🗃️ Day 3: SQL Fundamentals for Data Engineering - Complete Guide

📚 What You'll Learn Today

- **PostgreSQL Setup** and configuration
- **Essential SQL Operations** for data engineering
- **Working with Real Kaggle Data** using SQL
- **Database Design Principles** for analytics
- **Your First Database-Driven Pipeline**

🎯 Learning Objectives

By the end of Day 3, you will:

1. Have PostgreSQL installed and configured properly
2. Master essential SQL operations for data engineering
3. Load and analyze real Kaggle datasets in PostgreSQL
4. Understand database design principles for analytics
5. Build your first database-driven data pipeline

📊 Real Kaggle Dataset for Day 3

Primary Dataset: Superstore Dataset - Dataset containing Sales & Profits of a Superstore

- **Kaggle Link**: kaggle.com/datasets/vivek468/superstore-dataset-final
- **Size**: 9,994 records
- **Format**: CSV
- **Columns**: 21 columns including Order Date, Customer details, Product info, Sales, Profit
- **Use Case**: Perfect for learning SQL JOINs, aggregations, and business analytics

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

- **Kaggle Link**: kaggle.com/datasets/bravehart101/sample-supermarket-dataset

---

🛠️ PostgreSQL Installation and Setup (30 minutes)

Step 1: Download and Install PostgreSQL

```bash
# Windows: Download from postgresql.org
# macOS: Using Homebrew
brew install postgresql

# Ubuntu/Linux
sudo apt update
sudo apt install postgresql postgresql-contrib

# Start PostgreSQL service
# macOS
brew services start postgresql

# Linux
sudo systemctl start postgresql
sudo systemctl enable postgresql
```

Step 2: Initial Configuration

```bash
# Create a database user (replace 'yourusername' with your username)
sudo -u postgres createuser --interactive yourusername

# Create a database
sudo -u postgres createdb dataengineering

# Connect to PostgreSQL
psql -d dataengineering
```

Step 3: Install pgAdmin (GUI Tool)

- Download from: pgadmin.org
- Alternative: Use command line `psql` tool
- VS Code Extension: PostgreSQL by ckolkman

Step 4: Connection Setup

```sql
-- Test your connection
SELECT version();

-- Create sample schema
CREATE SCHEMA IF NOT EXISTS analytics;
```

📁 Loading Kaggle Dataset into PostgreSQL

Step 1: Download the Superstore Dataset

```bash
# Using Kaggle API
kaggle datasets download -d vivek468/superstore-dataset-final --unzip

# This downloads: Sample - Superstore.csv
```

Step 2: Examine the Dataset Structure

```python
import pandas as pd

# Load and examine the dataset
df = pd.read_csv('Sample - Superstore.csv')
print("Dataset Info:")
print(f"Shape: {df.shape}")
print(f"Columns: {df.columns.tolist()}")
print("\nData types:")
print(df.dtypes)
print("\nSample data:")
print(df.head())
```

Step 3: Create PostgreSQL Table

```sql
-- Create the superstore table
CREATE TABLE IF NOT EXISTS superstore (
    row_id SERIAL PRIMARY KEY,
    order_id VARCHAR(50),
    order_date DATE,
    ship_date DATE,
    ship_mode VARCHAR(50),
    customer_id VARCHAR(50),
    customer_name VARCHAR(100),
    segment VARCHAR(50),
    country VARCHAR(50),
    city VARCHAR(100),
    state VARCHAR(50),
    postal_code VARCHAR(20),
    region VARCHAR(50),
    product_id VARCHAR(50),
    category VARCHAR(50),
    sub_category VARCHAR(50),
    product_name TEXT,
    sales DECIMAL(10,2),
    quantity INTEGER,
    discount DECIMAL(5,2),
    profit DECIMAL(10,2)
);
```

Step 4: Load Data into PostgreSQL

```sql
-- Load data from CSV (adjust path as needed)
COPY superstore(
    order_id, order_date, ship_date, ship_mode, customer_id,
    customer_name, segment, country, city, state, postal_code,
    region, product_id, category, sub_category, product_name,
    sales, quantity, discount, profit
)
FROM '/path/to/Sample - Superstore.csv'
DELIMITER ','
CSV HEADER;

-- Verify data load
SELECT COUNT(*) FROM superstore;
-- Should return 9,994 records
```

🔍 Essential SQL Operations for Data Engineering

1. Data Exploration and Validation

```sql
-- Basic data exploration
SELECT
    COUNT(*) as total_records,
    COUNT(DISTINCT customer_id) as unique_customers,
    COUNT(DISTINCT product_id) as unique_products,
    MIN(order_date) as earliest_order,
    MAX(order_date) as latest_order
FROM superstore;

-- Data quality checks
SELECT
    'order_id' as column_name,
    COUNT(*) as total_count,
    COUNT(order_id) as non_null_count,
    COUNT(*) - COUNT(order_id) as null_count
FROM superstore
UNION ALL
SELECT
    'customer_name',
    COUNT(*),
    COUNT(customer_name),
    COUNT(*) - COUNT(customer_name)
FROM superstore;

-- Check for duplicates
SELECT
    order_id,
    product_id,
    COUNT(*) as duplicate_count
FROM superstore
GROUP BY order_id, product_id
HAVING COUNT(*) > 1;
```

