



MapReduce y Hadoop Distributed File System

Fundamentos y conceptos previos



Índice de contenidos

- Introducción a MapReduce
- MapReduce vs esquema clásico
- Paralelización
- Fundamentos de MapReduce
- Distributed File System (DFS)
- Hadoop Distributed File System (HDFS)
- Hadoop
- Resumen





Introducción

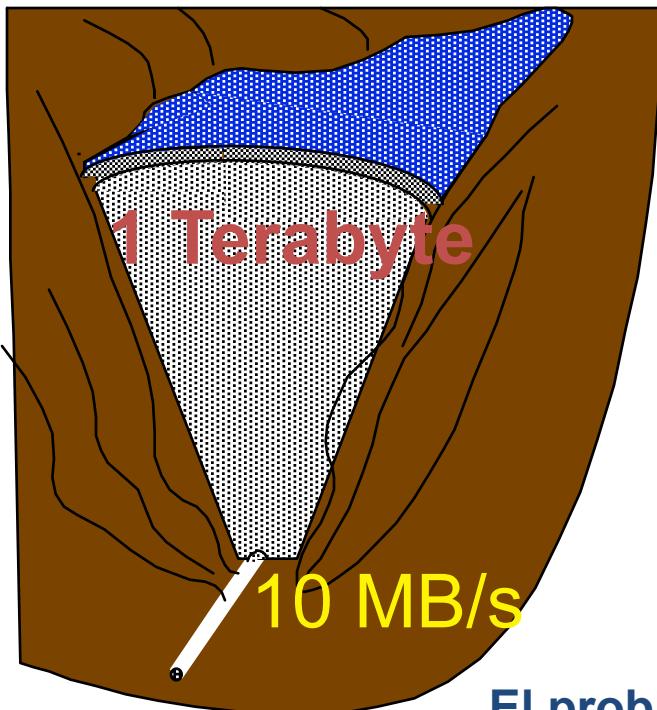
- ➊ Usuario → leer todo del mismo disco (lento)
- ➋ Mejor opción:
 - Lectura en paralelo: se lee de varios discos
 - Los datos se guardan de forma replicada para poder continuar ante un fallo ocasional de hardware
- ➌ MapReduce: permite separar la solución del problema de la representación física de los datos



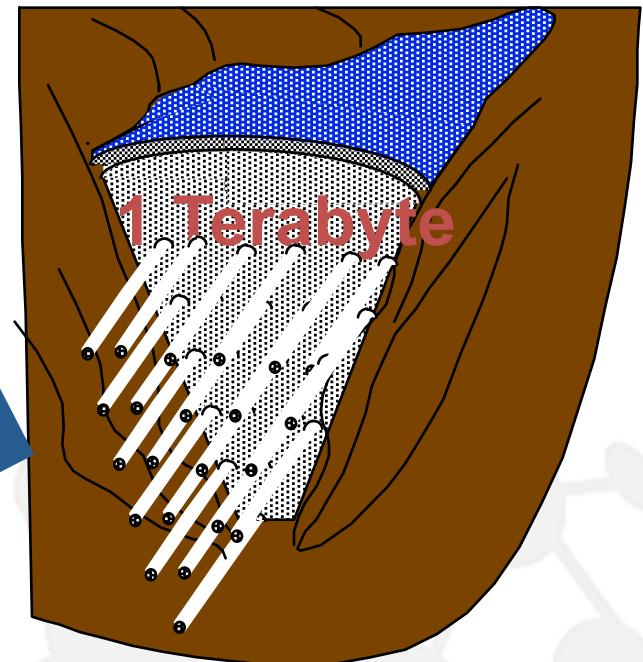


¿Por qué paralelizar?

a 10 MB/s
1.2 días para llevar la información a memoria



1,000 x paralelismo
1.5 minutos



El problema no es ya la capacidad de la memoria, sino el tiempo que tardamos en llevar la información desde el disco a su lugar de procesamiento





Fundamentos de MapReduce

- El rendimiento de las bases de datos tradicionales depende mucho del tiempo medio de búsqueda porque escriben y leen muchas veces del disco. Puesto que el patrón de acceso está dominado por las búsquedas, escribir o leer ficheros grandes puede no funcionar de forma muy eficiente
- MapReduce es un modelo “**write once read many**”. Trabaja mezclando y ordenando en memoria y de forma continua información que extrae del soporte. Por ello, el parámetro más importante en esta operación es la tasa de transferencia. Si se trabaja con ficheros de pequeño tamaño, no será lo más eficiente





Fundamentos de MapReduce

- Tiempo medio de búsqueda: tiempo que tarda un dispositivo en encontrar el primer dato que tiene que empezar a leer. Mejora de forma lenta, en torno a un **10%** al año
- Tasa de transferencia: velocidad a la que se puede transferir la información desde el disco a la memoria principal una vez que la información que interesa está localizada. Mejora de forma rápida, en torno a un **20%** al año



Esquema clásico vs MapReduce

- ➊ RDBMS: suelen usar una estructura de datos B-Tree para almacenar información. Por ello, tiene limitado la mejora en tiempos de búsqueda. Para actualizar pocos datos de forma continua funciona bien
- ➋ MapReduce es más potente cuando se quiere trabajar con grandes volúmenes de datos
 - Ficheros más habituales de 100 GBs a TBs
- ➌ Cuando más grande sea el conjunto de datos contra el que se lanza una consulta SQL, más tarda. Es difícil mejorar el tiempo de respuesta añadiendo máquinas al clúster. Por el contrario, un “job” MapReduce SÍ puede mejorar su velocidad de ejecución si se añaden más máquinas al clúster





Esquema clásico vs MapReduce

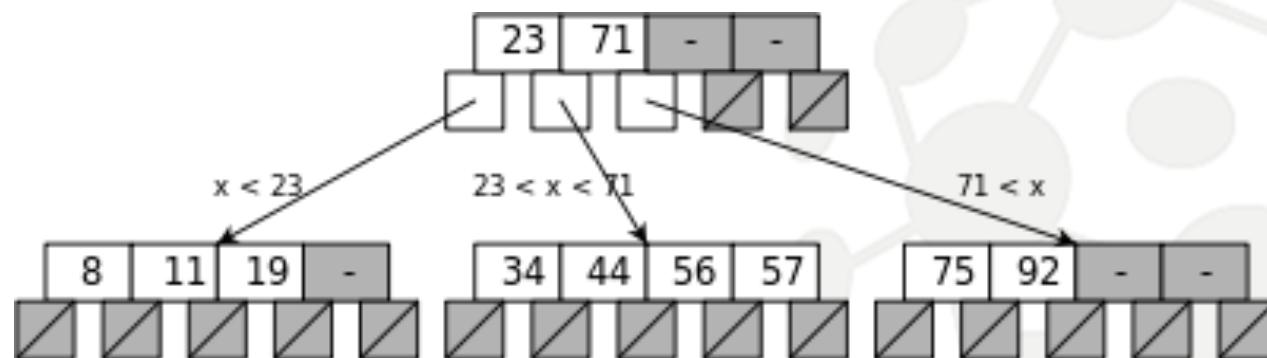
- ➊ Si la búsqueda es la actividad dominante, llevará tiempo leer o escribir grandes porciones de datos en una BBDD clásica
- ➋ Para actualizar grandes volúmenes de datos, B-Tree es menos eficiente que MapReduce, que usa sort / merge para reconstruir la BBDD
 - Una búsqueda para localizar cada objeto de actualización
 - En una RDBMS se actualiza pequeñas porciones de datos



B-Tree

Árboles balanceados de búsqueda

- Mantienen los datos ordenados
- Insertar y borrar muy rápidos $O(\log(n))$
- Exige re-balancear el árbol
- No eficiente si hay que actualizar grandes conjuntos de datos
- Se suelen organizar en función de las columnas de las tablas
 - ¿Qué pasa si hay más de 100 columnas?

