TCDM Práctica 3

José Antonio

- Introducción

Todos los programas de MapReduce con mrjob suelen seguir una estructura parecida.

Un programa de MapReduce con mrjob se define como una secuencia de pasos (steps). Cada paso normalmente representa un job MapReduce y se describe con MRStep(...). En cada paso podemos definir:

- mapper (opcional, pero lo normal es que exista)
- combiner (opcional)
- reducer (opcional; si no hay reducer, es un map-only job)

Al principio y fin de cada paso (MRStep) se aplica un protocolo que define la entrada y salida del paso. Por defecto, el protocolo de salida es JSONProtocol, pero como veremos lo podemos cambiar para adaptarlo a nuestras necesidades.

También es importante saber que después del mapper y combiner los pares clave-valor se ordenan por su clave antes de llegar al siguiente paso. También hay una opción para ordenarlos por valor SecondarySorting, pero no la vamos a usar, el flujo de un paso quedaría:

Input → Mapper → [SORT BY KEY] → Combiner → [SORT BY KEY] → Reducer → Output

Finalmente, otra característica de MRJob que tenemos que entender para comprender los ejercicios, es que normalmente tenemos varios reducer ejecutándose en paralelo, una clave con todos sus valores va a ir al mismo reducer. Por lo que si queremos que todos los datos vayan al mismo reducir los tenemos que poner con la misma clave.

- EJ1: so_count_answers_ordered.py:

- Primer MRStep
1º) mapper_count
Lee cada línea del CSV, ignora la cabecera y parsea la fila. Filtra para quedarse solo con los registros cuyo PostTypeId es 2 (es decir, respuestas). De cada respuesta extrae el ParentId (la pregunta a la que responde) y emite el par (ParentId, 1) para contar una respuesta más para ese ParentId.

2º) combiner_count
Para cada ParentId, suma localmente los 1 generados por el mapper en ese mismo nodo y emite (ParentId, suma_parcial). Esto reduce la cantidad de datos que llegan a los reducers.

3º) reducer_count
Recibe, para cada ParentId, todos los valores asociados (ya combinados o no) y los suma para obtener el total final de respuestas por pregunta. Emite (None, (ParentId,

total_respuestas)). Devolvemos los pares clave valor con clave a None y valor la tupla de pregunta y número de respuestas. Esto, como ya comentamos en la introducción, para que todos tengan la misma clave y vayan al mismo reducer, de manera que los podamos ordenar todos en conjunto en el siguiente paso.

- Segundo MRStep
1º) reducer_sort
Recibe un par clave-valor con la clave a None y todos los datos, los ordena por número de respuestas.

- EJ1: so_count_answers_topk.p

- Primer MRStep
1º) mapper_count
Lee cada línea del CSV, ignora la cabecera (líneas que empiezan por "Id") y parsea la fila con csv.reader. Filtra para quedarse solo con los registros cuyo PostTypeId es 2. De cada respuesta extrae el ParentId, comprueba que exista y emite el par (ParentId, 1) para contar una respuesta más asociada a esa pregunta.

2º) combiner_count
Para cada ParentId, suma localmente los 1 generados por el mapper en ese mismo nodo y emite (ParentId, suma_parcial).

3º) reducer_count
Recibe, para cada ParentId, todos los valores asociados (ya combinados o no) y los suma para obtener el total final de respuestas por pregunta. Emite (ParentId, total_respuestas).

- Segundo MRStep
1º) mapper_bucketize
Toma la salida del paso 1 (ParentId, count) y la reparte en “buckets” calculando bucket = ParentId % num_buckets (de esta manera siempre tendremos de 0 a num_buckets claves). Para cada elemento emite (bucket, (count, ParentId)), guardando el par (count, ParentId) para poder comparar por número de respuestas y desempatar por ParentId. Es decir para cada bucket un conjunto “aleatorio” de pares (nº respuestas, pregunta)

2º) reducer_topk_per_bucket
Para cada bucket, calcula un top-k local usando un min-heap de tamaño máximo k (donde k = --top-k).

