

# Tema 1 – Big Data y MapReduce – TCDM

T. Fernández, F. García, D. Sevilla

Máster en Tecnologías de Análisis de Datos Masivos: Big Data  
Universidad de Murcia

2022

- 1 Definición y conceptos sobre Big Data
- 2 Modelo de procesamiento MapReduce
- 3 Ejemplos de implementaciones

# 1. Definición y conceptos sobre Big Data

# ¿Qué es Big Data?

## Wikipedia

Big data is a term used to refer to the study and applications of data sets that are so big and complex that traditional data-processing application software are inadequate to deal with them. Big data challenges include capturing data, data storage, data analysis, search, sharing, transfer, visualization, querying, updating, information privacy and data source.

## Las 7 Vs del BigData

**Volume** enorme cantidad de datos generados

**Velocity** rapidez con la que se generan y mueven esos datos

**Variety** datos de múltiples tipos, estructurados y no-estructurados: texto, datos de sensores, audio, vídeo, click streams, ficheros de log, etc.

**Veracity** datos correctos o incorrectos

**Value** capacidad de extraer valor de los datos

**Variability** el significado de los datos puede variar con el tiempo

**Visualization** los datos deben poder ser comprendidos

# Aluvión de datos

El mundo está creando más bits que nunca: en 2018, 33 Zettabytes ( $33 \times 10^{21}$ ), y se estiman más de 2 Yottabytes ( $2 \times 10^{24}$ ) en 2035

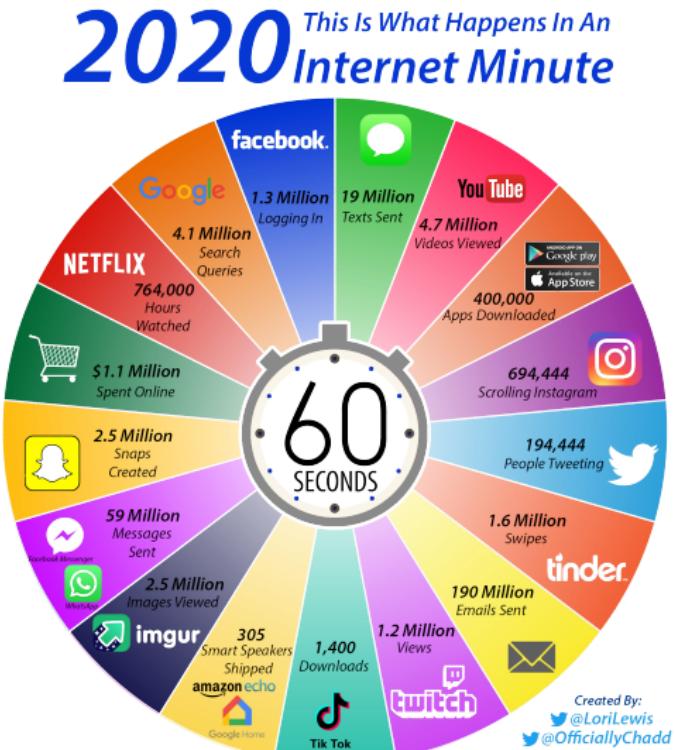


- En 2020, cada persona generará 1,7 MB cada segundo

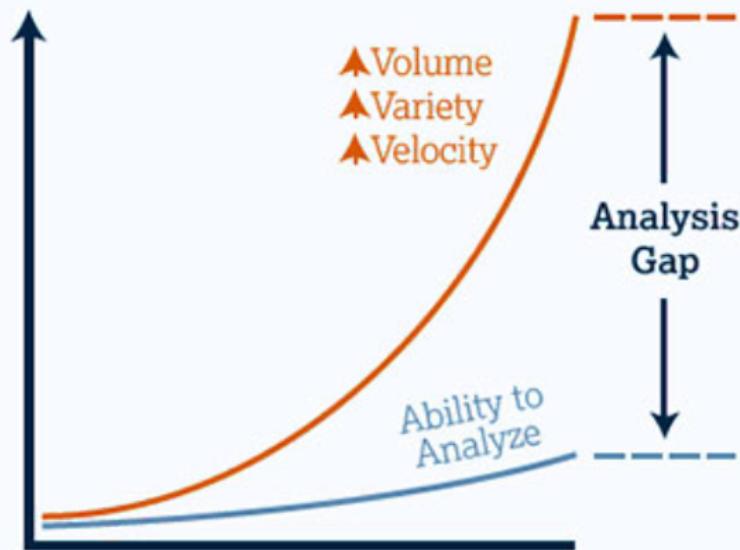
# Fuentes de datos

Origen de los datos:

- Redes sociales
- Redes de sensores
- Datos de GPS
- Datos de genoma
- Datos científicos
- Datos astronómicos (P.e. SKA generarán, en 2025, 750 TB/s  
~25 ZB por año)
- ...



