sql
-- Advanced ranking analysis
WITH customer_rankings AS (
    SELECT
        customer_name,
        customer_id,
        segment,
        region,
        SUM(sales) as total_sales,
        SUM(profit) as total_profit,
        COUNT(DISTINCT order_id) as total_orders,

        -- Different ranking functions
        ROW_NUMBER() OVER (ORDER BY SUM(sales) DESC) as sales_rank,
        RANK() OVER (ORDER BY SUM(sales) DESC) as sales_rank_with_ties,
        DENSE_RANK() OVER (ORDER BY SUM(sales) DESC) as dense_sales_rank,

        -- Ranking within segments
        ROW_NUMBER() OVER (PARTITION BY segment ORDER BY SUM(sales) DESC) as rank_in_s

        -- Percentile ranking
        PERCENT_RANK() OVER (ORDER BY SUM(sales)) as sales_percentile,
        NTILE(10) OVER (ORDER BY SUM(sales)) as sales_decile
    FROM superstore
    GROUP BY customer_name, customer_id, segment, region
)
SELECT
    customer_name,
    segment,
    region,
    total_sales,
    total_profit,
    sales_rank,
    rank_in_segment,
    sales_decile,
    CASE
        WHEN sales_decile >= 9 THEN 'Top 20%'
        WHEN sales_decile >= 7 THEN 'High Value'
        WHEN sales_decile >= 4 THEN 'Medium Value'
        ELSE 'Low Value'
    END as customer_tier
FROM customer_rankings
```

```
WHERE sales_rank <= 100
ORDER BY sales_rank;
```

2. Advanced Lag/Lead Analysis

```
sql
```

```sql
-- Customer behavior analysis with lag/lead
WITH customer_orders AS (
    SELECT
        customer_id,
        customer_name,
        order_date,
        sales,
        profit,
        category,

        -- Previous and next order analysis
        LAG(order_date, 1) OVER (PARTITION BY customer_id ORDER BY order_date) as prev_
        LEAD(order_date, 1) OVER (PARTITION BY customer_id ORDER BY order_date) as nex
        LAG(sales, 1) OVER (PARTITION BY customer_id ORDER BY order_date) as prev_orde

        -- First and last values
        FIRST_VALUE(order_date) OVER (PARTITION BY customer_id ORDER BY order_date) as
        LAST_VALUE(order_date) OVER (
            PARTITION BY customer_id
            ORDER BY order_date
            ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
        ) as last_order_date,

        -- Order sequence numbering
        ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date) as order_sequ
    FROM superstore
),
customer_insights AS (
    SELECT
        *,
        -- Calculate days between orders
        COALESCE(order_date - prev_order_date, 0) as days_since_last_order,
        COALESCE(next_order_date - order_date, 0) as days_to_next_order,

        -- Customer lifetime in days
        last_order_date - first_order_date as customer_lifetime_days,

        -- Sales trend analysis
        CASE
            WHEN prev_order_sales IS NULL THEN 'First Order'
            WHEN sales > prev_order_sales THEN 'Increasing'
            WHEN sales < prev_order_sales THEN 'Decreasing'
            ELSE 'Stable'
        END as sales_trend
    FROM customer_orders
)
SELECT
    customer_name,
    order_sequence,
    order_date,
    sales,
    days_since_last_order,
    sales_trend,
    customer_lifetime_days,

    -- Customer lifecycle stage
    CASE
        WHEN order_sequence = 1 THEN 'New Customer'
        WHEN days_since_last_order <= 30 THEN 'Active'
        WHEN days_since_last_order <= 90 THEN 'At Risk'
        ELSE 'Churned'
    END as customer_status
FROM customer_insights
WHERE customer_id IN (
    SELECT customer_id
    FROM customer_insights
    GROUP BY customer_id
    HAVING COUNT(*) >= 5
)
ORDER BY customer_name, order_sequence;
```

3. Moving Averages and Rolling Calculations

```
sql
```

```sql
-- Advanced time series analysis
WITH daily_sales AS (
    SELECT
        order_date,
        SUM(sales) as daily_sales,
        SUM(profit) as daily_profit,
        COUNT(DISTINCT order_id) as daily_orders,
        COUNT(DISTINCT customer_id) as daily_customers
    FROM superstore
    GROUP BY order_date
),
sales_analytics AS (
    SELECT
        order_date,
        daily_sales,
        daily_profit,
        daily_orders,
        daily_customers,

        -- Moving averages
        AVG(daily_sales) OVER (
            ORDER BY order_date
            ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
        ) as sales_7day_ma,

        AVG(daily_sales) OVER (
            ORDER BY order_date
            ROWS BETWEEN 29 PRECEDING AND CURRENT ROW
        ) as sales_30day_ma,

        -- Rolling sums
        SUM(daily_sales) OVER (
            ORDER BY order_date
            ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
        ) as sales_7day_rolling,

        -- Standard deviation for volatility
        STDDEV(daily_sales) OVER (
            ORDER BY order_date
            ROWS BETWEEN 29 PRECEDING AND CURRENT ROW
        ) as sales_30day_stddev,

        -- Min/Max in rolling window
        MIN(daily_sales) OVER (
            ORDER BY order_date
            ROWS BETWEEN 29 PRECEDING AND CURRENT ROW
        ) as sales_30day_min,

        MAX(daily_sales) OVER (
            ORDER BY order_date
            ROWS BETWEEN 29 PRECEDING AND CURRENT ROW
        ) as sales_30day_max
    FROM daily_sales
)
SELECT
    order_date,
    daily_sales,
    sales_7day_ma,
    sales_30day_ma,
    sales_30day_stddev,

    -- Trend analysis
    CASE
        WHEN daily_sales > sales_30day_ma + (2 * sales_30day_stddev) THEN 'Exceptional
        WHEN daily_sales > sales_30day_ma + sales_30day_stddev THEN 'High'
        WHEN daily_sales < sales_30day_ma - (2 * sales_30day_stddev) THEN 'Exceptional
        WHEN daily_sales < sales_30day_ma - sales_30day_stddev THEN 'Low'
        ELSE 'Normal'
    END as sales_performance,

