Extract - Transform - Load

Time Series Team

Ngày 12 tháng 8 năm 2025

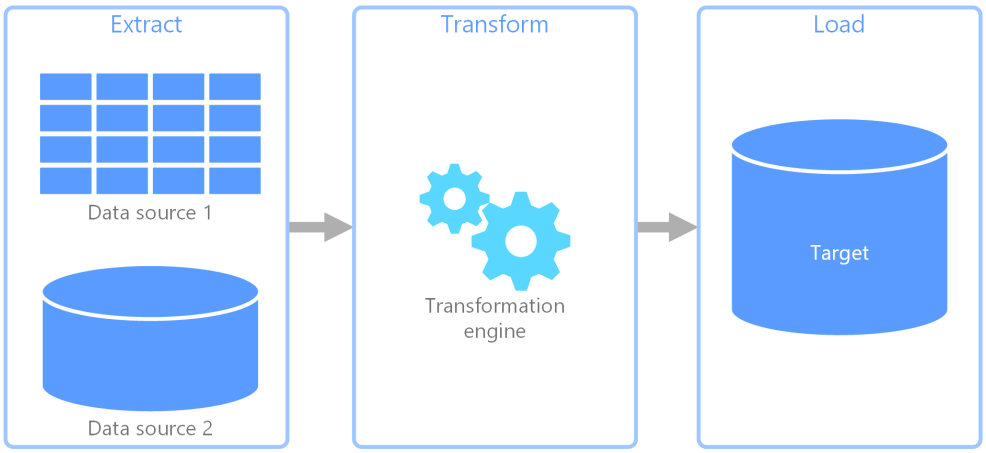
Mục lục

1. Tổng quan về ETL Pipeline

1.1 ETL là gì?

ETL (Extract, Transform, Load) là quy trình quan trọng trong Data Engineering, bao gồm ba giai đoạn chính:

- **Extract (Trích xuất)**: Thu thập dữ liệu từ nhiều nguồn khác nhau
- **Transform (Chuyển đổi)**: Làm sạch, xử lý và chuẩn hóa dữ liệu
- **Load (Tải)**: Lưu trữ dữ liệu đã xử lý vào Data Warehouse



Hình 1: Extract-Transform-Load

1.2 Tại sao cần ETL Pipeline?

Trong môi trường doanh nghiệp hiện đại, việc ra quyết định dựa trên dữ liệu (Data-driven Decision Making) đã trở thành yếu tố then chốt. Tuy nhiên, các tổ chức thường gặp phải những thách thức sau:

1. **Tốn thời gian**: Xây dựng báo cáo từ đầu, thu thập dữ liệu từ nhiều nguồn
2. **Nhiều người tham gia**: Dẫn đến sự không nhất quán và lỗi con người
3. **Khó tích hợp báo cáo**: Nhiều công cụ, nhiều thuật ngữ khác nhau
4. **Vấn đề không lường trước**: Số liệu không khớp, thông tin sai lệch

1.3 Kiến trúc dữ liệu hiện đại

ETL Pipeline là nền tảng của Business Intelligence, nằm trong hệ thống phân cấp:

- **Data Sourcing**: Thu thập dữ liệu từ các nguồn
- **Data Warehousing (ETL)**: Xử lý và lưu trữ dữ liệu
- **Data Mining**: Khai thác thông tin từ dữ liệu
- **Report Data Analysis**: Phân tích và báo cáo
- **Decision Making**: Ra quyết định dựa trên dữ liệu

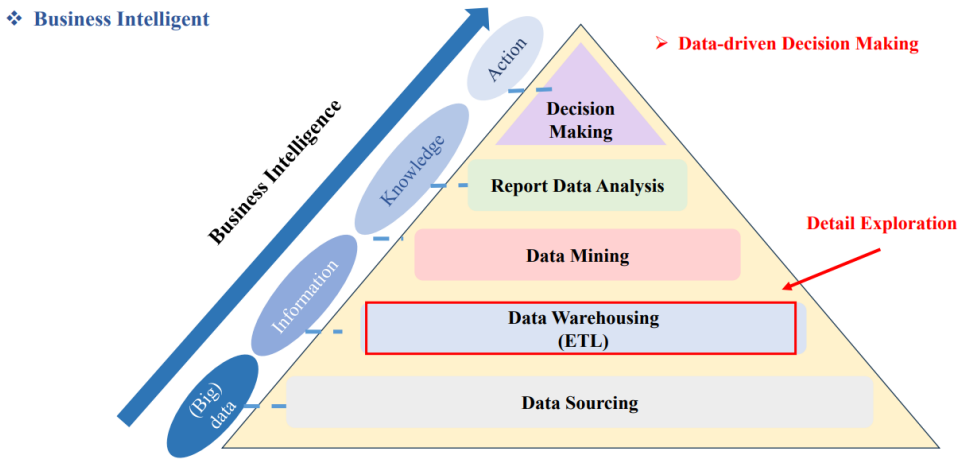
1.4 Kiến trúc dữ liệu hiện đại

1.4.1 Mô hình Business Intelligence Pyramid

Business Intelligence (BI) được tổ chức theo mô hình kim tự tháp, trong đó ETL Pipeline đóng vai trò nền tảng quan trọng. Mũi tên "Business Intelligence"biểu thị sự gia tăng giá trị và độ phức tạp từ dữ liệu thô đến knowledge và cuối cùng là action. Càng lên cao trong pyramid, chúng ta có:

- **Tăng giá trị business**: Từ raw data đến actionable insights
- **Giảm volume**: Từ massive datasets đến focused recommendations
- **Tăng context**: Từ isolated data points đến business-relevant information
- **Tăng impact**: Từ descriptive reports đến predictive và prescriptive analytics

Mô hình này minh họa sự tiến hóa từ dữ liệu thô đến insight và hành động cụ thể:



Hình 2: Business Intelligence Pyramid

Các tầng trong BI Pyramid:

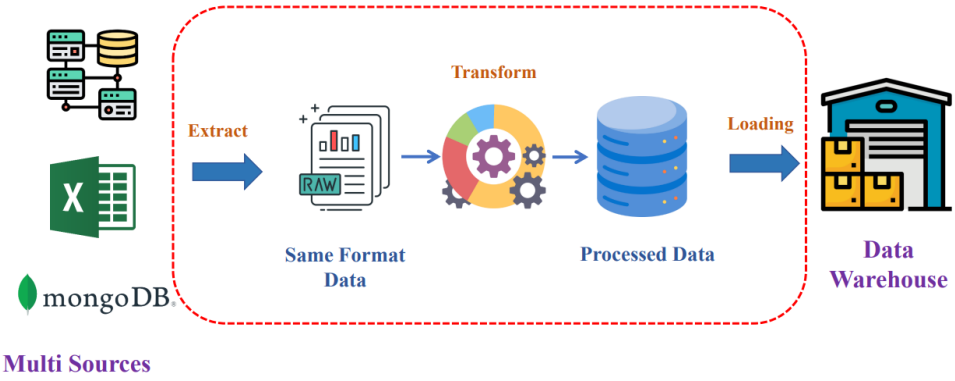
1. **Big Data (Tầng nền tảng)**:
   - Dữ liệu thô từ nhiều nguồn khác nhau
   - Volume lớn, đa dạng về format và structure
   - Yêu cầu khả năng lưu trữ và xử lý mạnh mẽ
2. **Data Sourcing (Thu thập dữ liệu)**:
   - Identification và connection đến các data sources
   - Data profiling và quality assessment
   - Establish data access patterns và security
3. **Data Warehousing - ETL (Xử lý và lưu trữ)**:
   - Extract, Transform, Load processes
   - Data integration và standardization

- Historical data storage và management

4. **Data Mining (Khai thác dữ liệu)**:
   - Pattern recognition và anomaly detection
   - Statistical analysis và machine learning
   - Predictive modeling và forecasting
5. **Report Data Analysis (Phân tích và báo cáo)**:
   - Descriptive analytics và visualization
   - KPI monitoring và dashboard creation
   - Ad-hoc analysis và self-service BI
6. **Decision Making (Ra quyết định)**:
   - Data-driven decision processes
   - Strategic planning dựa trên insights
   - Action planning và execution

1.4.2 Quy trình ETL tổng quát

ETL Pipeline là connecting bridge giữa raw data sources và business intelligence applications:



Hình 3: ETL Process Flow

Multi Sources (Đa nguồn dữ liệu):

- **Relational Databases**: MySQL, PostgreSQL, SQL Server, Oracle
- **File Systems**: Excel, CSV, JSON, XML, Parquet
- **NoSQL Databases**: MongoDB, Cassandra, Redis
- **Cloud Storage**: AWS S3, Google Cloud Storage, Azure Blob
- **APIs**: REST APIs, GraphQL, SOAP services
- **Streaming Data**: Kafka, Kinesis, Event Hubs

Extract Phase (Trích xuất):

- Kết nối và đọc dữ liệu từ multiple sources
- Handle different data formats và protocols
- Implement extraction strategies (full, incremental, CDC)
- Data validation và error handling

Transform Phase (Chuyển đổi):

- **Same Format Data**: Standardization về common format
- Data cleaning và quality improvement
- Business rule application
- Schema mapping và data type conversion
- Aggregation và calculation

Load Phase (Tải dữ liệu):

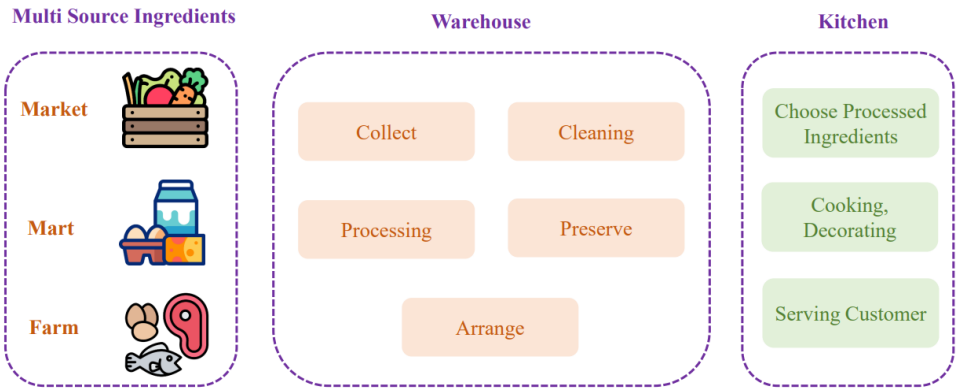
- **Processed Data**: Optimized cho analytical workloads
- Dimensional modeling (Star/Snowflake schema)
- Indexing và partitioning cho performance
- Data archival và retention policies

