Sistemas Distribuidos I
(75.74)

Modelado de Computo Distribuido

Implementación y Práctica de *MapReduce*

Docentes

- Pablo D. Roca
- Ezequiel Torres Feyuk

Agenda

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

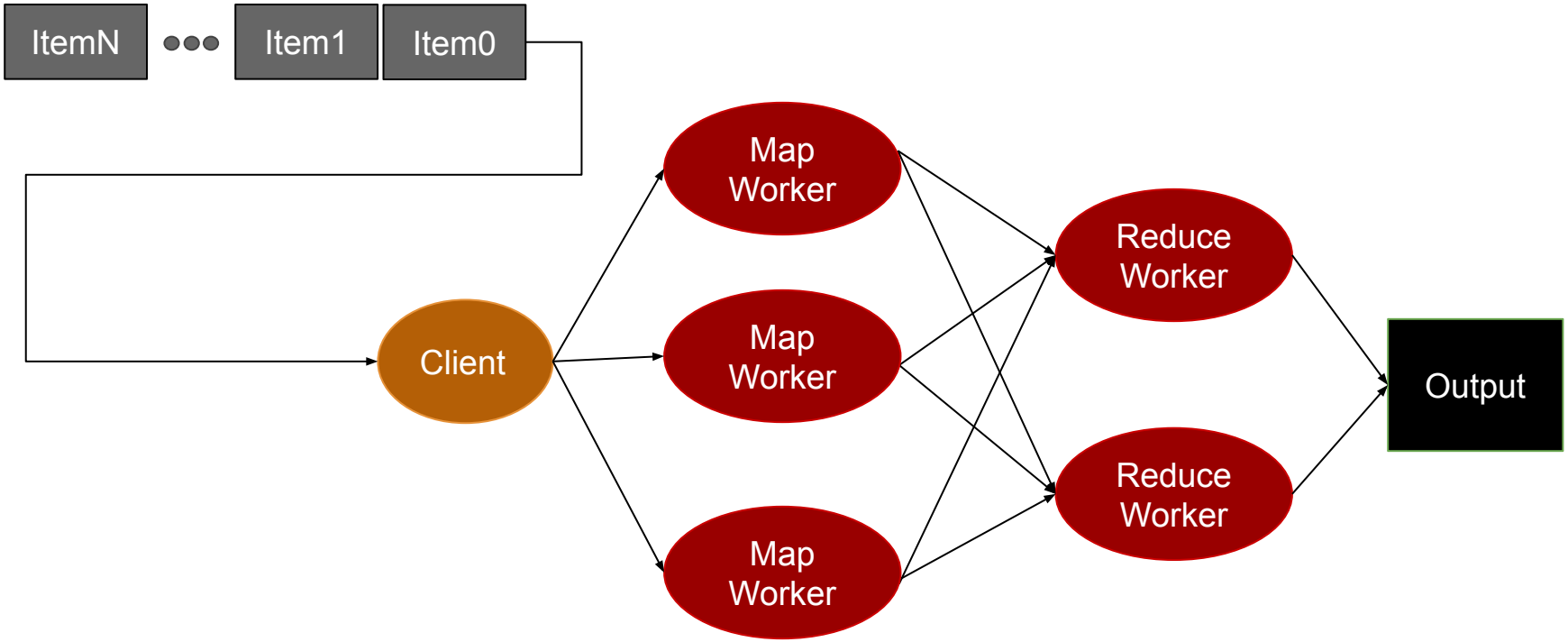
MapReduce | Motivaciones

- 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 I

- 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))))`
- Open Source
- Implementaciones
  - Apache Hadoop
  - Google MapReduce (Deprecado. Reemplazado por tools como Dataflow, Dataproc, empleo intensivo de BigQuery)

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***
- **Master**
  - Encargado de partir la data en # ***chunks***
  - Envía los ***chunks*** a los Workers
  - Recibe los resultados de todos los Workers
- **Workers**
  - Recibe ***chunks*** del Master
  - Procesa el ***chunk*** recibido
  - Envía el resultado del procesamiento al Master