## Information Explosion



# Retos tecnológicos (I)

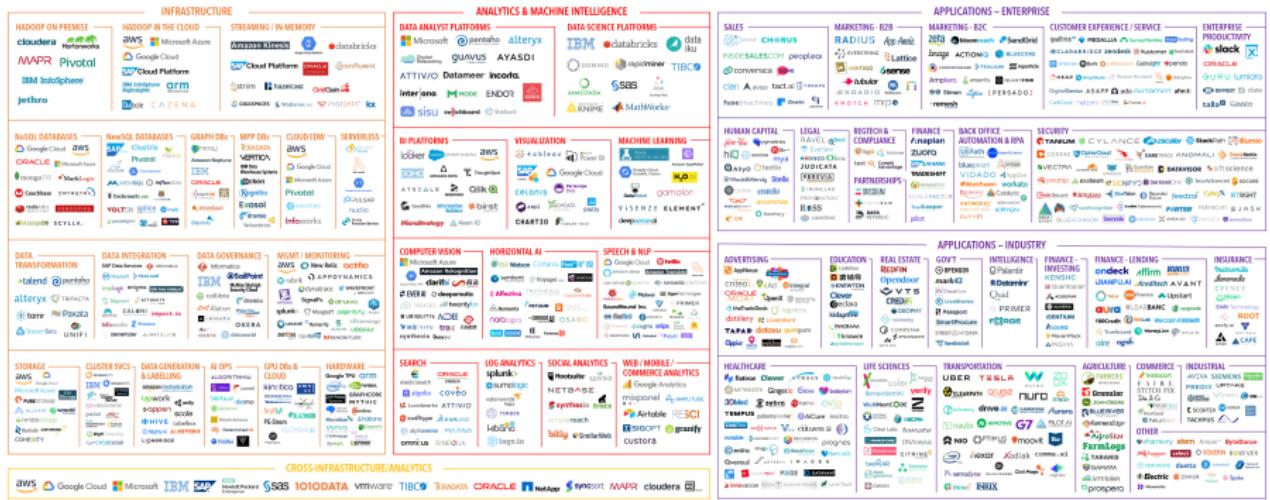
- Búsqueda y refinamiento (Lucene, Solr, Nutch, Elasticsearch, OpenRefine)
- Serialización (JSON, BSON, Apache Thrift, Apache Avro, Google Protocol Buffers)
- Sistemas de almacenamiento (HDFS, GFS, Lustre, Amazon S3)
- Servidores (Amazon EC2, Google Cloud Platform, Azure, OpenShift, Heroku, Tanzu, Ambari)
- Procesamiento (Hadoop, Hive, Pig, Spark, Flink, Beam, Disco, Dask, Tez, Cascading, Azkaban, Oozie, mrjob, Flume, Storm, Kinesis, Samza)

# Retos tecnológicos (II)

- Bases de datos (HBase, Cassandra, Accumulo, MongoDB, CouchDB, Riak, Amazon DynamoDB, Google BigTable, Drill, Kylin, Parquet, Sqoop)
- Análisis y BI (R, Greenplum, Splunk, Datameer, Kylin, Tableau, Jethro, PowerBI)
- Lenguaje natural (Natural Language Toolkit, Apache OpenNLP, Perldoop, Intelligent Tagging)
- Machine learning y deep learning (Weka, Apache Mahout, SciKit-learn, Pytorch, TensorFlow, caffe, Keras, Microsoft Cognitive Toolkit, BigDL, MXNet, Edward)
- Visualización (R, D3.js, Google Data Studio)

# BigData Landscape

## DATA & AI LANDSCAPE 2019



July 16, 2019 - FINAL 2019 VERSION

© Matt Turck (@mattturck), Lisa Xu (@lisaxu92), & FirstMark (@firstmarkcap) mattturck.com/data2019

FIRSTMARK  
EARLY STAGE VENTURE CAPITAL

# Programando para el Big Data: historia

- Circa 2002
  - Se busca mitigar el riesgo de pérdidas de grandes workloads distribuidas debidas a fallos de los discos en *commodity hardware*
- Google publica 2 papers:
  - Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google File System", 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.  
<http://research.google.com/archive/gfs.html>,
  - Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI'04, San Francisco, CA, December, 2004.  
<http://research.google.com/archive/mapreduce.html>
- Circa 2005
  - Doug Cutting y Mike Cafarella crean Hadoop

# Procesamiento del Big Data

Necesitamos:

- Escalabilidad con grandes volúmenes de datos
  - 1000's de máquinas
  - 10.000's de discos
- Equipos y redes de bajo coste
  - poco fiables
  - elevadas latencias
- Facilidad de programación
- Tolerancia a fallos automática



# Problemas en la ejecución

- Los nodos baratos fallan, sobre todo si hay muchos
  - Tiempo medio entre fallos para 1 nodo = 3 años
  - Tiempo medio entre fallos para 1000 nodos = 1 día
  - **Solución:** implementar tolerancia a fallos en el sistema
- Red de bajo coste = elevada latencia
  - **Solución:** llevar la computación a los datos
    - Minimizar el movimiento de los datos
- Dificultad de programación
  - **Solución:** un modelo de programación paralelo intensivo en datos
    - Los usuarios escriben funciones simples
    - El sistema distribuye el trabajo y gestiona los fallos

## **2. Modelo de procesamiento MapReduce**

# ¿Qué es MapReduce?

Modelo de programación data-parallel diseñado para escalabilidad y tolerancia a fallos en grandes sistemas de *commodity hardware*

- Basado en la combinación de operaciones Map y Reduce

# ¿Qué es MapReduce?

Modelo de programación data-parallel diseñado para escalabilidad y tolerancia a fallos en grandes sistemas de *commodity hardware*

- Basado en la combinación de operaciones Map y Reduce

Diseñado originalmente por Google (2004)

- Usado en múltiples operaciones
- Manejo de varios petabytes diarios

Popularizado por la implementación open source Apache Hadoop

- Usado por múltiples organizaciones como Facebook, Twitter, eBay, LinkedIn, Rackspace, Yahoo!, AWS, etc.

# ¿Para qué se usa?

- En Google:
  - Construcción de índices para el buscador (pagerank)
  - Clustering de artículos en Google News
  - Búsqueda de rutas en Google Maps
  - Traducción estadística
- En Facebook:
  - Minería de datos
  - Optimización de ads
  - Detección de spam
  - Gestión de logs
- En investigación:
  - Análisis astronómico, bioinformática, física de partículas, simulación climática, procesamiento del lenguaje natural, ...

# Modelo de programación

Inspirado en la programación funcional

- Operación Map:

```
square n = n * n
map op [] = []
map op (x:xs) = op x : map op xs
map (square) [1..5] => [1,4,9,16,25]
```

- Operación Reduce:

```
reduce op (x:[]) = x
reduce op (x:xs) = (op x (reduce op xs))

reduce (+) (map (square) [1..5]) => 55
```

# Modelo de programación

Entrada y salida: listas de pares clave/valor

- El programador especifica las funciones map y reduce
- Función Map: genera claves/valores intermedios