Data Warehouse (Kho dữ liệu):

- Central repository cho enterprise data
- Optimized cho OLAP operations
- Support cho reporting và analytics tools
- Data governance và security implementation

1.4.3 Data Warehouse như một nhà hàng

Để hiểu rõ hơn về vai trò của Data Warehouse, chúng ta có thể tưởng tượng nó như một nhà hàng chuyên nghiệp:



Hình 4: Data Warehouse as Restaurant

Multi-Source Ingredients (Nguyên liệu đa nguồn):
Giống như một nhà hàng cần nguyên liệu từ nhiều nguồn khác nhau:

- **Excel**: Như market stalls - dữ liệu nhỏ lẻ, manual
- **MySQL/PostgreSQL**: Như wholesale suppliers - dữ liệu structured, reliable
- **Supabase**: Như modern food distributors - cloud-based, scalable
- **APIs**: Như direct farm connections - real-time, fresh data

Data Warehouse (Kho lưu trữ):

- **Collect**: Thu thập nguyên liệu từ các nguồn
- **Cleaning**: Làm sạch, loại bỏ impurities
- **Processing**: Chế biến theo standardized recipes (business rules)
- **Preserve**: Bảo quản trong điều kiện optimal
- **Arrange**: Sắp xếp theo organized structure

Serving Customers (Phục vụ người dùng):
Data Warehouse phục vụ nhiều types of "customers":

- **Business Analysts**: Như regular diners - cần standard reports
- **Data Scientists**: Như professional chefs - cần raw ingredients cho custom analysis
- **Executives**: Như VIP customers - cần high-level dashboards và insights
- **Operational Teams**: Như catering orders - cần scheduled, automated deliveries

Quality Control (Kiểm soát chất lượng):

- **Data Quality**: Như food safety standards
- **Performance Monitoring**: Như service time tracking
- **Customer Satisfaction**: Như feedback collection từ users
- **Continuous Improvement**: Như menu updates và process optimization

1.4.4 Lợi ích của kiến trúc ETL

Business Benefits:

1. **Single Source of Truth**: Centralized data repository
2. **Data Consistency**: Standardized definitions và calculations
3. **Historical Analysis**: Time-series data cho trend analysis
4. **Improved Performance**: Optimized cho analytical queries
5. **Data Governance**: Centralized security và access control

Technical Benefits:

1. **Scalability**: Handle growing data volumes
2. **Reliability**: Robust error handling và recovery
3. **Maintainability**: Modular design với clear separation of concerns
4. **Monitoring**: Comprehensive logging và alerting
5. **Flexibility**: Support cho multiple output formats và destinations

Với foundation này, ETL Pipeline trở thành backbone của modern data-driven organization, enabling chuyển đổi từ reactive reporting sang proactive, insight-driven decision making.

2. Data Extraction - Trích xuất dữ liệu

2.1 Quy trình Extraction

Data Extraction (Trích xuất dữ liệu) là giai đoạn đầu tiên trong quy trình ETL (Extract - Transform - Load), đảm nhiệm việc lấy dữ liệu từ các nguồn khác nhau như cơ sở dữ liệu, API, file logs, hoặc các hệ thống ứng dụng. Quá trình này cần đảm bảo tính đầy đủ, chính xác và kịp thời của dữ liệu để phục vụ cho các bước xử lý tiếp theo. Các phương pháp trích xuất dữ liệu phổ biến gồm:

- **Pull (Active)**: Hệ thống ETL chủ động gửi yêu cầu đến nguồn dữ liệu theo lịch định sẵn hoặc theo sự kiện để *kéo* dữ liệu về.
  - *Ưu điểm*: Kiểm soát được thời điểm và tần suất lấy dữ liệu; dễ tích hợp với nhiều loại nguồn.
  - *Nhược điểm*: Có thể gây tải cao lên hệ thống nguồn nếu tần suất truy vấn lớn; độ trễ phụ thuộc vào lịch lấy dữ liệu.
  - *Ví dụ*: Hệ thống ETL chạy cron job mỗi 1 giờ để truy vấn bảng giao dịch từ cơ sở dữ liệu bán hàng.
- **Push (Inactive)**: Nguồn dữ liệu chủ động *đẩy* dữ liệu đến hệ thống ETL khi có dữ liệu mới hoặc khi đạt điều kiện nhất định.
  - *Ưu điểm*: Độ trễ thấp; giảm tải cho hệ thống ETL vì không cần truy vấn liên tục.
  - *Nhược điểm*: Cần cấu hình và hỗ trợ từ hệ thống nguồn; khó kiểm soát nếu nguồn gửi dữ liệu sai định dạng.
  - *Ví dụ*: Ứng dụng thương mại điện tử gửi sự kiện đơn hàng mới qua Webhook đến hệ thống ETL ngay khi đơn được tạo.
- **Incremental**: Chỉ trích xuất phần dữ liệu mới hoặc thay đổi so với lần trích xuất trước đó, thường dựa trên các cột như `last_modified` hoặc `created_at`.
  - *Ưu điểm*: Tiết kiệm thời gian và tài nguyên; giảm dung lượng dữ liệu cần xử lý.
  - *Nhược điểm*: Cần cơ chế theo dõi trạng thái lần trích xuất cuối; dễ bỏ sót nếu cột thời gian không được cập nhật chính xác.
  - *Ví dụ*: ETL chỉ lấy các bản ghi giao dịch có `updated_at` lớn hơn thời điểm trích xuất gần nhất.
- **Change Data Capture (CDC)**: Theo dõi và ghi lại mọi thay đổi (INSERT, UPDATE, DELETE) trên dữ liệu nguồn bằng log hoặc trigger ở mức cơ sở dữ liệu.