    -- Volatility measure
    CASE
        WHEN sales_30day_stddev / NULLIF(sales_30day_ma, 0) > 0.5 THEN 'High Volatility
        WHEN sales_30day_stddev / NULLIF(sales_30day_ma, 0) > 0.2 THEN 'Medium Volatil
        ELSE 'Low Volatility'
    END as volatility_level
FROM sales_analytics
WHERE order_date >= (SELECT MIN(order_date) + INTERVAL '30 days' FROM daily_sales)
ORDER BY order_date;
```

🔗 Complex JOINs and Subqueries

1. Advanced Multi-Table Analysis

sql

```sql
-- Create dimension tables for complex analysis
-- Customer dimension
CREATE TABLE dim_customers AS
SELECT DISTINCT
    customer_id,
    customer_name,
    segment,
    city,
    state,
    region,
    country,
    -- Customer metrics
    (SELECT MIN(order_date) FROM superstore s2 WHERE s2.customer_id = s1.customer_id)
    (SELECT MAX(order_date) FROM superstore s2 WHERE s2.customer_id = s1.customer_id)
    (SELECT COUNT(*) FROM superstore s2 WHERE s2.customer_id = s1.customer_id) as tota
    (SELECT SUM(sales) FROM superstore s2 WHERE s2.customer_id = s1.customer_id) as li
FROM superstore s1;

-- Product dimension
CREATE TABLE dim_products AS
SELECT DISTINCT
    product_id,
    product_name,
    category,
    sub_category,
    -- Product metrics
    (SELECT AVG(sales) FROM superstore s2 WHERE s2.product_id = s1.product_id) as avg_
    (SELECT SUM(quantity) FROM superstore s2 WHERE s2.product_id = s1.product_id) as t
    (SELECT COUNT(DISTINCT customer_id) FROM superstore s2 WHERE s2.product_id = s1.pr
FROM superstore s1;

-- Time dimension
CREATE TABLE dim_time AS
WITH date_range AS (
    SELECT generate_series(
        (SELECT MIN(order_date) FROM superstore),
        (SELECT MAX(order_date) FROM superstore),
        '1 day'::interval
    )::date as date_value
)
SELECT
    date_value,
    EXTRACT(YEAR FROM date_value) as year,
    EXTRACT(MONTH FROM date_value) as month,
    EXTRACT(DAY FROM date_value) as day,
    EXTRACT(QUARTER FROM date_value) as quarter,
    EXTRACT(DOW FROM date_value) as day_of_week,
    TO_CHAR(date_value, 'Day') as day_name,
    TO_CHAR(date_value, 'Month') as month_name,
    CASE
        WHEN EXTRACT(DOW FROM date_value) IN (0, 6) THEN 'Weekend'
        ELSE 'Weekday'
    END as day_type
FROM date_range;

-- Complex multi-table analysis
WITH customer_product_affinity AS (
    SELECT
        c.customer_id,
        c.customer_name,
        c.segment,
        p.category,
        p.sub_category,
        COUNT(*) as purchase_frequency,
        SUM(s.sales) as category_spend,
        AVG(s.sales) as avg_order_value,

        -- Rank categories by customer preference
        ROW_NUMBER() OVER (
            PARTITION BY c.customer_id
            ORDER BY COUNT(*) DESC, SUM(s.sales) DESC
        ) as category_preference_rank
    FROM superstore s
    JOIN dim_customers c ON s.customer_id = c.customer_id
    JOIN dim_products p ON s.product_id = p.product_id
    JOIN dim_time t ON s.order_date = t.date_value
    WHERE t.year >= 2020
    GROUP BY c.customer_id, c.customer_name, c.segment, p.category, p.sub_category
),
customer_segments AS (
    SELECT
        customer_id,
        customer_name,
        segment,

        -- Primary category (most frequent purchases)
        MAX(CASE WHEN category_preference_rank = 1 THEN category END) as primary_categ
        -- Category diversity (number of different categories purchased)
        COUNT(DISTINCT category) as category_diversity,

        -- Total spend and frequency
        SUM(category_spend) as total_spend,
        SUM(purchase_frequency) as total_frequency,

        -- Calculate category concentration (how focused customer is)
        MAX(category_spend) / SUM(category_spend) as category_concentration
    FROM customer_product_affinity
    GROUP BY customer_id, customer_name, segment
)
SELECT
    cs.customer_name,
    cs.segment,
    cs.primary_category,
    cs.category_diversity,
    cs.total_spend,
    cs.total_frequency,
    ROUND(cs.category_concentration * 100, 2) as concentration_pct,

    -- Customer behavior classification
    CASE
        WHEN cs.category_concentration > 0.8 THEN 'Specialist'
        WHEN cs.category_concentration > 0.5 THEN 'Focused'
        ELSE 'Diversified'
    END as shopping_behavior,

    -- Value classification
    CASE
        WHEN cs.total_spend > 15000 THEN 'High Value'
        WHEN cs.total_spend > 5000 THEN 'Medium Value'
        ELSE 'Low Value'
    END as value_segment
FROM customer_segments cs
WHERE cs.total_frequency >= 5
ORDER BY cs.total_spend DESC;
```

2. Correlated Subqueries for Advanced Analytics

```
sql
```

```sql
-- Advanced customer analysis with correlated subqueries
SELECT
    c.customer_name,
    c.segment,
    c.region,
    c.lifetime_value,

    -- Compare to segment average
    (
        SELECT AVG(lifetime_value)
        FROM dim_customers c2
        WHERE c2.segment = c.segment
    ) as segment_avg_value,

    -- Customer rank within segment
    (
        SELECT COUNT(*) + 1
        FROM dim_customers c2
        WHERE c2.segment = c.segment
        AND c2.lifetime_value > c.lifetime_value
    ) as rank_in_segment,

    -- Most purchased category
    (
        SELECT p.category
        FROM superstore s
        JOIN dim_products p ON s.product_id = p.product_id
        WHERE s.customer_id = c.customer_id
        GROUP BY p.category
        ORDER BY SUM(s.sales) DESC
        LIMIT 1
    ) as favorite_category,

    -- Average days between orders
    (
        SELECT AVG(days_between)
        FROM (
            SELECT
                order_date - LAG(order_date) OVER (ORDER BY order_date) as days_betwee
            FROM superstore s2
            WHERE s2.customer_id = c.customer_id
        ) t
        WHERE days_between IS NOT NULL
    ) as avg_days_between_orders,

