Sistemas Distribuidos I
(75.74)

Modelado de Computo Distribuido

Implementación y Práctica de *MapReduce*

Docentes

- Pablo D. Roca
- Ezequiel Torres Feyuk
- Guido Albarello
- Ana Czarnitzki
- Cristian Raña

Agenda

- **MapReduce**
- Ejemplos: WordCount / WordFrequencies / Union / Intersect / Join

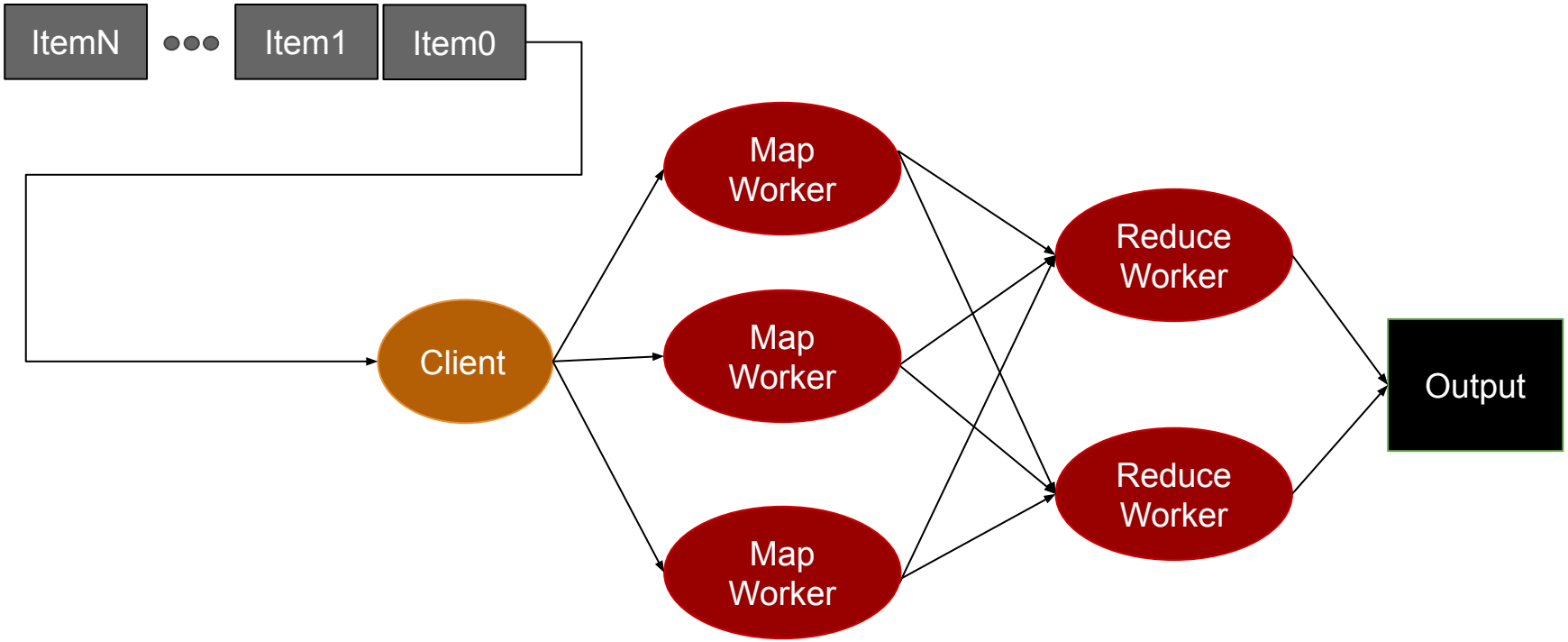
MapReduce | Motivaciones

- Programación tradicional: ejecutada en ambientes serializados
- ***Parallel Computing:*** Partir procesamiento en partes que puedan ser ejecutadas concurrentemente en múltiples cores
- **Desafío**
  - No todos los problemas pueden ser paralelizados
  - Concepto de camino crítico
- **Idea:**
  - Identificar **Tareas** que puedan ser ejecutadas concurrentemente
  - Identificar **Grupos de datos** que puedan ser procesados de forma concurrente