- *Ưu điểm*: Ghi nhận chính xác và đầy đủ các thay đổi; phù hợp cho đồng bộ dữ liệu thời gian thực.
- *Nhược điểm*: Cấu hình phức tạp; có thể ảnh hưởng hiệu năng của hệ thống nguồn nếu không tối ưu.
- *Ví dụ*: Sử dụng Debezium để lắng nghe binlog của MySQL, phát hiện mọi thay đổi và gửi đến Kafka.

2.2 Case Study: Iowa Liquor Sales Dataset

Chúng ta sẽ sử dụng bộ dữ liệu Iowa Liquor Sales làm ví dụ thực tế. Dataset này chứa thông tin về việc mua bán rượu tại Iowa với các trường chính:

```
# Data Structure of Iowa Liquor Sales
columns = [
    'invoice_line_no',
    'date',
    'store',
    'name',
    'address',
    'city',
    'zipcode',
    'store_location',
    'county_number',
    'county',
    'category',
    'category_name',
    'vendor_no',
    'vendor_name',
    'item_number',
    'item_description',
    'pack',
    'bottle_volume_ml',
    'state_bottle_cost',
    'state_bottle_retail',
    'sale_bottles',
    'sale_dollars',
    'sale_liters',
    'sale_gallons'
]
```

2.3 Xử lý vấn đề Composite Key Collisions

Khi extract dữ liệu từ nhiều nguồn, thường gặp vấn đề trùng lặp composite key:

```
-- Create stagging table to extract data
CREATE TABLE Staging_Sales (
    invoice_line_no TEXT,
    date TEXT,
    store TEXT,
    name TEXT,
    address TEXT,
    city TEXT,
    zipcode TEXT,
    store_location TEXT,
    county_number TEXT,
    county TEXT,
    category TEXT,
```

```
    category_name TEXT,
    vendor_no TEXT,
    vendor_name TEXT,
    item_number TEXT,
    item_description TEXT,
    pack TEXT,
    bottle_volume_ml TEXT,
    state_bottle_cost TEXT,
    state_bottle_retail TEXT,
    sale_bottles TEXT,
    sale_dollars TEXT,
    sale_liters TEXT,
    sale_gallons TEXT,
    file_name TEXT NOT NULL,
    processed_timestamp DATETIME NOT NULL,
    record_source TEXT NOT NULL
);
```

2.4 Batch Processing

Để xử lý hiệu quả, dữ liệu được chia thành các batch nhỏ:

```
def process_csv_in_batches(file_path, batch_size=10000):
    """
    Process CSV by batch_size
    """
    import pandas as pd

    chunk_iter = pd.read_csv(file_path, chunksize=batch_size)

    for i, chunk in enumerate(chunk_iter):
        # Add metadata
        chunk['file_name'] = file_path
        chunk['processed_timestamp'] = pd.Timestamp.now()
        chunk['record_source'] = 'batch_processing'

        # Save into staging area
        chunk.to_sql('staging_sales', conn,
                     if_exists='append', index=False)

        print(f"Processed batch {i+1}: {len(chunk)} records")
```

3. Data Transformation - Chuyển đổi dữ liệu

Chuyển đổi dữ liệu (Data Transformation) là giai đoạn quan trọng trong quy trình xử lý dữ liệu, nhằm làm sạch và biến đổi dữ liệu thô thành dữ liệu có chất lượng và phù hợp cho các bước phân tích hoặc lưu trữ. Quá trình này thường bao gồm hai bước chính: **Data Cleaning (Làm sạch dữ liệu)** và **Data Processing (Xử lý dữ liệu)**.

Data Cleaning - Làm sạch dữ liệu

Trong bước này, dữ liệu được kiểm tra và xử lý để loại bỏ hoặc sửa các lỗi và vấn đề ảnh hưởng đến chất lượng dữ liệu như:

- **Duplicated (Trùng lặp)**: Xác định và loại bỏ các bản ghi dữ liệu bị trùng để tránh gây sai lệch trong phân tích.

