🏢 Day 13: Data Warehousing Concepts - ETL vs ELT & Modern Architecture

📚 What You'll Learn Today (Concept-First Approach)

Primary Focus: Understanding data warehouse architecture and the paradigm shift from ETL to ELT **Secondary Focus:** Hands-on dimensional modeling with star schema design **Dataset for Context:** Retail Analytics Dataset from Kaggle for comprehensive business modeling

🎯 Learning Philosophy for Day 13

"Understand the business before building the warehouse"

We'll start with data warehousing fundamentals, explore dimensional modeling principles, understand ETL vs ELT trade-offs, and design production-ready data warehouse architecture for modern analytics.

🌟 The Data Warehouse Revolution: From Chaos to Clarity

🤔 The Problem: Data Silos and Analytical Chaos

Scenario: You're working for a retail company with data scattered across multiple systems...

Without Data Warehouse (Siloed Data):

```
┌─────────────┐    ┌─────────────┐    ┌─────────────┐
│   Sales     │    │  Inventory  │    │  Customer   │
│  Database   │    │   System    │    │    CRM      │
│             │    │             │    │             │
│ - Orders    │    │ - Stock     │    │ - Profiles  │
│ - Payments  │    │ - Suppliers │    │ - Support   │
│ - Returns   │    │ - Warehouses│    │ - Marketing │
└─────────────┘    └─────────────┘    └─────────────┘

❌ Problems:
- Inconsistent data formats across systems
- No single source of truth
- Complex queries requiring multiple database joins
- Different update schedules and data freshness
- Difficult to perform cross-functional analysis
- Business users can't self-serve analytics
```

With Data Warehouse (Unified Analytics):

```
┌─────────────────────────────────────────────────────────┐
│                  DATA WAREHOUSE                          │
│                                                         │
│  ┌─────────────┐  ┌─────────────┐  ┌─────────────┐     │
│  │   Sales     │  │  Customer   │  │   Product   │     │
│  │   Facts     │  │ Dimension   │  │ Dimension   │     │
│  │             │  │             │  │             │     │
│  │ - OrderID   │  │ - CustID    │  │ - ProductID │     │
│  │ - Quantity  │  │ - Name      │  │ - Name      │     │
│  │ - Revenue   │  │ - Segment   │  │ - Category  │     │
│  │ - Date      │  │ - Location  │  │ - Brand     │     │
│  └─────────────┘  └─────────────┘  └─────────────┘     │
│                                                         │
│ ✅ Benefits:                                            │
│ - Single source of truth                               │
│ - Optimized for analytical queries                     │
│ - Consistent data models                               │
│ - Historical data preservation                         │
│ - Business-friendly schema design                      │
└─────────────────────────────────────────────────────────┘
```

💡 The Data Warehouse Solution: Analytics-First Design

Think of a Data Warehouse like this:

- **Operational Database:** Like a busy restaurant kitchen (optimized for orders)
- **Data Warehouse:** Like a corporate boardroom (optimized for decisions)

🏗️ Understanding Data Warehouse Architecture (Visual Approach)

🎨 The Modern Data Warehouse Stack

```
┌─────────────────────────────────────────────────────────┐
│                   MODERN DATA WAREHOUSE                  │
│                                                         │
│  ┌─────────────────────────────────────────────────────┐ │
│  │              PRESENTATION LAYER                     │ │
│  │  ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌─────────┐ │ │
│  │  │ Tableau  │ │ Power BI │ │  Jupyter │ │   API   │ │ │
│  │  │Dashboard │ │ Reports  │ │Notebooks │ │Endpoints│ │ │
│  │  └──────────┘ └──────────┘ └──────────┘ └─────────┘ │ │
│  └─────────────────────────────────────────────────────┘ │
│                           ▲                             │
│  ┌────────────────────────▼──────────────────────────────┐ │
│  │              DATA WAREHOUSE LAYER                   │ │
│  │  ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌─────────┐ │ │
│  │  │   Fact   │ │Dimension │ │Aggregate │ │  Data   │ │ │
│  │  │  Tables  │ │  Tables  │ │  Tables  │ │  Marts  │ │ │
│  │  └──────────┘ └──────────┘ └──────────┘ └─────────┘ │ │
│  └─────────────────────────────────────────────────────┘ │
│                           ▲                             │
│  ┌────────────────────────▼──────────────────────────────┐ │
│  │              INTEGRATION LAYER                      │ │
│  │  ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌─────────┐ │ │
│  │  │   ETL    │ │   ELT    │ │  Data    │ │Staging  │ │ │
│  │  │Pipelines │ │Processes │ │Quality   │ │ Areas   │ │ │
│  │  └──────────┘ └──────────┘ └──────────┘ └─────────┘ │ │
│  └─────────────────────────────────────────────────────┘ │
│                           ▲                             │
│  ┌────────────────────────▼──────────────────────────────┐ │
│  │                DATA SOURCES                         │ │
│  │  ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌─────────┐ │ │
│  │  │   OLTP   │ │   APIs   │ │  Files   │ │Streaming│ │ │
│  │  │Databases │ │  REST    │ │CSV/JSON  │ │ Kafka   │ │ │
│  │  └──────────┘ └──────────┘ └──────────┘ └─────────┘ │ │
│  └─────────────────────────────────────────────────────┘ │
└─────────────────────────────────────────────────────────┘
```

🧠 Key Data Warehouse Components

1. Source Systems (OLTP):

- Operational databases optimized for transactions
- Real-time applications and user interfaces

- Normalized structure for data integrity

2. Staging Area:

- Temporary storage for raw extracted data
- Data quality checks and cleansing
- Error handling and logging

3. Data Warehouse (OLAP):

- Optimized for analytical queries
- Denormalized structure for query performance
- Historical data preservation

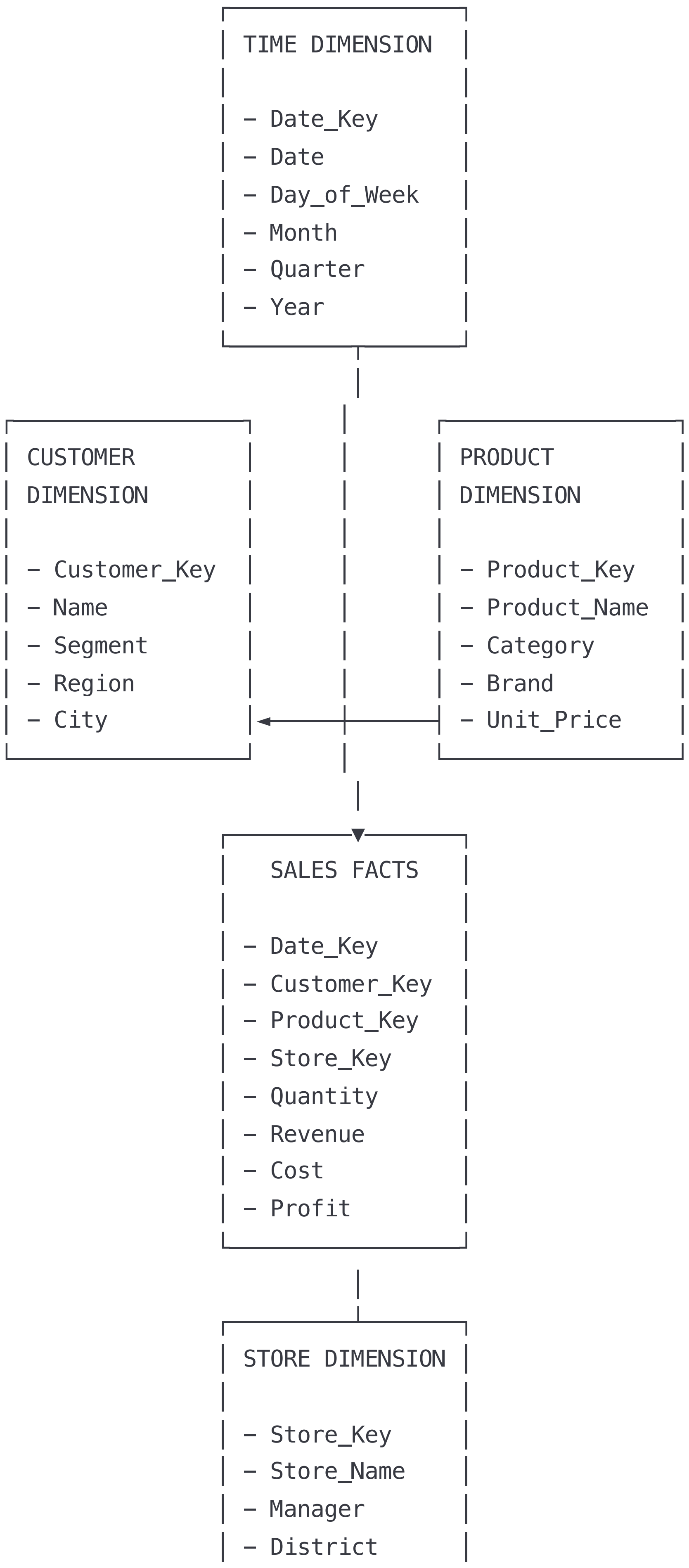
4. Data Marts:

- Subject-specific subsets of warehouse
- Department or function-focused views
- Simplified access for business users

📊 Dimensional Modeling Fundamentals

🧩 Understanding Facts and Dimensions

The Star Schema Pattern:


TIME DIMENSION
- Date_Key
- Date
- Day_of_Week
- Month
- Quarter
- Year
CUSTOMER DIMENSION
- Customer_Key
- Name
- Segment
- Region
- City
PRODUCT DIMENSION
- Product_Key
- Product_Name
- Category
- Brand
- Unit_Price
SALES FACTS
- Date_Key
- Customer_Key
- Product_Key
- Store_Key
- Quantity
- Revenue
- Cost
- Profit
STORE DIMENSION
- Store_Key
- Store_Name
- Manager
- District

```
                           │ - Square_Feet  │
                           └────────────────┘
```

Fact Table Characteristics:

- Contains measurable business events
- Numeric, additive measures (sales, quantities, costs)
- Foreign keys to dimension tables
- Grain defines the level of detail

Dimension Table Characteristics:

- Contains descriptive attributes
- Provides context for facts
- Slowly changing over time
- Denormalized for query performance

🔄 ETL vs ELT: The Great Paradigm Shift

⚖️ Understanding the Fundamental Difference

ETL (Extract, Transform, Load) - Traditional Approach:

```
┌─────────────┐    ┌─────────────┐    ┌─────────────┐
│   SOURCE    │    │   ETL       │    │    DATA     │
│   SYSTEMS   │───▶│   SERVER    │───▶│  WAREHOUSE  │
│             │    │             │    │             │
│ - Orders    │    │ Transform:  │    │ - Facts     │
│ - Customers │    │ • Clean     │    │ - Dims      │
│ - Products  │    │ • Join      │    │ - Aggs      │
│             │    │ • Aggregate │    │             │
└─────────────┘    └─────────────┘    └─────────────┘

Characteristics:
✅ Data validated before loading
✅ Lower storage requirements
✅ Consistent data quality
❌ Limited by ETL server capacity
❌ Longer development cycles
❌ Less flexibility for new analyses
```

ELT (Extract, Load, Transform) - Modern Approach:

```
┌─────────────┐    ┌─────────────┐    ┌─────────────┐
│   SOURCE    │    │    DATA     │    │  ANALYTICS  │
│   SYSTEMS   │───▶│  WAREHOUSE  │───▶│    LAYER    │
│             │    │             │    │             │
│ - Orders    │    │ Raw Data:   │    │ - Views     │
│ - Customers │    │ • All fields│    │ - Models    │
│ - Products  │    │ • Full hist │    │ - Reports   │
│             │    │ • JSON docs │    │             │
└─────────────┘    └─────────────┘    └─────────────┘
                          │                 ▲
                          └─── Transform ───┘

Characteristics:
✅ Leverage cloud warehouse power
✅ Faster time to insights
✅ Schema flexibility
✅ Better for exploratory analytics
❌ Higher storage costs
❌ Requires data governance
```

🎯 ETL vs ELT Decision Framework

Choose ETL when:

- Limited cloud warehouse budget
- Strict data quality requirements
- Complex business logic transformations
- Regulatory compliance needs
- Smaller data volumes

Choose ELT when:

- Cloud-native architecture
- Large data volumes
- Frequent schema changes
- Self-service analytics requirements
- Need for raw data exploration

🚀 Hands-On Data Warehouse Design

📱 Setting Up PostgreSQL for Data Warehousing

Step 1: Create Data Warehouse Environment

```yaml
# docker-compose.yml
version: '3.8'
services:
  postgres-dw:
    image: postgres:15
    container_name: retail-warehouse
    environment:
      POSTGRES_DB: retail_dw
      POSTGRES_USER: dw_admin
      POSTGRES_PASSWORD: warehouse123
    ports:
      - "5432:5432"
    volumes:
      - postgres_dw_data:/var/lib/postgresql/data
      - ./init-scripts:/docker-entrypoint-initdb.d/

  pgadmin:
    image: dpage/pgadmin4:latest
    container_name: pgadmin-dw
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@warehouse.com
      PGADMIN_DEFAULT_PASSWORD: admin123
    ports:
      - "8080:80"
    depends_on:
      - postgres-dw

volumes:
  postgres_dw_data:
```

Step 2: Download Retail Analytics Dataset

Dataset Source: Retail Analytics Dataset on Kaggle

Files in Dataset:

- `sales_data.csv` - Transaction-level sales data

- `customer_data.csv` - Customer demographics and segments
- `product_data.csv` - Product catalog with categories
- `store_data.csv` - Store information and geography

📊 Designing the Star Schema

Step 3: Create Dimensional Model

```
sql
```

```sql
-- init-scripts/01_create_schema.sql

-- Create schemas for different layers
CREATE SCHEMA IF NOT EXISTS staging;
CREATE SCHEMA IF NOT EXISTS warehouse;
CREATE SCHEMA IF NOT EXISTS mart;

-- Set search path
SET search_path TO warehouse;

-- ============================
-- DIMENSION TABLES
-- ============================

-- Date Dimension (most important dimension)
CREATE TABLE dim_date (
    date_key INTEGER PRIMARY KEY,
    date_actual DATE NOT NULL,
    day_of_week INTEGER NOT NULL,
    day_name VARCHAR(10) NOT NULL,
    day_of_month INTEGER NOT NULL,
    day_of_year INTEGER NOT NULL,
    week_of_year INTEGER NOT NULL,
    month_number INTEGER NOT NULL,
    month_name VARCHAR(10) NOT NULL,
    quarter_number INTEGER NOT NULL,
    quarter_name VARCHAR(2) NOT NULL,
    year_number INTEGER NOT NULL,
    is_weekend BOOLEAN NOT NULL,
    is_holiday BOOLEAN DEFAULT FALSE,
    fiscal_year INTEGER,
    fiscal_quarter INTEGER,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Customer Dimension
CREATE TABLE dim_customer (
    customer_key SERIAL PRIMARY KEY,
    customer_id VARCHAR(50) NOT NULL UNIQUE,
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    full_name VARCHAR(200),
    email VARCHAR(255),
    phone VARCHAR(20),
    gender VARCHAR(10),
    age_group VARCHAR(20),
    birth_date DATE,

    -- Geographic attributes
    address_line1 VARCHAR(255),
    address_line2 VARCHAR(255),
    city VARCHAR(100),
    state VARCHAR(100),
    postal_code VARCHAR(20),
    country VARCHAR(100),
    region VARCHAR(100),

    -- Behavioral attributes
    customer_segment VARCHAR(50),
    loyalty_tier VARCHAR(20),
    registration_date DATE,
    first_purchase_date DATE,

    -- SCD Type 2 fields
    effective_date DATE NOT NULL DEFAULT CURRENT_DATE,
    expiry_date DATE DEFAULT '9999-12-31',
    is_current BOOLEAN DEFAULT TRUE,

    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Product Dimension
CREATE TABLE dim_product (
    product_key SERIAL PRIMARY KEY,
    product_id VARCHAR(50) NOT NULL,
    product_name VARCHAR(255) NOT NULL,
    product_description TEXT,

    -- Product hierarchy
    category_l1 VARCHAR(100),  -- Electronics
    category_l2 VARCHAR(100),  -- Mobile Phones
    category_l3 VARCHAR(100),  -- Smartphones
    brand VARCHAR(100),
    manufacturer VARCHAR(100),

    -- Product attributes
    color VARCHAR(50),
    size VARCHAR(50),
    weight DECIMAL(10,2),
    unit_of_measure VARCHAR(20),

    -- Pricing information
    list_price DECIMAL(10,2),
    standard_cost DECIMAL(10,2),

    -- Product lifecycle
    launch_date DATE,
    discontinue_date DATE,
    is_active BOOLEAN DEFAULT TRUE,

    -- SCD Type 2 fields
    effective_date DATE NOT NULL DEFAULT CURRENT_DATE,
    expiry_date DATE DEFAULT '9999-12-31',
    is_current BOOLEAN DEFAULT TRUE,

    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Store Dimension
CREATE TABLE dim_store (
    store_key SERIAL PRIMARY KEY,
    store_id VARCHAR(50) NOT NULL UNIQUE,
    store_name VARCHAR(255) NOT NULL,
    store_type VARCHAR(50),  -- Mall, Street, Online

    -- Geographic information
    address_line1 VARCHAR(255),
    city VARCHAR(100),
    state VARCHAR(100),
    postal_code VARCHAR(20),
    country VARCHAR(100),
    region VARCHAR(100),
    district VARCHAR(100),

    -- Store characteristics
    store_size_sqft INTEGER,
    parking_spaces INTEGER,
    number_of_employees INTEGER,
    -- Management
    store_manager VARCHAR(200),
    district_manager VARCHAR(200),

    -- Operational dates
    opening_date DATE,
    closing_date DATE,
    is_active BOOLEAN DEFAULT TRUE,

    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- ============================
-- FACT TABLES
-- ============================

-- Sales Fact Table (Transaction Grain)
CREATE TABLE fact_sales (
    sales_key SERIAL PRIMARY KEY,

    -- Dimension Keys
    date_key INTEGER NOT NULL REFERENCES dim_date(date_key),
    customer_key INTEGER NOT NULL REFERENCES dim_customer(customer_key),
    product_key INTEGER NOT NULL REFERENCES dim_product(product_key),
    store_key INTEGER NOT NULL REFERENCES dim_store(store_key),

    -- Transaction identifiers
    transaction_id VARCHAR(100) NOT NULL,
    line_item_number INTEGER NOT NULL,

    -- Measures (Facts)
    quantity_sold INTEGER NOT NULL,
    unit_price DECIMAL(10,2) NOT NULL,
    discount_amount DECIMAL(10,2) DEFAULT 0,
    gross_amount DECIMAL(12,2) NOT NULL,
    tax_amount DECIMAL(10,2) DEFAULT 0,
    net_amount DECIMAL(12,2) NOT NULL,

    -- Cost measures
    unit_cost DECIMAL(10,2),
    total_cost DECIMAL(12,2),
    gross_profit DECIMAL(12,2),
    -- Additional attributes (degenerate dimensions)
    payment_method VARCHAR(50),
    promotion_code VARCHAR(50),
    sales_channel VARCHAR(50),

    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    -- Ensure grain uniqueness
    UNIQUE(transaction_id, line_item_number)
);

-- Daily Sales Summary Fact (Aggregate Fact)
CREATE TABLE fact_daily_sales_summary (
    summary_key SERIAL PRIMARY KEY,

    -- Dimension Keys
    date_key INTEGER NOT NULL REFERENCES dim_date(date_key),
    store_key INTEGER NOT NULL REFERENCES dim_store(store_key),

    -- Aggregated Measures
    total_transactions INTEGER NOT NULL,
    total_line_items INTEGER NOT NULL,
    total_customers INTEGER NOT NULL,
    total_quantity INTEGER NOT NULL,
    total_gross_amount DECIMAL(15,2) NOT NULL,
    total_discount_amount DECIMAL(15,2) NOT NULL,
    total_net_amount DECIMAL(15,2) NOT NULL,
    total_tax_amount DECIMAL(15,2) NOT NULL,
    total_cost DECIMAL(15,2) NOT NULL,
    total_profit DECIMAL(15,2) NOT NULL,

    -- Derived measures
    average_transaction_value DECIMAL(10,2),
    average_items_per_transaction DECIMAL(5,2),

    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    -- Ensure grain uniqueness
    UNIQUE(date_key, store_key)
);

-- Create indexes for performance
CREATE INDEX idx_fact_sales_date ON fact_sales(date_key);
CREATE INDEX idx_fact_sales_customer ON fact_sales(customer_key);
CREATE INDEX idx_fact_sales_product ON fact_sales(product_key);
CREATE INDEX idx_fact_sales_store ON fact_sales(store_key);
CREATE INDEX idx_fact_sales_transaction ON fact_sales(transaction_id);

-- Composite indexes for common query patterns
CREATE INDEX idx_fact_sales_date_store ON fact_sales(date_key, store_key);
CREATE INDEX idx_fact_sales_date_product ON fact_sales(date_key, product_key);
```