- Si el heap aún no tiene k elementos, inserta (count, ParentId).
- Si ya está lleno, compara el nuevo par con el mínimo del heap (heap[0]) y, si es mayor, lo reemplaza (heapreplace).
- Al final, emite los candidatos del top-k local como (None, (count, ParentId)) para enviarlos a una única clave común en el siguiente paso.

- Tercer MRStep
1º) reducer_topk_global

Misma lógica que el ejercicio anterior, recibe todos los candidatos (count, ParentId) bajo la clave None y repite la lógica de heap para mantener únicamente los k mejores globales. Finalmente, ordena los k elementos del heap en orden descendente (sorted(..., reverse=True)) y emite la salida final como (ParentId, count) ya ordenada de mayor a menor número de respuestas (desempate por ParentId porque el par comparado es (count, ParentId)).

- EJ2: so_bytagyear.py

Único MRStep
1º) mapper
Lee cada línea del CSV ignora la cabecera si empieza por Id y parsea la fila con csv.reader Filtra para quedarse solo con los registros cuyo PostTypeId es 1 es decir preguntas Extrae CreationDate y toma los primeros 4 caracteres como año si no son dígitos descarta el registro, extrae el campo Tags, comprueba que exista y obtiene la lista de etiquetas con una expresión regular que extrae el texto entre < y >, para cada tag emite la clave en formato "(año,tag)" junto con el valor 1 para contar una pregunta más en ese año para ese tag

2º) combiner
Para cada clave "(año,tag)" suma localmente los unos producidos por el mapper en ese nodo y emite ("(año,tag)", suma_parcial) reduciendo datos en el shuffle

3º) reducer
Recibe para cada clave "(año,tag)" todos los valores asociados y los suma para obtener el total final de preguntas por tag y año Emite como salida la clave "(año,tag)" y el conteo convertido a string en formato "(año,tag)\tN".

La clave la tratamos como una cadena de texto y no como una tupla porque por defecto la salida de una tupla está entre corchetes al ser el OUTPUT_PROTOCOL por defecto JSONProtocol:
[2018, "mysqlconnector"] 16
[2018, "mysqlconnector.jar"] 4
[2018, "mysqldb"] 9
Por eso ponemos el OUTPUT_PROTOCOL como TextProtocol. Para definir nosotros el formato de salida como un texto.

- EJ3: so_join.py

Único MRStep
1º) mapper
Lee líneas de dos ficheros distintos que vienen prefijadas para distinguir el origen en formato "P|<csv...>" para Posts y "U|<csv...>" para Users. Si la línea no cumple el patrón con barra vertical en la segunda posición se descarta. Luego separa el tipo de registro rec_type como el primer carácter y raw como el resto después de "X|". Para registros de tipo P parsea el CSV y filtra para quedarse solo con preguntas y extrae OwnerUserId. Convierte OwnerUserId a entero y toma el Id de la pregunta, emite como clave el OwnerUserId y como

valor una tupla marcada ("P", question_id, "") para indicar que ese usuario es autor de esa pregunta. Para registros de tipo U parsea el CSV extrae UserId y si falta descarta Convierte UserId a entero extrae Reputation y DisplayName, emite como clave el UserId y como valor una tupla marcada ("U", rep, name) para aportar los datos del usuario.

En otras palabras, el reducer va a recibir un par clave-valor por cada usuario, donde la clave es el usuario y los valores es una lista de tuplas donde hay una de tipo “U” con la información de su reputación y el resto de tuplas de tipo P con los id’s de las preguntas que ha formulado.

