MapReduce

Informe de la Séptima Práctica de Laboratorio

Bases de RDD API de Spark e Implementación de Algoritmos MapReduce

Diego Moreno y Daniel Lagos

Grupo 13 CAIM

Universitat Politècnica de Catalunya

Noviembre 2025

Índice

Construir índice invertido

Objetivos

Usando RDDs, escribe una función para construir un índice invertido para *titles_rdd*. Debería dar como salida un RDD que contenga todos los pares (*término*, *lista_de_publicaciones*). Cada palabra que no es una *stopword* debe aparecer exactamente una sola vez en la salida, y la lista de publicaciones debe estar ordenada mediante un *sortByKey()*.

Procedimiento y Resultados

A partir del código proporcionado por el enunciado hemos elaborado el siguiente código para llevar a cabo el índice invertido:

```
def inverted_index(rdd):
    return (rdd
        .flatMap(lambda tup: ( (word, tup[0]) for word in
tokens(tup[1]) if word not in stopwords_broadcast.value ))
        .groupByKey()
        .mapValues(lambda ids: sorted(list(ids)))
        .sortByKey()
    )
```

En resumidas cuentas, lo que hace esta función es leer las palabras de cada título descartando stopwords y guardarse pares de la palabra junto al id del documento en el que se ha encontrado. Posteriormente agrupa todas las parejas según la palabra, con lo cual se juntaran todos los IDs de documentos en los que aparece cada palabra. Finalmente, ordena de menor a mayor la lista de IDs de cada palabra y ordena alfabéticamente el índice completo para facilitar las búsquedas.

Computar Pagerank con Damping

Objetivos

Usando RDDs, escribe una función para computar un *Pagerank* con *damping*. Asumimos que cada nodo trabajador tiene una memoria de coste lineal, $\Theta(n)$, que será suficiente para almacenar un *Pagerank* o vector de grado, pero no lo suficiente para almacenar la matriz de adyacencia entera.

Algunas subtareas que deberemos resolver con Spark son:

- Computar un RDD con el grado de salida de todos los nodos.
- Computar las probabilidades resultantes de un único paso aleatorio en el grafo, dado el vector *pr* actual.

De lo demás (el stopping, damping y teletransporte) se ocupa el driver, es decir, el programa principal.

Procedimiento y Resultados

```
def get_sinks(outdeg_rdd):
    # Collect the out degrees into the driver program's memory
    outdeg = np.zeros(n)
    for (key, val) in outdeg_rdd.collect():
        outdeg[key] = val

    sinks = np.where(outdeg == 0)[0]
    print("%i dead-end nodes" % len(sinks))
    outdeg[sinks] = 1                              # avoid division by
zero
    return sinks, outdeg

def compute_outdegrees_rdd(edges_rdd):
    return edges_rdd.map(lambda x: (x[0], 1)).reduceByKey(lambda a, b:
a + b)

def pagerank(edges_rdd, n, damping, teleport=None, tol=1e-5,
max_iters=10):
    # Pagerank vector, initially uniform
    pr = np.full(n, 1.0 / n)

    outdeg_rdd_calc = compute_outdegrees_rdd(edges_rdd)
    sinks, outdeg = get_sinks(outdeg_rdd_calc)

    # Teleport vector, uniform if not provided
    if teleport is None:
        teleport = np.ones(n)

    # while not (termination condition):
    for i in range(max_iters):
        pr_nowhere = np.sum(pr[sinks])
        pr_teleport = (1.0 - damping) + damping * pr_nowhere
```

```
        # Compute probabilities without teleportation at the next step
        pr_divided = sc.broadcast(pr / outdeg)
        step = (edges_rdd
            .map(lambda x: (x[1], pr_divided.value[x[0]]))
            .reduceByKey(lambda a, b: a + b)
        )

        # Now account for teleportation
        pr = np.full(n, pr_teleport / n)
        for key, val in step.collect():
            pr[key] += damping * val

        print(f"Iteracion {i+1} completada")

    return pr
```

Compute_outdegrees calcula el grau de sortida de cada node i retorna un RDD de parells de la forma [doc_id, degree]. Per la seva banda, **get_sinks** retorna tots aquells nodes que no tenen cap sortida (sinks) i, a més, encara que incorrecte, assigna el seu grau a 1 per no caure en divisions per 0 a l'hora de fer càlculs.