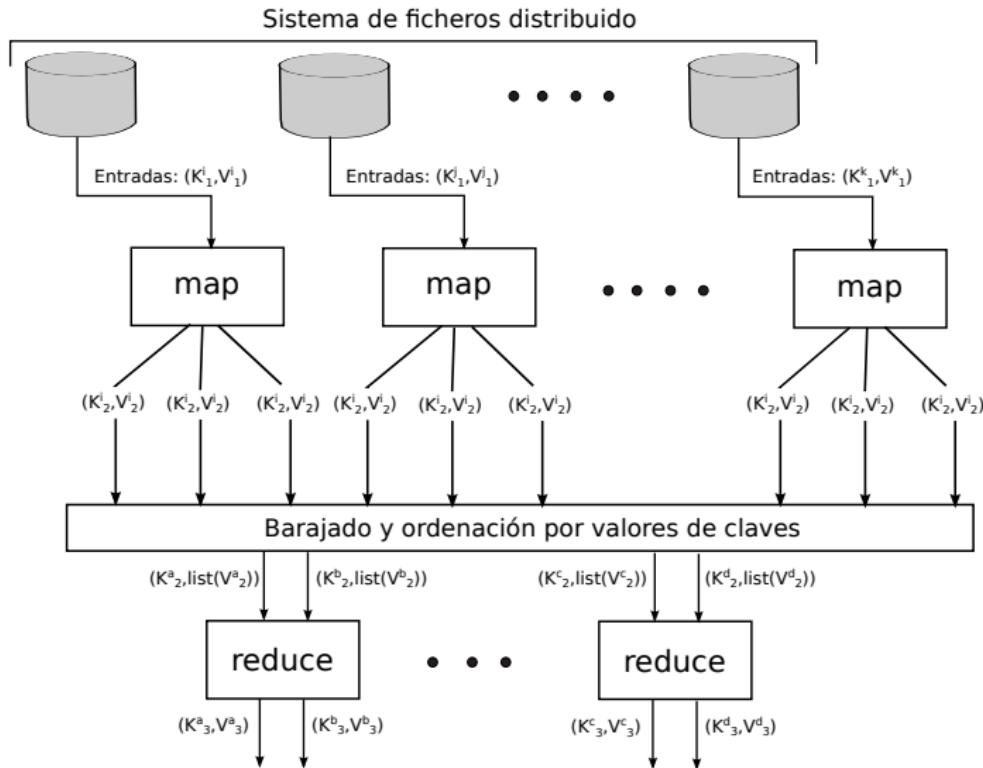
$$\text{map}(K_1, V_1) \rightarrow \text{list}(K_2, V_2)$$

- Para cada  $K_1$  y  $V_1$  se obtiene una lista intermedia
- $(K_2, V_2)$  es un par clave/valor intermedio
- Función Reduce: combina los valores intermedios para cada clave particular

$$\text{reduce}(K_2, \text{list}(V_2)) \rightarrow (K_3, \text{list}(V_3))$$

- Para cada clave de salida  $K_3$  se genera una lista de valores
  - $\text{list}(V_3)$  suele tener un único valor
  - A menudo,  $K_3 = K_2$

# Automatización del proceso



# Ejemplos de uso

1 Definición y conceptos sobre Big Data

2 Modelo de procesamiento MapReduce

- Ejemplos de uso
- Características de la ejecución
- Optimizaciones

3 Ejemplos de implementaciones

# Ejemplo: WordCount

Cuenta las ocurrencias de cada palabra en ficheros de texto

- **Entrada:**

$$K_1 = \emptyset, V_1 = \text{línea}$$

- **Salida:**

*pares (palabra, nº de ocurrencias)*

# Ejemplo: WordCount

Cuenta las ocurrencias de cada palabra en ficheros de texto

- **Entrada:**

$$K_1 = \emptyset, V_1 = \text{línea}$$

- **Salida:**

pares (*palabra, nº de ocurrencias*)

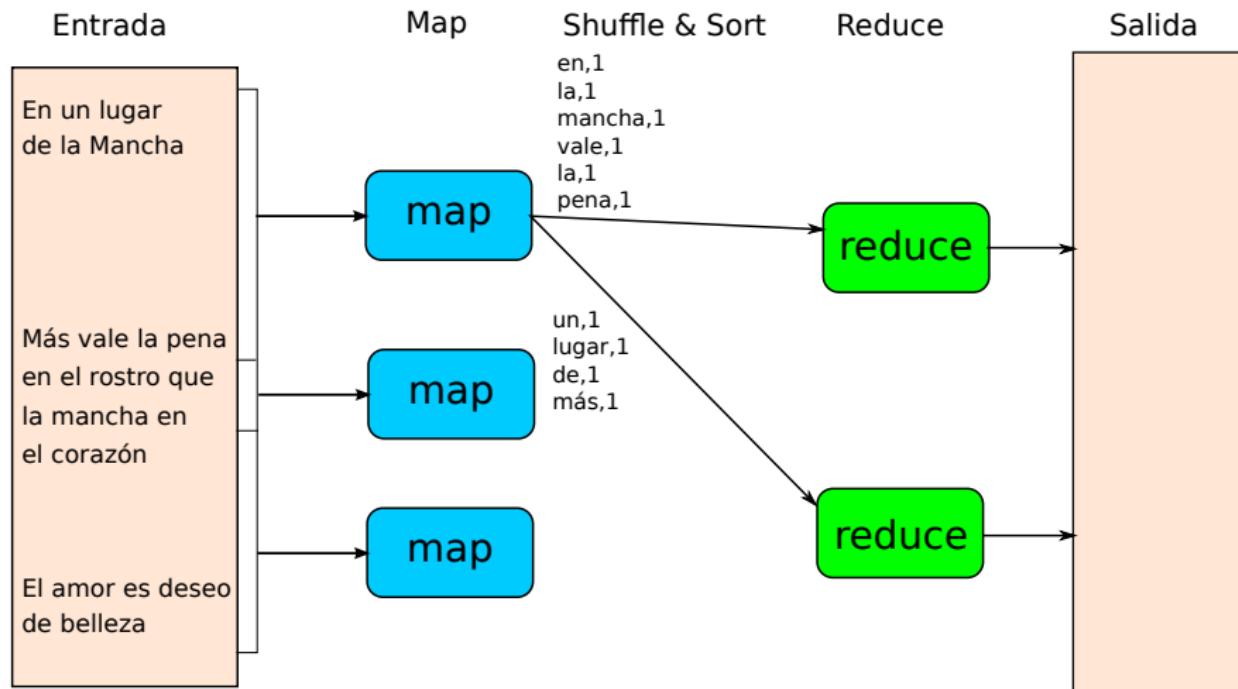
- Pseudocódigo del Map

```
map(key, value):  
//key: nada, value: línea de texto  
    for each word w in value emit(w, 1)
```