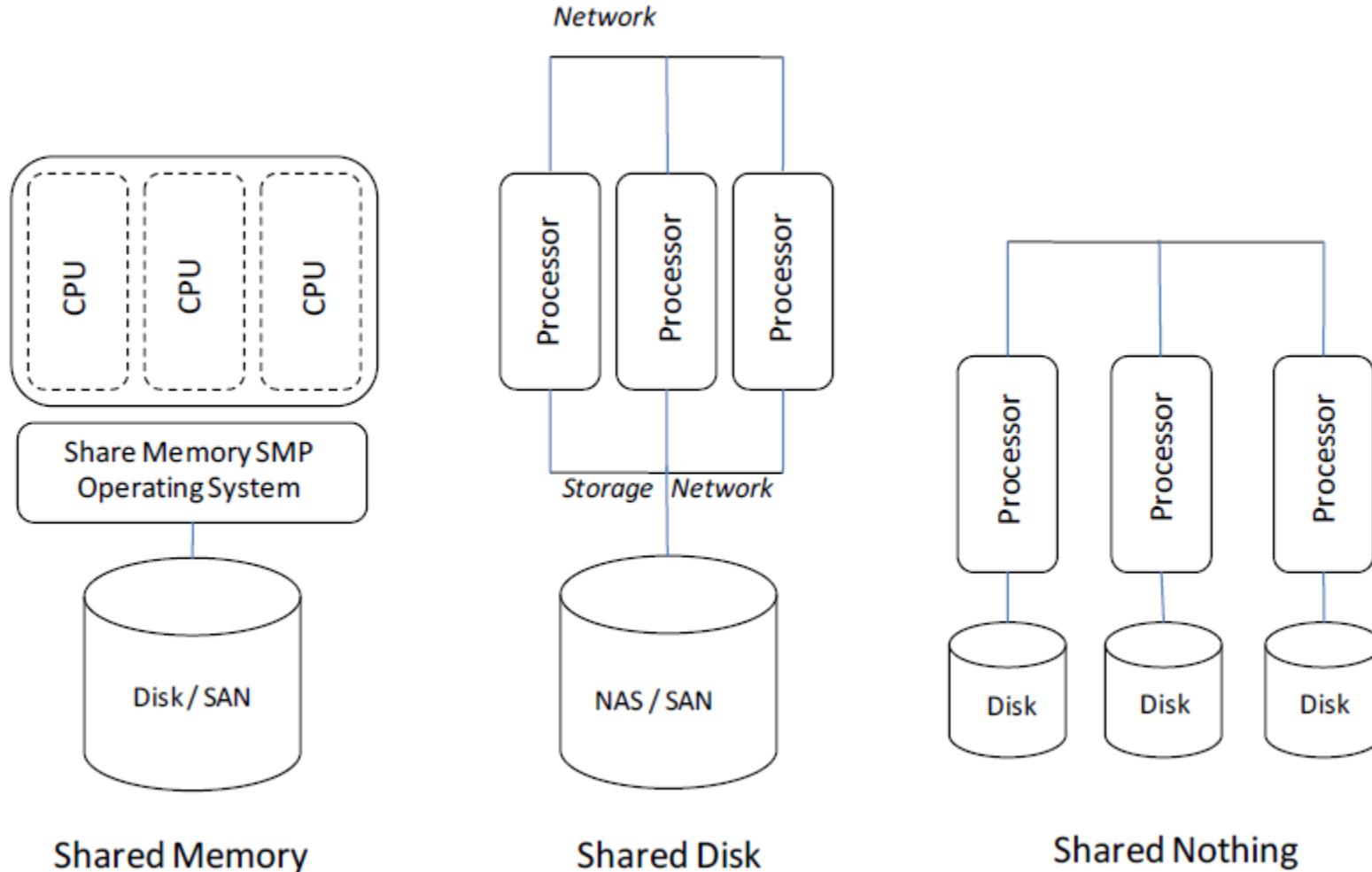


Fuente imagen: Wikipedia





Bases de datos paralelas





Diferencias

- ➊ BBDD NoSQL no admiten full joins
- ➋ Shared Memory: No muy escalable
- ➌ Shared nothing: más escalable pero hay que integrar y asegurar consistencia datos
- ➍ In-memory databases (para tiempo real con joins complejos). Por supuesto no podemos llegar a petabytes, sino “sólo” a cientos de terabytes.





RDBMS vs MapReduce

	RDBMS	MapReduce
Data Size	GBs	PBs
Access	Interactive & Batch	Batch
Updates	Read/Write many times	Write once / Read many times
Structure	Static schema	Dynamic Schema
Integrity	High*	Low
Scaling	NonLinear	Linear

* La normalización de la BBDD favorece la integridad de los datos (evitar datos no válidos). Por el contrario, MapReduce no tiene esos mecanismos





MapReduce (trucos)

- ➊ Datos REPLICADOS: permite continuar trabajando ante fallos en un nodo del clúster (asumimos que los fallos de los nodos se producen de forma independiente entre esos nodos, lo que implica que se distribuyen de forma particular en cada máquina)
- ➋ Data locality: se colocan los datos junto al nodo de cómputo, así me consigue un acceso local a los datos (más rápido que un acceso remoto)
- ➌ Los resultados se combinan salvando el cuello de botella que puede suponer el ancho de banda
- ➍ A diferencia de MPI, los mecanismos de transferencias de datos son invisibles al programador y además **gestiona los errores que puedan producirse**
- ➎ Shared-nothing architecture: tareas independientes y recuperables



Algunas dificultades sobre los esquemas de procesamiento paralelo

● En general y no necesariamente en MapReduce:

- A veces no logramos desprendernos de la influencia de los esquemas tradicionales en cuestiones como indexación, optimización del almacenamiento...
- La optimización de consultas puede ser compleja
- En ocasiones se exige re-procesamiento, re-elaboración, re-trabajo
- Para evitar estos inconvenientes es imprescindible un trabajo fino de diseño del código





Medidas (paralelización)

Speed-Up

Eficiencia

$$S_p = \frac{T_1}{T_p}$$

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

p número de procesadores

T_1 tiempo serial (1 procesador)

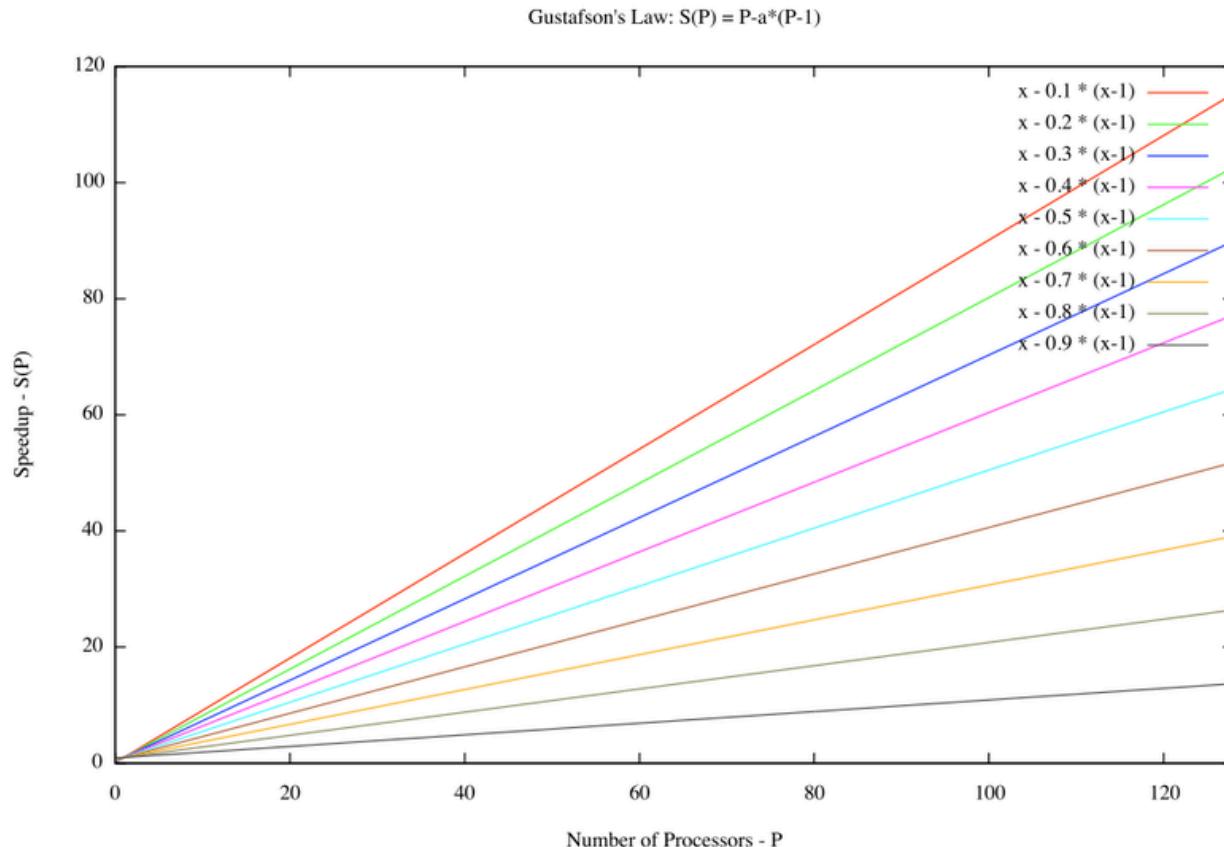
T_p tiempo empleado para la ejecución en p procesadores

Un cuello de botella a superar según se amplía el número de procesadores es el coste de la comunicaciones



Ley de Gustafson

- Cualquier problema suficientemente grande puede ser eficientemente paralelizado



Fuente imagen: Wikipedia

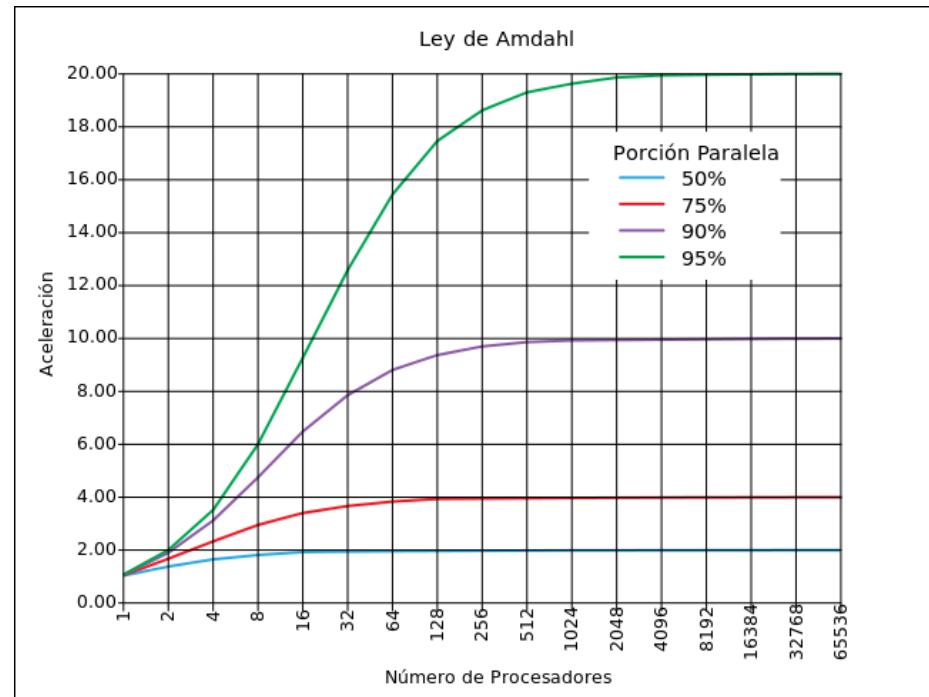


MapReduce y HDFS



Ley de Amdahl

- La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente



Fuente imagen: Wikipedia

- Dicho de otra forma: se pone límite a la mejora por Speedup que se puede tener debido a la paralelización, **para un conjunto de datos de tamaño fijo**, puesto que siempre habrá una parte que no se puede paralelizar que limita esta mejora





¿Estamos diciendo cosas distintas?