MapReduce | Introducción

- Paradigma de *parallel computing*
- Desarrollado en 2004 por Google
- Ligeramente basado en la idea de funciones **map** y **reduce** de LISP
  - **`map f[a,b,c] => [f(a),f(b),f(c)]`**
    - `map sqrt[a,b,c] => [sqrt(a),sqrt(b),sqrt(c)]`
  - **`reduce f[a,b,c] => f(a,b,c)`**
    - `reduce sum[1,2,3] => sum(1,sum(2,sum(3,sum(NULL))))`
- Implementaciones
  - Apache Hadoop
  - Amazon EMR
  - Google MapReduce (for AppEngine)

MapReduce | Arquitectura Naive



MapReduce | Parallel Computing - Caso ideal (Master-Worker)

- No existe dependencia entre los datos
- Datos pueden ser partidos en ***chunks*** del mismo tamaño
- Cada proceso pueden trabajar con un ***chunk/shard***
- **Master**
  - Encargado de partir la data en #***chunks***
  - Envía ubicación de los ***chunks*** a los Workers
  - Recibe ubicación de los resultados de todos los Worker
- **Workers**
  - Recibe ubicación de los ***chunks*** del Master
  - Procesa el ***chunk***
  - Envía el ubicación del resultado de procesamiento al Master

