Aula 8

Aula 8: ETL com SQL e Python

1. Introdução ao Processo de ETL

Na aula anterior vimos modelagem de dados. Nesta aula vamos nos concentrar no processo de **ETL** (Extract, Transform, Load) utilizando PostgreSQL para a extração e transformação e Python para o consumo final dos dados. O foco é demonstrar o fluxo completo e as decisões que tomamos em cada etapa.

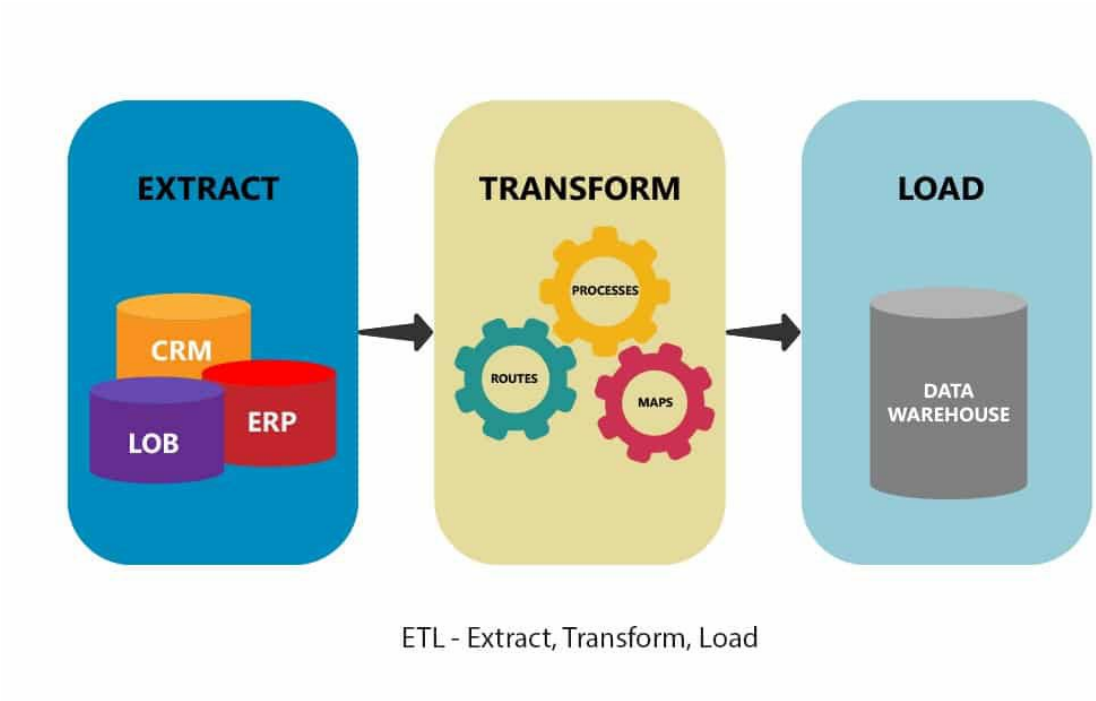

Figura 1: Fluxo de ETL.

2. Conceitos Fundamentais

2.1 O que é ETL?

O processo ETL envolve:

- **Extract**: coleta de dados de várias fontes;

- **Transform**: limpeza, padronização e agregações;
- **Load**: armazenamento do dado preparado para análise.

2.2 Views e Materialized Views

View = consulta armazenada (tabela virtual).
Materialized View = resultado persistido de uma consulta (melhora performance, precisa de REFRESH).