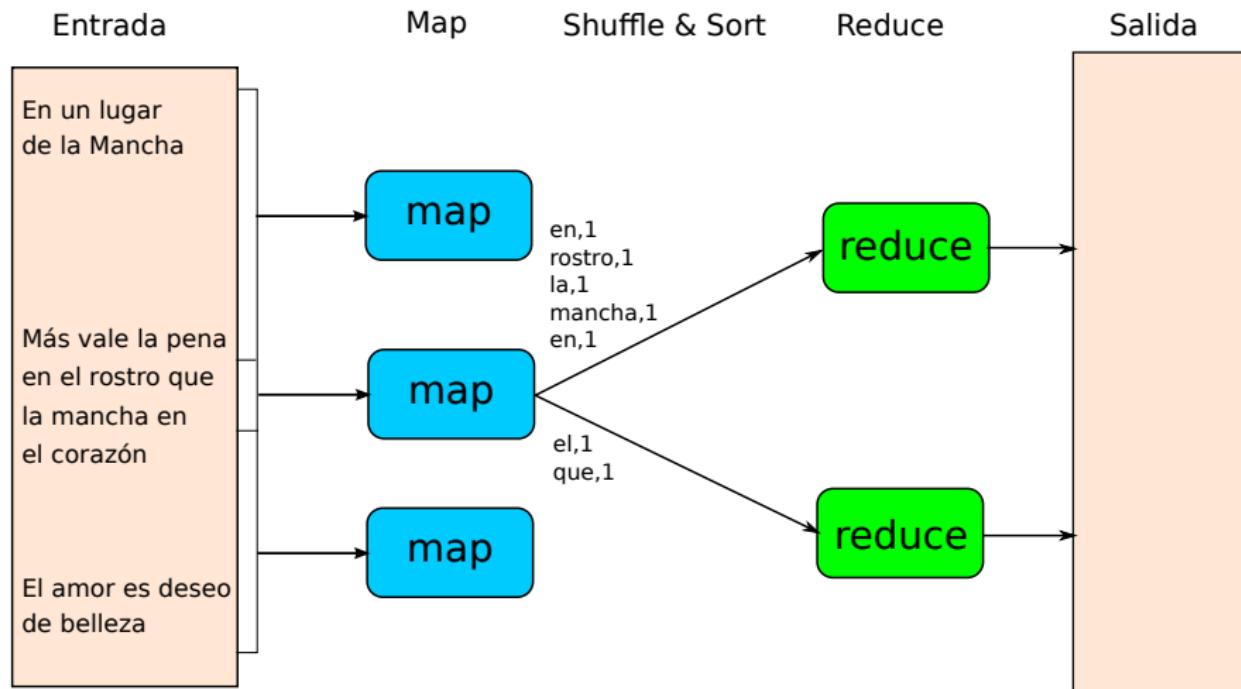
- Pseudocódigo del Reduce

```
reduce(key, values):  
//key: palabra; values: un iterador sobre los 1s  
    emit(key, sum(values))
```