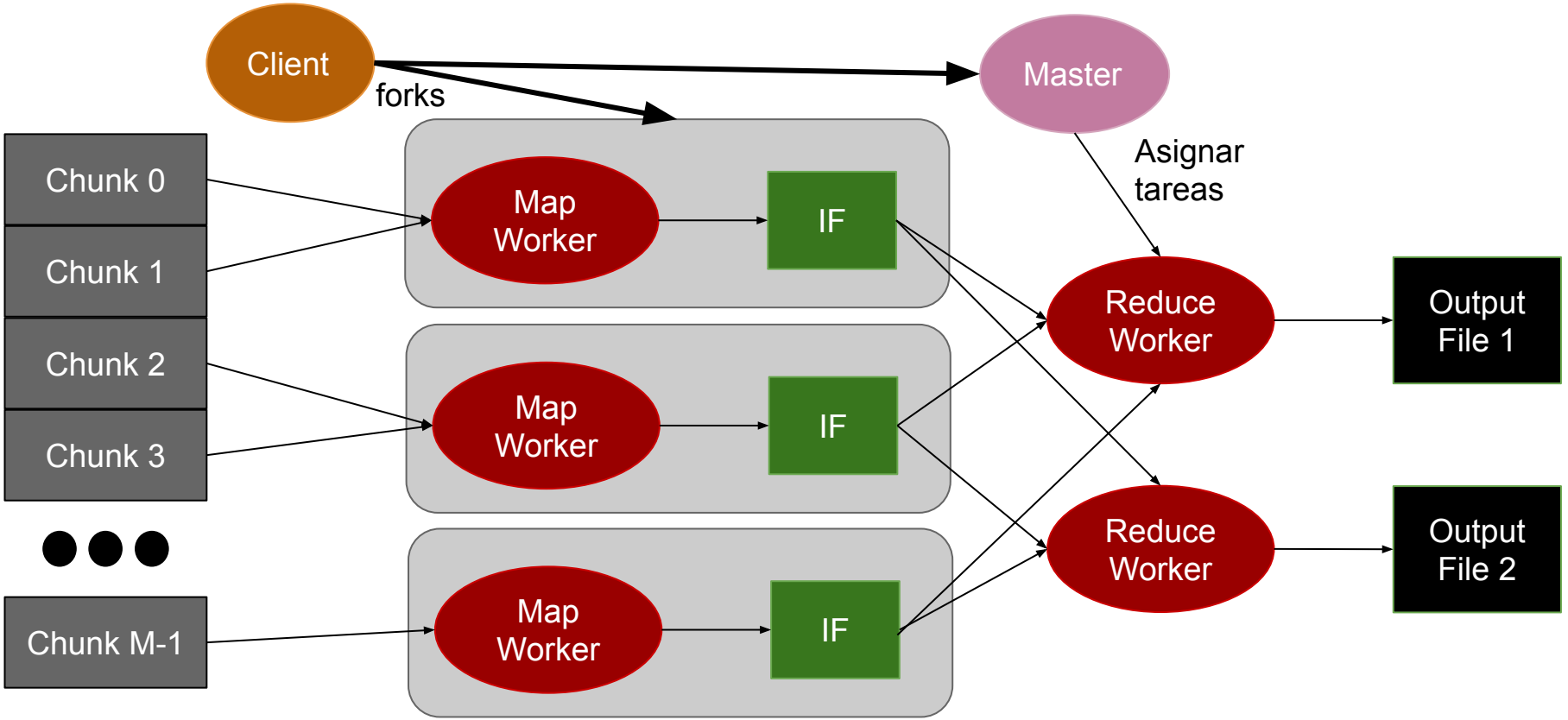
MapReduce | Función Map

- **Map: (input shard) → intermediate(key/value pairs)**
  - Data es particionada automáticamente en **K *chunks*** y procesada por M workers ejecutando la función **map**
  - Función **Map** proporcionada por el usuario es ejecutada en todos los ***chunks*** de data
  - Usuario decide **cómo filtrar la data provista en los *chunks***
  - Librería MapReduce agrupa todos los valores asociados con una misma key y envía ubicación de los datos al **Master Process**

MapReduce | Función Reduce

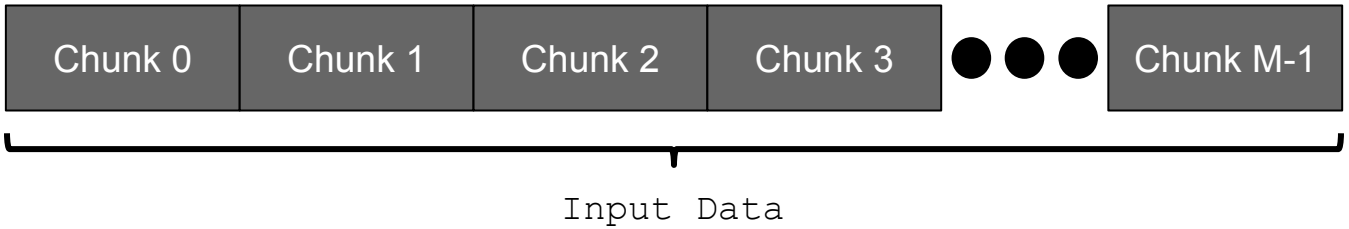
- **Reduce: intermediate(key/value pairs) → result files**
  - Función **Reduce** realiza una agregación de los datos para obtener un resultado final (result file)
  - Función **Reduce** es llamada por cada **Unique Key**
  - Realiza un *merge* de los datos recibidos para formar un set de datos menor
  - Función **Reduce** es distribuida particionando las keys en R Reduce workers
  - La cantidad de R workers es especificada por el usuario

MapReduce | Arquitectura



MapReduce | Paso 1

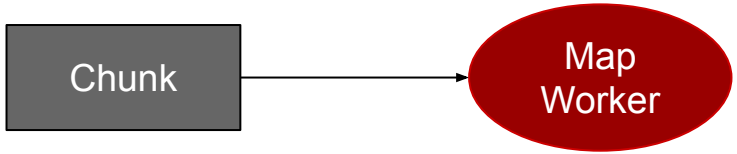
- **Partir la datos datos de entrada en N *chunks***
  - Por lo general *chunks* de 64MB (configurable)