- **Filtering (Lọc dữ liệu)**: Lọc bỏ các dữ liệu không hợp lệ hoặc không cần thiết dựa trên điều kiện cụ thể.
- **Missing (Dữ liệu thiếu)**: Xử lý các giá trị bị thiếu bằng cách điền giá trị thay thế, loại bỏ hoặc giữ nguyên tùy theo mục đích.
- **Invalid (Dữ liệu không hợp lệ)**: Phát hiện và sửa các dữ liệu sai định dạng, nằm ngoài phạm vi cho phép hoặc không hợp lệ.
- **Type Casting (Chuyển đổi kiểu dữ liệu)**: Đảm bảo các trường dữ liệu có kiểu dữ liệu phù hợp (ví dụ: số nguyên, chuỗi, ngày tháng) để dễ dàng xử lý và phân tích.

Data Processing - Xử lý dữ liệu

Sau khi dữ liệu đã được làm sạch, bước xử lý dữ liệu sẽ biến đổi dữ liệu để phục vụ cho các mục tiêu phân tích, bao gồm:

- **Enrichment (Làm giàu dữ liệu)**: Bổ sung thêm thông tin hoặc thuộc tính mới cho dữ liệu hiện có để tăng giá trị khai thác.
- **Aggregate (Tổng hợp)**: Tập hợp dữ liệu theo các nhóm hoặc tiêu chí nhất định để tạo ra các chỉ số tổng quan, ví dụ như tổng doanh số theo tháng.
- **Normalize (Chuẩn hóa)**: Điều chỉnh dữ liệu về cùng một thang đo hoặc định dạng để dễ so sánh và phân tích.
- **Integrate (Tích hợp)**: Kết hợp dữ liệu từ nhiều nguồn khác nhau thành một bộ dữ liệu thống nhất.
- **Derived Column (Cột dẫn xuất)**: Tạo thêm các cột dữ liệu mới dựa trên phép tính hoặc kết hợp các cột hiện có, ví dụ như tính tổng giá trị hoặc phân loại nhóm.

Nhờ vào các bước làm sạch và xử lý này, dữ liệu được chuyển đổi trở nên đồng nhất, chính xác và có ý nghĩa hơn, từ đó giúp nâng cao hiệu quả trong việc phân tích và ra quyết định.

3.1 Data Cleaning - Làm sạch dữ liệu

3.1.1 Xử lý dữ liệu trùng lặp

```python
import pandas as pd

# Find and Solve duplicate
df_full_duplicated = df[df.duplicated(keep=False)]
print(f"No. duplicate records: {len(df_full_duplicated)}")

# Drop duplicate
df_cleaned = df.drop_duplicates()

# Solve duplicate by key
df_special_duplicated = df[df.duplicated(
    subset=['invoice_line_no', 'store'], keep=False
)]

df_final = df[
    df.duplicated(subset=['invoice_line_no', 'store'], keep='first') |
    ~df.duplicated(subset=['invoice_line_no', 'store'], keep=False)
]
```

3.1.2 Xử lý dữ liệu thiếu (Missing Data)

```
# Check null data
null_columns = df.columns[df.isnull().sum() > 0].tolist()
print("Columns having null value:", null_columns)

# Fill missing data
strategies = {
    'address': 'Unknown',
    'city': 'Unknown',
    'zipcode': 'Unknown',
    'county_number': 'Unknown',
    'county': 'Unknown',
    'category': 'Unknown',
    'category_name': 'Unknown'
}

df_filled = df.fillna(value=strategies, inplace=True)

# Delete columns that have missing values
critical_columns = [
    'state_bottle_cost', 'state_bottle_retail',
    'sale_bottles', 'sale_dollars', 'sale_liters', 'sale_gallons'
]

# Delete columns based on business logic
df_cleaned = df.dropna(subset=critical_columns)
```

3.1.3 Xử lý dữ liệu không hợp lệ

```
# Check numeric columns
numeric_cols = [
    'pack', 'bottle_volume_ml', 'state_bottle_cost',
    'state_bottle_retail', 'sale_bottles', 'sale_dollars',
    'sale_liters', 'sale_gallons'

for col in numeric_cols:
    df[col] = pd.to_numeric(df[col], errors='coerce')

# Delete invalid data (< 1)
df_valid = df[(df[numeric_cols] >= 1).all(axis=1)]
```

3.2 Type Casting - Chuyển đổi kiểu dữ liệu

```
numeric_cols = [
    'state_bottle_cost', 'state_bottle_retail', 'sale_bottles',
    'sale_dollars', 'sale_liters', 'sale_gallons'
]

for col in numeric_cols:
    df[col] = pd.to_numeric(df[col], errors='coerce')

# Change data type into datetime
df['date'] = pd.to_datetime(df['date'], errors='coerce')
```