🔄 Implementing ETL Pipeline

Step 4: Create Staging Tables

```
sql
```

```sql
-- init-scripts/02_create_staging.sql

SET search_path TO staging;

-- Staging tables mirror source system structure
CREATE TABLE stg_customers (
    customer_id VARCHAR(50),
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    email VARCHAR(255),
    phone VARCHAR(20),
    gender VARCHAR(10),
    birth_date VARCHAR(20),  -- Raw date as string
    address VARCHAR(500),
    city VARCHAR(100),
    state VARCHAR(100),
    postal_code VARCHAR(20),
    country VARCHAR(100),
    segment VARCHAR(50),
    registration_date VARCHAR(20),
    file_name VARCHAR(255),
    load_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE stg_products (
    product_id VARCHAR(50),
    product_name VARCHAR(255),
    category VARCHAR(200),
    brand VARCHAR(100),
    price VARCHAR(20),  -- Raw price as string
    cost VARCHAR(20),   -- Raw cost as string
    description TEXT,
    launch_date VARCHAR(20),
    file_name VARCHAR(255),
    load_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE stg_stores (
    store_id VARCHAR(50),
    store_name VARCHAR(255),
    store_type VARCHAR(50),
    address VARCHAR(500),
    city VARCHAR(100),
    state VARCHAR(100),
    postal_code VARCHAR(20),
    size_sqft VARCHAR(20),
    manager VARCHAR(200),
    opening_date VARCHAR(20),
    file_name VARCHAR(255),
    load_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE stg_sales (
    transaction_id VARCHAR(100),
    transaction_date VARCHAR(20),
    customer_id VARCHAR(50),
    product_id VARCHAR(50),
    store_id VARCHAR(50),
    quantity VARCHAR(20),
    unit_price VARCHAR(20),
    discount VARCHAR(20),
    payment_method VARCHAR(50),
    promotion_code VARCHAR(50),
    file_name VARCHAR(255),
    load_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Step 5: Data Loading and Transformation Scripts

```
python
```

```python
# scripts/etl_pipeline.py

import pandas as pd
import psycopg2
from datetime import datetime, timedelta
import logging
from pathlib import Path

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

class RetailDataWarehouseETL:
    def __init__(self, db_config):
        self.db_config = db_config
        self.conn = None

    def connect_db(self):
        """Connect to PostgreSQL database"""
        try:
            self.conn = psycopg2.connect(**self.db_config)
            self.conn.autocommit = False
            logger.info("Database connection established")
        except Exception as e:
            logger.error(f"Database connection failed: {e}")
            raise

    def execute_sql(self, sql, params=None):
        """Execute SQL statement"""
        with self.conn.cursor() as cursor:
            cursor.execute(sql, params)
            return cursor.fetchall() if cursor.description else None

    def load_staging_data(self, data_dir):
        """Load raw data into staging tables"""
        logger.info("Starting staging data load...")

        # Load customers
        customers_df = pd.read_csv(f"{data_dir}/customer_data.csv")
        self._load_dataframe_to_table(customers_df, 'staging.stg_customers', 'customer

        # Load products
        products_df = pd.read_csv(f"{data_dir}/product_data.csv")
        self._load_dataframe_to_table(products_df, 'staging.stg_products', 'product_da

        # Load stores
        stores_df = pd.read_csv(f"{data_dir}/store_data.csv")
        self._load_dataframe_to_table(stores_df, 'staging.stg_stores', 'store_data.csv

        # Load sales
        sales_df = pd.read_csv(f"{data_dir}/sales_data.csv")
        self._load_dataframe_to_table(sales_df, 'staging.stg_sales', 'sales_data.csv')

        self.conn.commit()
        logger.info("Staging data load completed")

    def _load_dataframe_to_table(self, df, table_name, file_name):
        """Load dataframe to database table"""
        # Add metadata columns
        df['file_name'] = file_name
        df['load_timestamp'] = datetime.now()

        # Convert to lowercase column names
        df.columns = [col.lower().replace(' ', '_') for col in df.columns]

        # Load to database
        cursor = self.conn.cursor()

        # Clear existing data
        cursor.execute(f"TRUNCATE TABLE {table_name}")

        # Insert data
        columns = list(df.columns)
        placeholders = ','.join(['%s'] * len(columns))
        insert_sql = f"""
            INSERT INTO {table_name} ({','.join(columns)})
            VALUES ({placeholders})
        """

        for _, row in df.iterrows():
            cursor.execute(insert_sql, tuple(row))

        logger.info(f"Loaded {len(df)} rows into {table_name}")

    def populate_date_dimension(self, start_date='2020-01-01', end_date='2025-12-31'):
        """Populate date dimension with date range"""
        logger.info("Populating date dimension...")
```

```
cursor = self.conn.cursor()

# Clear existing data
cursor.execute("TRUNCATE TABLE warehouse.dim_date")

# Generate date range
start = datetime.strptime(start_date, '%Y-%m-%d')
end = datetime.strptime(end_date, '%Y-%m-%d')

current_date = start
while current_date <= end:
    date_key = int(current_date.strftime('%Y%m%d'))

    # Calculate date attributes
    day_of_week = current_date.weekday() + 1  # 1=Monday, 7=Sunday
    day_name = current_date.strftime('%A')
    month_name = current_date.strftime('%B')
    quarter = (current_date.month - 1) // 3 + 1
    is_weekend = current_date.weekday() >= 5

    # Fiscal year (assuming April-March)
    if current_date.month >= 4:
        fiscal_year = current_date.year
    else:
        fiscal_year = current_date.year - 1

    fiscal_quarter = ((current_date.month - 4) % 12) // 3 + 1

    insert_sql = """
        INSERT INTO warehouse.dim_date (
            date_key, date_actual, day_of_week, day_name, day_of_month,
            day_of_year, week_of_year, month_number, month_name,
            quarter_number, quarter_name, year_number, is_weekend,
            fiscal_year, fiscal_quarter
        ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
    """

    cursor.execute(insert_sql, (
        date_key, current_date.date(), day_of_week, day_name,
        current_date.day, current_date.timetuple().tm_yday,
        current_date.isocalendar()[1], current_date.month, month_name,
        quarter, f'Q{quarter}', current_date.year, is_weekend,
        fiscal_year, fiscal_quarter
```

```
            ))

            current_date += timedelta(days=1)

        self.conn.commit()
        logger.info("Date dimension populated successfully")

    def transform_and_load_dimensions(self):
        """Transform staging data and load into dimension tables"""
        logger.info("Starting dimension table loads...")