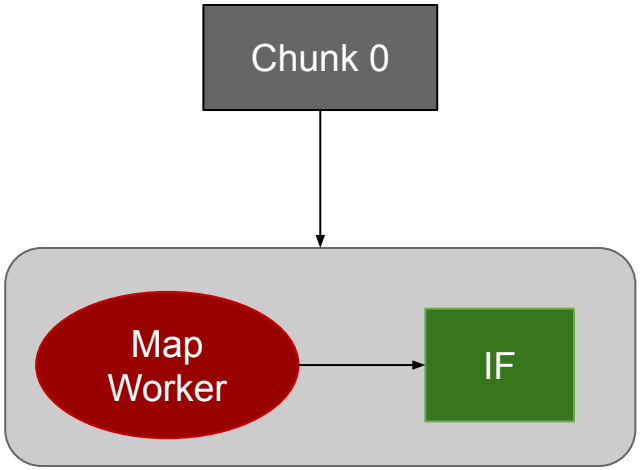


MapReduce | Paso 2

- ***Fork* de Procesos en Cluster**
  - 1 master: Actua de Scheduler y Coordinador
  - Muchos Workers (Mappers y Reducers)
- **Mappers y Reducers**
  - Tantos Mappers como ***Chunks***
  - R reducers (configurados por el usuario)

- ***Map* de Shards en Mappers**
  - Worker lee Input data
  - Filtra los datos recibidos en formato **key/value**
  - Ejecuta función provista por el usuario sobre cada par **key-value** que haya pasado el filtro
    - Por cada par produce un *valor intermedio*





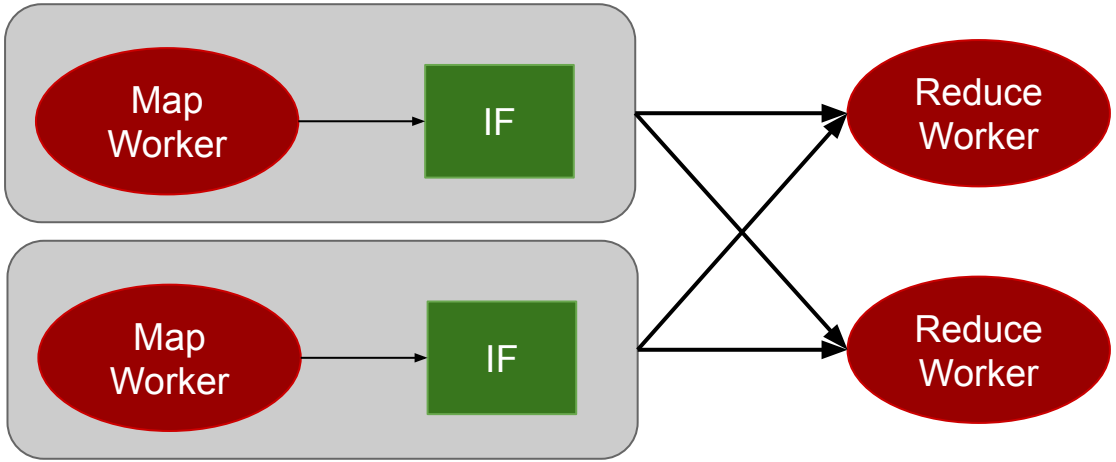
- **Creación de Archivos Intermedios (Intermediate Files o IF)**
  - Cada key/value intermedio producido es buffereado y escrito periódicamente en disco local
  - La data es particionada en R regiones utilizando una función de particionamiento
  - Notifica al **proceso master** cuando haya terminado de procesar el ***chunk*** de datos
  - Envía al proceso **master** ubicación del archivo intermedio
  - **Proceso master** envía ubicación de **IF** a **Reduce workers**

MapReduce | Paso 4b (Particionamiento)