3.3 Data Processing - Xử lý dữ liệu

3.3.1 Derived Columns - Tạo cột phái sinh

```
# Create columns based on business logic
df['revenue'] = df['sale_dollars']
df['cost'] = df['state_bottle_cost'] * df['sale_bottles']
df['profit'] = (df['state_bottle_retail'] - df['state_bottle_cost']) * df['
    sale_bottles']
df['total_bottles_sold'] = df['sale_bottles']
df['total_volume_sold_in_liters'] = df['sale_liters']

# Calculate advance metrics
df['profit_margin'] = (
    (df['profit'] / df['revenue'] * 100)
    .round(2)
    .where(df['revenue'] > 0, 0)
)

df['average_bottle_price'] = (
    (df['sale_dollars'] / df['sale_bottles'])
    .round(2)
    .where(df['sale_bottles'] > 0, 0)
)

df['volume_per_bottle_sold'] = (
    (df['sale_liters'] / df['sale_bottles'])
    .round(2)
    .where(df['sale_bottles'] > 0, 0)
)
```

4. Data Loading - Tải dữ liệu

4.1 Dimension và Fact Tables

4.1.1 Thiết kế Star Schema

Trong Data Warehouse, chúng ta sử dụng mô hình Star Schema với:

Fact Table (sales_fact):

```
CREATE TABLE sales_fact (
    sales_key INTEGER PRIMARY KEY,
    invoice_line_no TEXT,
    store INTEGER,
    date_key INTEGER,
    store_key INTEGER,
    item_key INTEGER,
    vendor_key INTEGER,
    revenue DECIMAL,
    profit DECIMAL,
    cost DECIMAL,
    total_bottles_sold INTEGER,
    total_volume_sold_in_liters DECIMAL,
    profit_margin DECIMAL,
    average_bottle_price DECIMAL,
    volume_per_bottle_sold DECIMAL,
    processed_timestamp DATETIME
);
```

Dimension Tables:

```sql
-- Date Dimension
CREATE TABLE date_dim (
    date_key INTEGER PRIMARY KEY,
    date DATE,
    year INTEGER,
    month INTEGER,
    day INTEGER,
    quarter INTEGER,
    weekday TEXT
);

-- Store Dimension
CREATE TABLE store_dim (
    store_key INTEGER PRIMARY KEY,
    store_id INTEGER,
    address TEXT,
    city TEXT,
    zipcode TEXT,
    store_location TEXT,
    county_number TEXT,
    county TEXT,
    start_date DATE,
    end_date DATE,
    is_active BOOLEAN
);

-- Item Dimension
CREATE TABLE item_dim (
    item_key INTEGER PRIMARY KEY,
    item_number TEXT,
    item_description TEXT,
    category TEXT,
    category_name TEXT,
    pack FLOAT,
    bottle_volume_ml FLOAT,
    state_bottle_cost DECIMAL,
    state_bottle_retail DECIMAL,
    start_date DATE,
    end_date DATE,
    is_active BOOLEAN
);

-- Vendor Dimension
CREATE TABLE vendor_dim (
    vendor_key INTEGER PRIMARY KEY,
    vendor_no TEXT,
    vendor_name TEXT,
    start_date DATE,
    end_date DATE,
    is_active BOOLEAN
);
```

4.2 Slowly Changing Dimensions (SCD)

Slowly Changing Dimensions (SCD) là khái niệm dùng để mô tả các chiều dữ liệu trong kho dữ liệu mà thông tin trong các chiều này thay đổi không thường xuyên hoặc thay đổi chậm theo thời gian. Ví dụ về các chiều như vậy thường là thông tin khách hàng, sản phẩm, địa điểm, nhân viên, v.v.

Trong quá trình quản lý dữ liệu, việc xử lý các chiều này rất quan trọng vì chúng không chỉ đơn thuần chứa dữ liệu hiện tại mà còn cần giữ được lịch sử các thay đổi theo thời gian để phục vụ phân tích và báo cáo chính xác.

4.3 Tại sao cần quan tâm đến Slowly Changing Dimensions?

- Các chiều dữ liệu thường lưu các thuộc tính mô tả đối tượng như tên, địa chỉ, trạng thái, hoặc các thuộc tính khác.
- Khi các thuộc tính này thay đổi (ví dụ khách hàng đổi địa chỉ, nhân viên thay đổi chức vụ), ta cần quyết định:
  - Có giữ lại thông tin cũ không?
  - Nếu giữ thì bằng cách nào?
  - Nếu thay đổi thì có ảnh hưởng đến dữ liệu báo cáo không?

4.4 Các loại Slowly Changing Dimensions phổ biến

1. **SCD Type 0 (Không thay đổi)**
   Không bao giờ cập nhật dữ liệu cũ, giữ nguyên giá trị ban đầu. Thường dùng khi không cần theo dõi lịch sử thay đổi.
2. **SCD Type 1 (Ghi đè dữ liệu cũ)**
   Khi có thay đổi, dữ liệu cũ bị ghi đè bởi dữ liệu mới, không giữ lịch sử thay đổi. Ví dụ: sửa lỗi chính tả trong tên khách hàng.
3. **SCD Type 2 (Giữ lịch sử đầy đủ)**
   Tạo bản ghi mới mỗi khi có thay đổi, giữ lại lịch sử của các bản ghi cũ. Thường dùng các trường như `start_date`, `end_date` hoặc `is_current` để xác định phiên bản hiện tại. Đây là loại SCD phổ biến nhất.
4. **SCD Type 3 (Lưu một số phiên bản thay đổi)**
   Chỉ lưu một số thuộc tính cũ và mới trong cùng một bản ghi, không giữ lịch sử đầy đủ. Ví dụ, có thể lưu giá trị cũ và giá trị hiện tại của một thuộc tính cụ thể.
5. **Các loại SCD mở rộng khác** (Type 4, Type 6, ...)
   Kết hợp các phương pháp trên hoặc lưu lịch sử trong bảng riêng biệt.