        # Load Customer Dimension
        self._load_customer_dimension()

        # Load Product Dimension
        self._load_product_dimension()

        # Load Store Dimension
        self._load_store_dimension()

        self.conn.commit()
        logger.info("Dimension tables loaded successfully")

    def _load_customer_dimension(self):
        """Load customer dimension with SCD Type 2"""
        logger.info("Loading customer dimension...")

        transform_sql = """
            INSERT INTO warehouse.dim_customer (
                customer_id, first_name, last_name, full_name, email, phone,
                gender, birth_date, city, state, postal_code, country,
                customer_segment, registration_date
            )
            SELECT DISTINCT
                customer_id,
                first_name,
                last_name,
                CONCAT(first_name, ' ', last_name) as full_name,
                email,
                phone,
                gender,
                CASE
                    WHEN birth_date ~ '^[0-9]{4}-[0-9]{2}-[0-9]{2}
                    THEN birth_date::DATE
                    ELSE NULL
                END as birth_date,
                city,
                state,
                postal_code,
                country,
                segment as customer_segment,
                CASE
                    WHEN registration_date ~ '^[0-9]{4}-[0-9]{2}-[0-9]{2}
                    THEN registration_date::DATE
                    ELSE NULL
                END as registration_date
            FROM staging.stg_customers
            WHERE customer_id IS NOT NULL
            ON CONFLICT (customer_id) DO NOTHING;
        """

        self.execute_sql(transform_sql)
        logger.info("Customer dimension loaded")

    def _load_product_dimension(self):
        """Load product dimension with hierarchical categories"""
        logger.info("Loading product dimension...")

        transform_sql = """
            INSERT INTO warehouse.dim_product (
                product_id, product_name, category_l1, brand, list_price,
                standard_cost, launch_date, product_description
            )
            SELECT DISTINCT
                product_id,
                product_name,
                SPLIT_PART(category, '>', 1) as category_l1,
                brand,
                CASE
                    WHEN price ~ '^[0-9]+\.?[0-9]*
                    THEN price::DECIMAL(10,2)
                    ELSE NULL
                END as list_price,
                CASE
                    WHEN cost ~ '^[0-9]+\.?[0-9]*
                    THEN cost::DECIMAL(10,2)
                    ELSE NULL
                END as standard_cost,
            CASE
                WHEN launch_date ~ '^[0-9]{4}-[0-9]{2}-[0-9]{2}
                THEN launch_date::DATE
                ELSE NULL
            END as launch_date,
            description as product_description
        FROM staging.stg_products
        WHERE product_id IS NOT NULL;
        """

        self.execute_sql(transform_sql)
        logger.info("Product dimension loaded")

    def _load_store_dimension(self):
        """Load store dimension"""
        logger.info("Loading store dimension...")

        transform_sql = """
        INSERT INTO warehouse.dim_store (
            store_id, store_name, store_type, city, state, postal_code,
            store_size_sqft, store_manager, opening_date
        )
        SELECT DISTINCT
            store_id,
            store_name,
            store_type,
            city,
            state,
            postal_code,
            CASE
                WHEN size_sqft ~ '^[0-9]+
                THEN size_sqft::INTEGER
                ELSE NULL
            END as store_size_sqft,
            manager as store_manager,
            CASE
                WHEN opening_date ~ '^[0-9]{4}-[0-9]{2}-[0-9]{2}
                THEN opening_date::DATE
                ELSE NULL
            END as opening_date
        FROM staging.stg_stores
        WHERE store_id IS NOT NULL;
        """
    self.execute_sql(transform_sql)
    logger.info("Store dimension loaded")

def load_fact_tables(self):
    """Load fact tables with proper dimension key lookups"""
    logger.info("Loading fact tables...")

    # Load Sales Facts
    self._load_sales_facts()

    # Generate Daily Summary Facts
    self._load_daily_summary_facts()

    self.conn.commit()
    logger.info("Fact tables loaded successfully")

def _load_sales_facts(self):
    """Load sales fact table"""
    logger.info("Loading sales fact table...")

    transform_sql = """
        INSERT INTO warehouse.fact_sales (
            date_key, customer_key, product_key, store_key,
            transaction_id, line_item_number, quantity_sold,
            unit_price, discount_amount, gross_amount, net_amount,
            payment_method, promotion_code
        )
        SELECT
            -- Date key lookup
            COALESCE(d.date_key, 19000101) as date_key,

            -- Dimension key lookups
            COALESCE(c.customer_key, -1) as customer_key,
            COALESCE(p.product_key, -1) as product_key,
            COALESCE(s.store_key, -1) as store_key,

            -- Transaction details
            st.transaction_id,
            ROW_NUMBER() OVER (PARTITION BY st.transaction_id ORDER BY st.load_tim

            -- Measures
            CASE
                WHEN st.quantity ~ '^[0-9]+
                THEN st.quantity::INTEGER
                ELSE 1
            END as quantity_sold,

            CASE
                WHEN st.unit_price ~ '^[0-9]+\.?[0-9]*
                THEN st.unit_price::DECIMAL(10,2)
                ELSE 0
            END as unit_price,

            CASE
                WHEN st.discount ~ '^[0-9]+\.?[0-9]*
                THEN st.discount::DECIMAL(10,2)
                ELSE 0
            END as discount_amount,

            -- Calculated measures
            (CASE
                WHEN st.quantity ~ '^[0-9]+ AND st.unit_price ~ '^[0-9]+\.?[0-9]*
                THEN st.quantity::INTEGER * st.unit_price::DECIMAL(10,2)
                ELSE 0
            END) as gross_amount,

            (CASE
                WHEN st.quantity ~ '^[0-9]+ AND st.unit_price ~ '^[0-9]+\.?[0-9]*
                THEN (st.quantity::INTEGER * st.unit_price::DECIMAL(10,2)) - st.di
                ELSE 0
            END) as net_amount,

            st.payment_method,
            st.promotion_code

        FROM staging.stg_sales st

        -- Date dimension lookup
        LEFT JOIN warehouse.dim_date d ON d.date_actual =
            CASE
                WHEN st.transaction_date ~ '^[0-9]{4}-[0-9]{2}-[0-9]{2}
                THEN st.transaction_date::DATE
                ELSE NULL
            END

        -- Customer dimension lookup
        LEFT JOIN warehouse.dim_customer c ON c.customer_id = st.customer_id
            AND c.is_current = TRUE
```

```
            -- Product dimension lookup
            LEFT JOIN warehouse.dim_product p ON p.product_id = st.product_id
                AND p.is_current = TRUE

            -- Store dimension lookup
            LEFT JOIN warehouse.dim_store s ON s.store_id = st.store_id

            WHERE st.transaction_id IS NOT NULL;
        """

        self.execute_sql(transform_sql)
        logger.info("Sales fact table loaded")

    def _load_daily_summary_facts(self):
        """Generate daily summary facts from transaction facts"""
        logger.info("Loading daily summary fact table...")

        transform_sql = """
            INSERT INTO warehouse.fact_daily_sales_summary (
                date_key, store_key, total_transactions, total_line_items,
                total_customers, total_quantity, total_gross_amount,
                total_discount_amount, total_net_amount,
                average_transaction_value, average_items_per_transaction
            )
            SELECT
                fs.date_key,
                fs.store_key,
                COUNT(DISTINCT fs.transaction_id) as total_transactions,
                COUNT(*) as total_line_items,
                COUNT(DISTINCT fs.customer_key) as total_customers,
                SUM(fs.quantity_sold) as total_quantity,
                SUM(fs.gross_amount) as total_gross_amount,
                SUM(fs.discount_amount) as total_discount_amount,
                SUM(fs.net_amount) as total_net_amount,
                AVG(fs.net_amount) as average_transaction_value,
                AVG(fs.quantity_sold) as average_items_per_transaction
            FROM warehouse.fact_sales fs
            GROUP BY fs.date_key, fs.store_key;
        """

        self.execute_sql(transform_sql)
        logger.info("Daily summary fact table loaded")
```

```python
    def run_full_etl(self, data_dir):
        """Run complete ETL process"""
        logger.info("Starting full ETL process...")

        try:
            self.connect_db()

            # Step 1: Load staging data
            self.load_staging_data(data_dir)

            # Step 2: Populate date dimension
            self.populate_date_dimension()

            # Step 3: Load dimension tables
            self.transform_and_load_dimensions()

            # Step 4: Load fact tables
            self.load_fact_tables()

            logger.info("ETL process completed successfully!")

        except Exception as e:
            logger.error(f"ETL process failed: {e}")
            if self.conn:
                self.conn.rollback()
            raise
        finally:
            if self.conn:
                self.conn.close()

# Usage example
if __name__ == "__main__":
    db_config = {
        'host': 'localhost',
        'port': 5432,
        'database': 'retail_dw',
        'user': 'dw_admin',
        'password': 'warehouse123'
    }

    etl = RetailDataWarehouseETL(db_config)
    etl.run_full_etl('data/retail_dataset')
```

📊 Advanced Data Warehouse Patterns

🔄 Slowly Changing Dimensions (SCD)

SCD Type 2 Implementation for Customer Changes:

```sql
-- Function to handle SCD Type 2 updates
CREATE OR REPLACE FUNCTION update_customer_scd()
RETURNS TRIGGER AS $
BEGIN
    -- Check if this is an update to an existing customer
    IF TG_OP = 'UPDATE' THEN
        -- Close the current record
        UPDATE warehouse.dim_customer
        SET expiry_date = CURRENT_DATE - 1,
            is_current = FALSE,
            updated_at = CURRENT_TIMESTAMP
        WHERE customer_id = NEW.customer_id
          AND is_current = TRUE;
    END IF;

    -- Insert new version
    INSERT INTO warehouse.dim_customer (
        customer_id, first_name, last_name, full_name,
        email, phone, city, state, customer_segment,
        effective_date, is_current
    ) VALUES (
        NEW.customer_id, NEW.first_name, NEW.last_name,
        CONCAT(NEW.first_name, ' ', NEW.last_name),
        NEW.email, NEW.phone, NEW.city, NEW.state,
        NEW.customer_segment, CURRENT_DATE, TRUE
    );

    RETURN NEW;
END;
$ LANGUAGE plpgsql;

-- Create trigger for SCD Type 2
CREATE TRIGGER customer_scd_trigger
    AFTER INSERT OR UPDATE ON staging.stg_customers
    FOR EACH ROW EXECUTE FUNCTION update_customer_scd();
```

📈 Data Mart Creation

Creating Subject-Specific Data Marts:

```
sql
```

```sql
-- Create Sales Analytics Data Mart
CREATE SCHEMA IF NOT EXISTS mart;

-- Monthly Sales Performance Mart
CREATE TABLE mart.monthly_sales_performance AS
SELECT
    d.year_number,
    d.month_number,
    d.month_name,
    s.store_name,
    s.region,
    p.category_l1,
    p.brand,