- **Reducción de set de datos**
  - La función Reduce del usuario será llamada una vez por unique key generada por los Map workers
  - Keys/values deben ser agrupados por **Key** y se debe decidir que Reduce Worker se encargará de procesar cada key
- **Función de Partición**
  - Decide **cuál** de los R workers va a trabajar con cada **Key**
  - Función default: **hash(key) mod R**
  - **Map workers** particionan la data por **Keys** con esta función
  - Cada Reduce Worker va a leer la partición que desea de cada **Map Worker**

MapReduce | Paso 5

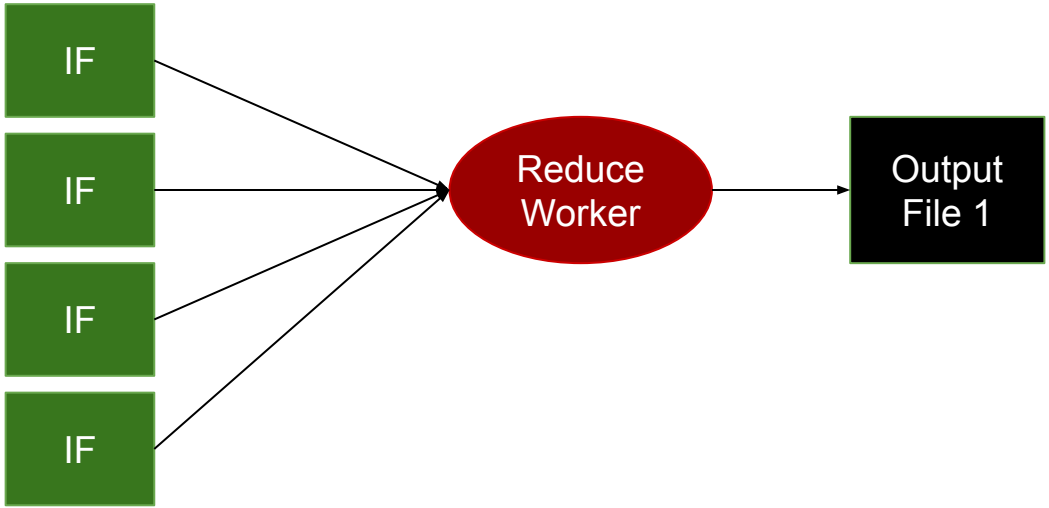
- **Reduce Workers** reciben ubicación de los archivos intermedios para la partición que deben procesar
  - Uso de RPC para leer data del disco local de los Mappers