3. Cenário de Exemplo: Sistema de Pedidos e Clientes

Definimos um mini-modelo relacional com duas tabelas principais: `customers` e `orders`. Abaixo estão as queries para criar essas tabelas.

```
-- Criação das tabelas
CREATE TABLE customers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100),
    created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    customer_id INT REFERENCES customers(id),
    order_date TIMESTAMP DEFAULT NOW(),
    total NUMERIC(10,2)
);
```

3.1 Script para popular as tabelas (Carga)

Geraremos dados fictícios para simular clientes e pedidos. Use o bloco abaixo no seu PostgreSQL.

```
-- Popula customers e orders com PL/pgSQL
DO $$
DECLARE
    i INT;
    customer_count INT := 30;
    order_count INT := 200;
    total NUMERIC(10,2);
BEGIN
    -- Inserindo clientes
```

```
    FOR i IN 1..customer_count LOOP
        INSERT INTO customers (name, email)
        VALUES (
            CONCAT('Cliente ', i),
            CONCAT('cliente', i, '@exemplo.com')
        );
    END LOOP;

    -- Inserindo pedidos
    FOR i IN 1..order_count LOOP
        total := ROUND(CAST(100 + RANDOM() * 2000 AS NUMERIC), 2);
        INSERT INTO orders (customer_id, order_date, total)
        VALUES (
            (SELECT id FROM customers ORDER BY RANDOM() LIMIT 1),
            NOW() - (INTERVAL '1 day' * FLOOR(RANDOM() * 90)),
            total
        );
    END LOOP;
END $$;
```

4. Etapa de Transformação (Transform)

Agora que temos dados brutos, vamos transformá-los em métricas úteis: total de pedidos e valor gasto por cliente.

```
-- Total de pedidos e valor total por cliente
```

4.1 View para o resumo de clientes

Criamos uma `View` para facilitar o reuso desta transformação.

```
-- Criação de uma View de resumo de clientes
CREATE VIEW customer_summary AS
SELECT
    c.id AS customer_id,
    c.name,
    COUNT(o.id) AS total_pedidos,
    SUM(o.total) AS valor_total
FROM customers c
```

```
LEFT JOIN orders o ON c.id = o.customer_id
GROUP BY c.id, c.name;
```

Consultar a view:

```
SELECT * FROM customer_summary ORDER BY valor_total DESC LIMIT 20;
```

4.2 Materialized View para otimização

Se essa agregação for usada frequentemente (dashboards, relatórios), usamos uma materialized view.

```
-- Criação da Materialized View
CREATE MATERIALIZED VIEW customer_summary_mv AS
SELECT
    c.id AS customer_id,
    c.name,
    COUNT(o.id) AS total_pedidos,
    SUM(o.total) AS valor_total
FROM customers c
LEFT JOIN orders o ON c.id = o.customer_id
GROUP BY c.id, c.name;

-- Atualizar a materialized view quando necessário
REFRESH MATERIALIZED VIEW customer_summary_mv;
```

4.3 Views auxiliares: comportamento mensal

Vamos criar um exemplo de transformação para obter métricas mensais por cliente (útil para análises de coorte/retention).

```
-- Agregação mensal por cliente
SELECT
    customer_id,
    DATE_TRUNC('month', order_date) AS month,
    COUNT(*) AS orders_count,
    SUM(total) AS total_spent,
    AVG(total) AS avg_order_value
FROM orders
GROUP BY customer_id, DATE_TRUNC('month', order_date)
ORDER BY customer_id, month;
```

Salvar como view:

```
CREATE VIEW customer_monthly AS
SELECT
    customer_id,
    DATE_TRUNC('month', order_date) AS month,
    COUNT(*) AS orders_count,
    SUM(total) AS total_spent,
    AVG(total) AS avg_order_value
FROM orders
GROUP BY customer_id, DATE_TRUNC('month', order_date);
```

5. Etapa de Carga e Consumo (Load)

A etapa final do ETL é disponibilizar os dados para consumo. Vamos conectar um notebook Python e ler a materialized view.

```
# Notebook Python: ler a materialized view
import pandas as pd
import sqlalchemy

engine = sqlalchemy.create_engine("postgresql://usuario:senha@localhost:5432/etl_db")

# Ler a MV
df = pd.read_sql("SELECT * FROM customer_summary_mv", engine)

df.head()
```

5.1 Visualização rápida

Exemplo simples (barra horizontal) com matplotlib:

```
import matplotlib.pyplot as plt

top_customers = df.sort_values("valor_total", ascending=False).head(10)

plt.barh(top_customers["name"], top_customers["valor_total"])
plt.xlabel("Valor Total (R$)")
plt.ylabel("Cliente")
plt.title("Top 10 Clientes por Valor Total de Pedidos")
plt.gca().invert_yaxis()
plt.show()
```