    -- Sales Metrics
    SUM(f.quantity_sold) as total_quantity,
    SUM(f.gross_amount) as total_gross_sales,
    SUM(f.discount_amount) as total_discounts,
    SUM(f.net_amount) as total_net_sales,
    COUNT(DISTINCT f.transaction_id) as total_transactions,
    COUNT(DISTINCT f.customer_key) as unique_customers,

    -- Performance Metrics
    AVG(f.net_amount) as avg_transaction_value,
    SUM(f.net_amount) / SUM(f.quantity_sold) as avg_price_per_unit,
    SUM(f.discount_amount) / SUM(f.gross_amount) * 100 as discount_percentage

FROM warehouse.fact_sales f
JOIN warehouse.dim_date d ON f.date_key = d.date_key
JOIN warehouse.dim_store s ON f.store_key = s.store_key
JOIN warehouse.dim_product p ON f.product_key = p.product_key
JOIN warehouse.dim_customer c ON f.customer_key = c.customer_key

GROUP BY
    d.year_number, d.month_number, d.month_name,
    s.store_name, s.region, p.category_l1, p.brand;

-- Customer Analytics Mart
CREATE TABLE mart.customer_analytics AS
SELECT
    c.customer_key,
    c.customer_id,
    c.full_name,
    c.customer_segment,
    c.city,
    c.state,
    c.region,

    -- Customer Behavior Metrics
    COUNT(DISTINCT f.transaction_id) as total_transactions,
    SUM(f.quantity_sold) as total_items_purchased,
    SUM(f.net_amount) as total_spent,
    AVG(f.net_amount) as avg_transaction_value,

    -- Recency, Frequency, Monetary (RFM)
    MAX(d.date_actual) as last_purchase_date,
    CURRENT_DATE - MAX(d.date_actual) as days_since_last_purchase,
    COUNT(DISTINCT d.date_actual) as purchase_frequency,
    SUM(f.net_amount) as monetary_value,

    -- Product Preferences
    MODE() WITHIN GROUP (ORDER BY p.category_l1) as preferred_category,
    MODE() WITHIN GROUP (ORDER BY p.brand) as preferred_brand,

    -- Channel Preferences
    MODE() WITHIN GROUP (ORDER BY s.store_type) as preferred_channel

FROM warehouse.dim_customer c
JOIN warehouse.fact_sales f ON c.customer_key = f.customer_key
JOIN warehouse.dim_date d ON f.date_key = d.date_key
JOIN warehouse.dim_product p ON f.product_key = p.product_key
JOIN warehouse.dim_store s ON f.store_key = s.store_key

WHERE c.is_current = TRUE
GROUP BY
    c.customer_key, c.customer_id, c.full_name,
    c.customer_segment, c.city, c.state, c.region;
```

🚀 Modern ELT Implementation

☁️ Cloud-Native ELT with dbt

dbt Models for ELT Transformation:

yaml

```yaml
# dbt_project.yml
name: 'retail_warehouse'
version: '1.0.0'
config-version: 2

model-paths: ["models"]
analysis-paths: ["analysis"]
test-paths: ["tests"]
seed-paths: ["data"]
macro-paths: ["macros"]
snapshot-paths: ["snapshots"]

target-path: "target"
clean-targets:
  - "target"
  - "dbt_packages"

models:
  retail_warehouse:
    staging:
      +materialized: view
    warehouse:
      +materialized: table
    marts:
      +materialized: table
```

Staging Models (models/staging/stg_sales.sql):

```sql
-- models/staging/stg_sales.sql
{{ config(materialized='view') }}

SELECT
    transaction_id,
    CAST(transaction_date AS DATE) as transaction_date,
    customer_id,
    product_id,
    store_id,
    CAST(quantity AS INTEGER) as quantity,
    CAST(unit_price AS DECIMAL(10,2)) as unit_price,
    CAST(COALESCE(discount, '0') AS DECIMAL(10,2)) as discount_amount,
    payment_method,
    promotion_code,
    load_timestamp

FROM {{ source('staging', 'stg_sales') }}
WHERE transaction_id IS NOT NULL
  AND transaction_date IS NOT NULL
```

Dimension Models (models/warehouse/dim_customer.sql):

sql

```sql
-- models/warehouse/dim_customer.sql
{{ config(materialized='table') }}

WITH customer_with_row_number AS (
    SELECT *,
        ROW_NUMBER() OVER (
            PARTITION BY customer_id
            ORDER BY load_timestamp DESC
        ) as rn
    FROM {{ source('staging', 'stg_customers') }}
)

SELECT
    {{ dbt_utils.generate_surrogate_key(['customer_id']) }} as customer_key,
    customer_id,
    first_name,
    last_name,
    CONCAT(first_name, ' ', last_name) as full_name,
    email,
    phone,
    gender,
    CAST(birth_date AS DATE) as birth_date,
    city,
    state,
    postal_code,
    country,
    segment as customer_segment,
    CAST(registration_date AS DATE) as registration_date,
    CURRENT_DATE as effective_date,
    DATE '9999-12-31' as expiry_date,
    TRUE as is_current,
    CURRENT_TIMESTAMP as created_at

FROM customer_with_row_number
WHERE rn = 1  -- Latest version only
  AND customer_id IS NOT NULL
```

Fact Models (models/warehouse/fact_sales.sql):

```
sql
```

```sql
-- models/warehouse/fact_sales.sql
{{ config(materialized='table') }}

SELECT
    {{ dbt_utils.generate_surrogate_key(['transaction_id', 'product_id']) }} as sales_

    -- Dimension Keys
    d.date_key,
    c.customer_key,
    p.product_key,
    s.store_key,

    -- Transaction Details
    st.transaction_id,
    ROW_NUMBER() OVER (PARTITION BY st.transaction_id ORDER BY st.product_id) as line_

    -- Measures
    st.quantity as quantity_sold,
    st.unit_price,
    st.discount_amount,
    st.quantity * st.unit_price as gross_amount,
    (st.quantity * st.unit_price) - st.discount_amount as net_amount,

    -- Additional Attributes
    st.payment_method,
    st.promotion_code,
    CURRENT_TIMESTAMP as created_at

FROM {{ ref('stg_sales') }} st

-- Dimension Lookups
LEFT JOIN {{ ref('dim_date') }} d
    ON d.date_actual = st.transaction_date

LEFT JOIN {{ ref('dim_customer') }} c
    ON c.customer_id = st.customer_id
    AND c.is_current = TRUE

LEFT JOIN {{ ref('dim_product') }} p
    ON p.product_id = st.product_id
    AND p.is_current = TRUE
```

```
  LEFT JOIN {{ ref('dim_store') }} s
      ON s.store_id = st.store_id
```

🔧 Data Quality Testing with dbt

Schema Tests (schema.yml):

yaml

```yaml
# models/schema.yml
version: 2

models:
  - name: dim_customer
    description: "Customer dimension with SCD Type 2"
    columns:
      - name: customer_key
        description: "Surrogate key for customer"
        tests:
          - unique
          - not_null
      - name: customer_id
        description: "Business key from source system"
        tests:
          - not_null
      - name: email
        description: "Customer email address"
        tests:
          - unique
          - not_null

  - name: fact_sales
    description: "Sales transaction facts"
    columns:
      - name: sales_key
        description: "Surrogate key for sales fact"
        tests:
          - unique
          - not_null
      - name: quantity_sold
        description: "Quantity of items sold"
        tests:
          - not_null
          - dbt_utils.accepted_range:
              min_value: 0
              max_value: 1000
      - name: net_amount
        description: "Net sales amount"
        tests:
          - not_null
          - dbt_utils.accepted_range:
              min_value: 0
```

```
sources:
  - name: staging
    description: "Staging area for raw data"
    tables:
      - name: stg_sales
        description: "Raw sales data"
        columns:
          - name: transaction_id
            tests:
              - not_null
              - unique
```

Custom Data Quality Tests (tests/assert_sales_balance.sql):

```sql
-- tests/assert_sales_balance.sql
-- Assert that total sales in facts match staging

SELECT
    'staging' as source,
    COUNT(*) as transaction_count,
    SUM(CAST(quantity AS INTEGER) * CAST(unit_price AS DECIMAL(10,2))) as total_amount
FROM {{ source('staging', 'stg_sales') }}

UNION ALL

SELECT
    'warehouse' as source,
    COUNT(*) as transaction_count,
    SUM(gross_amount) as total_amount
FROM {{ ref('fact_sales') }}

HAVING COUNT(DISTINCT total_amount) > 1  -- Fail if amounts don't match
```

📊 Performance Optimization Strategies

🚀 Indexing Strategy for Analytics

sql

```sql
-- Performance optimization indexes

-- Fact table indexes for common query patterns
CREATE INDEX idx_fact_sales_date_customer
    ON warehouse.fact_sales(date_key, customer_key);

CREATE INDEX idx_fact_sales_date_product
    ON warehouse.fact_sales(date_key, product_key);

CREATE INDEX idx_fact_sales_store_date
    ON warehouse.fact_sales(store_key, date_key);

-- Covering indexes for summary queries
CREATE INDEX idx_fact_sales_covering
    ON warehouse.fact_sales(date_key, store_key)
    INCLUDE (net_amount, quantity_sold);

-- Dimension table indexes for lookups
CREATE INDEX idx_dim_customer_business_key
    ON warehouse.dim_customer(customer_id, is_current);

CREATE INDEX idx_dim_product_category
    ON warehouse.dim_product(category_l1, is_current);