2. Business Analytics Queries

```sql
-- Sales performance by category
SELECT
    category,
    COUNT(*) as order_count,
    SUM(sales) as total_sales,
    AVG(sales) as avg_order_value,
    SUM(profit) as total_profit,
    AVG(profit/sales) * 100 as profit_margin_pct
FROM superstore
GROUP BY category
ORDER BY total_sales DESC;

-- Top performing customers
SELECT
    customer_name,
    COUNT(DISTINCT order_id) as total_orders,
    SUM(sales) as total_spent,
    AVG(sales) as avg_order_value,
    SUM(profit) as profit_generated
FROM superstore
GROUP BY customer_name
ORDER BY total_spent DESC
LIMIT 10;

-- Monthly sales trends
SELECT
    DATE_TRUNC('month', order_date) as month,
    COUNT(DISTINCT order_id) as orders,
    SUM(sales) as monthly_sales,
    SUM(profit) as monthly_profit
FROM superstore
GROUP BY DATE_TRUNC('month', order_date)
ORDER BY month;
```

3. Advanced SQL for Data Engineering

sql

```sql
-- Window functions for analytics
SELECT
    customer_name,
    order_date,
    sales,
    -- Running total of sales per customer
    SUM(sales) OVER (
        PARTITION BY customer_id
        ORDER BY order_date
        ROWS UNBOUNDED PRECEDING
    ) as running_total,
    -- Rank orders by sales within each customer
    ROW_NUMBER() OVER (
        PARTITION BY customer_id
        ORDER BY sales DESC
    ) as sales_rank
FROM superstore
ORDER BY customer_name, order_date;

-- Common Table Expressions (CTEs) for complex analysis
WITH customer_metrics AS (
    SELECT
        customer_id,
        customer_name,
        COUNT(*) as total_orders,
        SUM(sales) as total_sales,
        AVG(sales) as avg_order_value,
        MAX(order_date) as last_order_date
    FROM superstore
    GROUP BY customer_id, customer_name
),
customer_segments AS (
    SELECT
        *,
        CASE
            WHEN total_sales > 10000 THEN 'High Value'
            WHEN total_sales > 5000 THEN 'Medium Value'
            ELSE 'Low Value'
        END as customer_segment
    FROM customer_metrics
)
SELECT
    customer_segment,
```

```sql
    COUNT(*) as customer_count,
    AVG(total_sales) as avg_customer_value,
    SUM(total_sales) as segment_revenue
FROM customer_segments
GROUP BY customer_segment
ORDER BY avg_customer_value DESC;
```

4. Data Engineering Specific Operations

```sql
-- Create dimension tables for analytics
CREATE TABLE dim_customers AS
SELECT DISTINCT
    customer_id,
    customer_name,
    segment,
    city,
    state,
    region,
    country
FROM superstore;

CREATE TABLE dim_products AS
SELECT DISTINCT
    product_id,
    category,
    sub_category,
    product_name
FROM superstore;

-- Create fact table
CREATE TABLE fact_sales AS
SELECT
    ROW_NUMBER() OVER () as sale_id,
    order_id,
    order_date,
    ship_date,
    customer_id,
    product_id,
    sales,
    quantity,
    discount,
    profit,
    ship_mode
FROM superstore;

-- Add indexes for performance
CREATE INDEX idx_fact_sales_customer ON fact_sales(customer_id);
CREATE INDEX idx_fact_sales_product ON fact_sales(product_id);
CREATE INDEX idx_fact_sales_date ON fact_sales(order_date);
```

🚀 Building Your First Database-Driven Pipeline

Let's create a complete ETL pipeline that processes the Superstore data:

python

```python
import pandas as pd
import psycopg2
from sqlalchemy import create_engine
import logging
from datetime import datetime

class SuperstoreETLPipeline:
    def __init__(self, db_connection_string):
        self.db_connection = db_connection_string
        self.engine = create_engine(db_connection_string)

        # Configure logging
        logging.basicConfig(level=logging.INFO)
        self.logger = logging.getLogger(__name__)

    def extract_data(self, csv_file_path):
        """Extract data from CSV file"""
        try:
            self.logger.info(f"Extracting data from {csv_file_path}")
            df = pd.read_csv(csv_file_path)
            self.logger.info(f"Extracted {len(df)} records")
            return df
        except Exception as e:
            self.logger.error(f"Error extracting data: {e}")
            raise

    def transform_data(self, df):
        """Transform and clean the data"""
        self.logger.info("Starting data transformation")

        # Data cleaning
        df_clean = df.copy()

        # Handle missing values
        df_clean = df_clean.dropna(subset=['Order ID', 'Customer ID'])

        # Standardize column names
        df_clean.columns = [col.lower().replace(' ', '_').replace('-', '_')
                           for col in df_clean.columns]

        # Convert dates
        df_clean['order_date'] = pd.to_datetime(df_clean['order_date'])
        df_clean['ship_date'] = pd.to_datetime(df_clean['ship_date'])
```

```python
        # Calculate derived metrics
        df_clean['profit_margin'] = (df_clean['profit'] / df_clean['sales']) * 100
        df_clean['days_to_ship'] = (df_clean['ship_date'] - df_clean['order_date']).dt

        # Add time dimensions
        df_clean['order_year'] = df_clean['order_date'].dt.year
        df_clean['order_month'] = df_clean['order_date'].dt.month
        df_clean['order_quarter'] = df_clean['order_date'].dt.quarter
        df_clean['order_day_of_week'] = df_clean['order_date'].dt.day_name()

        self.logger.info(f"Transformation complete. Final shape: {df_clean.shape}")
        return df_clean

    def load_data(self, df, table_name='superstore_processed'):
        """Load data into PostgreSQL"""
        try:
            self.logger.info(f"Loading data into {table_name}")
            df.to_sql(
                table_name,
                self.engine,
                if_exists='replace',
                index=False,
                method='multi'
            )
            self.logger.info(f"Successfully loaded {len(df)} records")
            return True
        except Exception as e:
            self.logger.error(f"Error loading data: {e}")
            raise

    def create_analytics_views(self):
        """Create useful views for analytics"""
        views_sql = [
            # Monthly sales summary
            """
            CREATE OR REPLACE VIEW monthly_sales_summary AS
            SELECT
                order_year,
                order_month,
                COUNT(DISTINCT order_id) as total_orders,
                COUNT(DISTINCT customer_id) as unique_customers,
                SUM(sales) as total_sales,
                SUM(profit) as total_profit,
```

```
    AVG(profit_margin) as avg_profit_margin
FROM superstore_processed
GROUP BY order_year, order_month
ORDER BY order_year, order_month;
""",

# Customer segmentation
"""
CREATE OR REPLACE VIEW customer_segmentation AS
WITH customer_stats AS (
    SELECT
        customer_id,
        customer_name,
        COUNT(*) as total_orders,
        SUM(sales) as total_sales,
        AVG(sales) as avg_order_value,
        SUM(profit) as total_profit
    FROM superstore_processed
    GROUP BY customer_id, customer_name
)
SELECT
    *,
    CASE
        WHEN total_sales >= 15000 THEN 'VIP'
        WHEN total_sales >= 5000 THEN 'High Value'
        WHEN total_sales >= 1000 THEN 'Medium Value'
        ELSE 'Low Value'
    END as customer_segment
FROM customer_stats;
""",

# Product performance
"""
CREATE OR REPLACE VIEW product_performance AS
SELECT
    category,
    sub_category,
    COUNT(*) as total_orders,
    SUM(quantity) as units_sold,
    SUM(sales) as total_revenue,
    SUM(profit) as total_profit,
    AVG(profit_margin) as avg_margin,
    AVG(discount) as avg_discount
FROM superstore_processed
```

```
            GROUP BY category, sub_category
            ORDER BY total_revenue DESC;
            """
        ]

        try:
            with self.engine.connect() as conn:
                for view_sql in views_sql:
                    conn.execute(view_sql)
                    conn.commit()
            self.logger.info("Analytics views created successfully")
        except Exception as e:
            self.logger.error(f"Error creating views: {e}")
            raise

    def generate_data_quality_report(self):
        """Generate data quality report"""
        quality_checks = [
            "SELECT COUNT(*) as total_records FROM superstore_processed",
            "SELECT COUNT(*) as null_order_ids FROM superstore_processed WHERE order_i
            "SELECT COUNT(*) as null_customer_ids FROM superstore_processed WHERE cust
            "SELECT COUNT(*) as negative_sales FROM superstore_processed WHERE sales <
            "SELECT COUNT(*) as future_dates FROM superstore_processed WHERE order_dat
        ]

        results = {}
        try:
            with self.engine.connect() as conn:
                for check in quality_checks:
                    result = conn.execute(check).fetchone()
                    check_name = check.split('as ')[1].split(' FROM')[0]
                    results[check_name] = result[0]

            self.logger.info("Data Quality Report:")
            for check, value in results.items():
                self.logger.info(f"  {check}: {value}")

            return results
        except Exception as e:
            self.logger.error(f"Error generating quality report: {e}")
            raise

    def run_pipeline(self, csv_file_path):
        """Execute the complete ETL pipeline"""
```

```
        self.logger.info("Starting Superstore ETL Pipeline")
        start_time = datetime.now()

        try:
            # Extract
            raw_data = self.extract_data(csv_file_path)

            # Transform
            processed_data = self.transform_data(raw_data)

            # Load
            self.load_data(processed_data)

            # Create analytics views
            self.create_analytics_views()

            # Generate quality report
            self.generate_data_quality_report()

            end_time = datetime.now()
            duration = end_time - start_time

            self.logger.info(f"Pipeline completed successfully in {duration}")
            return True

        except Exception as e:
            self.logger.error(f"Pipeline failed: {e}")
            return False

# Usage example
if __name__ == "__main__":
    # Database connection string
    db_connection = "postgresql://username:password@localhost:5432/dataengineering"

    # Initialize pipeline
    pipeline = SuperstoreETLPipeline(db_connection)

    # Run the pipeline
    success = pipeline.run_pipeline("Sample - Superstore.csv")

    if success:
        print("🎉 ETL Pipeline completed successfully!")
```

```
    else:
        print("❌ Pipeline failed. Check logs for details.")
```