NO

- La Ley de Amdahl se basa en una carga de trabajo o tamaño de entrada prefijados. Esto implica que la parte secuencial de un programa no cambia con respecto al número de procesadores de la máquina, sin embargo, la parte paralelizable es uniformemente distribuida en el número de procesadores.
- Por eso es importante el comienzo en la Ley de Gustafson:
 - “... cualquier problema suficientemente grande..”





Ley de Moore

- Moore (1975): el número de transistores en un chip se duplica cada dos años
- Se sigue cumpliendo hoy
- Además de proyectar cómo aumenta la complejidad de los chips (medida por transistores contenidos en un chip de computador), la ley de Moore sugiere también una disminución de los costos
- Atendiendo a esta ley → hay potencial tecnológico suficiente para explotar los datos que tenemos a nuestra disposición
- Tecnologías como MapReduce y Hadoop facilitan el aprovechamiento de los recursos tecnológicos a nuestro alcance





MapReduce y lenguajes de programación

- ➊ Existen librerías para:
 - C++, Java, Ruby y Python
- ➋ ¡Ojo! con las distintas versiones
 - De 1.x a 2.x puede ser necesario modificar y re-compilar
 - De 1.0.x a 1.0.x debería respetar compatibilidad





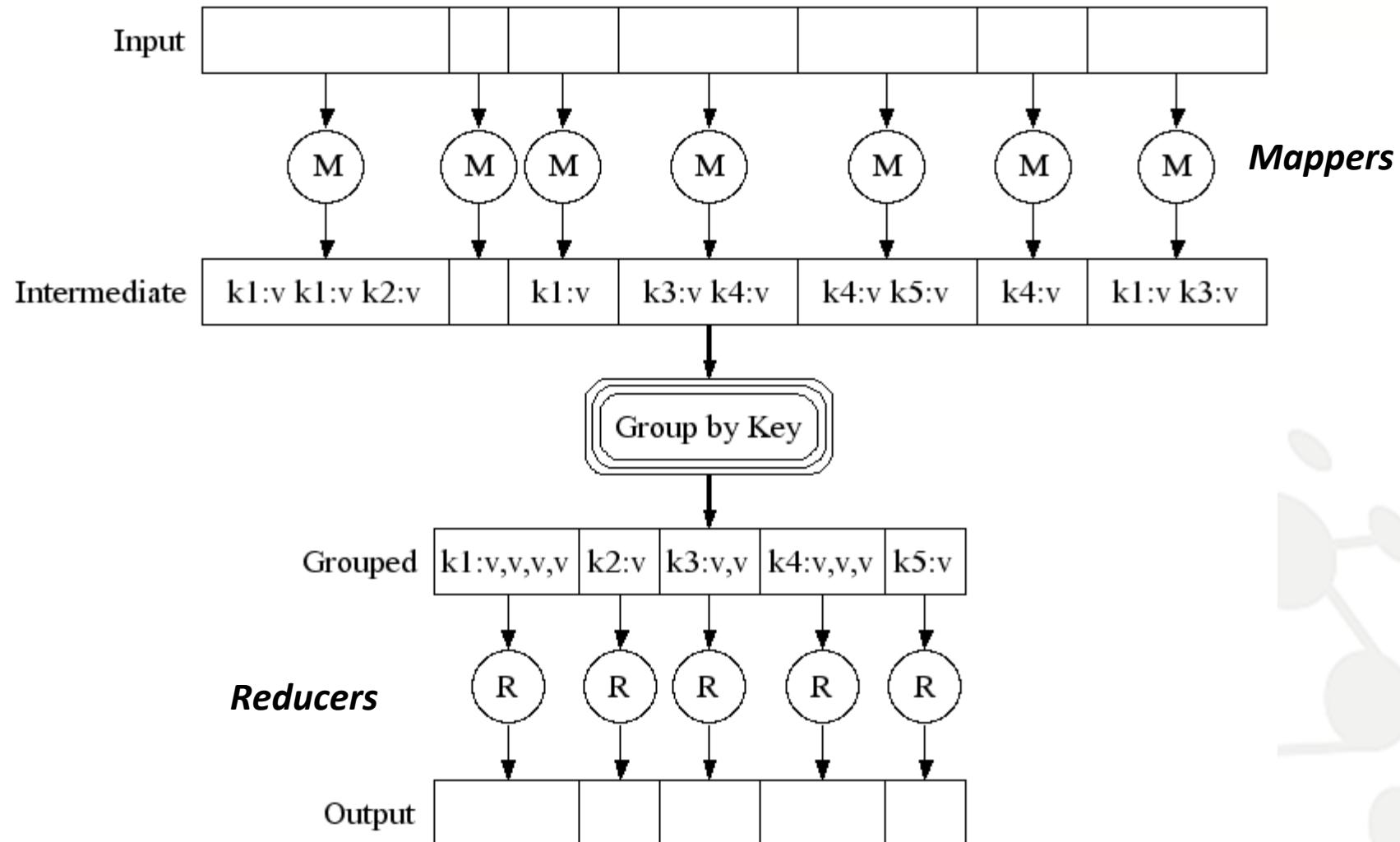
MapReduce con más detalle

- La entrada y la salida son conjuntos de pares clave/valor
- Es responsabilidad del programador diseñar estas dos funciones:
 - **map(K1, V1) -> list(K2, V2)**
 - Genera una lista de pares intermedios clave/valor para cada par de entrada clave/valor
 - **reduce(K2, list(V2)) -> list(K3, V3)**
 - Genera la lista de resultados para todos los valores intermedios asociados a la misma clave (tiene la misión de agrupar los resultados parciales)