La función pagerank calcula iterativamente la relevancia de cada página web simulando la navegación de usuarios aleatorios sobre el grafo. En cada ciclo, gestiona la "masa perdida" en los nodos sin salida (sinks) y aplica un factor de amortiguación (damping) para garantizar que el sistema converja y no se pierda energía. El núcleo del cálculo utiliza el modelo MapReduce de Spark que difunde la importancia actual de cada nodo dividida por sus enlaces salientes, distribuye esos valores a los destinos mediante map y suma todas las contribuciones recibidas en cada página con reduceByKey para actualizar el vector global de probabilidades.

Preguntas adicionales

1. **La lógica de teletransporte podría simplificarse si computamos la matriz Google por adelantado. ¿Qué ocurriría entonces?**

   Pese a que computar la matriz por adelantado podría parecer idóneo en teoría, en la práctica este procedimiento aumenta la complejidad del procedimiento a $O(n^2)$,

haciéndolo impráctico para grafos de gran tamaño. Por este motivo, se recomienda la teletransportación implícita.

2. **Comparamos con nuestra solución secuencial del Lab 6. ¿Son iguales? ¿Cuál es más rápida? ¿Por qué creemos que es el caso?**

   En teoría, los resultados obtenidos en los Labs 6 y 7 deben ser equivalentes. El método secuencial suele resultar más eficiente en grafos que caben en una sola máquina, debido a que genera menos overhead. Si el grafo es demasiado pesado como para caber en memoria RAM o se busca paralelizar en el cluster, la implementación distributiva es la más rápida.

   El motivo principal de este fenómeno es el overhead de comunicación y coordinación del sistema distribuido. En un sistema distribuido, hay un conjunto de tareas que el programa debe llevar a cabo, aunque el tamaño del grafo sea pequeño, como el shuffling, la serialización y deserialización de objetos, escribir y leer del disco cuando falta RAM, entre otros. Este coste se añade al coste O(m) del resto de la ejecución distribuida.

3. **Supón que hemos precomputado un RDD con la lista de próximos vecinos para cada vértice usando *groupByKey* y lo hemos usado en el bucle principal en vez de sumar contribuciones de aristas individuales. ¿Es esta aproximación siempre más rápida o puede volverse problemática?**

No siempre será más rápido. La implementación puede reducir costes de algunas operaciones, como por ejemplo evitar tener que transponer el grafo en cada iteración, pero también puede introducir nuevos costes, como un coste añadido de precomputación, un mayor uso de memoria y disco, o casos de ineficiencia resultantes de un particionado inadecuado, ya que el puntero del Pagerank y el RDD de incoming no comparten la misma partitioner.

4. **¿Y si *n* fuera demasiado grande como para almacenar el Pagerank entero o los vectores de grado en memoria? ¿Qué haríamos?**

   En ese caso, hay varias opciones disponibles.

   Por ejemplo, se podrían distribuir mediante un RDD con persistencia MEMORY_AND_DISC. Para ello, se presentarían el puntero del Pagerank y el grado como un RDD de pares (IDnodo, valor) persistido: persist(MEMORY_AND_DISC).

Otra opción es implementar frameworks de grafo, ya que están diseñados para almacenar el estado de los vértices de forma distributiva y eficiente, además que soportan persistencia y optimizadores de mensajería.

También se puede aplicar una decomposición por bloques, particionando el grafo en submatrices de adyacencia. De esta manera, se guardarían los subvectores de Pagerank por bloque y las multiplicaciones se llevarán a cabo de bloque a bloque. La comunicación se reduciría y se evita tener todo el vector presente en una única máquina.