2º) reducer
Recibe agrupado por owner_id todas las tuplas de ese usuario tanto de tipo U como de tipo P, recorre los valores guardando la reputación y el nombre cuando ve el registro marcado con U y acumulando en una lista los question_ids cuando ve registros marcados con P. Para cada question_id acumulado emite como salida de texto con TextProtocol la clave qid y como valor el string "(rep,name)" quedando en formato "qid\t(rep,name)" donde qid es el identificador de la pregunta y rep y name pertenecen al autor

- EJ4: so_tagcooc.py

- Único MRStep

1º) mapper
Filtramos las preguntas y obtenemos la lista de etiquetas con una expresión regular que extrae el texto entre < y >, si hay menos de 2 tags descarta el registro. Para asegurar consistencia elimina duplicados con set y ordena las etiquetas, luego genera todos los pares únicos de etiquetas por pregunta usando combinations(tags, 2) y para cada par emite (tag1,tag2) como clave en formato "t1,t2" junto con el valor 1 para contar una ocurrencia de ese par.

2º) combiner
Para cada clave "t1,t2" suma localmente los unos producidos por el mapper en ese nodo y emite ("t1,t2", suma_parcial) reduciendo datos en el shuffle

3º) reducer
Recibe para cada par de tags todos los valores asociados y los suma para obtener el total final de coocurrencias, emite como salida de texto con TextProtocol la clave "t1,t2" y el conteo convertido a string en formato "t1,t2\tN"

- EJ5: so_tecnology_evolution.py

Primer MRStep
1º) mapper_join_inputs
Para cada Post devolvemos una tupla clave-valor, donde si el post es una pregunta el valor va a ser el año y mes, año-mes, de formulación y los tags, si el post es una respuesta solo

el año-mes. En ambos casos la clave va a ser el id de la pregunta qid (si es una pregunta su id, si es una respuesta la pregunta a la que está asociada). Además etiquetamos si es pregunta Q o respuesta A.

2º) reducer_join_emit_activity
Recibe para cada qid la mezcla de registros de tipo Q y A Recupera el mes de la pregunta y su tags_field y acumula los meses de las respuestas, extrae las etiquetas del tags_field con la expresión regular TAG_RE elimina duplicados y ordena para consistencia, luego calcula actividad ponderada por (tag, ym) contando 1.0 por cada tag en el mes de la pregunta y 0.5 por cada tag en el mes de cada respuesta usando los tags de la pregunta. Para reducir emisiones hace una pequeña agregación local en un diccionario y finalmente emite como clave (tag, ym) y como valor la actividad float. Este paso usa JSONProtocol internamente para poder transportar tuplas como claves entre pasos.

Segundo MRStep
1º) combiner_sum_activity
Para cada clave (tag, ym) suma localmente las actividades parciales generadas por el paso anterior en ese nodo y emite (tag, ym) con suma_parcial reduciendo datos en el shuffle

2º) reducer_sum_activity
Recibe para cada (tag, ym) todos los valores de actividad y los suma para obtener la actividad total final de ese tag en ese mes, emite (tag, ym) con actividad_total

Tercer MRStep
1º) mapper_to_tag
Toma la salida agregada (tag, ym) -> activity y la reagrupa por tag emitiendo como clave el tag y como valor (ym, activity) para poder procesar la serie temporal completa de cada tecnología

2º) reducer_monthly_deltas
Recibe para cada tag todos los pares (ym, activity) los convierte en lista y los ordena por ym aprovechando que YYYY-MM ordena bien como string, recorre en orden y calcula para cada mes la diferencia delta entre la actividad del mes actual y la del mes anterior y emite tag -> (ym, delta) como salida intermedia para el siguiente paso

Cuarto MRStep
1º) mapper_pass_deltas
Reemite sin cambios tag -> (ym, delta) para que toda la información de cada tag vaya al mismo reducer.

2º) reducer_classify_trend
Recibe para cada tag todos los (ym, delta) los ordena por mes toma los últimos 6 deltas o menos si no hay tantos y calcula el promedio avg. Asigna la tendencia como CRECIMIENTO si avg es positivo DECLIVE si avg es negativo o ESTABLE si avg es cero. Formatea el promedio con un decimal y emite la salida final con TextProtocol como Tag \t tendencia \t promedio_delta \t meses_considerados en una única línea por tag