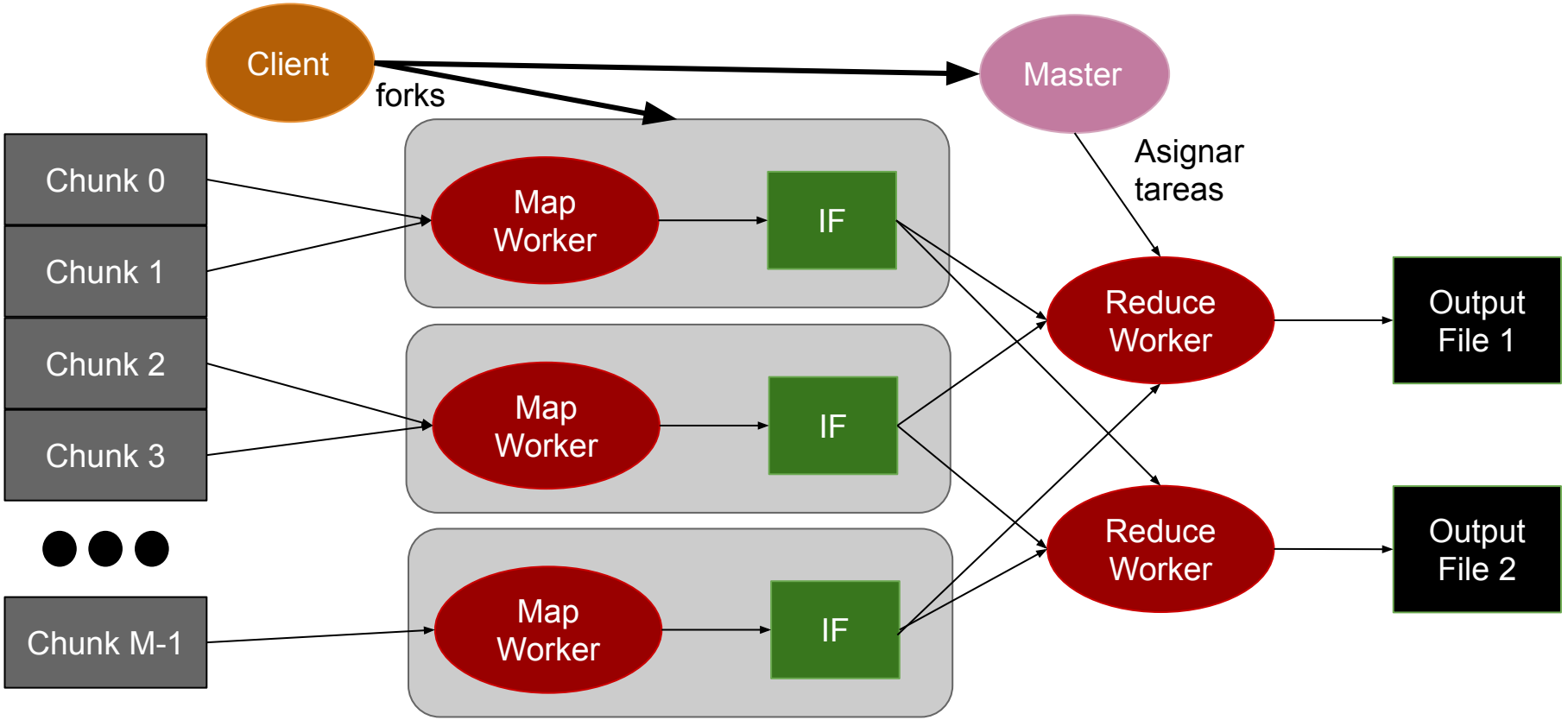
- Cuando los Reduce Workers reciben la data intermedia de cada partición
  - Ordenan la data por **key**
  - Todas las ocurrencias de la misma **Key** son agrupadas

MapReduce | Paso 6

- Reduce worker lee data agrupada por Key...
  - Aplica función del usuario **Reduce** sobre cada set de datos
  - El output de la función es almacenado en un *output file*
  - Se despierta a User process y se le indica ubicación de los *output files*



MapReduce | Arquitectura



Agenda

- MapReduce
- **Ejemplos: WordCount / WordFrequency / Union / Intersect / Join**

MapReduce | Word Count

- Contar la cantidad de ocurrencias de palabras en un texto
- Ejercicio en [HackerRank]

```
map(string key, string value):


reduce(string key, list value):
```

MapReduce | Word Count

- Contar la cantidad de ocurrencias de palabras en un texto
- Ejercicio en HackerRank

```
map(string key, string value):
    // key: document name
    // value: document as a multiline string
    for word w in value:
        emitIntermediate(w, 1)

reduce(string key, list value):
    // key: word
    // value: list of “1s” associated with the words
    emit(key, len(value))
```

MapReduce | Word Frequency

- Contar la frecuencia de una palabra en un documento

```
map(string key, string value):


reduce(string key, list value):
```

MapReduce | Word Frequency

- Contar la frecuencia de una palabra en un documento

```
map(string key, string value):
     // key: document name
     // key: document as a multiline string
     for word w in value:
         emitIntermediate(w, 1)

reduce(string key, list value):
     // key: word
     // value: list of “1s” associated with the words
     emit(key, count(value) / totalWords)
```