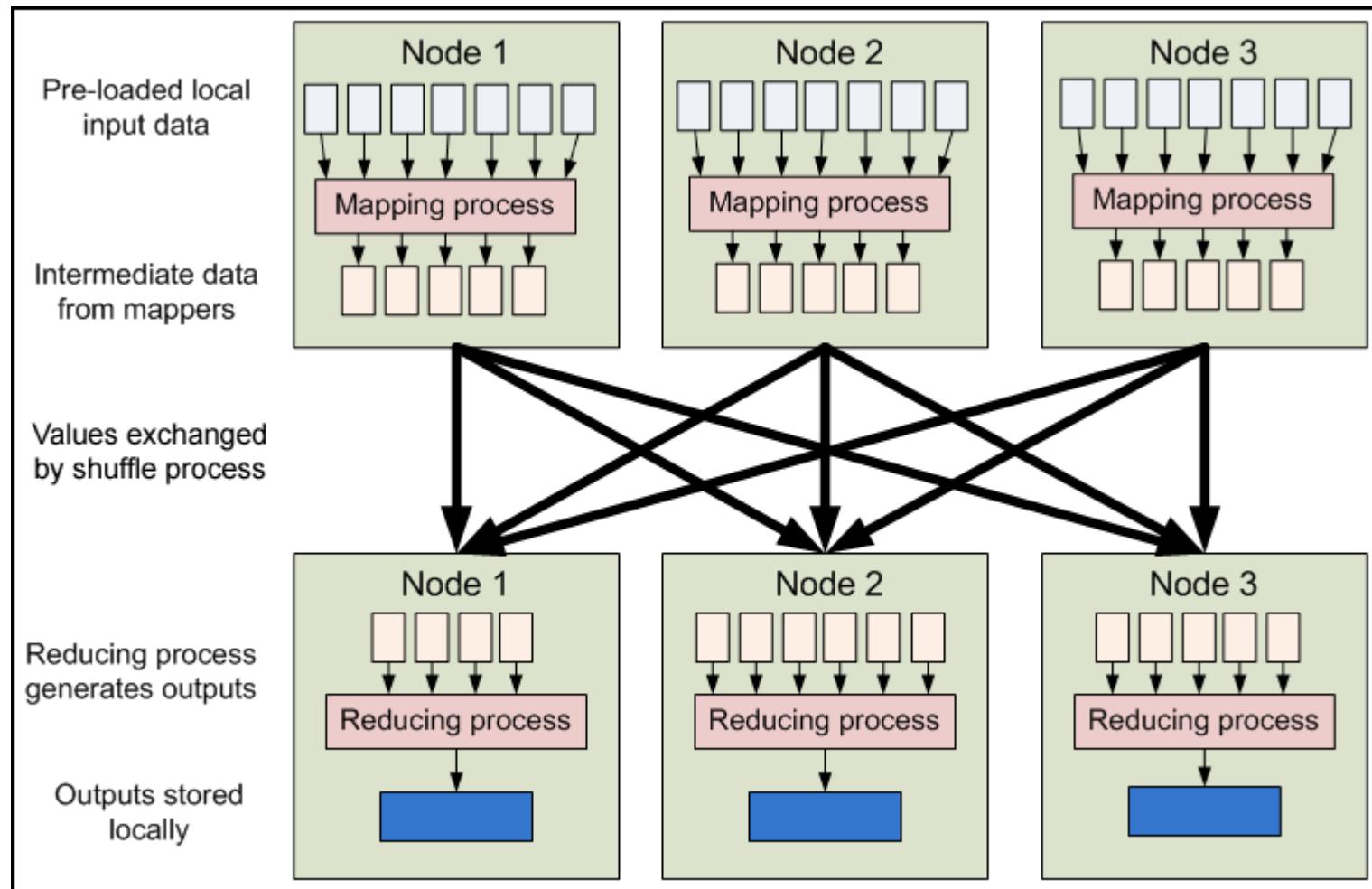




MapReduce: detalle

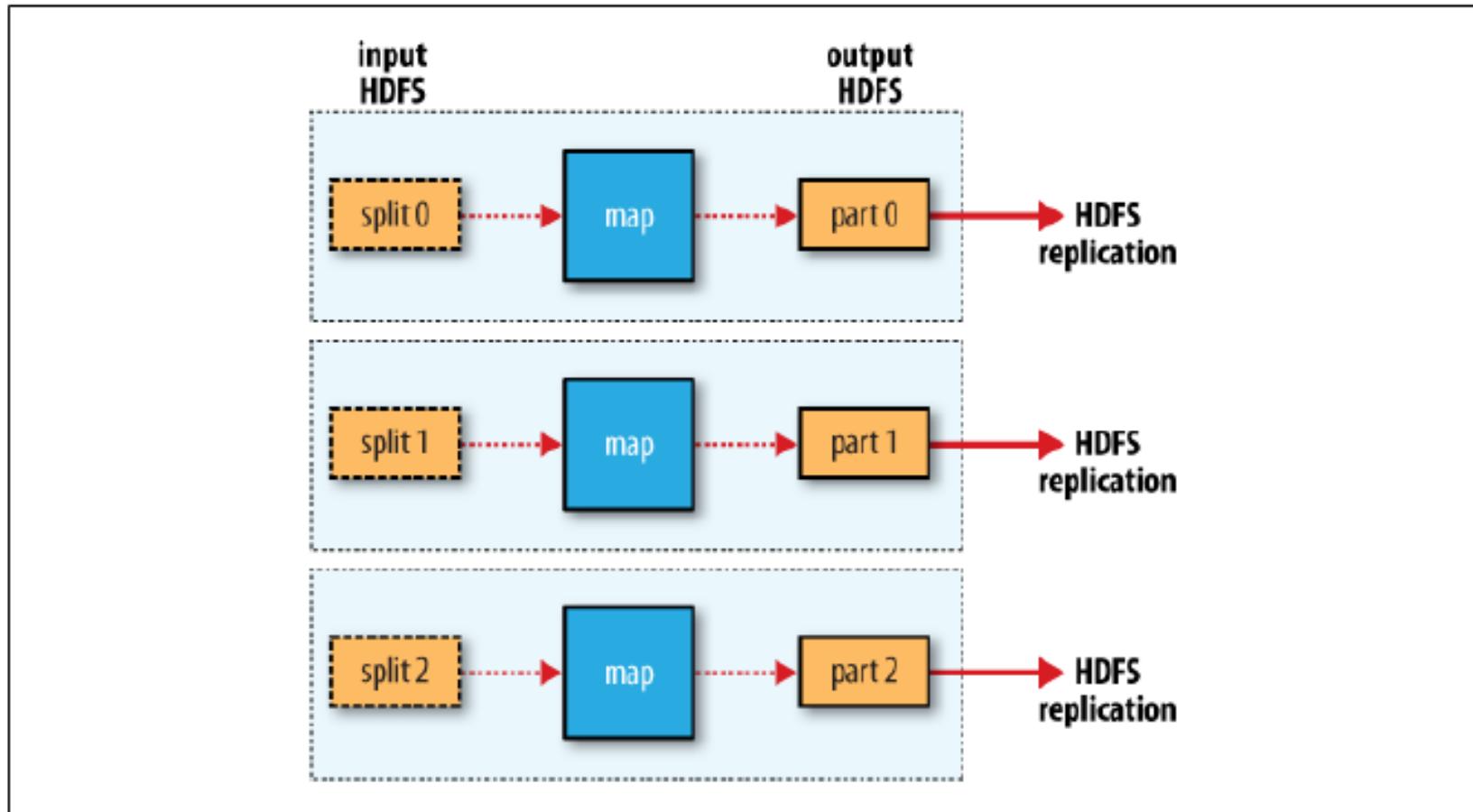


Más detalle (from Yahoo!)