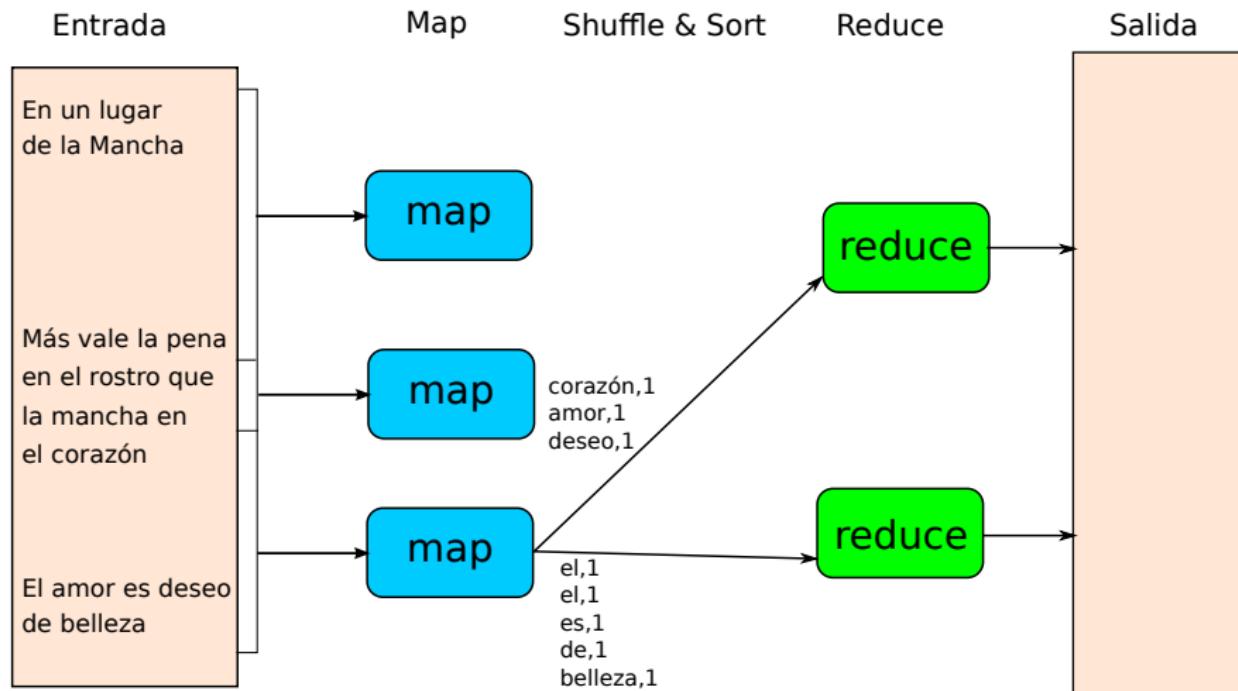
# Ejemplo: WordCount



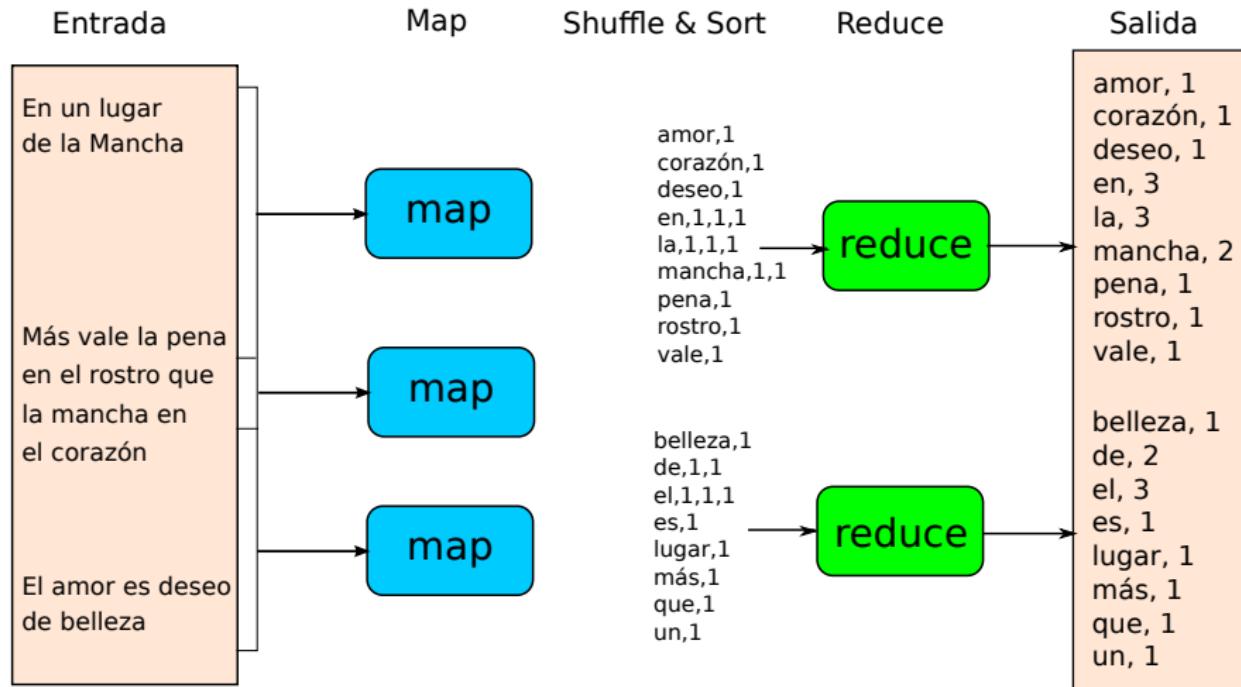
# Ejemplo: WordCount



# Ejemplo: WordCount



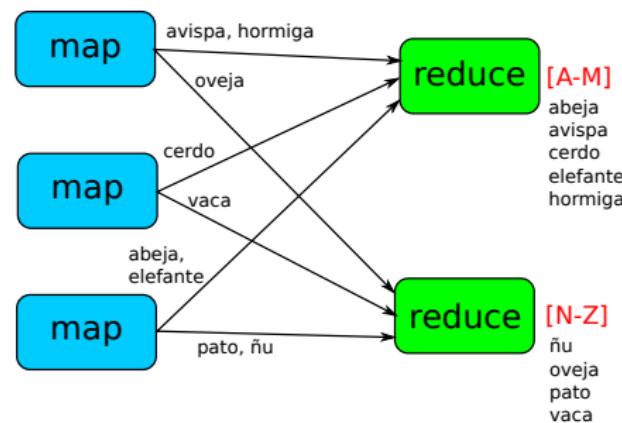
# Ejemplo: WordCount



# Ejemplo: Sort distribuido

- **Entrada:** Pares (id, valor)
- **Salida:** Pares ordenados por id
- Map: Función identidad
- Reduce: Función identidad
- Función de particionado  $P$ :

$$k_1 < k_2 \Rightarrow P(k_1) < P(k_2)$$



## Ejemplo: índice inverso

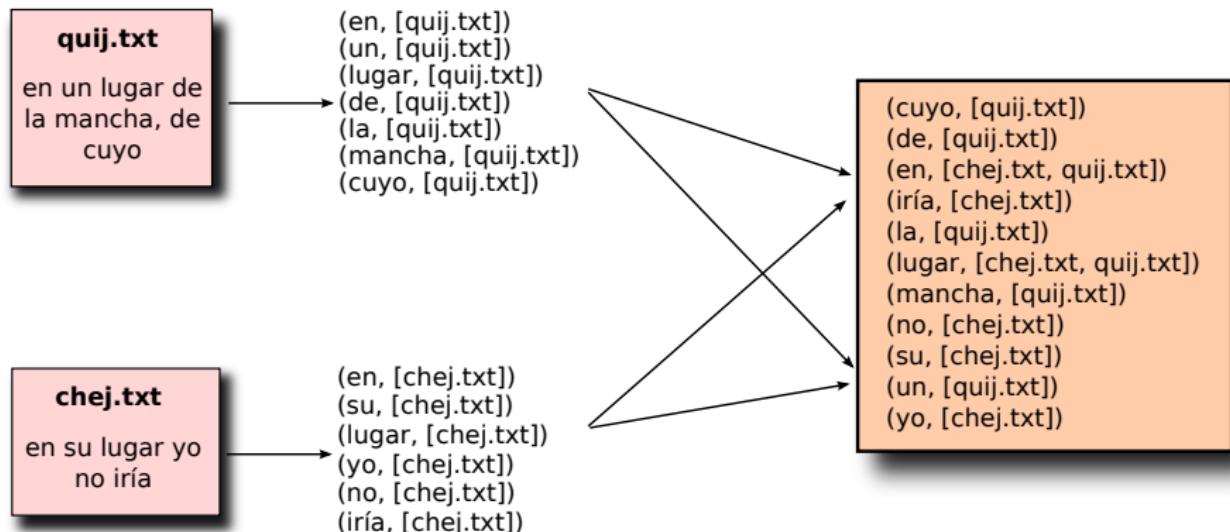
- **Entrada:** Pares (nombre\_fichero, línea\_de\_texto)
- **Salida:** Pares (palabra, [ficheros\_conteniendo\_palabra])
- Pseudocódigo del Map

```
map(key, value):  
    // key: nombre fichero; value: línea de texto  
    for each word w in value emit(w, key)
```

