Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
Primer Semestre de 2024

IIC 2440 – Procesamiento de Datos Masivos
Tarea 2

1. Enunciado

En esta tarea vamos a programar un algoritmo basado en map-reduce para encontrar ocurrencias de un patrón de grafos. La gracia de tu algoritmo es que la cantidad de reducers a usar va a ser un parámetro: si el subgrafo tiene ℓ nodos entonces tu algoritmo podrá funcionar usando b^ℓ reducers, donde b es cualquier entero.

Trabajaremos sobre knowledge graphs, asumiendo siempre que tienen un conjunto V de nodos, L de etiquetas de aristas y que sus aristas son un subconjunto de $X \times L \times V$.

1.1. Precalentamiento: triángulos

Un triangulo dirigido contempla aristas (x, R, y), (y, S, z), (z, T, x), para variables x, y, z y etiquetas de aristas R, S, T.

En un grafo, vamos a identificar el triángulo con una tupla de tres nodos: (n_1, n_2, n_3), que correspoinden a la respuesta de la consulta del patrón del triángulo: los nodos que hacen match a las variables x, y e z respectivamene: es decir, el triángulo (n_1, n_2, n_3) en un grafo corresponde a aristas (n_1, R, n_2), (n_2, S, n_3), (n_3, T, n_1).

Para procesar triángulos con map-reduce usamos la siguiente estrategia, que requiere de una función de hash $h : V \to 0, \ldots, b-1$.

La idea es agrupar todos los triángulos en los b^3 grupos dados por $\{0, \ldots, b-1\}^3$: desde el $(0, 0, 0)$ al $(b-1, b-1, b-1)$. Para eso, vamos a ir hasheando nodos de forma de dirigir los nodos solo a los potenciales reducers que podrían encontrar un triángulo con esos nodos.

Fase de Map. Cada mapper tendrá una cierta cantidad de aristas del grafo. Para cada arista (n_1, R, n_2), el mapper debe hacer lo siguiente.

1. Hashear los nodos: $b_1 = h(n_1), b_2 = h(n_2)$.
2. Como el triángulo que formemos solo puede mapear la variable x a n_1 e y a n_2[1], la arista (n_1, R, n_2) solo podría formar parte de un tríangulo de la forma (n_1, n_2, a) para un $a \in V$. Entonces, generamos b llaves $(b_1, b_2, 0), \ldots, (b_1, b_2, b-1)$, y emitimos b mensajes con el contenido de la arista, uno para cada llave: $((b_1, b_2, 0), (n_1, R, n_2)), \ldots, ((b_1, b_2, b-1), (n_1, R, n_2))$.

Fase de Reduce. Los reducers reciben siempre los mensajes correspondientes a alguna llave (b_1, b_2, b_3). Pero en el valor de esa llave se encuentran aristas. Todas estas aristas forman un pequeño grafo , y el reducer emite como respuesta todas las tuplas (n_1, n_2, n_3) correspondientes a los triángulos que detecta en su grafo.

1.2. Precalentamiento2: triángulos que usen cualquier arista

El patrón $(x, u, y), (y, v, z), (z, w, x)$ usa variables en los nodos y en las aristas. Como consulta, entrega todos los triángulos entre tres nodos, independiente de que arista los conecta.

Ahora cuando un mapper quiere saber que hacer con una arista (n_1, r, n_2), esta arista podría hacer match con tres partes del grafo: (x, u, y), o bien (y, v, z), o bien (z, w, y). Entonces, el mapper debería emitir los siguientes mensajes:

[1]Esto por que la única arista con etiqueta R en el triángulo es (x, R, y).

- Primero, pensando en que n_1 puede hacer match con x y n_2 con y, el mapper debe generar llaves $(h(n_1), h(n_2), 0), dots, (h(n_1), h(n_2), b-1)$, y emitir los mensajes que corresponden a esa llave con el valor (n_1, r, n_2).
- Pensando en que n_1 puede hacer match con y y n_2 con z, el mapper debe generar llaves $(0, h(n_1), h(n_2)), dots, (b-1, h(n_1), h(n_2))$, y emitir los mensajes que corresponden a esa llave con el valor (n_1, r, n_2).
- Finalmente, pensando en que n_1 puede hacer match con z y n_2 con x, el mapper debe generar llaves $(h(n_2), 0, h(n_1)), dots, (h(n_2), b-1, h(n_1))$, y emitir los mensajes que corresponden a esa llave con el valor (n_1, r, n_2).

2. Problemas a resolver

- Implementa una función que reciba un grafo en Neo4j y genere una RDD con las aristas de ese grafo. **NOTA: Vamos a aprender a trabajar con Neo4j el 05/07. Por mientras puedes usar un arreglo de aristas en el github del curso como una RDD de prueba.**
- Implementa un programa en PySpark que entregue todos los triángulos (como tuplas de tres nodos) en el grafo usando b^3 reducers, donde b es un parámetro. Para esta primera parte puedes asumir que tu grafo solo usa una etiqueta de arista (en el grafo de prueba, esa etiqueta corresponde al numero 11).
- Asume ahora que recibes un subgrafo como tres arreglos: un arreglo A con las variables, otro L con los tipos de aristas, y una matriz M de tamaño $|A| \times |L| \times |A|$ que tiene un uno en la posicion (x, R, y) si y solo si (x, R, y) es una arista de tu subgrafo.
- Implementa un programa en PySpark que reciba un patrón que tiene solo variables, y exactamente cuatro variables, y entregue todos los matches de ese patrón (como tuplas de 4 nodos) en el grafo usando b^4 reducers, donde b es un parámetro.

3. Detalles académicos

3.1. El video que debes entregar

Debes entregar un video donde nos expliques la solución general que desarrollaste. Nos importa (1) que nos expliques sin código (a alto nivel) lo que codificaste, y (2) que nos muestres en tu collab y nos cuentes como tu solución se implementa en PySpark.

3.2. Información importante

Esta tarea debe resolverse en grupos de dos personas. El formato de entrega consta de los siguientes archivos:

- Un link a un repositorio público en Github donde se encuentre todo el código necesario para correr los códigos. Debes subir un notebook, que se pueda probar en Google Colab, donde este claramente definido el algoritmo para el triángulo y para los subgrafos en general.

  Importante: nos interesa que nos muestres que tu código sea escalable, por lo que debes probarlos con grafos de tamaño considerable. Debes darnos las instrucciones de como correr todo en Google Colab en el Readme del repositorio.
- El link del video donde expliques tu solución.

Fechas. La fecha de entrega de la tarea es el 18 de Junio, a las 20:00 hrs.