Tabla de Contenidos

TEC | Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

IC-4302 – Bases de Datos II

Proyecto 2

Red Social de Viajes

Estudiantes:

Esteban Josué Solano Araya – 2021579468

Pablo Mesén Alvarado – 2023071259

Daniel Zeas Brown – 2023147474

Profesor:

Keneth Obando Rodríguez

22 de noviembre de 2024

II Semestre 2024

Estos se limpiaron y se transformaron en valores numéricos (float) para asegurar la compatibilidad con Neo4j.

3. **Creación de nodos y relaciones en Neo4j**

   Los datos fueron cargados en Neo4j mediante transacciones Cypher, creando:

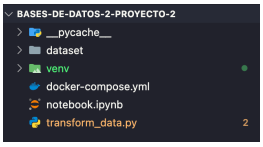
   - Nodos para clientes (Customer).
   - Nodos para productos (Product).
   - Relación entre ambos nodos con las transacciones (TRANSACTION).

   Cada transacción incluye propiedades como transaction_id, transaction_date y standard_cost.

1. Introducción

Este proyecto analiza transacciones simuladas utilizando servicios integrados como Neo4j, Spark y PostgreSQL. Los datos se cargan inicialmente en Neo4j, se procesan mediante Spark para generar métricas clave, y se almacenan en PostgreSQL para facilitar consultas adicionales.

2. Estructura del Proyecto



3. Cómo se cargaron los datos en Neo4j

1. **Lectura del archivo CSV**

   Utilizamos la biblioteca pandas para leer el archivo Transactions.csv, seleccionando las columnas relevantes:

   - transaction_id
   - product_id
   - customer_id
   - transaction_date
   - standard_cost

   Eliminamos filas con valores faltantes en las columnas clave para garantizar que los datos cargados sean consistentes.

2. **Limpieza de datos**

   - La columna standard_cost contenía valores formateados con caracteres como '$' y ','.

4. Consultas utilizadas en Neo4j

4.1 Extracción de datos

Los datos fueron extraídos desde Neo4j con la siguiente consulta Cypher:

```
MATCH (c:Customer)-[t:TRANSACTION]->(p:Product)
RETURN c.customer_id AS customer_id, p.product_id AS product_id, t.standard_cost AS
standard_cost
```

Esta consulta generó una tabla con tres columnas: customer_id, product_id, y standard_cost. Los datos se procesaron con Python usando el driver oficial de Neo4j:

```
def execute_cypher_query(query):
    with driver.session() as session:
        result = session.run(query)
        return [dict(record) for record in
result]
```

5. Análisis con Spark

5.1 Importación de librerías y Conexión a PostgreSQL

- pyspark.sql.SparkSession: Se usa para inicializar una sesión de Spark, que permite realizar análisis de datos distribuidos.
- neo4j.GraphDatabase: Facilita la conexión con una base de datos Neo4j y la ejecución de consultas Cypher.
- pandas: Ayuda a manipular y transformar los datos en estructuras tabulares.
- psycopg2: Proporciona una interfaz para conectarse y realizar consultas con PostgreSQL.

La función psycopg2.connect establece la conexión con una base de datos PostgreSQL utilizando:

- dbname: Nombre de la base de datos.

5.3 Creación de la sesión de Spark

SparkSession.builder configura la aplicación Spark con:

- appName: El nombre de la aplicación Spark.
- master: Define el modo de ejecución como local.

```
# Crear la sesión de Spark
spark = SparkSession.builder.appName("Spark").master("local").getOrCreate()
```

5.4 Función para ejecutar consultas Cypher

La función recibe una consulta Cypher (query) como parámetro y ejecuta esta consulta en el servidor Neo4j.

- driver.session(): Abre una sesión temporal con Neo4j.
- session.run(query): Ejecuta la consulta Cypher.
- list(): Convierte los resultados en una lista para su manejo posterior.

```
def execute_cypher_query(query):
    with driver.session() as session:
        return list(session.run(query))
```

5.5 Ejecución de consulta Cypher

La consulta busca nodos de tipo Customer y Product conectados por relaciones TRANSACTION. Devuelve tres atributos:

- c.customer_id: Identificador del cliente.
- p.product_id: Identificador del producto.