📊 Advanced SQL Analytics Examples

1. Time Series Analysis

sql

```sql
-- Sales trend analysis with moving averages
WITH daily_sales AS (
    SELECT
        order_date,
        SUM(sales) as daily_sales,
        COUNT(DISTINCT order_id) as daily_orders
    FROM superstore_processed
    GROUP BY order_date
),
sales_with_ma AS (
    SELECT
        order_date,
        daily_sales,
        daily_orders,
        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
    FROM daily_sales
)
SELECT
    order_date,
    daily_sales,
    sales_7day_ma,
    sales_30day_ma,
    CASE
        WHEN daily_sales > sales_30day_ma * 1.2 THEN 'High'
        WHEN daily_sales < sales_30day_ma * 0.8 THEN 'Low'
        ELSE 'Normal'
    END as sales_performance
FROM sales_with_ma
ORDER BY order_date;
```

2. Customer Cohort Analysis

```sql
-- Customer cohort analysis
WITH customer_orders AS (
    SELECT
        customer_id,
        order_date,
        sales,
        ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order_date) as order_numb
    FROM superstore_processed
),
first_orders AS (
    SELECT
        customer_id,
        order_date as first_order_date,
        DATE_TRUNC('month', order_date) as cohort_month
    FROM customer_orders
    WHERE order_number = 1
),
customer_activities AS (
    SELECT
        f.cohort_month,
        DATE_TRUNC('month', c.order_date) as activity_month,
        COUNT(DISTINCT c.customer_id) as active_customers
    FROM first_orders f
    JOIN customer_orders c ON f.customer_id = c.customer_id
    GROUP BY f.cohort_month, DATE_TRUNC('month', c.order_date)
)
SELECT
    cohort_month,
    activity_month,
    active_customers,
    EXTRACT(MONTH FROM AGE(activity_month, cohort_month)) as months_since_first_order
FROM customer_activities
ORDER BY cohort_month, activity_month;
```

3. Product Recommendation Analysis

```sql
-- Market basket analysis (products frequently bought together)
WITH order_products AS (
    SELECT
        order_id,
        product_name,
        category
    FROM superstore_processed
),
product_pairs AS (
    SELECT
        a.product_name as product_a,
        b.product_name as product_b,
        COUNT(*) as frequency
    FROM order_products a
    JOIN order_products b ON a.order_id = b.order_id
    WHERE a.product_name < b.product_name
    GROUP BY a.product_name, b.product_name
    HAVING COUNT(*) >= 5
)
SELECT
    product_a,
    product_b,
    frequency,
    RANK() OVER (ORDER BY frequency DESC) as ranking
FROM product_pairs
ORDER BY frequency DESC
LIMIT 20;
```

📈 Performance Optimization for Data Engineering

1. Indexing Strategy

sql

```sql
-- Create indexes for common query patterns
CREATE INDEX CONCURRENTLY idx_superstore_customer_date
ON superstore_processed(customer_id, order_date);

CREATE INDEX CONCURRENTLY idx_superstore_product_category
ON superstore_processed(category, sub_category);

CREATE INDEX CONCURRENTLY idx_superstore_sales_range
ON superstore_processed(sales) WHERE sales > 1000;

-- Partial index for recent data
CREATE INDEX CONCURRENTLY idx_superstore_recent_orders
ON superstore_processed(order_date, sales)
WHERE order_date >= '2020-01-01';
```

2. Query Optimization

sql

```sql
-- Optimized query with proper joins and filtering
EXPLAIN ANALYZE
SELECT
    c.customer_name,
    c.segment,
    p.category,
    SUM(f.sales) as total_sales,
    COUNT(*) as order_count
FROM fact_sales f
JOIN dim_customers c ON f.customer_id = c.customer_id
JOIN dim_products p ON f.product_id = p.product_id
WHERE f.order_date >= '2023-01-01'
    AND c.segment = 'Consumer'
    AND p.category = 'Technology'
GROUP BY c.customer_name, c.segment, p.category
HAVING SUM(f.sales) > 1000
ORDER BY total_sales DESC;
```

3. Materialized Views for Analytics

```sql
-- Create materialized view for heavy aggregations
CREATE MATERIALIZED VIEW mv_monthly_product_performance AS
SELECT
    DATE_TRUNC('month', order_date) as month,
    category,
    sub_category,
    COUNT(*) as order_count,
    SUM(sales) as total_sales,
    SUM(profit) as total_profit,
    AVG(profit/sales) as avg_margin
FROM superstore_processed
GROUP BY DATE_TRUNC('month', order_date), category, sub_category;

-- Create index on materialized view
CREATE INDEX idx_mv_monthly_product_month_category
ON mv_monthly_product_performance(month, category);

-- Refresh materialized view (run daily)
REFRESH MATERIALIZED VIEW mv_monthly_product_performance;
```