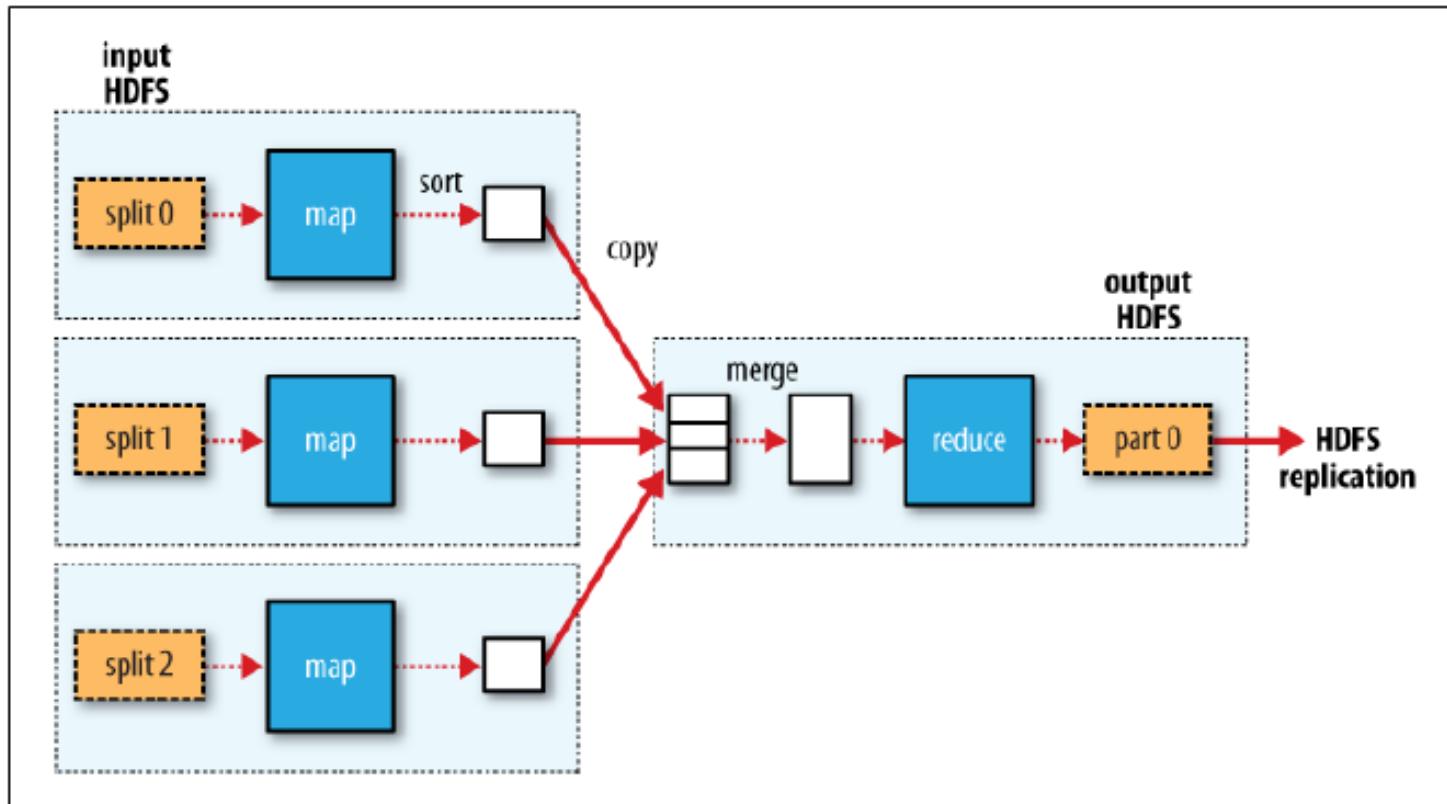


MapReduce con cero Reducers



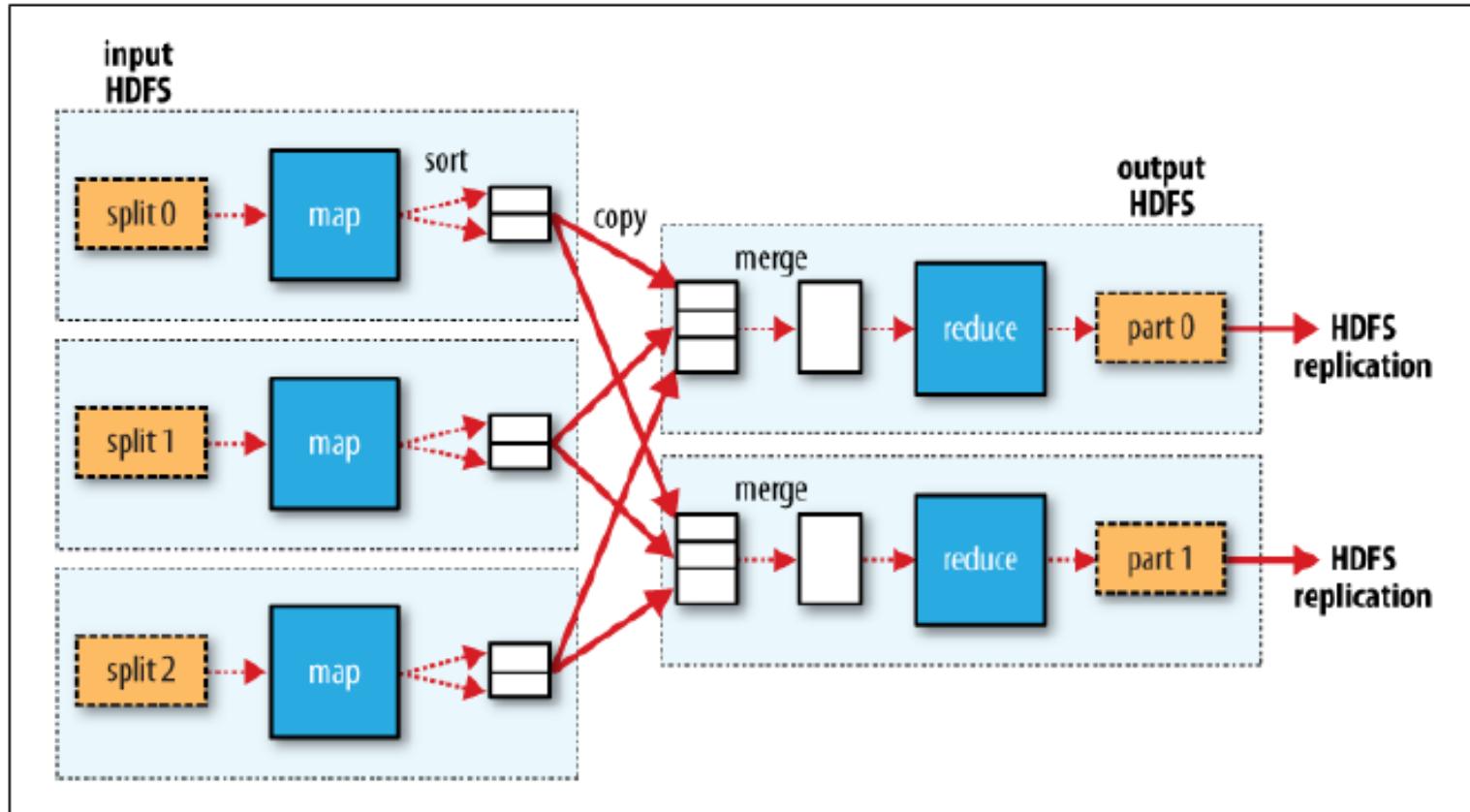


MapReduce con un Reducer





MapReduce con dos Reducers





Distributed File System (DFS)

- Ventajas en la tolerancia a fallos.
- Se descomponen en trozos los archivos grandes:
 - Google File System (GFS), donde encontramos los Chunks Serves y
 - Hadoop Distributed File System (HDFS), donde encontramos los Data Nodes
- GFS:
 - Los Chunks Serves están replicados (se debe mantener la consistencia entre réplicas). **Generalmente el ratio de réplica es 3**
 - Hay un master node (GFS) que organiza dónde va cada cosa. Si un acceso falla, busca la siguiente réplica para acceder. Leer es muy fácil, lo difícil es actualizar. Pero una réplica es señalada como la copia principal (réplica primaria), si cae la primaria se asigna otra réplica como primaria. Ahí es donde primero se escribe y después en las demás. Si la escritura falla en una réplica, la réplica primaria lo intenta de nuevo hasta que tiene éxito en todas.
- GFS y HDFS son muy parecidos.
 - Hay un Master Node que es una especie de índice. Cuando la extensión del índice es muy amplia se debe abrir otro nodo.





HDFS

En HDFS el tamaño por defecto del bloque es de 64 MB frente a 4 u 8 KB en un sistema tradicional

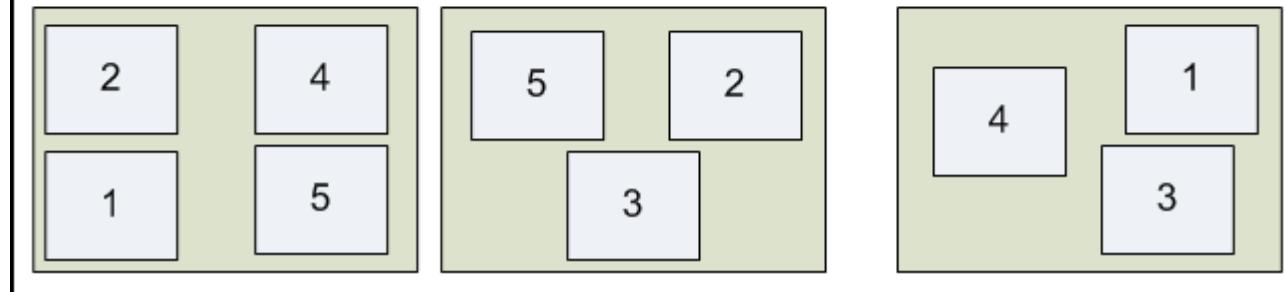
NameNode:
Stores metadata only

METADATA:
/user/aaron/foo → 1, 2, 4
/user/aaron/bar → 3, 5

En este ejemplo el factor de replicación es 2

Establece la correspondencia entre el fichero que se almacena y el ID de los bloques de datos que lo contienen

DataNodes: Store blocks from files



Fuente: <http://developer.yahoo.com/hadoop/tutorial/module2.html>





HDFS: características

- Tamaño de bloque de hasta 128 MB vs los tradicionales 4 u 8 KB
- Gracias a esta decisión, un fichero “cabe” en menos bloques (porque cada bloque es más grandes)
- Por ello, se precisan menos metadatos para indicar dónde se está guardando cada fichero
- También permite incrementar la velocidad de lectura, ya que se guarda más información junta y de forma secuencial
- Por ello, HDFS está preparado para almacenar ficheros muy grandes. Por el contrario, NTFS o EXT están diseñados para guardar miles de ficheros pequeños





HDFS: características

- En HDFS, un fichero de 100 MB se almacenaría en 2 bloques. Por ello, HDFS puede almacenar sin problemas ficheros de varios GB de extensión
- Otra cuestión importante es que HDFS espera que los bloques de datos se lean siempre desde el principio hasta el final, este enfoque beneficia a MapReduce
- Los ficheros almacenados en HDFS NO forman parte del sistema de ficheros local de las máquinas. HDFS trabaja en un entorno (namespace) separado de los ficheros del SO local
- Los bloques HDFS se almacenan en un directorio particular gestionado por el DataNode Service





HDFS: características

- ➊ No se puede interactuar con los ficheros HDFS a través de herramientas típicas de Linux como ls, cp, mv...
- ➋ Por ello, HDFS posee sus propios comandos para interactuar con los ficheros almacenados.
- ➌ HDFS se basa en el modelo de “***write once read many***”
- ➍ Para abrir un fichero, se conecta con el NameNode y se pide la localización de los bloques donde se almacena el fichero. También se identifica al DataNode que tiene cada bloque
- ➎ Se puede leer los datos procedentes de los servidores DataNode, además en paralelo





HDFS: control de errores

- ➊ Todo el sistema está preparado ante posibles fallos, se emplea redundancia (la información está replicada) para recuperar la información precisa ante caídas del sistema reestableciendo el servicio
- ➋ Los metadatos almacenados en el NameNode deben ser restaurados manualmente, pero es raro que este fallo se produzca
- ➌ No obstante, se deben realizar backups periódicos de los metadatos del NameNode para reaccionar correctamente ante posibles fallos





¿Por qué YARN?

- La versión clásica de MapReduce presenta cuellos de botella serios en la escalabilidad de un cluster grande
- Un punto crítico de la versión anterior de Hadoop era el JobTracker encargado de gestionar los trabajos que se lanzaban sobre el clúster a través de las aplicaciones externas
- Si el JobTracker fallaba o se bloqueaba (lo que no era muy frecuente) y había que reiniciarlo se podía perder el estado de los trabajos en ejecución

Solución

- Yet Another Resource Negotiator (YARN)

Esencialmente:

- Divide las responsabilidades entre los recursos disponibles
- ***Optimiza el uso del clúster y potencia la escalabilidad***
- Compatible con la ejecución de procesos MRv1