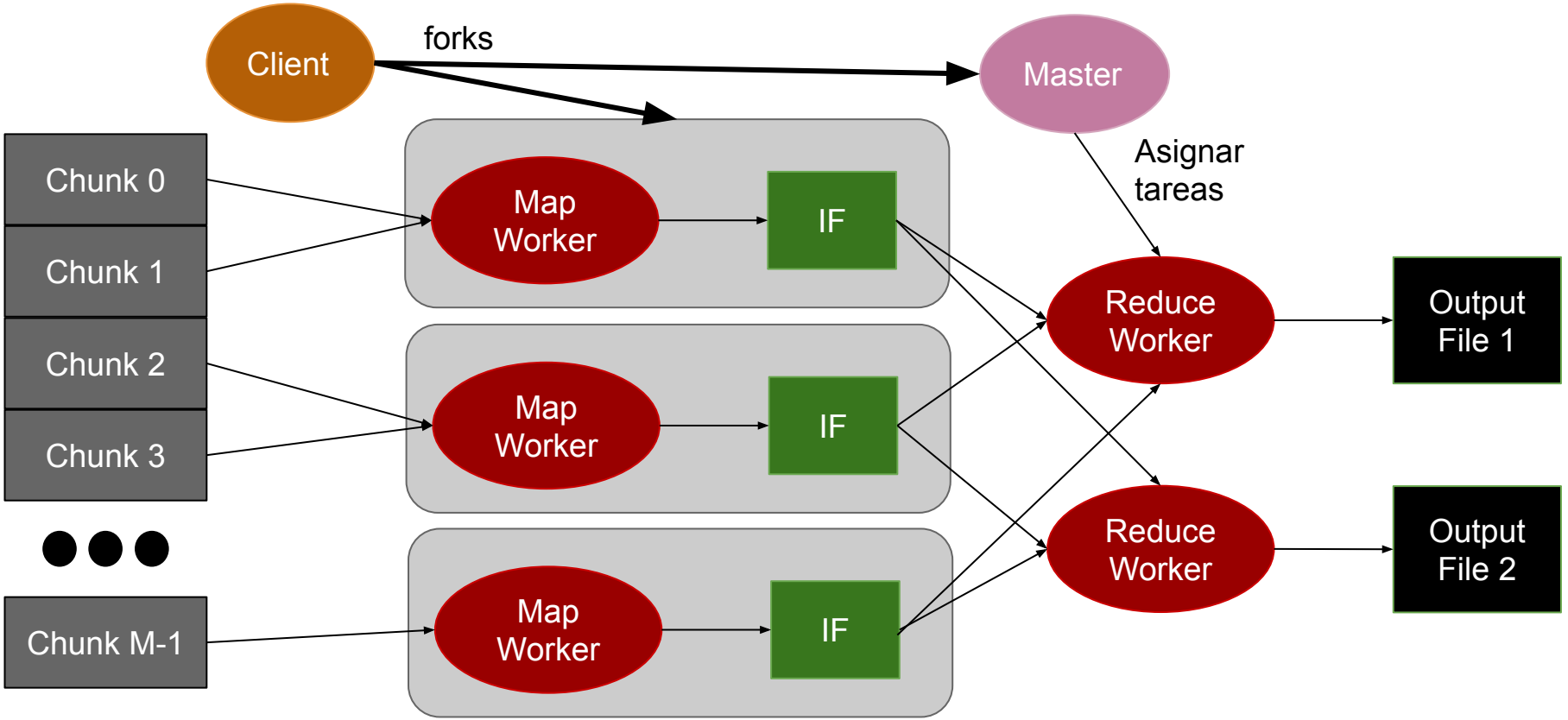
MapReduce | Función Map

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

MapReduce | Función Reduce

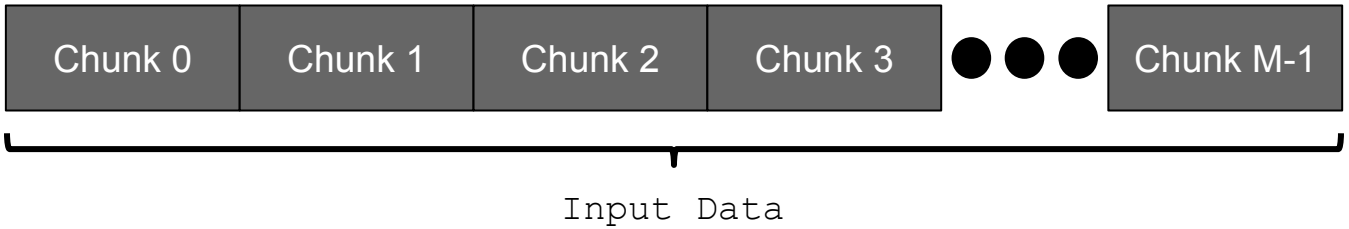
- **Reduce: intermediate(key/value pairs) → result files**
  - Recibe una key intermedia y un set de valores
  - Realiza un *merge* de los datos recibidos para formar un set de datos menor
  - Función **Reduce** es distribuida particionando las keys intermedias en R workers
  - La cantidad de R workers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

MapReduce | Arquitectura



MapReduce | Paso 1

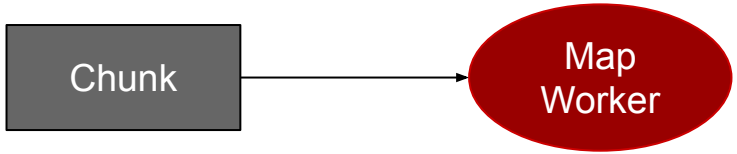
- **Partir la datos datos de entrada en N *chunks***
  - Por lo general *chunks* de 64MB (recuerdan HDFS?)