📚 Essential Resources for Day 3

📖 PostgreSQL Documentation and Learning

1. **PostgreSQL Official Tutorial**
    - Source: postgresql.org/docs
    - PostgreSQL is an advanced and open-source relational database management system and is used as a database for many web applications, mobile and analytics applications
2. **PostgreSQL Tutorial (PostgreSQLTutorial.com)**
    - Source: postgresqltutorial.com
    - This website provides you with everything you need to know to get started with PostgreSQL quickly and effectively
3. **W3Schools PostgreSQL Tutorial**
    - Source: w3schools.com/postgresql
    - In this tutorial you get a step by step guide on how to install and create a PostgreSQL database

🎥 Video Courses

1. **SQL and PostgreSQL: The Complete Developer's Guide - Udemy**
    - This is the only course online that will teach you how to design a database, store complex data, optimize your queries, everything that is needed for operating a production, scalable database
2. **Creating PostgreSQL Databases - DataCamp**
    - Learn to create PostgreSQL databases with this four-hour course. You'll explore the structure, data types, and normalization of databases using PostgreSQL

📊 Practice Datasets

1. **Primary Dataset**: Superstore Dataset (Day 3)
    - **Source**: kaggle.com/datasets/vivek468/superstore-dataset-final
    - **Records**: 9,994 sales transactions
    - **Use Case**: Complete SQL learning from basics to advanced analytics
2. **Alternative Datasets**:
    - **Sample Superstore**: kaggle.com/datasets/bravehart101/sample-supermarket-dataset
    - **Tableau Superstore**: kaggle.com/datasets/truongdai/tableau-sample-superstore

🛠️ Tools and Software

1. **PostgreSQL Installation**
    - **Windows/Mac**: postgresql.org/download
    - **Docker**: `docker run --name postgres -e POSTGRES_PASSWORD=password -p 5432:5432 -d postgres`
2. **GUI Tools**
    - **pgAdmin**: pgadmin.org
    - **DBeaver**: dbeaver.io (Free, multi-platform)
    - **VS Code Extension**: PostgreSQL by ckolkman
3. **Python Libraries**
    - **psycopg2**: PostgreSQL adapter for Python
    - **SQLAlchemy**: SQL toolkit and ORM
    - **pandas**: Data manipulation with SQL integration

✅ Day 3 Practical Tasks

Task 1: Environment Setup (45 minutes)

- [ ] Install PostgreSQL and pgAdmin
- [ ] Create database and user
- [ ] Test connection with sample query
- [ ] Download Superstore dataset from Kaggle

Task 2: Data Loading (30 minutes)

- [ ] Create superstore table with proper schema
- [ ] Load CSV data using COPY command
- [ ] Verify data integrity and record count
- [ ] Run basic data exploration queries

Task 3: SQL Fundamentals Practice (60 minutes)

- [ ] Write SELECT queries with filtering and sorting
- [ ] Practice GROUP BY and aggregate functions
- [ ] Create JOINs between related data
- [ ] Use window functions for analytics

Task 4: Advanced Analytics (45 minutes)

- [ ] Build customer segmentation analysis
- [ ] Create time series sales trends
- [ ] Implement cohort analysis
- [ ] Generate business insights

Task 5: Pipeline Integration (30 minutes)

- [ ] Connect Python to PostgreSQL
- [ ] Run the ETL pipeline script
- [ ] Create analytics views
- [ ] Generate data quality reports

🎯 Common SQL Patterns for Data Engineering

1. Data Quality Checks

```sql
-- Comprehensive data quality assessment
SELECT
    'Total Records' as metric,
    COUNT(*)::text as value
FROM superstore
UNION ALL
SELECT
    'Null Order IDs',
    COUNT(*)::text
FROM superstore WHERE order_id IS NULL
UNION ALL
SELECT
    'Duplicate Records',
    (COUNT(*) - COUNT(DISTINCT (order_id, product_id, customer_id)))::text
FROM superstore
UNION ALL
SELECT
    'Invalid Dates',
    COUNT(*)::text
FROM superstore WHERE order_date > ship_date;
```

2. ETL Validation Queries

```sql
-- Source vs Target validation
WITH source_stats AS (
    SELECT
        COUNT(*) as record_count,
        SUM(sales) as total_sales,
        COUNT(DISTINCT customer_id) as unique_customers
    FROM superstore
),
target_stats AS (
    SELECT
        COUNT(*) as record_count,
        SUM(sales) as total_sales,
        COUNT(DISTINCT customer_id) as unique_customers
    FROM superstore_processed
)
SELECT
    'Records Match' as check_type,
    (s.record_count = t.record_count)::text as result
FROM source_stats s, target_stats t
UNION ALL
SELECT
    'Sales Match',
    (ABS(s.total_sales - t.total_sales) < 0.01)::text
FROM source_stats s, target_stats t;
```