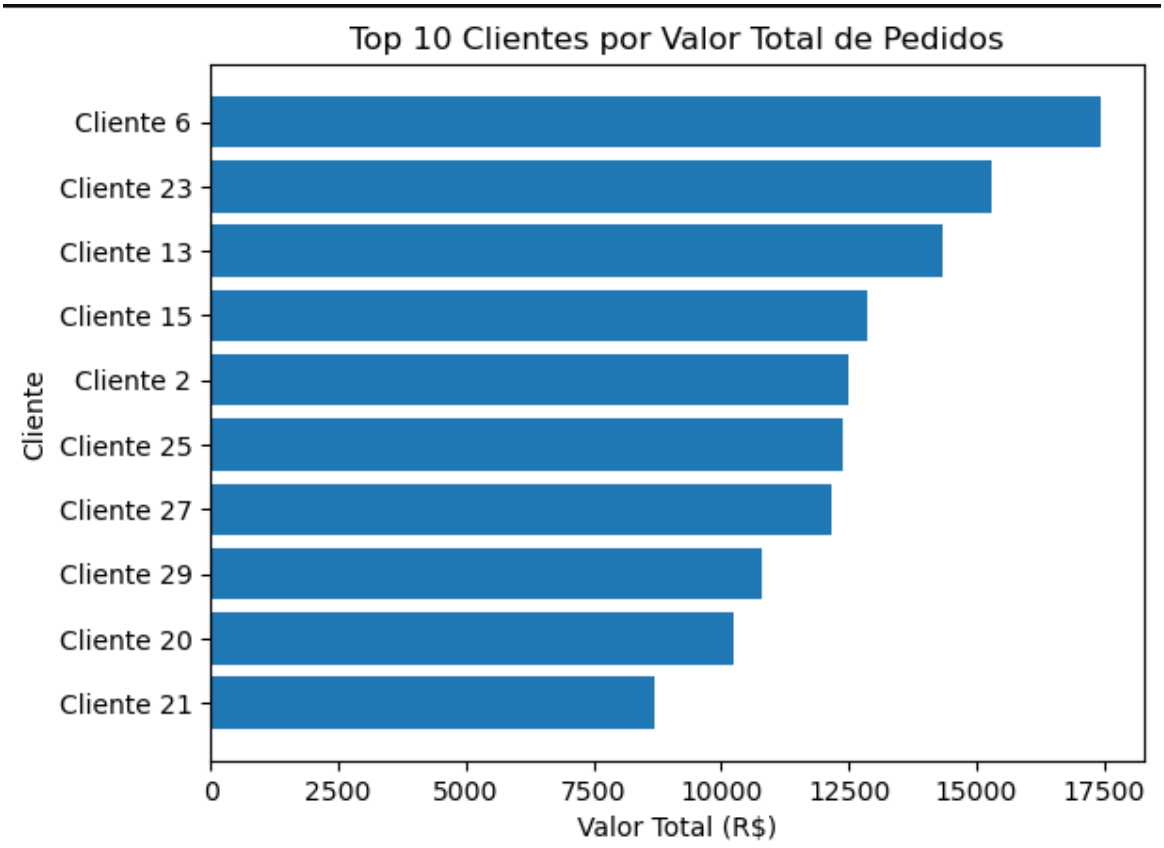

Figura 2: Top 10 clientes por valor de pedido

6. Boas práticas de ETL

- **Auditoria**: mantenha colunas de `created_at` e `updated_at` nas tabelas e logs de execução do ETL.
- **Idempotência**: scripts que possam rodar várias vezes sem duplicar dados.
- **Backfill**: estratégia para importar dados históricos.
- **Atualizações incrementais**: evite recriar toda a base; atualize apenas registros novos/alterados.
- **Documentação**: comente queries complexas e mantenha um changelog do pipeline.