    -- Has made recent purchase (last 90 days)
    EXISTS (
        SELECT 1
        FROM superstore s3
        WHERE s3.customer_id = c.customer_id
        AND s3.order_date >= CURRENT_DATE - INTERVAL '90 days'
    ) as recent_customer
FROM dim_customers c
WHERE c.total_orders >= 3
ORDER BY c.lifetime_value DESC;
```

📈 Common Table Expressions (CTEs) Advanced Patterns

1. Hierarchical Data Processing

A Common Table Expression (CTE) is a temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. CTEs make complex queries easier to read, write, and maintain by breaking them down into logical, manageable parts.

sql

```sql
-- Hierarchical product category analysis
WITH RECURSIVE category_hierarchy AS (
    -- Base case: main categories
    SELECT
        category as path,
        category,
        sub_category,
        0 as level,
        category as root_category
    FROM dim_products
    WHERE category IS NOT NULL

    UNION ALL

    -- Recursive case: sub-categories
    SELECT
        ch.path || ' > ' || p.sub_category as path,
        p.category,
        p.sub_category,
        ch.level + 1,
        ch.root_category
    FROM category_hierarchy ch
    JOIN dim_products p ON ch.category = p.category
    WHERE p.sub_category IS NOT NULL
    AND ch.level < 2
),
category_performance AS (
    SELECT
        ch.path,
        ch.level,
        ch.root_category,
        COUNT(DISTINCT s.product_id) as product_count,
        COUNT(DISTINCT s.customer_id) as customer_count,
        SUM(s.sales) as total_sales,
        SUM(s.profit) as total_profit,
        AVG(s.sales) as avg_order_value,

        -- Profit margin
        CASE
            WHEN SUM(s.sales) > 0 THEN SUM(s.profit) / SUM(s.sales) * 100
            ELSE 0
        END as profit_margin_pct
    FROM category_hierarchy ch
    LEFT JOIN dim_products p ON (ch.category = p.category AND ch.sub_category = p.sub_
    LEFT JOIN superstore s ON p.product_id = s.product_id
    GROUP BY ch.path, ch.level, ch.root_category
)
SELECT
    REPEAT('  ', level) || path as hierarchy_display,
    product_count,
    customer_count,
    total_sales,
    total_profit,
    ROUND(profit_margin_pct, 2) as profit_margin_pct,

    -- Performance ranking within level
    RANK() OVER (PARTITION BY level ORDER BY total_sales DESC) as sales_rank_in_level
FROM category_performance
WHERE total_sales > 0
ORDER BY root_category, level, total_sales DESC;
```

2. Complex Time Series Analysis with CTEs

sql

```sql
-- Multi-layered time series analysis
WITH monthly_base AS (
    -- Base monthly aggregations
    SELECT
        DATE_TRUNC('month', order_date) as month,
        category,
        segment,
        SUM(sales) as monthly_sales,
        SUM(profit) as monthly_profit,
        COUNT(DISTINCT customer_id) as monthly_customers,
        COUNT(DISTINCT order_id) as monthly_orders
    FROM superstore s
    JOIN dim_products p ON s.product_id = p.product_id
    JOIN dim_customers c ON s.customer_id = c.customer_id
    GROUP BY DATE_TRUNC('month', order_date), category, segment
),
monthly_trends AS (
    -- Add trend calculations
    SELECT
        *,
        LAG(monthly_sales, 1) OVER (
            PARTITION BY category, segment
            ORDER BY month
        ) as prev_month_sales,

        LAG(monthly_sales, 12) OVER (
            PARTITION BY category, segment
            ORDER BY month
        ) as same_month_last_year,

        AVG(monthly_sales) OVER (
            PARTITION BY category, segment
            ORDER BY month
            ROWS BETWEEN 11 PRECEDING AND CURRENT ROW
        ) as trailing_12_month_avg
    FROM monthly_base
),
monthly_insights AS (
    -- Calculate growth rates and trends
    SELECT
        *,
        CASE
            WHEN prev_month_sales > 0 THEN
```

```sql
                ((monthly_sales - prev_month_sales) / prev_month_sales) * 100
            ELSE NULL
        END as mom_growth_pct,

        CASE
            WHEN same_month_last_year > 0 THEN
                ((monthly_sales - same_month_last_year) / same_month_last_year) * 100
            ELSE NULL
        END as yoy_growth_pct,

        CASE
            WHEN trailing_12_month_avg > 0 THEN
                ((monthly_sales - trailing_12_month_avg) / trailing_12_month_avg) * 10
            ELSE NULL
        END as vs_12mo_avg_pct
    FROM monthly_trends
),
performance_classification AS (
    -- Classify performance
    SELECT
        *,
        CASE
            WHEN mom_growth_pct > 20 THEN 'Accelerating'
            WHEN mom_growth_pct > 5 THEN 'Growing'
            WHEN mom_growth_pct > -5 THEN 'Stable'
            WHEN mom_growth_pct > -20 THEN 'Declining'
            ELSE 'Steep Decline'
        END as mom_trend,

        CASE
            WHEN yoy_growth_pct > 15 THEN 'Strong Growth'
            WHEN yoy_growth_pct > 5 THEN 'Moderate Growth'
            WHEN yoy_growth_pct > -5 THEN 'Flat'
            ELSE 'Declining'
        END as yoy_trend
    FROM monthly_insights
)
SELECT
    month,
    category,
    segment,
    monthly_sales,
    monthly_profit,
    monthly_customers,
    ROUND(mom_growth_pct, 2) as mom_growth_pct,
    ROUND(yoy_growth_pct, 2) as yoy_growth_pct,
    mom_trend,
    yoy_trend,

    -- Overall performance score
    CASE
        WHEN mom_trend IN ('Accelerating', 'Growing') AND yoy_trend IN ('Strong Growth
        WHEN mom_trend IN ('Growing', 'Stable') AND yoy_trend IN ('Moderate Growth', '
        WHEN mom_trend = 'Stable' AND yoy_trend = 'Flat' THEN 'Steady'
        WHEN mom_trend IN ('Declining', 'Steep Decline') OR yoy_trend = 'Declining' TH
        ELSE 'Mixed'
    END as overall_performance
FROM performance_classification
WHERE month >= DATE_TRUNC('month', CURRENT_DATE) - INTERVAL '24 months'
ORDER BY category, segment, month DESC;
```

3. Advanced Cohort Analysis with CTEs