4.5 Ví dụ minh họa SCD Type 2

Giả sử bảng khách hàng lưu thông tin địa chỉ. Khi khách hàng thay đổi địa chỉ, thay vì cập nhật trực tiếp trên bản ghi hiện tại, ta tạo thêm một bản ghi mới với địa chỉ mới và đánh dấu bản ghi cũ là không còn hiệu lực (ví dụ `end_date` = ngày thay đổi).

CustomerID	Name	Address	Start_Date	End_Date	Is_Current
1001	Nguyễn An	123 Đường A	2020-01-01	2022-06-30	False
1001	Nguyễn An	456 Đường B	2022-07-01	NULL	True

Bảng 1: Ví dụ SCD Type 2 cho quản lý địa chỉ khách hàng

Điều này giúp khi phân tích doanh số theo thời gian, ta biết chính xác khách hàng ở đâu trong từng giai đoạn.

4.6 Ví dụ minh họa các SCD khác

4.6.1 SCD Type 1 - Overwrite

Ghi đè dữ liệu cũ bằng dữ liệu mới:

```
-- Example: Update store address
UPDATE store_dim
SET address = 'New Address',
    city = 'New City'
WHERE store_id = 123;
```

4.6.2 SCD Type 2 - Add New Row

Thêm bản ghi mới và giữ lại lịch sử:

```
def process_scd_type2(df, dim_table, key_col, attributes, conn):
    """
    Handle Slowly Changing Dimension Type 2
    """
    # Create a copy to avoid SettingWithCopyWarning
    df = df.copy()

    if current_dim.empty:
        # Initial load - all records are new
        df = df.assign(
            start_date=datetime.now().date(),
            end_date=None,
            is_active=True
        )
        df.to_sql(dim_table, conn, if_exists='append', index=False)
        return

    # Find new and changed records
    merged = df.merge(current_dim, on=key_col, how='outer',
                      suffixes=('', '_current'), indicator=True)

    # New records
    new_records = merged[merged['_merge'] == 'left_only'][key_col + attributes].copy()
    if not new_records.empty:
        new_records = new_records.assign(
            start_date=datetime.now().date(),
            end_date=None,
            is_active=True
        )
        new_records.to_sql(dim_table, conn, if_exists='append', index=False)

    # Changed records
    changed_records = merged[merged['_merge'] == 'both'].copy()
    for attr in attributes:
        changed_records = changed_records[
            changed_records[attr] != changed_records[f'{attr}_current']
        ]

    if not changed_records.empty:
        # Expire old record versions
        conn.execute(text(f"""
            UPDATE {dim_table}
            SET end_date = :end_date, is_active = 0
            WHERE {key_col} = :key_val AND is_active = 1
```

```
        """), {'end_date': datetime.now().date(),
               'key_val': row[key_col]})

        # Insert new versions
        new_versions = changed_records[key_col + attributes].copy()
        new_versions = new_versions.assign(
            start_date=datetime.now().date(),
            end_date=None,
            is_active=True
        )
        new_versions.to_sql(dim_table, conn, if_exists='append', index=False)
```

5. Advanced ETL Concepts

5.1 Change Data Capture (CDC)

CDC theo dõi và ghi lại mọi thay đổi trong database:

```
def implement_cdc_tracking():
    """
    Implement CDC tracking with timestamp and checksum
    """
    # Add CDC columns to the staging table
    cdc_columns = {
        'cdc_timestamp': 'TIMESTAMP DEFAULT CURRENT_TIMESTAMP',
        'cdc_operation': 'VARCHAR(10)',  # INSERT, UPDATE, DELETE
        'cdc_checksum': 'VARCHAR(64)'   # MD5 hash of the record
    }

    # Trigger to automatically record CDC
    trigger_sql = """
    CREATE TRIGGER cdc_trigger_sales
    AFTER INSERT OR UPDATE OR DELETE ON sales_source
    FOR EACH ROW
    BEGIN
        IF TG_OP = 'INSERT' THEN
            INSERT INTO cdc_log VALUES (NEW.*, 'INSERT', CURRENT_TIMESTAMP);
        ELSIF TG_OP = 'UPDATE' THEN
            INSERT INTO cdc_log VALUES (NEW.*, 'UPDATE', CURRENT_TIMESTAMP);
        ELSIF TG_OP = 'DELETE' THEN
            INSERT INTO cdc_log VALUES (OLD.*, 'DELETE', CURRENT_TIMESTAMP);
        END IF;
    END;
    """
```