Separación de poderes

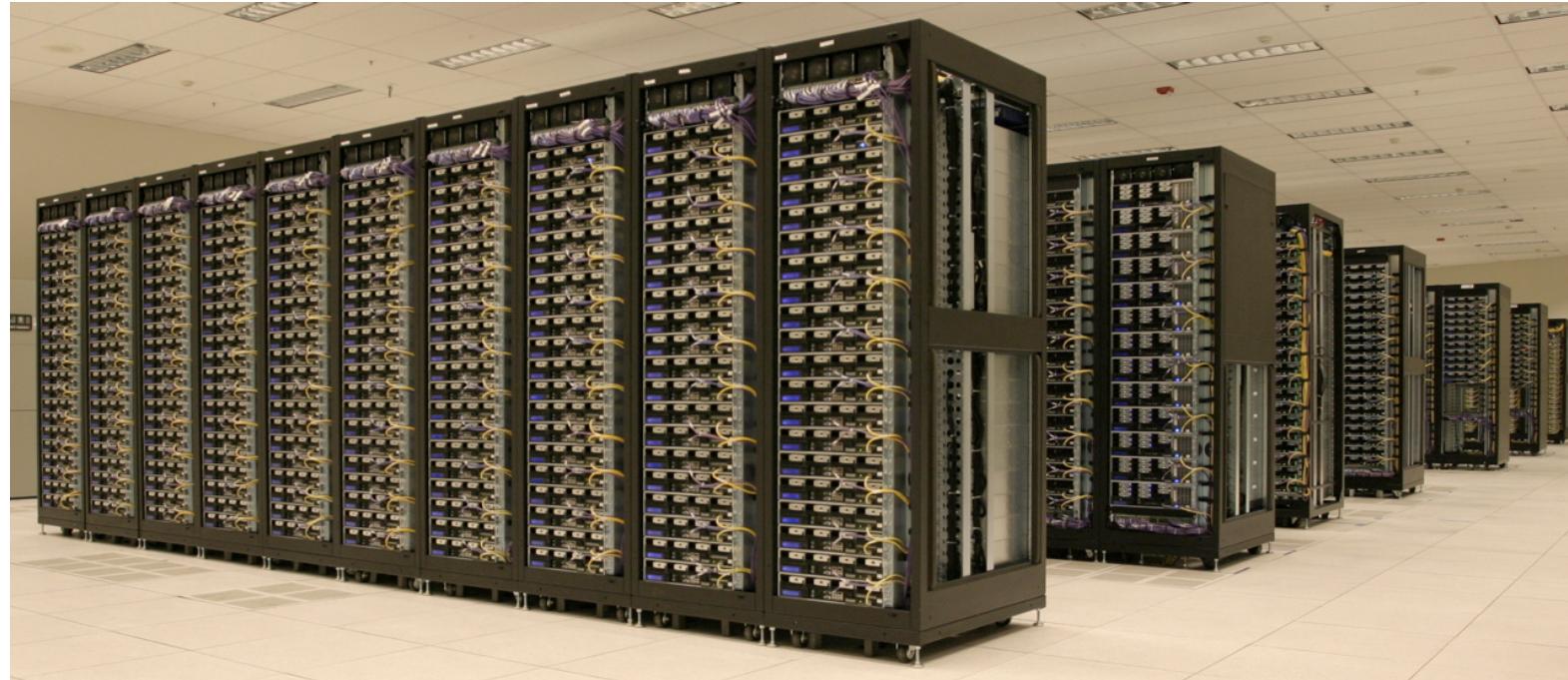
- YARN separa las dos funcionalidades del JobTracker
 - Gestión de recursos o Resource Manager (RM)
 - Monotorización y planificación de Jobs o Application Master (AM)
- El RM junto a un NodeManager (NM) por esclavo, distribuye los recursos entre todas las aplicaciones del sistema
- El AM es un framework que negocia los recursos con el RM y trabaja junto a los NMs para ejecutar y monotorizar las tareas



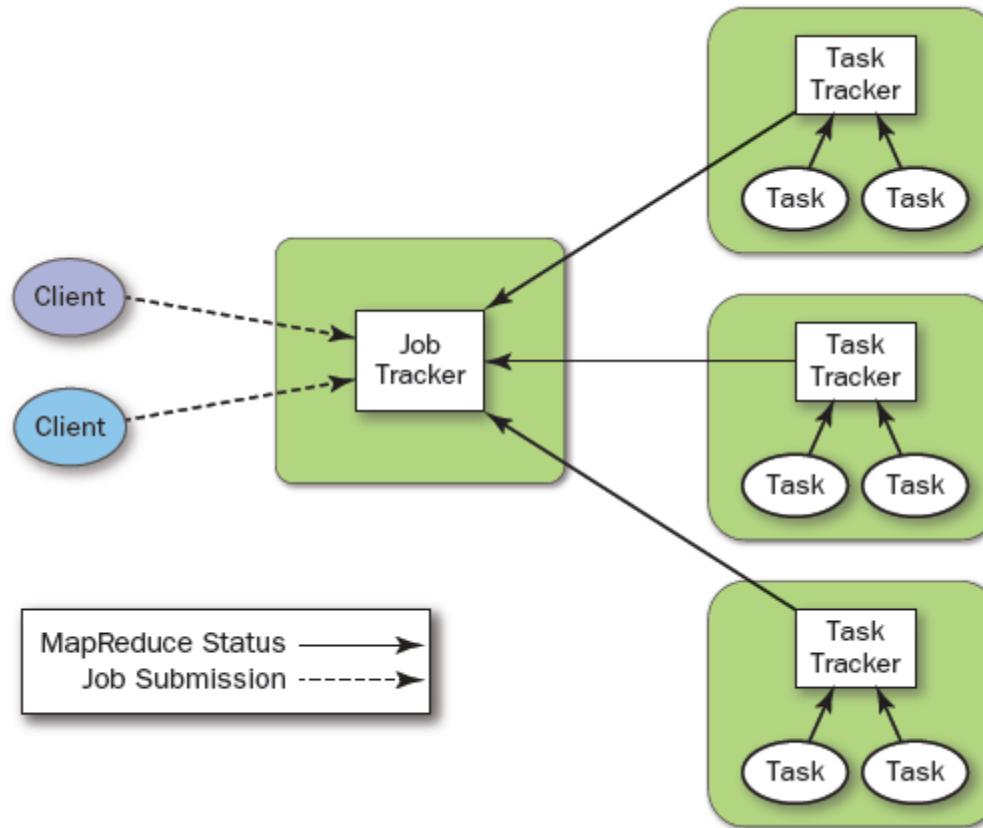


Jusfiticación YARN: Hadoop clusters & YAHOO!

- 10.000 máquinas para 10.000 procesos semanales
- El JobTracker presentaba un serio problema de cuello de botella a partir de los 4.000 nodos

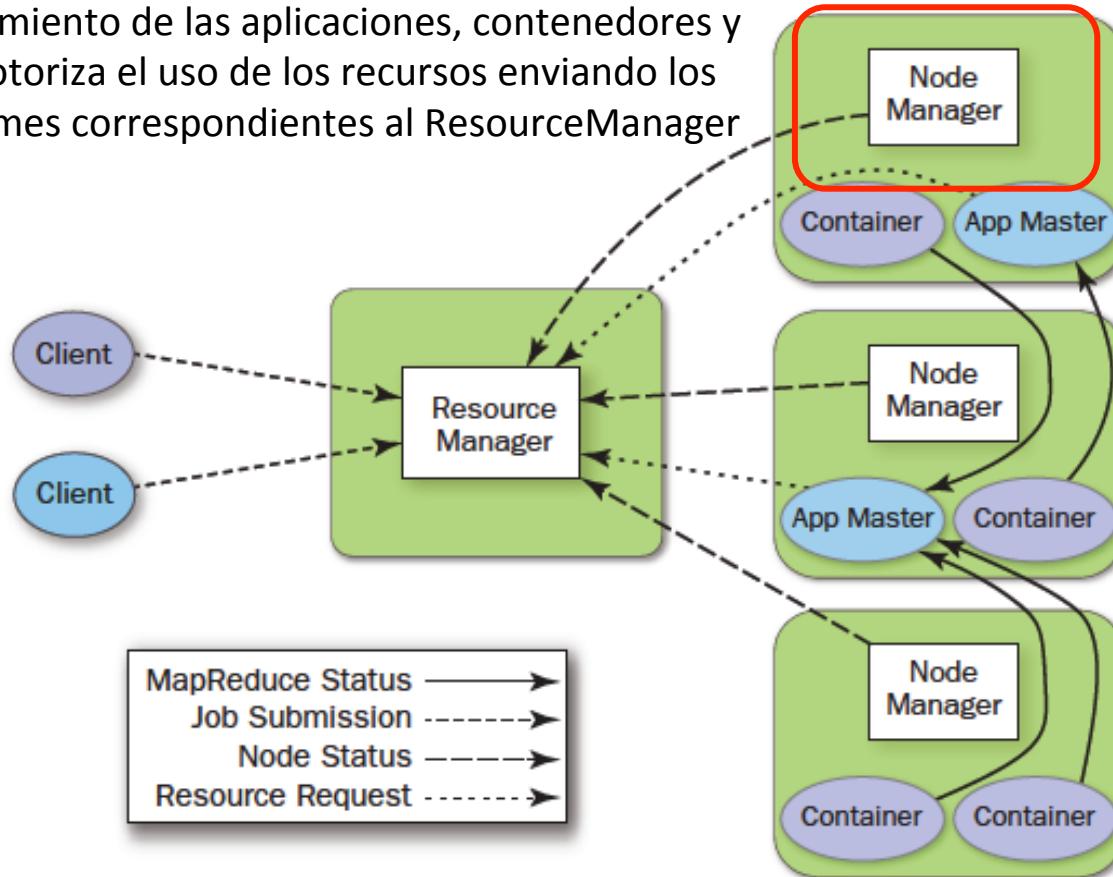


Entender el problema



Y la solución

Control local. El NodeManager gestiona el lanzamiento de las aplicaciones, contenedores y monotoriza el uso de los recursos enviando los informes correspondientes al ResourceManager