```
sql
```

```sql
-- Customer cohort analysis using CTEs
WITH customer_cohorts AS (
    -- Define customer cohorts by first purchase month
    SELECT
        customer_id,
        DATE_TRUNC('month', MIN(order_date)) as cohort_month
    FROM superstore
    GROUP BY customer_id
),
customer_activities AS (
    -- Track customer activities by month
    SELECT
        cc.cohort_month,
        cc.customer_id,
        DATE_TRUNC('month', s.order_date) as activity_month,
        SUM(s.sales) as monthly_sales,
        COUNT(DISTINCT s.order_id) as monthly_orders
    FROM customer_cohorts cc
    JOIN superstore s ON cc.customer_id = s.customer_id
    GROUP BY cc.cohort_month, cc.customer_id, DATE_TRUNC('month', s.order_date)
),
cohort_metrics AS (
    -- Calculate cohort metrics
    SELECT
        cohort_month,
        activity_month,
        EXTRACT(MONTH FROM AGE(activity_month, cohort_month)) as months_since_first_pu
        COUNT(DISTINCT customer_id) as active_customers,
        SUM(monthly_sales) as cohort_revenue,
        AVG(monthly_sales) as avg_customer_spend
    FROM customer_activities
    GROUP BY cohort_month, activity_month
),
cohort_sizes AS (
    -- Calculate initial cohort sizes
    SELECT
        cohort_month,
        COUNT(DISTINCT customer_id) as cohort_size
    FROM customer_cohorts
    GROUP BY cohort_month
),
cohort_retention AS (
    -- Calculate retention rates
    SELECT
        cm.cohort_month,
        cm.months_since_first_purchase,
        cs.cohort_size,
        cm.active_customers,
        cm.cohort_revenue,
        cm.avg_customer_spend,

        -- Retention rate
        ROUND(
            (cm.active_customers::DECIMAL / cs.cohort_size) * 100, 2
        ) as retention_rate,

        -- Revenue per original customer
        ROUND(
            cm.cohort_revenue / cs.cohort_size, 2
        ) as revenue_per_original_customer
    FROM cohort_metrics cm
    JOIN cohort_sizes cs ON cm.cohort_month = cs.cohort_month
)
SELECT
    cohort_month,
    months_since_first_purchase,
    cohort_size,
    active_customers,
    retention_rate,
    cohort_revenue,
    revenue_per_original_customer,

    -- Cohort performance classification
    CASE
        WHEN months_since_first_purchase = 0 THEN 'Acquisition'
        WHEN months_since_first_purchase <= 3 AND retention_rate >= 30 THEN 'Strong Ea
        WHEN months_since_first_purchase <= 3 AND retention_rate >= 15 THEN 'Moderate
        WHEN months_since_first_purchase <= 3 THEN 'Weak Early Retention'
        WHEN months_since_first_purchase <= 12 AND retention_rate >= 15 THEN 'Strong L
        WHEN months_since_first_purchase <= 12 AND retention_rate >= 8 THEN 'Moderate
        ELSE 'Weak Long-term'
    END as retention_classification
FROM cohort_retention
WHERE months_since_first_purchase <= 24
ORDER BY cohort_month, months_since_first_purchase;
```

⚡ Query Performance Optimization

1. Indexing Strategies for Data Engineering

```sql
-- Create strategic indexes for performance
-- Covering index for customer analysis
CREATE INDEX CONCURRENTLY idx_superstore_customer_analysis
ON superstore (customer_id, order_date)
INCLUDE (sales, profit, quantity);

-- Partial index for recent high-value transactions
CREATE INDEX CONCURRENTLY idx_superstore_recent_high_value
ON superstore (order_date, sales)
WHERE order_date >= '2020-01-01' AND sales > 1000;

-- Composite index for category analysis
CREATE INDEX CONCURRENTLY idx_superstore_category_segment
ON superstore (category, segment, order_date);

-- Function-based index for date analysis
CREATE INDEX CONCURRENTLY idx_superstore_year_month
ON superstore (EXTRACT(YEAR FROM order_date), EXTRACT(MONTH FROM order_date));

-- Check index usage
SELECT
    schemaname,
    tablename,
    indexname,
    idx_scan as index_scans,
    idx_tup_read as tuples_read,
    idx_tup_fetch as tuples_fetched
FROM pg_stat_user_indexes
WHERE tablename = 'superstore'
ORDER BY idx_scan DESC;
```

2. Query Optimization Techniques

```sql
-- Example: Optimized vs Unoptimized queries

-- BEFORE: Inefficient query
-- This query has multiple performance issues
SELECT DISTINCT
    c.customer_name,
    (SELECT SUM(s2.sales) FROM superstore s2 WHERE s2.customer_id = c.customer_id) as
    (SELECT COUNT(*) FROM superstore s3 WHERE s3.customer_id = c.customer_id) as order_
FROM (SELECT DISTINCT customer_id, customer_name FROM superstore) c
WHERE (SELECT SUM(s4.sales) FROM superstore s4 WHERE s4.customer_id = c.customer_id) >
ORDER BY (SELECT SUM(s5.sales) FROM superstore s5 WHERE s5.customer_id = c.customer_id

-- AFTER: Optimized query
-- Much more efficient with single table scan and proper aggregation
SELECT
    customer_name,
    SUM(sales) as total_sales,
    COUNT(*) as order_count
FROM superstore
GROUP BY customer_id, customer_name
HAVING SUM(sales) > 5000
ORDER BY total_sales DESC;

-- Performance comparison query
EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
SELECT
    customer_name,
    SUM(sales) as total_sales,
    COUNT(*) as order_count,
    AVG(sales) as avg_order_value,
    MAX(order_date) as last_order_date
FROM superstore
WHERE order_date >= '2020-01-01'
GROUP BY customer_id, customer_name
HAVING SUM(sales) > 1000
ORDER BY total_sales DESC
LIMIT 100;
```

3. Materialized Views for Performance