- Pseudocódigo del Reduce

```
reduce(key, values):  
    // key: palabra; value: nombres de ficheros  
    emit(key, sort(values))
```

# Ejemplo: índice inverso

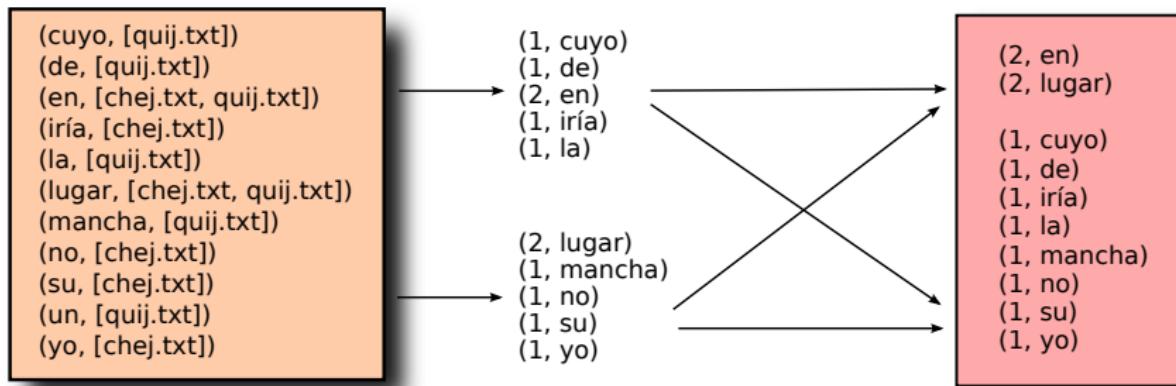


**Mejora:** eliminar duplicados en el map

# Ejemplo: palabras más frecuentes

- **Entrada:** Pares (nombre\_fichero, línea\_de\_texto)
- **Salida:** Lista de las palabras que aparecen en más ficheros
- Solución con dos fases Map-Reduce
  - Fase 1: ejemplo índice inverso
    - (palabra, [ficheros\_conteniendo\_palabra])
  - Fase 2:
    - $\text{Map}(\text{palabra}, [\text{ficheros}]) \rightarrow (\text{ocurrencias}, \text{palabra})$
    - Resultados ordenados por ocurrencias (como en sort distribuido)

# Ejemplo: palabras más frecuentes



# Características de la ejecución

## 1 Definición y conceptos sobre Big Data

## 2 Modelo de procesamiento MapReduce

- Ejemplos de uso
- Características de la ejecución
- Optimizaciones

## 3 Ejemplos de implementaciones

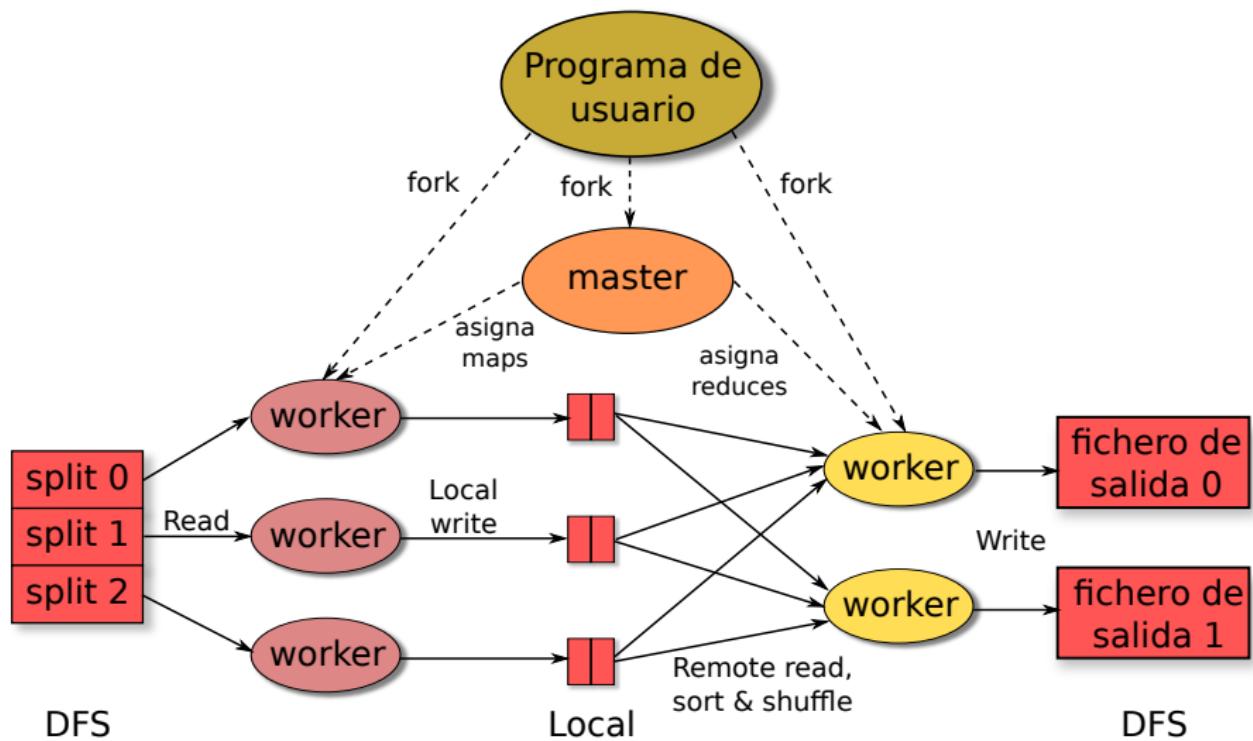
# Características de la ejecución

- Sistema de ficheros distribuido para gestionar los datos
  - GFS (Google File System), HDFS (Hadoop Distributed FS)
  - Ficheros divididos en bloques grandes (64–128 MB en HDFS), con replicación