3. Incremental Data Processing

```sql
-- Incremental data load pattern
CREATE TABLE IF NOT EXISTS data_load_log (
    load_id SERIAL PRIMARY KEY,
    table_name VARCHAR(100),
    load_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    records_processed INTEGER,
    max_date_processed DATE
);

-- Track last processed date
INSERT INTO data_load_log (table_name, records_processed, max_date_processed)
SELECT
    'superstore',
    COUNT(*),
    MAX(order_date)
FROM superstore_processed
WHERE order_date > (
    SELECT COALESCE(MAX(max_date_processed), '1900-01-01'::date)
    FROM data_load_log
    WHERE table_name = 'superstore'
);
```

📝 Day 3 Deliverables

1. Working PostgreSQL Environment ✅

- PostgreSQL installed and configured
- Database created with proper permissions
- GUI tool connected and functional
- Sample queries executed successfully

2. Loaded Kaggle Dataset ✅

- Superstore dataset downloaded from Kaggle
- Data loaded into PostgreSQL table
- Data integrity verified
- Basic exploration completed

3. SQL Skills Assessment

Rate yourself after today (1-10):

- [ ] Basic SQL queries: ___/10
- [ ] JOIN operations: ___/10
- [ ] Aggregate functions: ___/10
- [ ] Window functions: ___/10
- [ ] Performance optimization: ___/10

4. Analytics Deliverables

Create these queries and save results:

- [ ] Top 10 customers by revenue
- [ ] Monthly sales trends
- [ ] Product category performance
- [ ] Customer segmentation analysis
- [ ] Data quality report

5. GitHub Repository Update

```bash
# Commit your Day 3 work
git add .
git commit -m "Day 3: SQL fundamentals and PostgreSQL setup"
git push origin main
```

6. Learning Journal Entry

Create `day-03/learning-notes.md`:

```markdown
# Day 3: SQL Fundamentals - Learning Notes

## Key Concepts Mastered
- PostgreSQL installation and configuration
- Essential SQL operations for data engineering
- Loading real Kaggle datasets into databases
- Advanced analytics with window functions and CTEs
- Database-driven ETL pipeline development

## Practical Skills Gained
- Set up PostgreSQL development environment
- Loaded 9,994 Superstore records into database
- Created normalized dimension and fact tables
- Built complex analytical queries
- Implemented data quality validation

## Real-World Applications
- Customer segmentation and cohort analysis
- Sales performance tracking and forecasting
- Product recommendation systems
- Data warehouse design patterns

## Key SQL Patterns Learned
- Window functions for running calculations
- Common Table Expressions (CTEs) for complex logic
- Materialized views for performance optimization
- Indexing strategies for large datasets

## Challenges Overcome
- [Document specific difficulties and solutions]

## Business Insights from Superstore Data
- Technology category has highest profit margins
- Consumer segment drives most revenue
- Seasonal trends in office supplies
- Geographic performance variations

## Tomorrow's Preparation
- Review advanced SQL concepts
- Prepare for complex JOIN scenarios
- Set up additional practice datasets
```

🔗 Bonus: SQL Best Practices for Data Engineering

1. Query Organization and Readability

sql

```sql
-- Well-structured query example
WITH customer_metrics AS (
    -- Calculate customer-level metrics
    SELECT
        customer_id,
        customer_name,
        segment,
        COUNT(DISTINCT order_id) as total_orders,
        SUM(sales) as total_sales,
        AVG(sales) as avg_order_value,
        SUM(profit) as total_profit,
        MIN(order_date) as first_order,
        MAX(order_date) as last_order
    FROM superstore_processed
    WHERE order_date >= '2020-01-01'
    GROUP BY customer_id, customer_name, segment
),
customer_scoring AS (
    -- Apply business logic for customer scoring
    SELECT
        *,
        -- Recency score (days since last order)
        CASE
            WHEN last_order >= CURRENT_DATE - INTERVAL '30 days' THEN 5
            WHEN last_order >= CURRENT_DATE - INTERVAL '90 days' THEN 4
            WHEN last_order >= CURRENT_DATE - INTERVAL '180 days' THEN 3
            WHEN last_order >= CURRENT_DATE - INTERVAL '365 days' THEN 2
            ELSE 1
        END as recency_score,

        -- Frequency score (number of orders)
        CASE
            WHEN total_orders >= 20 THEN 5
            WHEN total_orders >= 10 THEN 4
            WHEN total_orders >= 5 THEN 3
            WHEN total_orders >= 2 THEN 2
            ELSE 1
        END as frequency_score,

        -- Monetary score (total sales)
        CASE
            WHEN total_sales >= 10000 THEN 5
            WHEN total_sales >= 5000 THEN 4
```

```
            WHEN total_sales >= 2000 THEN 3
            WHEN total_sales >= 500 THEN 2
            ELSE 1
        END as monetary_score
    FROM customer_metrics
)
-- Final RFM analysis
SELECT
    customer_name,
    segment,
    total_orders,
    total_sales,
    recency_score,
    frequency_score,
    monetary_score,
    (recency_score + frequency_score + monetary_score) as rfm_score,
    CASE
        WHEN (recency_score + frequency_score + monetary_score) >= 13 THEN 'Champions'
        WHEN (recency_score + frequency_score + monetary_score) >= 10 THEN 'Loyal Cust
        WHEN (recency_score + frequency_score + monetary_score) >= 7 THEN 'Potential L
        WHEN (recency_score + frequency_score + monetary_score) >= 5 THEN 'New Custome
        ELSE 'At Risk'
    END as customer_segment
FROM customer_scoring
ORDER BY rfm_score DESC, total_sales DESC;
```