Código Utilizado para cargar datos en Neo4j junto a explicación en cada punto.

```
from neo4j import GraphDatabase
import pandas as pd

# Configuración de conexión a Neo4j
uri = "bolt://localhost:7687"
username = "neo4j"
password = "badpassword"
driver = GraphDatabase.driver(uri, auth=(username, password))

# Leer el archivo CSV
csv_file = "./dataset/Transactions.csv"
data = pd.read_csv(csv_file, usecols=["transaction_id", "product_id", "customer_id", "transaction_date",
"standard_cost"])

# Limpieza de datos: eliminar filas con NaN
data_subset = data.dropna(subset=["transaction_id", "product_id", "customer_id", "transaction_date",
"standard_cost"])

# Limpieza de la columna 'standard_cost'
data_subset.loc[:, 'standard_cost'] = (
    data_subset['standard_cost']
    .replace('[$,]', '', regex=True)
    .replace('[,]', '.', regex=True)
    .astype(float)
)

# Función para crear nodos y relaciones en Neo4j
def create_transaction_relationships(tx, customer_id, product_id, transaction_id, transaction_date, standard_cost):
    tx.run(
        """
        MERGE (c:Customer {customer_id: $customer_id})
        MERGE (p:Product {product_id: $product_id})
        MERGE (c)-[:TRANSACTION {transaction_id: $transaction_id, transaction_date: $transaction_date,
standard_cost: $standard_cost}]->(p)
        """,
        customer_id=customer_id, product_id=product_id, transaction_id=transaction_id,
        transaction_date=transaction_date, standard_cost=standard_cost
    )

# Crear nodos y relaciones usando sesiones
with driver.session() as session:
    for _, row in data_subset.iterrows():
        session.execute_write(
            create_transaction_relationships,
            row["customer_id"], row["product_id"], row["transaction_id"],
            row["transaction_date"], row["standard_cost"]
        )

# Cerrar la conexión
driver.close()
```

- user y password: Credenciales de usuario.
- host y port: Dirección y puerto donde está corriendo el servidor.

```
from pyspark.sql import SparkSession
from neo4j import GraphDatabase
import pandas as pd
import psycopg2

conn = psycopg2.connect(
    dbname="pry02_db",
    user="postgres",
    password="password",
    host="postgres",
    port="5432"
)
```

5.2 Conexión a Neo4j e Inicialización de un cursor

Conexión a Neo4j:

- uri: Define la dirección del servidor Neo4j utilizando el protocolo bolt://.
- GraphDatabase.driver: Inicializa un controlador para conectarse al servidor.
- auth: Recibe un tuple con el nombre de usuario y contraseña para autenticar.

Inicialización de un cursor:

- conn.cursor crea un cursor que se utiliza para ejecutar comandos SQL en PostgreSQL.

```
uri = "bolt://neo4j:7687"
username = "neo4j"
password = "badpassword"

# Crear cursor para Postgres
cursor = conn.cursor()

# Crear la conexión con Neo4j
driver = GraphDatabase.driver(uri, auth=(username, password))
```

```
GTPC_data = []
for record in query_output:
    GTPC_data.append({
        "customer_id": record["c.customer_id"],
        "product_id": record["p.product_id"],
        "standard_cost": record["t.standard_cost"]
        })

GTPC_df = pd.DataFrame(GTPC_data)
GTPC_spark_df = spark.createDataFrame(GTPC_df)

# Consulta de Spark
GTPC_spark_query = GTPC_spark_df.groupBy("customer_id") \
    .sum("standard_cost") \
    .withColumnRenamed("sum(standard_cost)", "total_cost") \
    .orderBy("customer_id")
# Fin consulta de Spark

GTPC_spark_query.show()

GTPC_pandas_df = GTPC_spark_query.toPandas()
```

Inserción de GTPC en PostgreSQL

Creación de tabla:

- Crea una tabla llamada `GTPC` si no existe.
- Define dos columnas:
  - `customer_id`: Llave primaria.
  - `total_cost`: Representa el gasto total.

Inserción de datos:

- Convierte los datos procesados por Spark (almacenados en pandas) en una lista de tuplas para PostgreSQL.
- Usa executemany para insertar múltiples filas de manera eficiente.

- t.standard_cost: Costo de la transacción.

Ejecución de la consulta:

- Usa la función execute_cypher_query para enviar la consulta al servidor Neo4j y obtener los resultados en query_output.

```
query = """
MATCH (c:Customer)-[t:TRANSACTION]->(p:Product)
RETURN c.customer_id, p.product_id, t.standard_cost
"""
query_output = execute_cypher_query(query)
```

5.6 GTPC (Gasto Total por Cliente)

Preparación de los datos:

- Itera sobre los registros de query_output (resultado de Neo4j).
- Extrae y organiza los valores en un diccionario con las claves customer_id, product_id y standard_cost.
- Los diccionarios se almacenan en la lista GTPC_data.

Conversión a DataFrame:

- Pandas: Convierte GTPC_data en un DataFrame (estructura tabular).
- Spark: Transforma el DataFrame de pandas en un DataFrame compatible con Spark.

Análisis con Spark:

- Agrupa los datos por customer_id.
- Calcula la suma de standard_cost para cada cliente.
- Renombra la columna de la suma como total_cost.
- Ordena los resultados por customer_id.

```
PMC_data = []
for record in query_output:
    PMC_data.append({
        "customer_id": record["c.customer_id"],
        "product_id": record["p.product_id"]
        })

PMC_df = pd.DataFrame(PMC_data)
PMC_spark_df = spark.createDataFrame(PMC_df)

PMC_spark_query = PMC_spark_df.groupBy("product_id") \
    .count() \
    .withColumnRenamed("count", "total_count") \
    .orderBy("total_count", ascending=False)

PMC_spark_query.show()

PMC_pandas_df = PMC_spark_query.toPandas()
```

Inserción de GTPC en PostgreSQL

- Similar al caso anterior

5.8 PGPC (Promedio General Por Cliente)

Preparación de los datos:

- Similar a GTPC, organiza los resultados en una lista de diccionarios con customer_id y average_cost.