# Características de la ejecución

- Sistema de ficheros distribuido para gestionar los datos
  - GFS (Google File System), HDFS (Hadoop Distributed FS)
  - Ficheros divididos en bloques grandes (64–128 MB en HDFS), con replicación
- Arquitectura *master-worker*
- Los map workers procesan *splits* de datos
  - Se inician en el mismo nodo o rack que sus datos de entrada
  - Minimiza el uso de la red
- Los map workers graban su salida en el disco local antes de servirla a los reduce workers
  - Permite recuperar una tarea reduce en caso de fallo
- Los reduce workers graban su salida en el sistema distribuido
  - Replicación en los resultados
  - Un fichero de salida por cada reduce

# Características de la ejecución



# Diferencia entre bloques y splits

*Lorem ipsum dolor sit amet, consectetur adipiscing elit.*

*Aenean vehicula fermentum nunc, ut consequat tellus ultricies at. Mauris scelerisque magna imperdiet, tempor sem at, ultrices purus. Nulla scelerisque bibendum sollicitudin. Proin urna nibh, cursus at ipsum quis, aliquet aliquam dui. Suspendisse lacinia bibendum neque, nec rhoncus justo faucibus sed. Curabitur mauris dui, blandit at urna quis, accumsan blandit enim. Suspendisse sagittis, nisi eu euismod commodo, nibh mi condimentum enim,*

*nec lobortis dolor lacus nec orci. Praesent ut facilisis lectus. Nullam vitae leo elementum, pharetra turpis eget, dapibus odio.*

*Pellentesque dignissim, augue ut dapibus consequat, metus est feugiat magna, sed porttitor lorem ante et massa. Nam non lectus nec elit cursus finibus. Sed consectetur neque mollis elit varius fermentum. Mauris vel magna sapien. Nunc quis libero nulla. Duis eu libero eros. Quisque imperdiet fringilla malesuada.*

*Aenean facilisis est sed metus mollis tristique. Integer tincidunt tortor id lectus pulvinar lacinia. Suspendisse finibus enim sed mauris ullamcorper, vitae egestas orci lobortis. Donec id ullamcorper quam. Nulla consectetur interdum nisl, sed ultricies tellus eleifend ut. Aenean risus enim, accumsan et felis nec, consequat accumsan orci. Curabitur id porta felis. Curabitur faucibus turpis in purus ullamcorper, non efficitur velit tempus.*

*Sed molestie neque in feugiat elementum.*

**Bloques: tamaño cte. Splits: número entero de registros**

# Número de Maps y Reduces

- $M$  tareas map y  $R$  tareas reduce
- Hacer  $M + R$  mucho mayor que el número de PEs disponibles
  - Mejora la utilización del cluster

# Número de Maps y Reduces

- $M$  tareas map y  $R$  tareas reduce
- Hacer  $M + R$  mucho mayor que el número de PEs disponibles
  - Mejora la utilización del cluster
- Habitualmente,  $M$  igual al número de splits de entrada
  - Mejora el balance de carga dinámico y acelera la recuperación cuando falla un *worker*

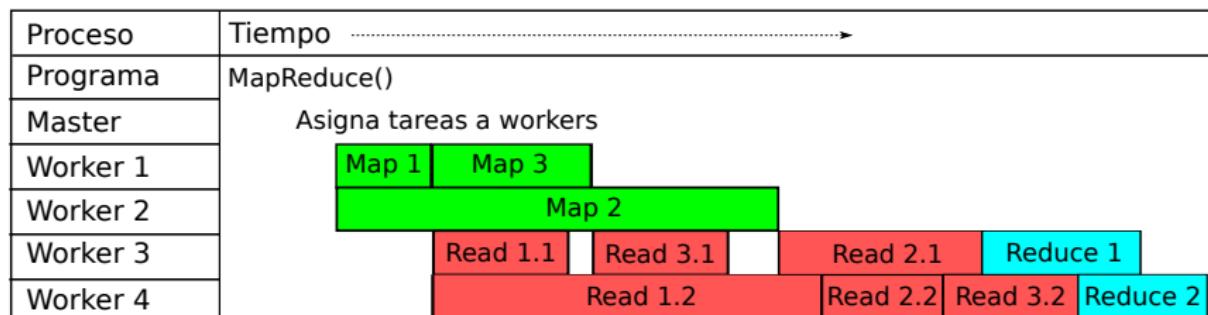
- $M$  tareas map y  $R$  tareas reduce
- Hacer  $M + R$  mucho mayor que el número de PEs disponibles
  - Mejora la utilización del cluster
- Habitualmente,  $M$  igual al número de splits de entrada
  - Mejora el balance de carga dinámico y acelera la recuperación cuando falla un *worker*
- $R$  menor que  $M$ 
  - Salida repartida en  $R$  ficheros
  - Valor ajustable por el usuario
  - Función del tipo de problema y los recursos disponibles (memoria, nº cores, ...)

# Coordinación de las tareas

- El Master se encarga de coordinar las diferentes tareas
  - Estado de cada tarea: *idle*, *in-progress*, *completed*
  - Las tareas *idle* se planifican a medida que van quedando workers libres
- Cuando una tarea map acaba, le envía al Master la localización y tamaño de sus  $R$  ficheros intermedios, uno por tarea reduce
- El Master envía esta información a las tareas reduce

# Solapamiento de tareas

- La fase de reduce no puede comenzar hasta que se complete la de map
- Se puede simultanear la ordenación/mezcla con la ejecución del map
  - Mejora el balance de carga dinámico



# Tolerancia a fallos

- Mediante *heartbeats*, el Master detecta fallos en los workers

# Tolerancia a fallos

- Mediante *heartbeats*, el Master detecta fallos en los workers
- Si falla una tarea map o reduce
  - La tarea se reintenta en otro nodo
    - Correcto para los maps porque no tienen dependencias
    - Correcto para los reduces porque las salidas de los maps están en disco
    - Necesario que las tareas no tengan efectos colaterales
  - Si la misma tarea falla repetidamente, el trabajo MapReduce se aborta y se notifica al usuario (ajustable)

# Tolerancia a fallos

- Mediante *heartbeats*, el Master detecta fallos en los workers
- Si falla una tarea map o reduce
  - La tarea se reintenta en otro nodo
    - Correcto para los maps porque no tienen dependencias
    - Correcto para los reduces porque las salidas de los maps están en disco
    - Necesario que las tareas no tengan efectos colaterales
  - Si la misma tarea falla repetidamente, el trabajo MapReduce se aborta y se notifica al usuario (ajustable)
- Si falla un nodo completo
  - Relanzar sus tareas en curso en otros nodos
  - Reejecutar cualquier map que se hubiera ejecutado en el nodo
    - Necesario pues las salidas de los maps se perdieron