2. Performance Monitoring Queries

sql

```sql
-- Query performance monitoring
SELECT
    query,
    calls,
    total_time,
    mean_time,
    rows,
    100.0 * shared_blks_hit / nullif(shared_blks_hit + shared_blks_read, 0) AS hit_per
FROM pg_stat_statements
WHERE query LIKE '%superstore%'
ORDER BY total_time DESC
LIMIT 10;

-- Table size monitoring
SELECT
    schemaname,
    tablename,
    attname as column_name,
    n_distinct,
    correlation,
    most_common_vals
FROM pg_stats
WHERE schemaname = 'public'
    AND tablename = 'superstore_processed'
ORDER BY n_distinct DESC;
```

3. Data Lineage and Documentation

sql

```sql
-- Create data lineage documentation
COMMENT ON TABLE superstore_processed IS 'Processed superstore sales data from Kaggle 
COMMENT ON COLUMN superstore_processed.customer_id IS 'Unique customer identifier. Lin
COMMENT ON COLUMN superstore_processed.profit_margin IS 'Calculated field: (profit/sal

-- Create data dictionary view
CREATE VIEW data_dictionary AS
SELECT
    t.table_name,
    c.column_name,
    c.data_type,
    c.is_nullable,
    col_description(pgc.oid, c.ordinal_position) as column_description
FROM information_schema.tables t
JOIN information_schema.columns c ON t.table_name = c.table_name
JOIN pg_class pgc ON pgc.relname = t.table_name
WHERE t.table_schema = 'public'
    AND t.table_type = 'BASE TABLE'
ORDER BY t.table_name, c.ordinal_position;
```

🎯 Advanced Data Engineering SQL Patterns

1. Slowly Changing Dimensions (SCD Type 2)

```sql
-- Implement SCD Type 2 for customer dimension
CREATE TABLE dim_customers_scd (
    surrogate_key SERIAL PRIMARY KEY,
    customer_id VARCHAR(50),
    customer_name VARCHAR(100),
    segment VARCHAR(50),
    city VARCHAR(100),
    state VARCHAR(50),
    region VARCHAR(50),
    effective_date DATE,
    expiry_date DATE,
    is_current BOOLEAN DEFAULT TRUE
);

-- Insert historical records
INSERT INTO dim_customers_scd (
    customer_id, customer_name, segment, city, state, region,
    effective_date, expiry_date, is_current
)
SELECT DISTINCT
    customer_id,
    customer_name,
    segment,
    city,
    state,
    region,
    MIN(order_date) as effective_date,
    '2999-12-31'::date as expiry_date,
    TRUE as is_current
FROM superstore_processed
GROUP BY customer_id, customer_name, segment, city, state, region;
```

2. Data Validation Framework

sql

```sql
-- Create data validation framework
CREATE TABLE data_quality_rules (
    rule_id SERIAL PRIMARY KEY,
    table_name VARCHAR(100),
    column_name VARCHAR(100),
    rule_type VARCHAR(50),
    rule_definition TEXT,
    threshold_value DECIMAL,
    is_active BOOLEAN DEFAULT TRUE
);

-- Insert validation rules
INSERT INTO data_quality_rules (table_name, column_name, rule_type, rule_definition, tl
('superstore_processed', 'sales', 'NOT_NULL', 'Sales amount should not be null', 0),
('superstore_processed', 'sales', 'POSITIVE', 'Sales amount should be positive', 0),
('superstore_processed', 'order_date', 'DATE_RANGE', 'Order date should be within reas
('superstore_processed', 'profit_margin', 'OUTLIER', 'Profit margin should be within -

-- Execute validation checks
CREATE OR REPLACE FUNCTION validate_data_quality()
RETURNS TABLE(table_name text, rule_type text, failed_records bigint) AS $
BEGIN
    RETURN QUERY
    -- Null checks
    SELECT 'superstore_processed'::text, 'NOT_NULL'::text, COUNT(*)
    FROM superstore_processed WHERE sales IS NULL
    UNION ALL
    -- Positive value checks
    SELECT 'superstore_processed'::text, 'POSITIVE'::text, COUNT(*)
    FROM superstore_processed WHERE sales <= 0
    UNION ALL
    -- Date range checks
    SELECT 'superstore_processed'::text, 'DATE_RANGE'::text, COUNT(*)
    FROM superstore_processed WHERE order_date < '1900-01-01' OR order_date > CURRENT_I
    UNION ALL
    -- Outlier checks
    SELECT 'superstore_processed'::text, 'OUTLIER'::text, COUNT(*)
    FROM superstore_processed WHERE profit_margin < -100 OR profit_margin > 500;
END;
$ LANGUAGE plpgsql;
```

```
-- Run validation
SELECT * FROM validate_data_quality();
```