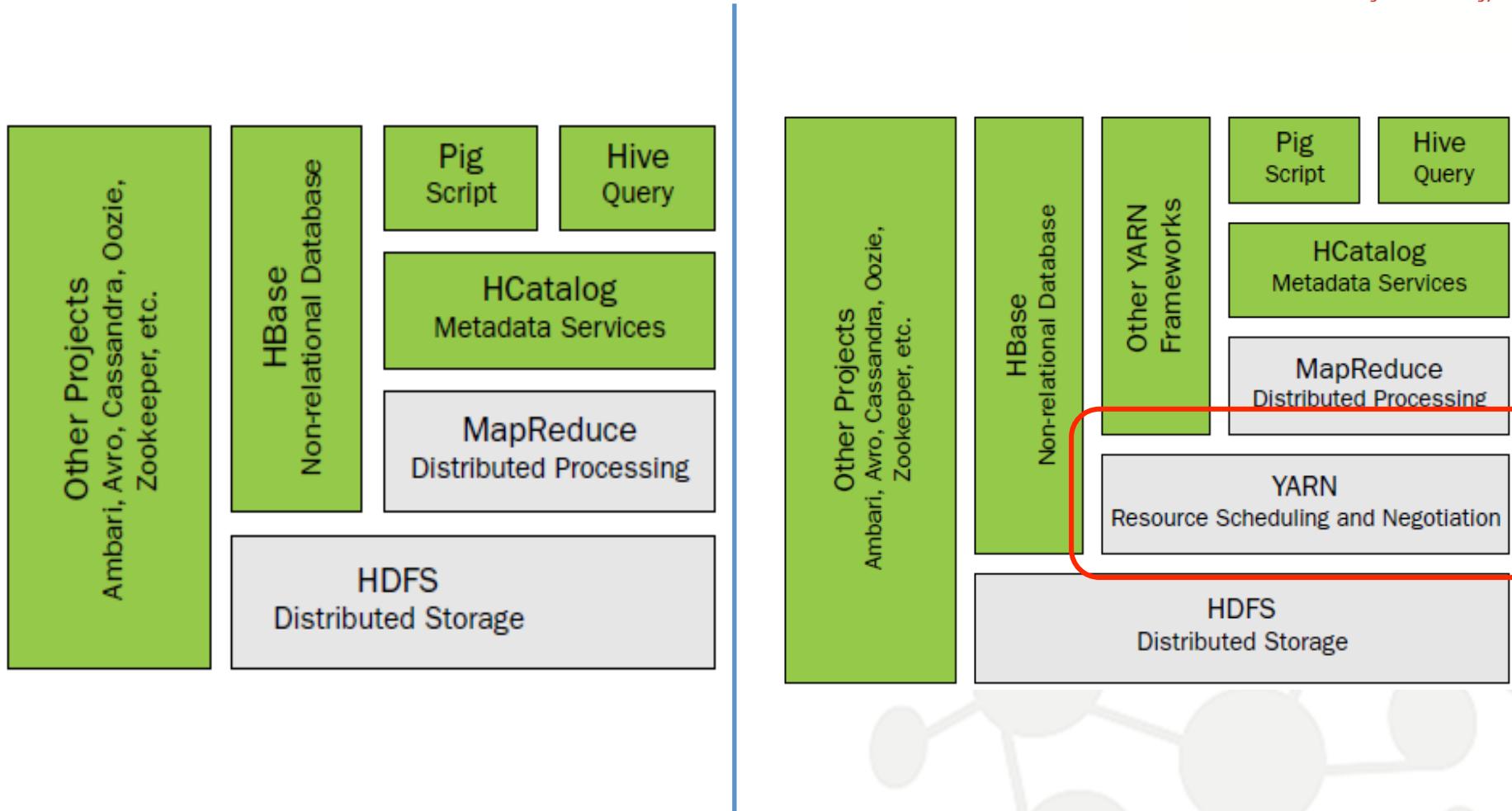


Se puede ejecutar el gestor de recursos en una máquina dedicada aparte





MRv1 vs YARN





Ejemplo típico: WordCount

- Input: Conjunto de (Document name, Document Contents)
- Output: Conjunto de(Word, Count(Word))
- map(k1, v1):

```
for each word w in v1
    emit(w, 1) //emite pares a los Reducers
```
- La función reduce procesa valores agrupados por una misma llave y *emite* otro cierto número de pares (clave, valor) de salida
- reduce(k2, v2_list):

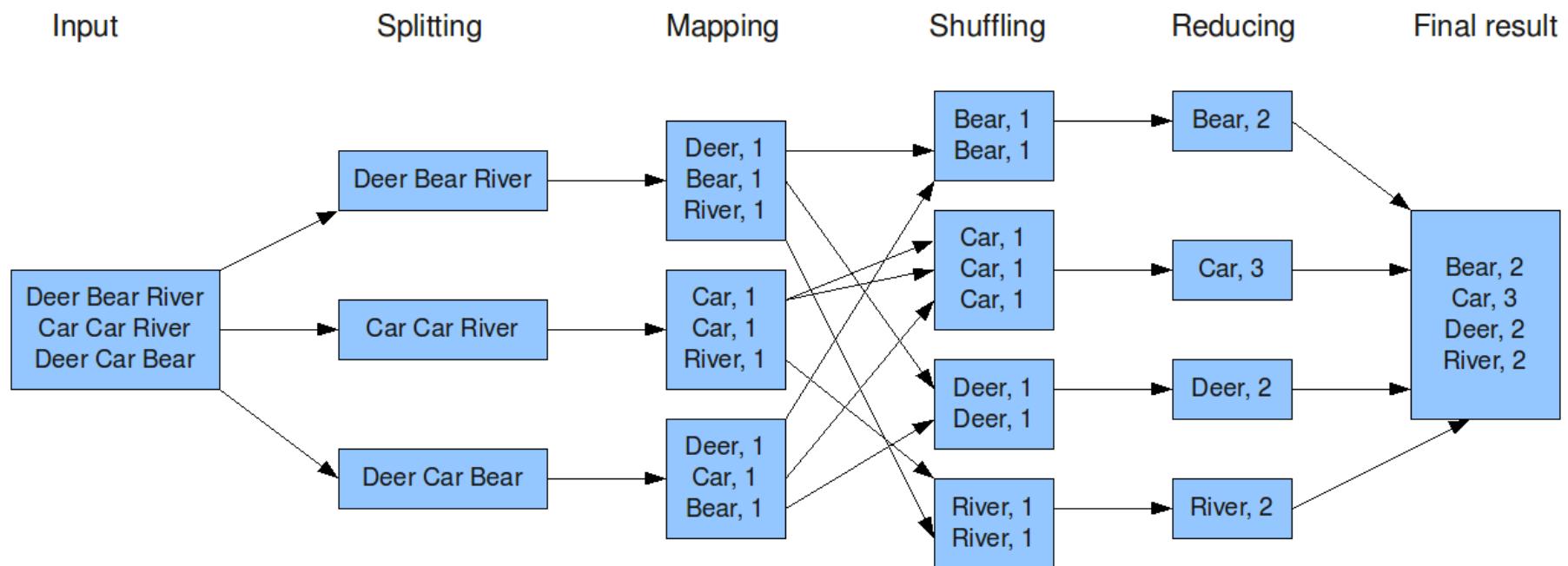
```
int result = 0;
for each v in v2_list
    result += v;
emit(k2, result) //emite resultados finales
```





MapReduce example

The overall MapReduce word count process





Optimización: Combiners

- A veces es posible realizar una agregación parcial en el lado del mapeo
- Esto permite reducir de forma significativa el volumen de los datos que se pasa al lado reduce
- **combine(K2, list(V2)) -> K2, list(V2)**
- En el caso del WordCount
 - **Combine == Reduce**





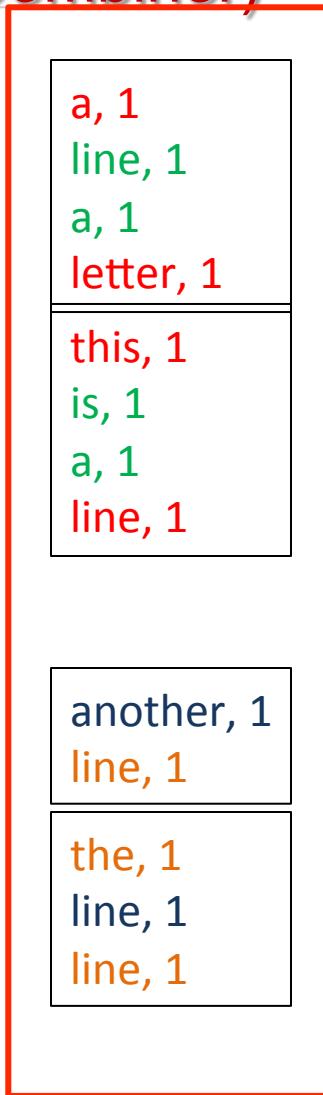
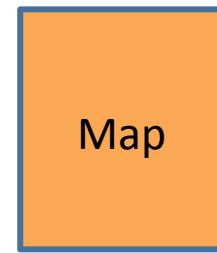
WordCount (Con Combiner)