```
sql
```

```sql
-- Create materialized views for heavy analytical queries
CREATE MATERIALIZED VIEW mv_customer_monthly_summary AS
WITH monthly_customer_metrics AS (
    SELECT
        customer_id,
        customer_name,
        segment,
        region,
        DATE_TRUNC('month', order_date) as month,
        SUM(sales) as monthly_sales,
        SUM(profit) as monthly_profit,
        COUNT(DISTINCT order_id) as monthly_orders,
        AVG(sales) as avg_order_value
    FROM superstore
    GROUP BY customer_id, customer_name, segment, region, DATE_TRUNC('month', order_da
)
SELECT
    *,
    LAG(monthly_sales, 1) OVER (
        PARTITION BY customer_id
        ORDER BY month
    ) as prev_month_sales,

    -- Growth calculations
    CASE
        WHEN LAG(monthly_sales, 1) OVER (PARTITION BY customer_id ORDER BY month) > 0
            ((monthly_sales - LAG(monthly_sales, 1) OVER (PARTITION BY customer_id ORD
             / LAG(monthly_sales, 1) OVER (PARTITION BY customer_id ORDER BY month)) *
        ELSE NULL
    END as month_over_month_growth,

    -- Running totals
    SUM(monthly_sales) OVER (
        PARTITION BY customer_id
        ORDER BY month
        ROWS UNBOUNDED PRECEDING
    ) as cumulative_sales
FROM monthly_customer_metrics;

-- Create indexes on materialized view
CREATE INDEX idx_mv_customer_monthly_customer_month
ON mv_customer_monthly_summary (customer_id, month);
```

```
CREATE INDEX idx_mv_customer_monthly_segment_month
ON mv_customer_monthly_summary (segment, month);

-- Refresh strategy (run this in your ETL pipeline)
REFRESH MATERIALIZED VIEW CONCURRENTLY mv_customer_monthly_summary;
```

🔬 Advanced Analytics Patterns

1. Market Basket Analysis

```
sql
```

```sql
-- Advanced market basket analysis
WITH order_products AS (
    SELECT
        order_id,
        product_name,
        category,
        sub_category,
        sales,
        profit
    FROM superstore
),
product_pairs AS (
    -- Find products bought together
    SELECT
        a.product_name as product_a,
        b.product_name as product_b,
        a.category as category_a,
        b.category as category_b,
        COUNT(*) as frequency,
        AVG(a.sales + b.sales) as avg_combined_value,
        SUM(a.profit + b.profit) as total_combined_profit
    FROM order_products a
    JOIN order_products b ON a.order_id = b.order_id
    WHERE a.product_name < b.product_name  -- Avoid duplicates and self-joins
    GROUP BY a.product_name, b.product_name, a.category, b.category
    HAVING COUNT(*) >= 5  -- Minimum support threshold
),
product_statistics AS (
    -- Calculate individual product statistics
    SELECT
        product_name,
        COUNT(DISTINCT order_id) as total_orders
    FROM order_products
    GROUP BY product_name
),
association_metrics AS (
    -- Calculate association rule metrics
    SELECT
        pp.*,
        psa.total_orders as orders_a,
        psb.total_orders as orders_b,
        (SELECT COUNT(DISTINCT order_id) FROM order_products) as total_unique_orders,
        -- Support: P(A ∩ B)
        pp.frequency::DECIMAL / (SELECT COUNT(DISTINCT order_id) FROM order_products)

        -- Confidence: P(B|A) = P(A ∩ B) / P(A)
        pp.frequency::DECIMAL / psa.total_orders as confidence_a_to_b,

        -- Confidence: P(A|B) = P(A ∩ B) / P(B)
        pp.frequency::DECIMAL / psb.total_orders as confidence_b_to_a,

        -- Lift: P(A ∩ B) / (P(A) * P(B))
        (pp.frequency::DECIMAL / (SELECT COUNT(DISTINCT order_id) FROM order_products)
        ((psa.total_orders::DECIMAL / (SELECT COUNT(DISTINCT order_id) FROM order_prod
         (psb.total_orders::DECIMAL / (SELECT COUNT(DISTINCT order_id) FROM order_prod
    FROM product_pairs pp
    JOIN product_statistics psa ON pp.product_a = psa.product_name
    JOIN product_statistics psb ON pp.product_b = psb.product_name
)
SELECT
    product_a,
    product_b,
    category_a,
    category_b,
    frequency,
    ROUND(support * 100, 3) as support_pct,
    ROUND(confidence_a_to_b * 100, 2) as confidence_a_to_b_pct,
    ROUND(confidence_b_to_a * 100, 2) as confidence_b_to_a_pct,
    ROUND(lift, 3) as lift,
    ROUND(avg_combined_value, 2) as avg_combined_value,

    -- Business interpretation
    CASE
        WHEN lift > 2 THEN 'Strong Association'
        WHEN lift > 1.5 THEN 'Moderate Association'
        WHEN lift > 1 THEN 'Weak Association'
        ELSE 'Negative Association'
    END as association_strength,

    -- Cross-category insights
    CASE
        WHEN category_a = category_b THEN 'Same Category'
        ELSE 'Cross Category'
    END as category_relationship
FROM association_metrics
WHERE lift > 1  -- Only show positive associations
```

```
ORDER BY lift DESC, frequency DESC
LIMIT 50;
```

2. Customer Lifetime Value Prediction

```
sql
```

```sql
-- Advanced CLV analysis with predictive elements
WITH customer_purchase_history AS (
    SELECT
        customer_id,
        customer_name,
        segment,
        region,
        MIN(order_date) as first_purchase_date,
        MAX(order_date) as last_purchase_date,
        COUNT(DISTINCT order_id) as total_orders,
        COUNT(DISTINCT DATE_TRUNC('month', order_date)) as active_months,
        SUM(sales) as total_revenue,
        AVG(sales) as avg_order_value,
        SUM(profit) as total_profit,

        -- Calculate customer lifespan in days
        MAX(order_date) - MIN(order_date) as customer_lifespan_days,

        -- Calculate average days between orders
        CASE
            WHEN COUNT(DISTINCT order_id) > 1 THEN
                (MAX(order_date) - MIN(order_date)) / (COUNT(DISTINCT order_id) - 1)
            ELSE NULL
        END as avg_days_between_orders,

        -- Recency (days since last purchase)
        CURRENT_DATE - MAX(order_date) as days_since_last_purchase
    FROM superstore
    GROUP BY customer_id, customer_name, segment, region
),
customer_clv_metrics AS (
    SELECT
        *,
        -- Purchase frequency (orders per month)
        CASE
            WHEN customer_lifespan_days > 0 THEN
                (total_orders::DECIMAL / (customer_lifespan_days::DECIMAL / 30.44))
            ELSE total_orders
        END as purchase_frequency_monthly,