-- Partial indexes for active records only
CREATE INDEX idx_dim_customer_current
    ON warehouse.dim_customer(customer_key)
    WHERE is_current = TRUE;
```

📊 Partitioning for Large Tables

sql

```sql
-- Partition fact table by date for better performance
CREATE TABLE warehouse.fact_sales_partitioned (
    LIKE warehouse.fact_sales INCLUDING ALL
) PARTITION BY RANGE (date_key);

-- Create monthly partitions
CREATE TABLE warehouse.fact_sales_2024_01
    PARTITION OF warehouse.fact_sales_partitioned
    FOR VALUES FROM (20240101) TO (20240201);

CREATE TABLE warehouse.fact_sales_2024_02
    PARTITION OF warehouse.fact_sales_partitioned
    FOR VALUES FROM (20240201) TO (20240301);

-- Function to create partitions automatically
CREATE OR REPLACE FUNCTION create_monthly_partition(table_name TEXT, start_date DATE)
RETURNS VOID AS $
DECLARE
    partition_name TEXT;
    start_key INTEGER;
    end_key INTEGER;
BEGIN
    partition_name := table_name || '_' || TO_CHAR(start_date, 'YYYY_MM');
    start_key := TO_CHAR(start_date, 'YYYYMMDD')::INTEGER;
    end_key := TO_CHAR(start_date + INTERVAL '1 month', 'YYYYMMDD')::INTEGER;

    EXECUTE format('CREATE TABLE warehouse.%I PARTITION OF warehouse.%I
                    FOR VALUES FROM (%s) TO (%s)',
                   partition_name, table_name, start_key, end_key);
END;
$ LANGUAGE plpgsql;
```

📈 Materialized Views for Performance

sql

```sql
-- Create materialized views for common aggregations
CREATE MATERIALIZED VIEW warehouse.mv_monthly_sales_summary AS
SELECT
    d.year_number,
    d.month_number,
    d.month_name,
    s.region,
    p.category_l1,

    SUM(f.net_amount) as total_sales,
    SUM(f.quantity_sold) as total_quantity,
    COUNT(DISTINCT f.customer_key) as unique_customers,
    COUNT(DISTINCT f.transaction_id) as total_transactions,
    AVG(f.net_amount) as avg_transaction_value

FROM warehouse.fact_sales f
JOIN warehouse.dim_date d ON f.date_key = d.date_key
JOIN warehouse.dim_store s ON f.store_key = s.store_key
JOIN warehouse.dim_product p ON f.product_key = p.product_key

GROUP BY
    d.year_number, d.month_number, d.month_name,
    s.region, p.category_l1;

-- Create indexes on materialized view
CREATE INDEX idx_mv_monthly_sales_date
    ON warehouse.mv_monthly_sales_summary(year_number, month_number);

CREATE INDEX idx_mv_monthly_sales_region
    ON warehouse.mv_monthly_sales_summary(region);

-- Refresh materialized view (can be automated)
REFRESH MATERIALIZED VIEW warehouse.mv_monthly_sales_summary;
```

📊 Business Intelligence Query Patterns

📈 Common Analytics Queries

1. Sales Trend Analysis:

```sql
-- Monthly sales trend with year-over-year comparison
WITH monthly_sales AS (
    SELECT
        d.year_number,
        d.month_number,
        d.month_name,
        SUM(f.net_amount) as monthly_sales,
        COUNT(DISTINCT f.customer_key) as unique_customers
    FROM warehouse.fact_sales f
    JOIN warehouse.dim_date d ON f.date_key = d.date_key
    WHERE d.year_number IN (2023, 2024)
    GROUP BY d.year_number, d.month_number, d.month_name
),
sales_with_lag AS (
    SELECT *,
        LAG(monthly_sales) OVER (
            PARTITION BY month_number
            ORDER BY year_number
        ) as previous_year_sales
    FROM monthly_sales
)
SELECT
    year_number,
    month_name,
    monthly_sales,
    previous_year_sales,
    CASE
        WHEN previous_year_sales > 0
        THEN ((monthly_sales - previous_year_sales) / previous_year_sales * 100)
        ELSE NULL
    END as yoy_growth_percentage,
    unique_customers
FROM sales_with_lag
ORDER BY year_number, month_number;
```

2. Customer Cohort Analysis:

```
sql
```

```sql
-- Customer cohort analysis - retention by month
WITH customer_first_purchase AS (
    SELECT
        c.customer_key,
        MIN(d.date_actual) as first_purchase_date,
        DATE_TRUNC('month', MIN(d.date_actual)) as cohort_month
    FROM warehouse.fact_sales f
    JOIN warehouse.dim_customer c ON f.customer_key = c.customer_key
    JOIN warehouse.dim_date d ON f.date_key = d.date_key
    GROUP BY c.customer_key
),
customer_purchases AS (
    SELECT
        c.customer_key,
        cfp.cohort_month,
        DATE_TRUNC('month', d.date_actual) as purchase_month,
        EXTRACT(MONTH FROM AGE(d.date_actual, cfp.first_purchase_date)) as month_numbe
    FROM warehouse.fact_sales f
    JOIN warehouse.dim_customer c ON f.customer_key = c.customer_key
    JOIN warehouse.dim_date d ON f.date_key = d.date_key
    JOIN customer_first_purchase cfp ON c.customer_key = cfp.customer_key
),
cohort_table AS (
    SELECT
        cohort_month,
        month_number,
        COUNT(DISTINCT customer_key) as customers
    FROM customer_purchases
    GROUP BY cohort_month, month_number
),
cohort_sizes AS (
    SELECT
        cohort_month,
        COUNT(DISTINCT customer_key) as cohort_size
    FROM customer_first_purchase
    GROUP BY cohort_month
)
SELECT
    ct.cohort_month,
    cs.cohort_size,
    ct.month_number,
    ct.customers,
    ROUND(ct.customers * 100.0 / cs.cohort_size, 2) as retention_percentage
```

```sql
  FROM cohort_table ct
  JOIN cohort_sizes cs ON ct.cohort_month = cs.cohort_month
  ORDER BY ct.cohort_month, ct.month_number;
```

3. Product Performance Analysis:

sql

```sql
-- Top performing products with profitability analysis
WITH product_performance AS (
    SELECT
        p.product_key,
        p.product_name,
        p.category_l1,
        p.brand,
        SUM(f.quantity_sold) as total_quantity,
        SUM(f.net_amount) as total_revenue,
        COUNT(DISTINCT f.customer_key) as unique_customers,
        COUNT(DISTINCT f.transaction_id) as transactions,
        AVG(f.unit_price) as avg_selling_price,
        p.standard_cost,
        SUM(f.quantity_sold) * p.standard_cost as total_cost,
        SUM(f.net_amount) - (SUM(f.quantity_sold) * p.standard_cost) as total_profit
    FROM warehouse.fact_sales f
    JOIN warehouse.dim_product p ON f.product_key = p.product_key
    WHERE p.is_current = TRUE
    GROUP BY p.product_key, p.product_name, p.category_l1, p.brand, p.standard_cost
)
SELECT
    product_name,
    category_l1,
    brand,
    total_quantity,
    total_revenue,
    total_profit,
    ROUND(total_profit / total_revenue * 100, 2) as profit_margin_percentage,
    unique_customers,
    ROUND(total_revenue / unique_customers, 2) as revenue_per_customer,
    RANK() OVER (ORDER BY total_profit DESC) as profit_rank,
    RANK() OVER (ORDER BY total_revenue DESC) as revenue_rank
FROM product_performance
WHERE total_revenue > 1000  -- Filter for significant products
ORDER BY total_profit DESC
LIMIT 20;
```

4. Store Performance Dashboard:

```
sql
```

```sql
-- Comprehensive store performance metrics
WITH store_metrics AS (
    SELECT
        s.store_key,
        s.store_name,
        s.region,
        s.store_type,
        s.store_size_sqft,

        -- Sales Metrics
        SUM(f.net_amount) as total_sales,
        SUM(f.quantity_sold) as total_items_sold,
        COUNT(DISTINCT f.transaction_id) as total_transactions,
        COUNT(DISTINCT f.customer_key) as unique_customers,
        COUNT(DISTINCT d.date_actual) as operating_days,

        -- Performance Ratios
        AVG(f.net_amount) as avg_transaction_value,
        SUM(f.net_amount) / COUNT(DISTINCT d.date_actual) as sales_per_day,
        SUM(f.net_amount) / s.store_size_sqft as sales_per_sqft,
        COUNT(DISTINCT f.customer_key) / COUNT(DISTINCT d.date_actual) as customers_pe

    FROM warehouse.fact_sales f
    JOIN warehouse.dim_store s ON f.store_key = s.store_key
    JOIN warehouse.dim_date d ON f.date_key = d.date_key
    WHERE d.date_actual >= CURRENT_DATE - INTERVAL '12 months'
    GROUP BY s.store_key, s.store_name, s.region, s.store_type, s.store_size_sqft
)
SELECT
    store_name,
    region,
    store_type,
    store_size_sqft,
    total_sales,
    total_transactions,
    unique_customers,
    ROUND(avg_transaction_value, 2) as avg_transaction_value,
    ROUND(sales_per_day, 2) as daily_sales_avg,
    ROUND(sales_per_sqft, 2) as sales_per_sqft,
    ROUND(customers_per_day, 1) as daily_customers_avg,

    -- Performance Rankings
    RANK() OVER (ORDER BY total_sales DESC) as sales_rank,
    RANK() OVER (ORDER BY sales_per_sqft DESC) as efficiency_rank,
    RANK() OVER (ORDER BY avg_transaction_value DESC) as aov_rank

FROM store_metrics
ORDER BY total_sales DESC;
```

🔄 Modern Data Stack Integration

🌐 Cloud Data Warehouse Architecture

Amazon Redshift Implementation:

```sql
-- Redshift-optimized table design
CREATE TABLE warehouse.fact_sales_redshift (
    sales_key BIGINT IDENTITY(1,1),
    date_key INTEGER NOT NULL,
    customer_key INTEGER NOT NULL,
    product_key INTEGER NOT NULL,
    store_key INTEGER NOT NULL,
    transaction_id VARCHAR(100) NOT NULL,
    line_item_number SMALLINT NOT NULL,
    quantity_sold INTEGER NOT NULL,
    unit_price DECIMAL(10,2) NOT NULL,
    discount_amount DECIMAL(10,2) DEFAULT 0,
    gross_amount DECIMAL(12,2) NOT NULL,
    net_amount DECIMAL(12,2) NOT NULL,
    payment_method VARCHAR(50),
    promotion_code VARCHAR(50),
    created_at TIMESTAMP DEFAULT GETDATE()
)
DISTKEY(customer_key)  -- Distribute by customer for customer analytics
SORTKEY(date_key, store_key);  -- Sort by date and store for time-series queries

-- Redshift column encoding for compression
ALTER TABLE warehouse.fact_sales_redshift
ALTER COLUMN date_key ENCODE az64,
ALTER COLUMN customer_key ENCODE az64,
ALTER COLUMN product_key ENCODE az64,
ALTER COLUMN store_key ENCODE az64,
ALTER COLUMN quantity_sold ENCODE az64,
ALTER COLUMN payment_method ENCODE lzo,
ALTER COLUMN promotion_code ENCODE lzo;
```

Snowflake Implementation:

```sql
-- Snowflake-optimized table design with clustering
CREATE OR REPLACE TABLE warehouse.fact_sales_snowflake (
    sales_key NUMBER AUTOINCREMENT,
    date_key NUMBER NOT NULL,
    customer_key NUMBER NOT NULL,
    product_key NUMBER NOT NULL,
    store_key NUMBER NOT NULL,
    transaction_id VARCHAR(100) NOT NULL,
    line_item_number NUMBER NOT NULL,
    quantity_sold NUMBER NOT NULL,
    unit_price NUMBER(10,2) NOT NULL,
    discount_amount NUMBER(10,2) DEFAULT 0,
    gross_amount NUMBER(12,2) NOT NULL,
    net_amount NUMBER(12,2) NOT NULL,
    payment_method VARCHAR(50),
    promotion_code VARCHAR(50),
    created_at TIMESTAMP_NTZ DEFAULT CURRENT_TIMESTAMP()
)
CLUSTER BY (date_key, store_key);  -- Snowflake clustering for performance

-- Create secure views for data access
CREATE OR REPLACE SECURE VIEW warehouse.vw_sales_summary AS
SELECT
    date_key,
    store_key,
    SUM(net_amount) as total_sales,
    COUNT(DISTINCT customer_key) as unique_customers,
    COUNT(*) as total_transactions
FROM warehouse.fact_sales_snowflake
GROUP BY date_key, store_key;
```

🔧 Data Pipeline Orchestration

Airflow DAG for Data Warehouse ETL:

```
python
```

```python
# dags/retail_warehouse_etl.py
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from airflow.operators.bash_operator import BashOperator
from airflow.providers.postgres.operators.postgres import PostgresOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'data-team',
    'depends_on_past': False,
    'start_date': datetime(2024, 1, 1),
    'email_on_failure': True,
    'email_on_retry': False,
    'retries': 2,
    'retry_delay': timedelta(minutes=5)
}

dag = DAG(
    'retail_warehouse_etl',
    default_args=default_args,
    description='Retail data warehouse ETL pipeline',
    schedule_interval='0 2 * * *',  # Daily at 2 AM
    catchup=False,
    max_active_runs=1,
    tags=['warehouse', 'etl', 'retail']
)

def extract_source_data(**context):
    """Extract data from source systems"""
    from scripts.etl_pipeline import RetailDataWarehouseETL

    db_config = {
        'host': 'localhost',
        'port': 5432,
        'database': 'retail_dw',
        'user': 'dw_admin',
        'password': 'warehouse123'
    }

    etl = RetailDataWarehouseETL(db_config)
    etl.connect_db()
    etl.load_staging_data('data/retail_dataset')
    return f"Extracted data for {context['ds']}"

def transform_and_load_dimensions(**context):
    """Transform and load dimension tables"""
    from scripts.etl_pipeline import RetailDataWarehouseETL

    db_config = {
        'host': 'localhost',
        'port': 5432,
        'database': 'retail_dw',
        'user': 'dw_admin',
        'password': 'warehouse123'
    }

    etl = RetailDataWarehouseETL(db_config)
    etl.connect_db()
    etl.transform_and_load_dimensions()

    return "Dimensions loaded successfully"

def load_fact_tables(**context):
    """Load fact tables"""
    from scripts.etl_pipeline import RetailDataWarehouseETL

    db_config = {
        'host': 'localhost',
        'port': 5432,
        'database': 'retail_dw',
        'user': 'dw_admin',
        'password': 'warehouse123'
    }

    etl = RetailDataWarehouseETL(db_config)
    etl.connect_db()
    etl.load_fact_tables()

    return "Facts loaded successfully"

def run_data_quality_checks(**context):
    """Run data quality validations"""
    import psycopg2

    conn = psycopg2.connect(
        host='localhost',
        port=5432,
        database='retail_dw',
        user='dw_admin',
        password='warehouse123'
    )

    cursor = conn.cursor()

    # Check 1: Ensure no orphaned facts
    cursor.execute("""
        SELECT COUNT(*) FROM warehouse.fact_sales f
        LEFT JOIN warehouse.dim_customer c ON f.customer_key = c.customer_key
        WHERE c.customer_key IS NULL
    """)
    orphaned_customers = cursor.fetchone()[0]

    # Check 2: Ensure reasonable sales amounts
    cursor.execute("""
        SELECT COUNT(*) FROM warehouse.fact_sales
        WHERE net_amount < 0 OR net_amount > 100000
    """)
    invalid_amounts = cursor.fetchone()[0]

    conn.close()

    if orphaned_customers > 0:
        raise ValueError(f"Found {orphaned_customers} orphaned customer records")

    if invalid_amounts > 0:
        raise ValueError(f"Found {invalid_amounts} invalid sales amounts")

    return "Data quality checks passed"

# Define tasks
extract_task = PythonOperator(
    task_id='extract_source_data',
    python_callable=extract_source_data,
    dag=dag
)

transform_dimensions_task = PythonOperator(
    task_id='transform_and_load_dimensions',
    python_callable=transform_and_load_dimensions,
    dag=dag
```

```python
)

load_facts_task = PythonOperator(
    task_id='load_fact_tables',
    python_callable=load_fact_tables,
    dag=dag
)

quality_check_task = PythonOperator(
    task_id='run_data_quality_checks',
    python_callable=run_data_quality_checks,
    dag=dag
)

# Refresh materialized views
refresh_views_task = PostgresOperator(
    task_id='refresh_materialized_views',
    postgres_conn_id='postgres_warehouse',
    sql="""
        REFRESH MATERIALIZED VIEW warehouse.mv_monthly_sales_summary;
        REFRESH MATERIALIZED VIEW warehouse.mv_customer_summary;
    """,
    dag=dag
)

# dbt run for ELT transformations (if using dbt)
dbt_run_task = BashOperator(
    task_id='dbt_run_transformations',
    bash_command='cd /opt/airflow/dbt && dbt run --profiles-dir .',
    dag=dag
)

# Set task dependencies
extract_task >> transform_dimensions_task >> load_facts_task >> quality_check_task
quality_check_task >> refresh_views_task
quality_check_task >> dbt_run_task
```

🛡️ Data Governance and Security

🔐 Row-Level Security Implementation

sql

```sql
-- Create roles for different user types
CREATE ROLE warehouse_admin;
CREATE ROLE regional_manager;
CREATE ROLE store_manager;
CREATE ROLE analyst;

-- Create policy for regional access
CREATE POLICY regional_sales_policy ON warehouse.fact_sales
    FOR SELECT
    TO regional_manager
    USING (
        store_key IN (
            SELECT store_key
            FROM warehouse.dim_store
            WHERE region = current_setting('app.current_region')
        )
    );

-- Create policy for store-level access
CREATE POLICY store_sales_policy ON warehouse.fact_sales
    FOR SELECT
    TO store_manager
    USING (
        store_key = current_setting('app.current_store_key')::INTEGER
    );

-- Enable row level security
ALTER TABLE warehouse.fact_sales ENABLE ROW LEVEL SECURITY;

-- Grant permissions
GRANT SELECT ON warehouse.fact_sales TO regional_manager;
GRANT SELECT ON warehouse.fact_sales TO store_manager;
GRANT SELECT ON warehouse.fact_sales TO analyst;
```

📊 Data Lineage Tracking

sql

```sql
-- Create data lineage tracking table
CREATE TABLE warehouse.data_lineage (
    lineage_id SERIAL PRIMARY KEY,
    source_table VARCHAR(255) NOT NULL,
    target_table VARCHAR(255) NOT NULL,
    transformation_logic TEXT,
    created_by VARCHAR(100) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Insert lineage records
INSERT INTO warehouse.data_lineage
(source_table, target_table, transformation_logic, created_by) VALUES
('staging.stg_customers', 'warehouse.dim_customer',
 'Clean and standardize customer data with SCD Type 2', 'etl_system'),
('staging.stg_sales', 'warehouse.fact_sales',
 'Join with dimensions and calculate derived measures', 'etl_system'),
('warehouse.fact_sales', 'mart.monthly_sales_performance',
 'Aggregate sales by month, store, and product category', 'dbt_system');