3. Automated Reporting Procedures

```sql
-- Create stored procedure for daily reporting
CREATE OR REPLACE FUNCTION generate_daily_sales_report(report_date DATE DEFAULT CURREN
RETURNS TABLE(
    metric_name TEXT,
    metric_value TEXT
) AS $
BEGIN
    RETURN QUERY
    WITH daily_metrics AS (
        SELECT
            COUNT(DISTINCT order_id) as orders,
            COUNT(DISTINCT customer_id) as customers,
            SUM(sales) as revenue,
            SUM(profit) as profit,
            AVG(sales) as avg_order_value
        FROM superstore_processed
        WHERE order_date = report_date
    )
    SELECT 'Total Orders'::TEXT, orders::TEXT FROM daily_metrics
    UNION ALL
    SELECT 'Unique Customers'::TEXT, customers::TEXT FROM daily_metrics
    UNION ALL
    SELECT 'Total Revenue'::TEXT, TO_CHAR(revenue, 'FM$999,999,999.00') FROM daily_met
    UNION ALL
    SELECT 'Total Profit'::TEXT, TO_CHAR(profit, 'FM$999,999,999.00') FROM daily_metri
    UNION ALL
    SELECT 'Average Order Value'::TEXT, TO_CHAR(avg_order_value, 'FM$999,999.00') FROM
END;
$ LANGUAGE plpgsql;

-- Usage
SELECT * FROM generate_daily_sales_report('2023-01-15');
```

📊 Data Visualization Integration

1. Queries for Dashboards

sql

```sql
-- Dashboard KPIs query
CREATE VIEW dashboard_kpis AS
WITH current_month AS (
    SELECT
        COUNT(DISTINCT order_id) as current_orders,
        SUM(sales) as current_revenue,
        COUNT(DISTINCT customer_id) as current_customers
    FROM superstore_processed
    WHERE DATE_TRUNC('month', order_date) = DATE_TRUNC('month', CURRENT_DATE)
),
previous_month AS (
    SELECT
        COUNT(DISTINCT order_id) as prev_orders,
        SUM(sales) as prev_revenue,
        COUNT(DISTINCT customer_id) as prev_customers
    FROM superstore_processed
    WHERE DATE_TRUNC('month', order_date) = DATE_TRUNC('month', CURRENT_DATE) - INTERV
)
SELECT
    c.current_orders,
    c.current_revenue,
    c.current_customers,
    ROUND(((c.current_orders::DECIMAL - p.prev_orders) / p.prev_orders * 100), 2) as o
    ROUND(((c.current_revenue - p.prev_revenue) / p.prev_revenue * 100), 2) as revenue_
    ROUND(((c.current_customers::DECIMAL - p.prev_customers) / p.prev_customers * 100)
FROM current_month c, previous_month p;
```

2. Time Series Data for Charts

```sql
-- Time series data optimized for visualization
CREATE VIEW sales_time_series 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,
    -- 7-day moving average
    AVG(SUM(sales)) OVER (
        ORDER BY order_date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) as sales_7day_avg,
    -- Year-over-year comparison
    LAG(SUM(sales), 365) OVER (ORDER BY order_date) as sales_yoy
FROM superstore_processed
GROUP BY order_date
ORDER BY order_date;
```

✅ Day 3 Checklist

- [ ] ✅ Install and configure PostgreSQL
- [ ] ✅ Download Superstore dataset from Kaggle
- [ ] ✅ Create database schema and load data
- [ ] ✅ Master basic SQL operations (SELECT, WHERE, GROUP BY)
- [ ] ✅ Learn advanced SQL (JOINs, CTEs, Window Functions)
- [ ] ✅ Build customer segmentation analysis
- [ ] ✅ Create time series analytics
- [ ] ✅ Implement ETL pipeline with Python + SQL
- [ ] ✅ Set up data quality validation
- [ ] ✅ Create analytics views and reports
- [ ] ✅ Update GitHub repository with SQL scripts
- [ ] ✅ Document learning progress and insights

🎯 Tomorrow's Preview: Day 4 - Advanced SQL

What to expect:

- Complex JOINs and subqueries

- Advanced window functions and analytics
- SQL performance optimization
- Working with multiple related datasets
- Building a complete data warehouse schema

Preparation:

- Review today's window functions
- Practice complex JOINs
- Prepare for performance tuning exercises

🎉 *Congratulations on completing Day 3! You now have a solid SQL foundation for data engineering. Tomorrow, we'll dive deeper into advanced SQL techniques and optimization strategies.*

Progress: 6% (3/50 days) | **Next**: Day 4 - Advanced SQL | **Skills**: Python ✅ + SQL ✅