        -- Customer lifetime value calculation
        -- CLV = Average Order Value × Purchase Frequency × Customer Lifespan
        CASE
```

```sql
            WHEN customer_lifespan_days > 0 AND avg_days_between_orders > 0 THEN
                avg_order_value * (total_orders::DECIMAL / (customer_lifespan_days::DE
                (customer_lifespan_days::DECIMAL / 30.44)
            ELSE total_revenue
        END as historical_clv,

        -- Predicted future value (simple model)
        CASE
            WHEN days_since_last_purchase <= 90 AND avg_days_between_orders > 0 THEN
                avg_order_value * (365.0 / avg_days_between_orders) * 2  -- Predict ne
            ELSE 0
        END as predicted_future_value
    FROM customer_purchase_history
    WHERE total_orders >= 2  -- Only customers with multiple purchases
),
customer_segmentation AS (
    SELECT
        *,
        -- RFM-based segmentation
        NTILE(5) OVER (ORDER BY days_since_last_purchase DESC) as recency_score,
        NTILE(5) OVER (ORDER BY purchase_frequency_monthly) as frequency_score,
        NTILE(5) OVER (ORDER BY avg_order_value) as monetary_score,

        -- CLV-based segmentation
        NTILE(10) OVER (ORDER BY historical_clv) as clv_decile,

        -- Customer status
        CASE
            WHEN days_since_last_purchase <= 30 THEN 'Active'
            WHEN days_since_last_purchase <= 90 THEN 'At Risk'
            WHEN days_since_last_purchase <= 180 THEN 'Dormant'
            ELSE 'Lost'
        END as customer_status
    FROM customer_clv_metrics
)
SELECT
    customer_name,
    segment,
    region,
    total_orders,
    ROUND(avg_order_value, 2) as avg_order_value,
    ROUND(purchase_frequency_monthly, 3) as monthly_frequency,
    days_since_last_purchase,
    ROUND(historical_clv, 2) as historical_clv,
    ROUND(predicted_future_value, 2) as predicted_future_value,
    ROUND(historical_clv + predicted_future_value, 2) as total_clv,

    -- RFM scores combined
    CONCAT(recency_score, frequency_score, monetary_score) as rfm_score,

    -- CLV segment
    CASE
        WHEN clv_decile >= 9 THEN 'Champions'
        WHEN clv_decile >= 7 THEN 'Loyal Customers'
        WHEN clv_decile >= 5 THEN 'Potential Loyalists'
        WHEN clv_decile >= 3 THEN 'New Customers'
        ELSE 'At Risk'
    END as clv_segment,

    customer_status,

    -- Business recommendations
    CASE
        WHEN clv_decile >= 8 AND customer_status = 'Active' THEN 'VIP Treatment'
        WHEN clv_decile >= 6 AND customer_status = 'At Risk' THEN 'Retention Campaign'
        WHEN clv_decile >= 4 AND customer_status = 'Active' THEN 'Upsell Opportunity'
        WHEN customer_status = 'Lost' AND historical_clv > 5000 THEN 'Win Back Campaig
        ELSE 'Standard Treatment'
    END as recommended_action
FROM customer_segmentation
ORDER BY total_clv DESC;
```

3. Time Series Forecasting Foundations

```
sql
```

```sql
-- Advanced time series analysis for forecasting
WITH daily_metrics AS (
    SELECT
        order_date,
        category,
        SUM(sales) as daily_sales,
        COUNT(DISTINCT order_id) as daily_orders,
        COUNT(DISTINCT customer_id) as daily_customers,
        AVG(sales) as avg_order_value
    FROM superstore
    WHERE order_date >= '2020-01-01'
    GROUP BY order_date, category
),
time_series_features AS (
    SELECT
        *,
        -- Day of week effects
        EXTRACT(DOW FROM order_date) as day_of_week,
        EXTRACT(DAY FROM order_date) as day_of_month,
        EXTRACT(MONTH FROM order_date) as month,
        EXTRACT(QUARTER FROM order_date) as quarter,

        -- Lag features
        LAG(daily_sales, 1) OVER (PARTITION BY category ORDER BY order_date) as sales_
        LAG(daily_sales, 7) OVER (PARTITION BY category ORDER BY order_date) as sales_
        LAG(daily_sales, 30) OVER (PARTITION BY category ORDER BY order_date) as sales_

        -- Moving averages
        AVG(daily_sales) OVER (
            PARTITION BY category
            ORDER BY order_date
            ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
        ) as ma_7day,

        AVG(daily_sales) OVER (
            PARTITION BY category
            ORDER BY order_date
            ROWS BETWEEN 29 PRECEDING AND CURRENT ROW
        ) as ma_30day,

        -- Trend calculation (simple linear)
        ROW_NUMBER() OVER (PARTITION BY category ORDER BY order_date) as time_index
    FROM daily_metrics
),
seasonal_analysis AS (
    SELECT
        *,
        -- Seasonal decomposition components
        AVG(daily_sales) OVER (PARTITION BY category, month) as seasonal_monthly,
        AVG(daily_sales) OVER (PARTITION BY category, day_of_week) as seasonal_weekly,
        AVG(daily_sales) OVER (PARTITION BY category) as overall_mean,

        -- Detrended and deseasonalized values
        daily_sales - ma_30day as detrended_sales,
        daily_sales / NULLIF(seasonal_monthly, 0) as deseasonalized_sales
    FROM time_series_features
),
forecast_base AS (
    SELECT
        *,
        -- Simple forecast components
        -- Trend component (using linear regression on time_index)
        -- Note: This is a simplified approach; real forecasting would use more sophis

        -- Combine trend + seasonal for basic forecast
        ma_30day + (seasonal_monthly - overall_mean) as basic_forecast,

        -- Calculate forecast confidence based on historical volatility
        STDDEV(daily_sales) OVER (
            PARTITION BY category
            ORDER BY order_date
            ROWS BETWEEN 29 PRECEDING AND CURRENT ROW
        ) as volatility_30day
    FROM seasonal_analysis
)
SELECT
    order_date,
    category,
    daily_sales,
    ma_7day,
    ma_30day,
    ROUND(basic_forecast, 2) as basic_forecast,
    ROUND(volatility_30day, 2) as volatility,