MapReduce | Word Frequency

- Contar la frecuencia de una palabra en un documento

```
map(string key, string value):
     // key: document name
     // key: document as a multiline string
     for word w in value:
         emitAll(“”, 1)
         emitIntermediate(w, 1)

reduce(string key, list value):
     // key: word
     // value: list of “1s” associated with the words
     totalWords = reduceAll(“”, sum)
     emit(key, count(value) / totalWords)
```

MapReduce | Union

- Dado N documentos, obtener palabras en uno o en ambos documentos

```
map(string key, string value):


reduce(string key, list value):
```

MapReduce | Union

- Dado N documentos, obtener palabras en uno o en ambos documentos

```
map(string key, string value):
     // key: document name
     // value: document as a multiline string
     dict = {}
     for word w in value:
         if not word in dict:
             dict[word] = 1
             emitIntermediate(word, key)

reduce(string key, list value):
     // key: placeholder
     // value:
     emit(key, key)
```

MapReduce | Intersect

- Dado N documentos, obtener palabras que se encuentren en todos ellos

```
map(string key, string value):


reduce(string key, list value): (word: pepe, list: [asd.txt, pepe.txt, asd.txt])
```

MapReduce | Intersect

- Dado N documentos, obtener palabras existentes en todos los docs

```
map(string key, string value):
     // key: document name
     // value: document as a multiline string
     for word in value:
         emitIntermediate(word, key)

reduce(string key, list value): // key: word, value: list of repetitions
    dictionary = {}
    for v in value:
         if not v in dictionary:
             dictionary[v] = true
    if len(dictionary.keys()) == amountDocuments:
        emit(key,key)
```

MapReduce | Intersect

```
map(string key, string value):
     // key: document name - value: document as a multiline string
     dict = {}
     emitAll(“”, key)
     for word w in value:
         if not word in dict:
             dict[word] = key
             emitIntermediate(word, key) → (word: pepe, document: asd.txt)

reduce(string key, list value): (word: pepe, list: [asd.txt, pepe.txt, asd.txt])
    amountDocuments = reduceAll(“”, (sort, uniq, len))
    dictionary = {}
    for v in value:
         if not v in dictionary:
             dictionary[v] = true
     if len(dictionary.keys()) == amountDocuments:
         emit(key, key)
```

MapReduce | Join

- Dados S1:=[shared_field, field_1] y S2:=[shared_field, field_2], obtener el conjunto final con la forma S3:=[shared_field, field_1, field_2]

```
map(string key, string value):


reduce(string key, list value): // key: join fields, value: tuples
```

MapReduce | Join

- Dados S1:=[shared_field, field_1] y S2:=[shared_field, field_2], obtener el conjunto final con la forma S3:=[shared_field, field_1, field_2]

```
map(string key, string value):
     // key: set names as S1|S2|S3, value: multiline csv as "shared_field,field_X"
     for line in value:
          fields = line.split(',')
          emitIntermediate(fields[0], (key, fields[1]))

reduce(string key, list value): // key: join fields, value: tuples
     data_fields = [key, S1_value, S2_value, None]
     for set_data in value:
          if set_data[0] == 'S1':
               data_fields[1] = set_data[1]
          else: // set_data[0] == 'S2':
               data_fields[2] = set_data[1]
     emit(key,','.join(data_fields))
```

Bibliografía

- MapReduce
  - http://static.googleusercontent.com/media/research.google.com/en//archive/papers/mapreduce-sigmetrics09-tutorial.pdf
- HackerRank - Distributed Systems
  - https://www.hackerrank.com/domains/distributed-systems, Ejercicios 'Relational MapReduce Patterns' y 'MapReduce Tutorials'
- McCool M., Robison A. D., Reinders J., Structured Parallel Programming Patterns for Efficient Computation, 2012, Elsevier-Morgan Kaufmann.
  - Capítulo 4.1 - Map
  - Capítulo 5.1 - Reduce