# Tolerancia a fallos

- Si una tarea no progresiona (*straggler* o *rezagada*)
  - Se lanza una segunda copia de la tarea en otro nodo (ejecución especulativa)
  - Se toma la salida de la tarea que acabe antes, y se mata a la otra
  - Situación bastante común en clusters grandes
    - Debidos a errores hardware, bugs software, fallos de configuración, etc.
    - Una tarea rezagada puede enlentecer de forma importante un trabajo

# Tolerancia a fallos

- Si una tarea no progresiona (*straggler* o *rezagada*)
  - Se lanza una segunda copia de la tarea en otro nodo (ejecución especulativa)
  - Se toma la salida de la tarea que acabe antes, y se mata a la otra
  - Situación bastante común en clusters grandes
    - Debidos a errores hardware, bugs software, fallos de configuración, etc.
    - Una tarea rezagada puede enlentecer de forma importante un trabajo
- Si falla el Master
  - Se intenta relanzar de nuevo
    - Las tareas en proceso o acabadas durante el reinicio, se relanzan
  - Si continua fallando, el trabajo se aborta y se le notifica al usuario

## 1 Definición y conceptos sobre Big Data

## 2 Modelo de procesamiento MapReduce

- Ejemplos de uso
- Características de la ejecución
- Optimizaciones

## 3 Ejemplos de implementaciones

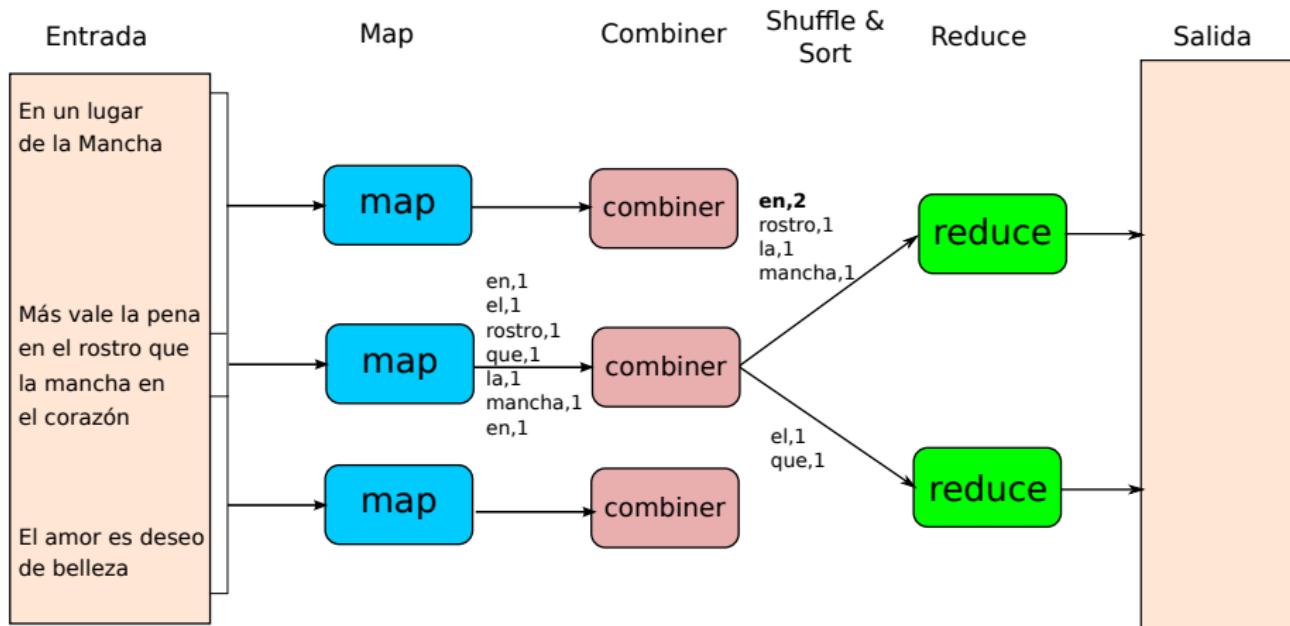
# Combinador

- Un combinador (*Combiner*) es una función de agregación local para las claves repetidas de cada map
- Puede ahorrar ancho de banda al reducir el tamaño de los datos intermedios del map

$$\text{combiner}(K_2^m, \text{list}(V_2^m)) \rightarrow (K_3^m, \text{list}(V_3^m))$$

- Habitualmente, misma función que el reducer
- Se ejecuta en el mismo nodo que el map
- Solo puede utilizarse si la función reduce es commutativa y asociativa

# Combinador



# Particionador

- El particionador (*shuffler*) por defecto es un hash de las claves:
  - $\text{hash}(K) \bmod R$
- Garantiza:
  - Claves iguales van al mismo reducer
  - Carga de los reducers relativamente bien balanceada (en muchos casos)
- Hay situaciones en las que puede interesar cambiarlo:
  - Ejemplo:  $\text{hash}(\text{hostame}(URL)) \bmod R$
  - Todas las URLs de cada host se juntan en el mismo fichero de salida

### **3. Ejemplos de implementaciones**

# Ejemplos de implementaciones

## Implementaciones opensource y en la nube

- Hadoop: Implementación open source de MapReduce
- Amazon Elastic MapReduce: ejecución simple de Apache Hadoop, Spark, HBase, Hive, y otras aplicaciones Big Data
- Microsoft Azure HDInsight: servicio en la nube totalmente administrado para el procesamiento Big Data
- Google Cloud Dataproc: Apache Hadoop y Apache Spark nativos en la nube

# Conclusiones

- El modelo de programación MapReduce oculta la complejidad de la distribución del trabajo y la tolerancia a fallos
- Principales aspectos de diseño:
  - Altamente escalable, maneja los fallos hardware
  - Reduce los costes del hardware, programación y administración
- No es adecuado para todos los problemas, pero cuando funciona puede ahorrar mucho tiempo
- Muy apropiado para la ejecución en la nube