    -- Forecast bounds (simple confidence intervals)
    ROUND(basic_forecast - (1.96 * volatility_30day), 2) as forecast_lower_bound,
    ROUND(basic_forecast + (1.96 * volatility_30day), 2) as forecast_upper_bound,
    -- Forecast accuracy metrics (for historical validation)
    CASE
        WHEN basic_forecast > 0 THEN
            ABS(daily_sales - basic_forecast) / basic_forecast * 100
        ELSE NULL
    END as forecast_error_pct,

    -- Trend classification
    CASE
        WHEN ma_7day > ma_30day * 1.1 THEN 'Strong Upward'
        WHEN ma_7day > ma_30day * 1.05 THEN 'Upward'
        WHEN ma_7day < ma_30day * 0.9 THEN 'Strong Downward'
        WHEN ma_7day < ma_30day * 0.95 THEN 'Downward'
        ELSE 'Stable'
    END as trend_direction
FROM forecast_base
WHERE order_date >= '2021-01-01'  -- Focus on recent data for better forecast quality
ORDER BY category, order_date;
```

📊 Production-Ready Query Templates

1. Automated Data Quality Monitoring

```
sql
```

```sql
-- Comprehensive data quality monitoring system
CREATE OR REPLACE FUNCTION run_data_quality_checks()
RETURNS TABLE(
    check_date DATE,
    table_name TEXT,
    check_type TEXT,
    check_description TEXT,
    expected_result TEXT,
    actual_result TEXT,
    status TEXT,
    severity TEXT
) AS $
BEGIN
    RETURN QUERY

    -- Completeness checks
    SELECT
        CURRENT_DATE as check_date,
        'superstore'::TEXT as table_name,
        'COMPLETENESS'::TEXT as check_type,
        'Order ID completeness'::TEXT as check_description,
        '100%'::TEXT as expected_result,
        ROUND((COUNT(order_id)::DECIMAL / COUNT(*)) * 100, 2)::TEXT || '%' as actual_r
        CASE WHEN COUNT(order_id) = COUNT(*) THEN 'PASS' ELSE 'FAIL' END as status,
        'HIGH'::TEXT as severity
    FROM superstore

    UNION ALL

    -- Validity checks
    SELECT
        CURRENT_DATE,
        'superstore'::TEXT,
        'VALIDITY'::TEXT,
        'Sales amount validity (positive values)',
        '100%'::TEXT,
        ROUND((COUNT(CASE WHEN sales > 0 THEN 1 END)::DECIMAL / COUNT(*)) * 100, 2)::T
        CASE WHEN COUNT(CASE WHEN sales <= 0 THEN 1 END) = 0 THEN 'PASS' ELSE 'FAIL' E
        'HIGH'::TEXT
    FROM superstore

    UNION ALL
    -- Consistency checks
    SELECT
        CURRENT_DATE,
        'superstore'::TEXT,
        'CONSISTENCY'::TEXT,
        'Order date before ship date',
        '100%'::TEXT,
        ROUND((COUNT(CASE WHEN order_date <= ship_date THEN 1 END)::DECIMAL / COUNT(*)
        CASE WHEN COUNT(CASE WHEN order_date > ship_date THEN 1 END) = 0 THEN 'PASS' E
        'MEDIUM'::TEXT
    FROM superstore

    UNION ALL

    -- Uniqueness checks
    SELECT
        CURRENT_DATE,
        'superstore'::TEXT,
        'UNIQUENESS'::TEXT,
        'Row ID uniqueness',
        '100%'::TEXT,
        CASE WHEN COUNT(*) = COUNT(DISTINCT row_id) THEN '100%'
             ELSE ROUND((COUNT(DISTINCT row_id)::DECIMAL / COUNT(*)) * 100, 2)::TEXT |
        END,
        CASE WHEN COUNT(*) = COUNT(DISTINCT row_id) THEN 'PASS' ELSE 'FAIL' END,
        'HIGH'::TEXT
    FROM superstore

    UNION ALL

    -- Timeliness checks
    SELECT
        CURRENT_DATE,
        'superstore'::TEXT,
        'TIMELINESS'::TEXT,
        'Recent data availability (last 7 days)',
        'YES'::TEXT,
        CASE WHEN MAX(order_date) >= CURRENT_DATE - INTERVAL '7 days' THEN 'YES' ELSE
        CASE WHEN MAX(order_date) >= CURRENT_DATE - INTERVAL '7 days' THEN 'PASS' ELSE
        'MEDIUM'::TEXT
    FROM superstore;
END;
$ LANGUAGE plpgsql;
```

```sql
-- Execute data quality checks
SELECT * FROM run_data_quality_checks();
```

2. Performance Monitoring Queries

sql

```sql
-- Query performance monitoring
CREATE VIEW query_performance_monitor AS
WITH query_stats AS (
    SELECT
        query,
        calls,
        total_time,
        mean_time,
        stddev_time,
        rows,
        100.0 * shared_blks_hit / nullif(shared_blks_hit + shared_blks_read, 0) AS hit
        100.0 * shared_blks_dirtied / nullif(shared_blks_hit + shared_blks_read, 0) AS
    FROM pg_stat_statements
    WHERE query NOT LIKE '%pg_stat%'
      AND query NOT LIKE '%information_schema%'
      AND calls > 5
),
slow_queries AS (
    SELECT *,
        CASE
            WHEN mean_time > 1000 THEN 'SLOW'
            WHEN mean_time > 500 THEN 'MODERATE'
            ELSE 'FAST'
        END as performance_category
    FROM query_stats
)
SELECT
    LEFT(query, 100) as query_preview,
    calls,
    ROUND(total_time::numeric, 2) as total_time_ms,
    ROUND(mean_time::numeric, 2) as avg_time_ms,
    ROUND(stddev_time::numeric, 2) as stddev_time_ms,
    rows,
    ROUND(hit_percent::numeric, 2) as cache_hit_percent,
    performance_category,

    -- Recommendations
    CASE
        WHEN hit_percent < 90 THEN 'Consider adding indexes'
        WHEN mean_time > 1000 AND calls > 100 THEN 'Priority optimization target'
        WHEN stddev_time > mean_time * 2 THEN 'Inconsistent performance - investigate'
        ELSE 'Performance acceptable'
    END as recommendation
```