5.2 Data Quality Monitoring

```
def data_quality_checks(df):
    """
    Check data quality
    """
    quality_report = {}

    # 1. Completeness Check
    quality_report['completeness'] = {
        'total_records': len(df),
        'missing_values': df.isnull().sum().to_dict(),
```

```
        'completeness_rate': ((len(df) - df.isnull().sum()) / len(df) * 100).to_dict()
    }

    # 2. Uniqueness Check
    duplicate_rate = df.duplicated().sum() / len(df) * 100
    quality_report['uniqueness'] = {
        'duplicate_records': df.duplicated().sum(),
        'duplicate_rate': duplicate_rate
    }

    # 3. Validity Check
    numeric_cols = df.select_dtypes(include=['number']).columns
    invalid_numeric = (df[numeric_cols] < 0).sum()
    quality_report['validity'] = {
        'invalid_numeric_values': invalid_numeric.to_dict()
    }

    # 4. Consistency Check
    business_rules_violations = []
    if 'profit' in df.columns and 'revenue' in df.columns:
        invalid_profit = (df['profit'] > df['revenue']).sum()
        business_rules_violations.append({
            'rule': 'profit_should_not_exceed_revenue',
            'violations': invalid_profit
        })

    quality_report['consistency'] = business_rules_violations

    return quality_report
```

5.3 Performance Optimization

5.3.1 Parallel Processing

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import multiprocessing as mp

def parallel_etl_processing(file_list, num_workers=None):
    """
    Parallel ETL processing for multiple files
    """
    if num_workers is None:
        num_workers = mp.cpu_count()

    def process_single_file(file_path):
        try:
            # Extract
            df = pd.read_csv(file_path)

            # Transform
            df_transformed = transform_data(df)

            # Load
            load_to_warehouse(df_transformed)

            return {"file": file_path, "status": "success", "records": len(df)}
        except Exception as e:
            return {"file": file_path, "status": "error", "error": str(e)}

```

```
    # Use ProcessPoolExecutor for CPU-intensive tasks
    with ProcessPoolExecutor(max_workers=num_workers) as executor:
        results = list(executor.map(process_single_file, file_list))

    return results
```

5.3.2 Memory Optimization

```
def memory_efficient_processing(file_path, chunk_size=50000):
    """
    Process large files with memory optimization
    """
    # Use categorical data type for string columns
    dtype_map = {
        'store': 'category',
        'name': 'category',
        'city': 'category',
        'county': 'category',
        'category_name': 'category',
        'vendor_name': 'category'
    }

    # Read only necessary columns
    required_columns = [
        'invoice_line_no', 'date', 'store', 'sale_dollars',
        'sale_bottles', 'state_bottle_cost', 'state_bottle_retail'
    ]

    chunk_iter = pd.read_csv(
        file_path,
        chunksize=chunk_size,
        dtype=dtype_map,
        usecols=required_columns,
        parse_dates=['date']
    )

    for chunk in chunk_iter:
        # Process each chunk
        processed_chunk = transform_chunk(chunk)

        # Load immediately to free memory
        load_chunk_to_warehouse(processed_chunk)

        # Explicitly delete to free memory
        del chunk, processed_chunk
```

5.4 Error Handling và Monitoring

```
import logging
from datetime import datetime

def setup_etl_logging():
    """
    Setup logging for ETL pipeline
    """
    logging.basicConfig(
        level=logging.INFO,
```

```
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        handlers=[
            logging.FileHandler(f'etl_pipeline_{datetime.now().strftime("%Y%m%d")}.log'),
            logging.StreamHandler()
        ]
    )
    return logging.getLogger('ETL_Pipeline')

def robust_etl_pipeline(source_config, transform_config, target_config):
    """
    ETL pipeline with robust error handling
    """
    logger = setup_etl_logging()
    pipeline_stats = {
        'start_time': datetime.now(),
        'processed_records': 0,
        'error_records': 0,
        'status': 'RUNNING'
    }

    try:
        logger.info("Starting ETL Pipeline")

        # Extract phase
        logger.info("Starting extraction phase")
        raw_data = extract_data(source_config)
        logger.info(f"Extracted {len(raw_data)} records")

        # Transform phase
        logger.info("Starting transformation phase")
        transformed_data, error_data = transform_data_with_error_handling(
            raw_data, transform_config
        )

        pipeline_stats['processed_records'] = len(transformed_data)
        pipeline_stats['error_records'] = len(error_data)

        if len(error_data) > 0:
            logger.warning(f"Found {len(error_data)} error records")
            # Save error records for review
            save_error_records(error_data)

        # Load phase
        logger.info("Starting load phase")
        load_data(transformed_data, target_config)

        pipeline_stats['status'] = 'SUCCESS'
        logger.info("ETL Pipeline completed successfully")

    except Exception as e:
        pipeline_stats['status'] = 'FAILED'
        logger.error(f"ETL Pipeline failed: {str(e)}")
        raise

    finally:
        pipeline_stats['end_time'] = datetime.now()
        pipeline_stats['duration'] = (
            pipeline_stats['end_time'] - pipeline_stats['start_time']
        ).total_seconds()
```

```
69
70          logger.info(f"Pipeline statistics: {pipeline_stats}")
71
72      return pipeline_stats
```