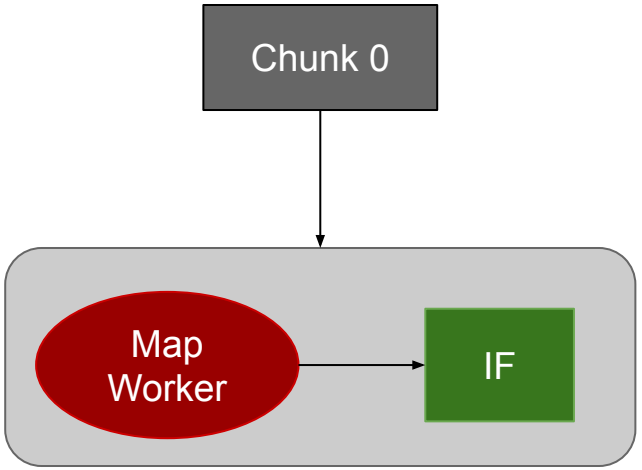
MapReduce | Paso 2

- ***Fork* de Procesos en Cluster**
  - 1 master: Actua de Scheduler y Coordinador
  - Muchos Workers (Mappers y Reducers)
  - Por lo general *chunks* de 64MB (recuerdan HDFS?)
- **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**

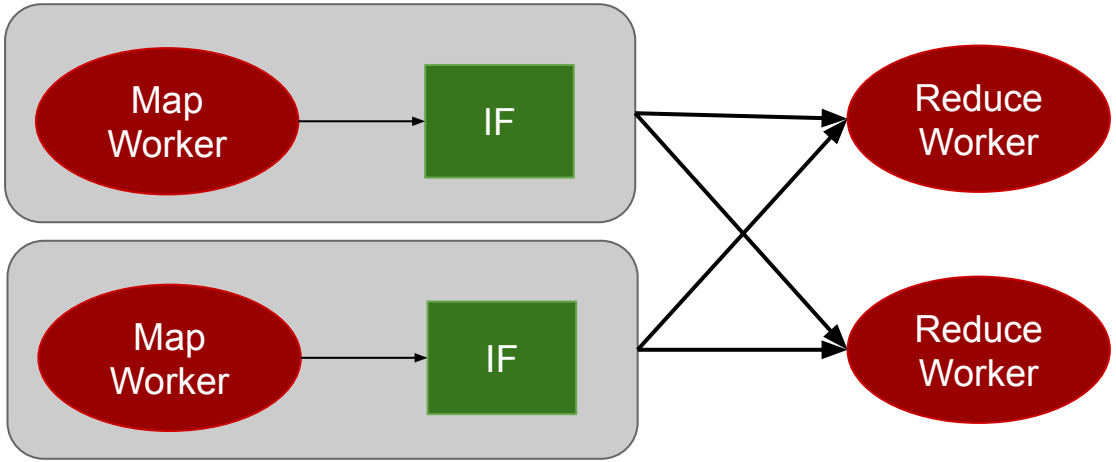


MapReduce | Paso 4a (Particionamiento)

- **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 cuales keys
- **Función de Partición**
  - Decide **cual** de los R workers va a trabajar con cada **Key**
  - Función default: **hash(key) mod R**
  - **Map workers** particionan la data por **Keys**
  - Cada Reduce Worker va leer la partición que desea de cada **Map Worker**

MapReduce | Paso 5

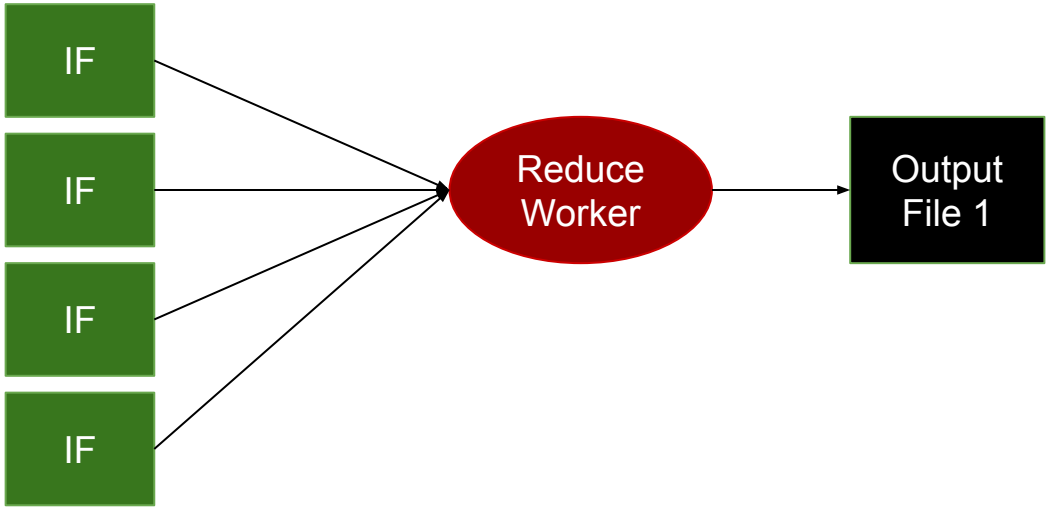
- **Reduce Workers** reciben ubicación de los archivos intermedios para la partición que deben procesar
  - Uso de RPC para leer data del disco local de los Mappers



- 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 recibe 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*



Agenda

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

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