```
FROM slow_queries
ORDER BY total_time DESC;
```

📚 Essential Resources for Day 4

📖 Advanced SQL Learning

1. **Advanced SQL Techniques for Data Scientists**
   - Explore 7 essential advanced SQL techniques, including CTEs, window functions, and more, to streamline your data analysis process
   - Source: SQLPad.io
2. **Window Functions and CTEs Guide**
   - Advanced SQL features such as Window Functions and Common Table Expressions (CTEs) provide powerful tools for performing complex data analysis and manipulation
   - Source: CSInfo360
3. **15 Advanced SQL Concepts**
   - Common Table Expressions (CTEs) and recursive queries can also be used to calculate running totals
   - Source: Airbyte

🎥 Video Courses

1. **SQL for Data Analysis: Advanced Querying - Udemy**
   - Learn advanced data analysis with SQL, and master topics like subqueries, CTEs, window functions, and more
   - Focus: Practical application of advanced SQL techniques
2. **Advanced SQL Techniques for Data Engineering**
   - Window Functions, Common Table Expressions (CTEs), and other complex methods are used in Advanced SQL approaches for Data Engineering
   - Source: Oracle PL/SQL Tutorial

📊 Practice Datasets

1. **Primary**: Sample Superstore Dataset
   - **Link**: kaggle.com/datasets/bravehart101/sample-supermarket-dataset
   - **Use**: Complex JOINs, performance optimization
2. **Secondary**: E-Commerce Transactions

- **Link**: kaggle.com/datasets/smayanj/e-commerce-transactions-dataset
- **Use**: Large dataset performance testing

🛠️ Tools and Performance

1. **PostgreSQL Documentation**
   - **Query Optimization**: Official PostgreSQL performance tuning guide
   - **Window Functions**: Comprehensive function reference
2. **pgAdmin and Performance Tools**
   - **Query Analysis**: EXPLAIN ANALYZE for query optimization
   - **Index Management**: Index usage statistics and recommendations

---

✅ Day 4 Practical Tasks

Task 1: Advanced Window Functions (60 minutes)

- [ ] Implement customer ranking analysis with multiple ranking functions
- [ ] Create moving averages and rolling calculations for time series
- [ ] Build lag/lead analysis for customer behavior tracking
- [ ] Practice FIRST_VALUE/LAST_VALUE for cohort analysis

Task 2: Complex CTEs and Joins (75 minutes)

- [ ] Build hierarchical product category analysis
- [ ] Create recursive CTEs for organizational data
- [ ] Implement multi-layered time series analysis
- [ ] Design complex multi-table analytical queries

Task 3: Performance Optimization (45 minutes)

- [ ] Create strategic indexes for query performance
- [ ] Analyze query execution plans with EXPLAIN
- [ ] Build materialized views for heavy analytics
- [ ] Implement query performance monitoring

Task 4: Advanced Analytics (90 minutes)

- [ ] Build market basket analysis with association rules
- [ ] Create customer lifetime value analysis
- [ ] Implement cohort retention analysis
- [ ] Design time series forecasting foundations

Task 5: Production Readiness (30 minutes)

- [ ] Set up automated data quality monitoring
- [ ] Create performance monitoring views
- [ ] Document query optimization strategies
- [ ] Build reusable analytical query templates

📝 Day 4 Deliverables

1. Advanced SQL Mastery ✅

- Complex window functions for sophisticated analytics
- Multi-layered CTEs for readable, maintainable queries
- Optimized query performance with proper indexing
- Production-ready analytical query templates

2. Real Business Analytics ✅

- Customer segmentation and lifetime value analysis
- Market basket analysis with association rules
- Time series forecasting foundations
- Cohort retention and behavior analysis

3. Performance Optimization ✅

- Strategic indexing for large datasets
- Materialized views for heavy analytical workloads
- Query performance monitoring and alerting
- Execution plan analysis and optimization

4. Skills Assessment

Rate yourself after today (1-10):

- [ ] Advanced window functions: ___/10
- [ ] Complex CTEs and recursive queries: ___/10
- [ ] Multi-table JOINs and subqueries: ___/10
- [ ] Query performance optimization: ___/10
- [ ] Production SQL patterns: ___/10

5. GitHub Repository Update

```bash
# Commit your Day 4 work
git add .
git commit -m "Day 4: Advanced SQL techniques and performance optimization"
git push origin main
```

6. Learning Journal Entry

Create `day-04/learning-notes.md`:

```markdown
# Day 4: Advanced SQL - Learning Notes

## Key Concepts Mastered
- Advanced window functions for complex analytics
- Common Table Expressions (CTEs) for modular queries
- Complex JOINs and correlated subqueries
- Query performance optimization strategies
- Production-ready analytical patterns

## Advanced Techniques Learned
- Market basket analysis with association rules
- Customer lifetime value calculation
- Cohort retention analysis
- Time series forecasting foundations
- Hierarchical data processing with recursive CTEs

## Performance Optimization Skills
- Strategic indexing for analytical workloads
- Materialized views for heavy computations
- Query execution plan analysis
- Performance monitoring and alerting

## Real Business Applications
- Customer segmentation and RFM analysis
- Product recommendation systems
- Sales forecasting and trend analysis
- Data quality monitoring automation

## Production Patterns Implemented
- Automated data quality checking functions
- Performance monitoring views
- Reusable analytical query templates
- Error handling and optimization strategies

## Tomorrow's Preparation
- Review data modeling concepts
- Prepare for dimensional modeling
- Study star schema design patterns
```

🎯 Tomorrow's Preview: Day 5 - Data Modeling

What to expect:

- Dimensional modeling and star schema design
- Fact and dimension table architecture
- Data warehouse design patterns
- Normalization vs denormalization strategies
- Building scalable data models for analytics

Preparation:

- Review relational database concepts
- Understand business requirements analysis
- Prepare for hands-on data warehouse design

🎉 *Congratulations on completing Day 4! You now have advanced SQL skills that rival experienced data engineers. Tomorrow, we'll learn how to design robust data models that scale.*

Progress: 8% (4/50 days) | **Next**: Day 5 - Data Modeling | **Skills**: Python ✅ + SQL ✅ + Advanced SQL ✅