a line a letter

this is a line

another line

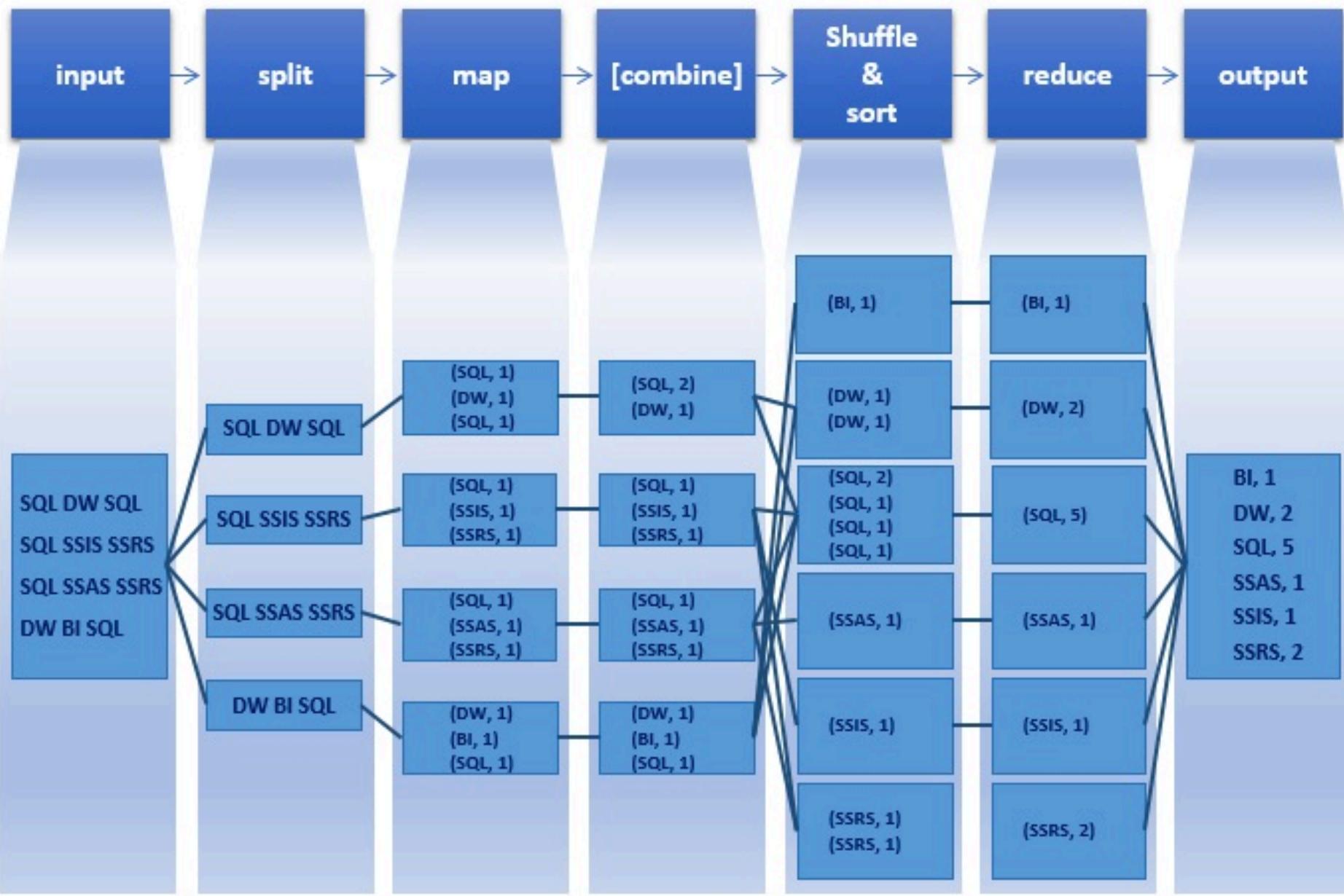
the line line



GROUP	COMBINE
a, (1,1) line, (1) letter, (1)	a, 2 line, 1 letter, 1
this, (1) is, (1) a, (1) line, (1)	this, 1 is, 1 a, 1 line, 1
another, (1) line, (1)	another, 1 line, 1
the, (1) line, (1,1)	the, 1 line, 2



MapReduce – Word Count Example Flow





Optimización: ejecuciones redundantes

- ➊ Los esclavos más lentos lastran el tiempo de ejecución del conjunto
- ➋ Esto puede pasar porque:
 - Hay otros procesos consumiendo recursos
 - Discos o red ineficiente, lentos...
- ➌ Solución: ***speculative execution***
 - Generar copias (redundancia) de las tareas más laboriosas (tiempo de ejecución más largo)
 - La copia que finaliza primero GANA.
 - Se aborta la ejecución del resto





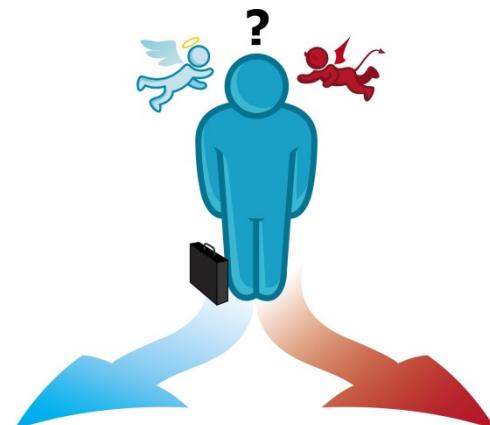
Número de Mappers y Reducers

- El número de mappers suele estar asociado al número de bloques DFS que ocupa el fichero a procesar. Por tanto, modificando el parámetro asociado al tamaño del bloque se puede obtener un número de mappers diferente
- Parámetros interesantes:
 - mapred.map.tasks: el máximo número de tareas mapper por job
 - mapred.child.java.opts: controla la cantidad de memoria reservada para cada tarea
 - mapred.reduce.tasks: número por defecto de tareas reducer por job. The preferred setting is 90%. This number is typically set to 99% of the cluster's reduce capacity so that if a node fails the reduces can still run in a single wave.
 - mapreduce.map.memory.mb y mapreduce.reduce.memory.mb: memoria (MBs) reservada para tareas map y reduce respectivamente, por defecto 1024



Otras consideraciones

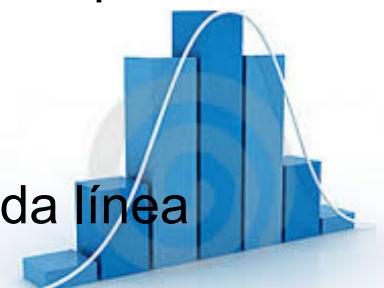
- ➊ Cada reducer genera un fichero/resultado propio
- ➋ Se recomienda generar el mínimo número de ficheros posibles
- ➌ Pero si sólo se configura 1 reducer = 1 resultado ¡no hay paralelismo en esta fase!
- ➍ Aprovechar las capacidades de parallelización suele implicar más velocidad
- ➎ Por tanto:





La complejidad de las soluciones mapReduce

- ➊ Un ejercicio típico para comprender el razonamiento mapReduce es “el histograma”
- ➋ En el histograma se representa la frecuencia absoluta para cada marca de clase
- ➌ Característica de un histograma de barras:
 - Representa la frecuencia (absoluta o relativa) mediante la altura de la barra la cual es proporcional a la frecuencia simple de la categoría que representa
- ➍ Entrada de datos
 - Un fichero de texto contenido un número en cada línea





Pasos para construir un histograma

1. Determinar el rango de los datos: máximo –mínimo
2. Determinar “n”, el número de marcas de clase deseado, cada número de la entrada deberá asignarse a una marca de clase
3. Establecer la longitud de la clase, el rango dividido entre el número de clases :
 - $\text{longitud} = (\text{máximo} - \text{mínimo})/n$
4. Construir los intervalos de las clases
 - $[\text{Mínimo}, \text{Mínimo} + \text{longitud}), [\text{Mínimo} + \text{longitud}, \text{Mínimo} + 2\text{longitud}) \dots [\text{Mínimo} + (n-1)\text{longitud}, \text{max}]$
5. Asignar cada número al intervalo que le corresponda



Ejercicio: ¿cómo se puede hacer? ¿cuántos procesos mapReduce?





Ecosistema Hadoop

- **Paquete base:**
 - HDFS
 - MapReduce
 - Hadoop Streaming: utilidad para permitir a MapReduce codificar en cualquier lenguaje: C, Perl, Python, C++, Bash...
- **Hive:** Data warehouse para facilitar el uso de Hadoop. Permite emplear código similar a SQL. Hive convierte el trabajo a MapReduce (no es un entorno ANSI-SQL al completo)
 - Hue: interfaz gráfica basada en navegador para trabajar en Hive
- **Pig:** entorno de programación de nivel alto para realizar codificación MapReduce. El lenguaje Pig es llamado Pig Latin
- **Sqoop:** proporciona transferencia de datos bidireccional entre Hadoop y una Base de Datos Relacional





Ecosistema Hadoop

- **Oozie**: gestor de flujos de trabajo para Hadoop
- **HBase**: BBDD NoSQL muy escalable
- **Flume**: recolector de datos en tiempo real para Hadoop y HBase
- **Whirr**: servicio en nube para Hadoop
- **Mahout**: software usado para análisis de datos
- **Fuse**: hace que el sistema HDFS parezca como un sistema de archivos normal para poder usar ls, rm, cd, y otros comandos en HDFS
- **Zookeeper**: software para realizar trabajos de sincronización en un cluster





Resumen

- La paralelización es esencial para resolver problemas complejos en tiempo y forma adecuado
- La aplicación del principio Divide y Vencerás (DyV) hace posible esta paralelización
- MapReduce es un modelo de programación que permite la resolución de problemas en paralelo aplicando el principio DyV
- Partes esenciales de MapReduce son el Mapper y el Reduce
- Existen diversas técnicas para optimizar el proceso de ejecución: combiners, lectura de datos local...
- Dos versiones disponibles: MRv1 y YARN o MRv2
- Hadoop presenta un amplio ecosistema proporcionando muchas y variadas utilidades