-- Function to track data processing
CREATE OR REPLACE FUNCTION track_data_processing(
    p_source_table VARCHAR(255),
    p_target_table VARCHAR(255),
    p_records_processed INTEGER,
    p_processing_duration INTERVAL
)
RETURNS VOID AS $
BEGIN
    INSERT INTO warehouse.data_processing_log (
        source_table, target_table, records_processed,
        processing_duration, processed_at
    ) VALUES (
        p_source_table, p_target_table, p_records_processed,
        p_processing_duration, CURRENT_TIMESTAMP
    );
END;
$ LANGUAGE plpgsql;
```

📊 Performance Monitoring and Optimization

📈 Query Performance Analysis

```
sql
```

```sql
-- Create function to analyze query performance
CREATE OR REPLACE FUNCTION analyze_warehouse_performance()
RETURNS TABLE (
    table_name TEXT,
    total_size_mb NUMERIC,
    index_size_mb NUMERIC,
    seq_scans BIGINT,
    index_scans BIGINT,
    rows_read BIGINT,
    rows_fetched BIGINT
) AS $
BEGIN
    RETURN QUERY
    SELECT
        schemaname||'.'||tablename as table_name,
        ROUND(pg_total_relation_size(schemaname||'.'||tablename) / 1024.0 / 1024.0, 2)
        ROUND(pg_indexes_size(schemaname||'.'||tablename) / 1024.0 / 1024.0, 2) as ind
        seq_scan as seq_scans,
        idx_scan as index_scans,
        seq_tup_read as rows_read,
        idx_tup_fetch as rows_fetched
    FROM pg_stat_user_tables
    WHERE schemaname IN ('warehouse', 'mart')
    ORDER BY pg_total_relation_size(schemaname||'.'||tablename) DESC;
END;
$ LANGUAGE plpgsql;

-- Monitor slow queries
CREATE TABLE warehouse.slow_query_log (
    log_id SERIAL PRIMARY KEY,
    query_text TEXT NOT NULL,
    execution_time_ms INTEGER NOT NULL,
    rows_examined INTEGER,
    rows_returned INTEGER,
    logged_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Function to log slow queries (can be called from application)
CREATE OR REPLACE FUNCTION log_slow_query(
    p_query_text TEXT,
    p_execution_time_ms INTEGER,
    p_rows_examined INTEGER,
    p_rows_returned INTEGER
)
RETURNS VOID AS $
BEGIN
    IF p_execution_time_ms > 5000 THEN  -- Log queries taking more than 5 seconds
        INSERT INTO warehouse.slow_query_log
        (query_text, execution_time_ms, rows_examined, rows_returned)
        VALUES (p_query_text, p_execution_time_ms, p_rows_examined, p_rows_returned);
    END IF;
END;
$ LANGUAGE plpgsql;
```

🔧 Automated Maintenance Tasks

```sql
-- Create maintenance procedures
CREATE OR REPLACE FUNCTION warehouse_maintenance()
RETURNS VOID AS $
BEGIN
    -- Update table statistics
    ANALYZE warehouse.fact_sales;
    ANALYZE warehouse.dim_customer;
    ANALYZE warehouse.dim_product;
    ANALYZE warehouse.dim_store;

    -- Refresh materialized views
    REFRESH MATERIALIZED VIEW warehouse.mv_monthly_sales_summary;

    -- Archive old staging data (keep last 30 days)
    DELETE FROM staging.stg_sales
    WHERE load_timestamp < CURRENT_DATE - INTERVAL '30 days';

    DELETE FROM staging.stg_customers
    WHERE load_timestamp < CURRENT_DATE - INTERVAL '30 days';

    -- Clean up old log entries (keep last 90 days)
    DELETE FROM warehouse.slow_query_log
    WHERE logged_at < CURRENT_DATE - INTERVAL '90 days';

    -- Log maintenance completion
    INSERT INTO warehouse.maintenance_log (maintenance_type, completed_at)
    VALUES ('automated_maintenance', CURRENT_TIMESTAMP);

END;
$ LANGUAGE plpgsql;

-- Schedule maintenance (this would typically be done via cron or Airflow)
-- SELECT warehouse_maintenance();
```

🌐 Cloud Migration Strategies

☁️ Migration from On-Premise to Cloud

Migration Planning Framework:

python

```python
# scripts/cloud_migration_plan.py

class CloudMigrationPlanner:
    def __init__(self):
        self.migration_phases = {
            'assessment': 'Analyze current data warehouse',
            'design': 'Design cloud-native architecture',
            'pilot': 'Migrate subset of data',
            'full_migration': 'Complete data migration',
            'optimization': 'Optimize for cloud performance'
        }

    def assess_current_warehouse(self):
        """Assess current data warehouse for cloud readiness"""
        assessment = {
            'data_volume': self._calculate_data_volume(),
            'query_patterns': self._analyze_query_patterns(),
            'performance_requirements': self._assess_performance_needs(),
            'security_requirements': self._evaluate_security_needs(),
            'compliance_requirements': self._check_compliance_needs()
        }
        return assessment

    def design_cloud_architecture(self, target_platform='snowflake'):
        """Design cloud data warehouse architecture"""
        if target_platform == 'snowflake':
            return self._design_snowflake_architecture()
        elif target_platform == 'redshift':
            return self._design_redshift_architecture()
        elif target_platform == 'bigquery':
            return self._design_bigquery_architecture()

    def _design_snowflake_architecture(self):
        return {
            'compute': {
                'warehouse_sizes': ['X-SMALL', 'SMALL', 'MEDIUM'],
                'auto_suspend': 60,  # seconds
                'auto_resume': True,
                'multi_cluster': True
            },
            'storage': {
                'time_travel': 90,  # days
                'fail_safe': 7,     # days
                'compression': 'automatic'
            },
            'security': {
                'encryption': 'AES-256',
                'network_policy': 'restrictive',
                'mfa_required': True
            }
        }

    def create_migration_timeline(self):
        """Create detailed migration timeline"""
        return {
            'week_1': 'Data assessment and architecture design',
            'week_2': 'Set up cloud environment and security',
            'week_3': 'Pilot migration with sample data',
            'week_4': 'Full data migration and testing',
            'week_5': 'Performance optimization and go-live',
            'week_6': 'Monitor and fine-tune'
        }

# Usage
planner = CloudMigrationPlanner()
assessment = planner.assess_current_warehouse()
architecture = planner.design_cloud_architecture('snowflake')
timeline = planner.create_migration_timeline()
```

📚 Essential Resources for Day 13

📖 Official Documentation

- **Dimensional Modeling:** The Data Warehouse Toolkit by Ralph Kimball
- **PostgreSQL Documentation:** https://www.postgresql.org/docs/
- **dbt Documentation:** https://docs.getdbt.com/
- **Apache Airflow:** https://airflow.apache.org/docs/

🛠️ Tools and Technologies

- **PostgreSQL:** Open-source RDBMS for data warehousing
- **dbt (data build tool):** Modern ELT framework
- **Apache Airflow:** Workflow orchestration
- **Snowflake:** Cloud data warehouse platform

- **Amazon Redshift:** AWS data warehouse service
- **Google BigQuery:** Google Cloud data warehouse

📊 Sample Datasets for Practice

- **Retail Analytics:** https://www.kaggle.com/datasets/manjeetsingh/retaildataset
- **Superstore Dataset:** https://www.kaggle.com/datasets/vivek468/superstore-dataset-final
- **E-commerce Transactions:** https://www.kaggle.com/datasets/smayanj/e-commerce-transactions-dataset

🎯 Key Design Principles

Data Warehouse Design: ✅ **Business-Driven:** Start with business requirements, not technical constraints ✅ **Dimensional Modeling:** Use facts and dimensions for analytical queries ✅ **Grain Declaration:** Clearly define the level of detail in fact tables ✅ **Conformed Dimensions:** Ensure consistency across data marts ✅ **Slowly Changing Dimensions:** Handle changes in dimensional data appropriately

ETL vs ELT Decision Factors:

- **Data Volume:** ELT scales better for large datasets
- **Transformation Complexity:** ETL better for complex business logic
- **Cloud Resources:** ELT leverages cloud compute power
- **Time to Insight:** ELT typically faster for exploratory analysis
- **Cost Considerations:** Balance compute vs storage costs

Performance Optimization: ✅ **Proper Indexing:** Create indexes for common query patterns ✅ **Partitioning:** Partition large tables by date or other logical divisions ✅ **Materialized Views:** Pre-aggregate common calculations ✅ **Query Optimization:** Analyze and optimize frequently-run queries ✅ **Resource Management:** Right-size compute resources for workloads

🚀 Tomorrow's Preview: Infrastructure as Code with Terraform

Day 26 Focus: Automated infrastructure management, environment consistency, and infrastructure versioning for data platforms.

Preparation: Think about how today's data warehouse concepts apply to infrastructure management and automated deployment patterns.

💡 Key Takeaways from Day 13

Conceptual Mastery:

- Understanding the difference between OLTP and OLAP systems
- Dimensional modeling as the foundation of analytics
- ETL vs ELT paradigms and when to use each
- Data warehouse architecture layers and components

Practical Skills:

- Designing star schema for business requirements
- Implementing SCD Type 2 for historical data tracking
- Building ETL pipelines with data quality checks
- Creating performance-optimized database structures

Business Impact:

- Single source of truth for consistent reporting
- Historical data preservation for trend analysis
- Self-service analytics capabilities for business users
- Scalable architecture for growing data volumes

The comprehensive journey through data warehousing concepts provides the foundation for understanding how data flows from operational systems to business insights, setting the stage for more advanced topics in modern data engineering.