Conversión a DataFrame:

- Idéntico al procedimiento de GTPC y PMC.

Análisis con Spark:

- Ordena los clientes por el ID para asegurar que los datos sean fáciles de interpretar y estén en orden lógico.

```
# Crear tabla en Postgres
cursor.execute("""
CREATE TABLE IF NOT EXISTS GTPC (
    customer_id INT PRIMARY KEY,
    total_cost FLOAT
)
""")
conn.commit()

# Insertar datos en la tabla
insert_query = "INSERT INTO GTPC (customer_id, total_cost) VALUES (%s, %s)"
data = [tuple(row) for row in GTPC_pandas_df.to_numpy()]
cursor.executemany(insert_query, data)
conn.commit()
```

5.7 PMC (Productos Más Comprados)

Preparación de los datos:

- Similar al caso de GTPC, organiza los resultados en una lista de diccionarios con las claves product_id y purchase_count.

Conversión a DataFrame:

- Mismo procedimiento que GTPC: se convierten los datos a un DataFrame de pandas y luego a uno de Spark.

Análisis con Spark:

- Ordena los productos por la columna purchase_count en orden descendente para conservar los productos más comprados en la parte superior.

```python
FCPC_data = []
for record in query_output:
    FCPC_data.append({
        "customer_id": record["c.customer_id"],
        "product_id": record["p.product_id"]
        })

FCPC_df = pd.DataFrame(FCPC_data)
FCPC_spark_df = spark.createDataFrame(FCPC_df)

FCPC_spark_query = FCPC_spark_df.groupBy("customer_id") \
    .count() \
    .withColumnRenamed("count", "total_transactions") \
    .orderBy("customer_id")

FCPC_spark_query.show()

FCPC_pandas_df = FCPC_spark_query.toPandas()
```

Inserción de GTPC en PostgreSQL

- Similar al caso anterior

```python
PGPC_data = []
for record in query_output:
    PGPC_data.append({
        "customer_id": record["c.customer_id"],
        "product_id": record["p.product_id"],
        "standard_cost": record["t.standard_cost"]
        })

PGPC_df = pd.DataFrame(PGPC_data)
PGPC_spark_df = spark.createDataFrame(PGPC_df)

PGPC_spark_query = PGPC_spark_df.groupBy("customer_id") \
    .avg("standard_cost") \
    .withColumnRenamed("avg(standard_cost)", "avg_cost") \
    .orderBy("customer_id")

PGPC_spark_query.show()

PGPC_pandas_df = PGPC_spark_query.toPandas()
```

Inserción de GTPC en PostgreSQL

- Similar al caso anterior

5.9 FCPC (Frecuencia de Compras Por Cliente)

Preparación de los datos:

- Similar a las otras métricas, organiza los resultados en una lista de diccionarios con customer_id y transaction_count.

Conversión a DataFrame:

- Mismo procedimiento que GTPC, PMC y PGPC.

Análisis con Spark:

- Ordena los datos por el ID del cliente para mantener la consistencia.

6.3 Promedio de Gasto por Cliente (PGPC) / average_spent_per_customer

	customer_id [PK] integer	avg_cost double precision
1	1	551.4872727272727
2	2	640.9366666666666
3	3	815.6775
4	4	413.575
5	5	584.7099999999999
6	6	397.028
7	7	258.42333333333335
8	8	495.78200000000004
9	9	500.74
10	10	329.09399999999994
11	11	523.4333333333334
12	12	407.74000000000007
13	13	485.3371428571428
14	14	265.1633333333336

Total rows: 1000 of 3494 Query complete 00:00:00.125

6.4 Frecuencia de Compra por Cliente / transaction_count_per_customer

	customer_id [PK] integer	total_transactions integer
1	1	11
2	2	3
3	3	8
4	4	2
5	5	6
6	6	5
7	7	3
8	8	10
9	9	6
10	10	5
11	11	6
12	12	7
13	13	7
14	14	3

Total rows: 1000 of 3494 Query complete 00:00:00.110

6. Resultados

6.1 Gasto Total por Cliente (GTPC) / total_spent_per_customer

	customer_id [PK] integer	total_cost double precision
1	1	6066.36
2	2	1922.81
3	3	6525.42
4	4	827.15
5	5	3508.2599999999998
6	6	1985.14
7	7	775.27
8	8	4957.820000000001
9	9	3004.44
10	10	1645.4699999999998
11	11	3140.6000000000004
12	12	2854.1800000000003
13	13	3397.3599999999997
14	14	795.49

Total rows: 1000 of 3494 Query complete 00:00:00.326

6.2 Productos más Comprados (PMC) / product_purchase_count

	product_id [PK] integer	total_count integer
1	0	1181
2	1	311
3	2	240
4	3	354
5	4	241
6	5	222
7	6	186
8	7	194
9	8	136
10	9	201
11	10	188
12	11	170
13	12	224
14	13	180

Total rows: 101 of 101 Query complete 00:00:00.156