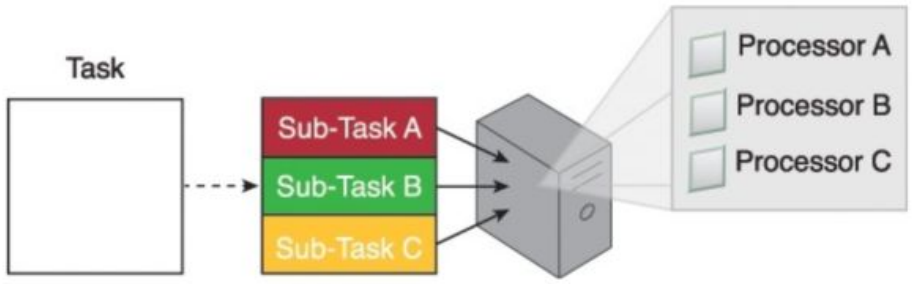
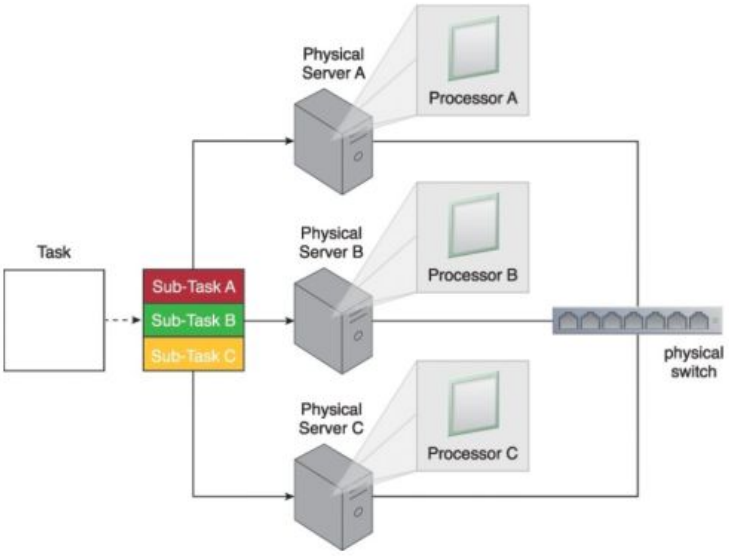
Capítulo 3

Conceptos y Técnicas de Procesamiento de Big Data

Procesamiento Paralelo



Procesamiento Distribuido



Actualización síncrona

- Con actualizaciones sincrónicas, los clientes se comunican directamente con la base de datos y bloquean hasta que se completa la actualización

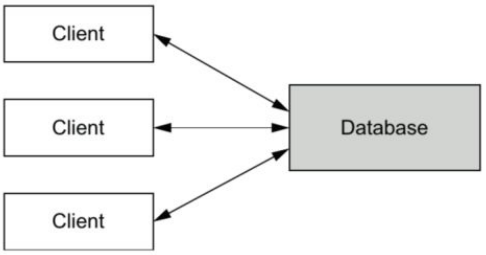


Fig: Arquitectura de capa de velocidad simple que utiliza actualizaciones sincrónicas

- Con actualizaciones síncronas, los clientes envían las actualizaciones a la cola e inmediatamente proceden con otras tareas.
- Después de un tiempo, el stream processor lee un lote de mensajes de la cola y aplica las actualizaciones

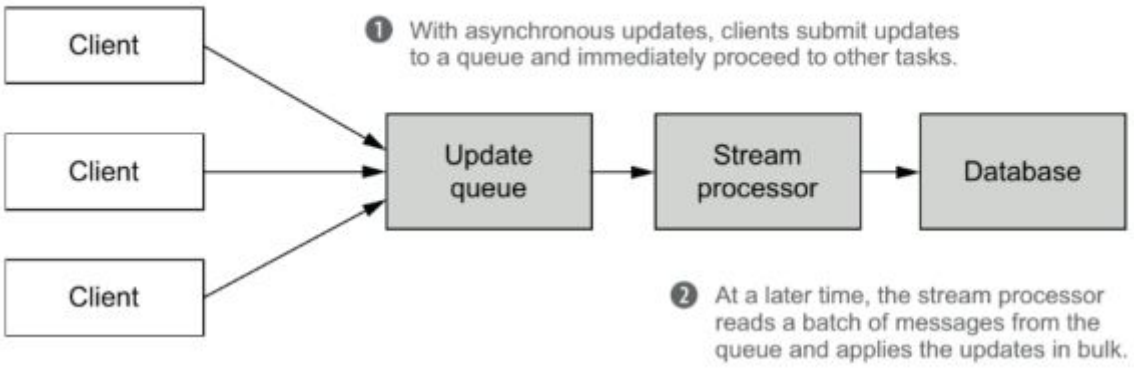


Fig x: Actualizaciones asincrónicas proporcionan mayor rendimiento y manejan fácilmente cargas variables

Colas y Procesamiento de Streams

- Arquitecturas Asíncronas
    - Colas
    - Stream Processing
- Procesamiento sin colas persistentes
    - Dispara y Olvida (Fire and Forget)
    - Tráfico?

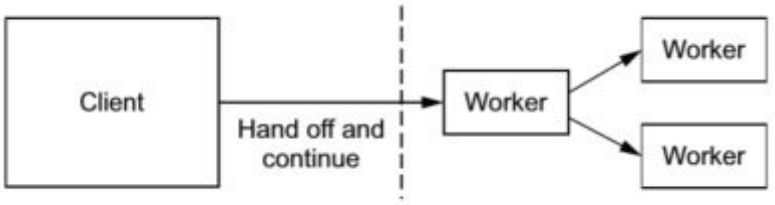


Fig: Para implementar procesamiento asincrónico sin colas, un cliente envía un evento sin monitorear si su procesamiento es exitoso.

Servidor de colas de único consumidor - Single-consumer queue server

- Los mensajes se eliminan de la cola cuando son confirmados
- Múltiples aplicaciones consumiendo los mismos eventos?
- El problema es que la cola controla que fue consumido y que no

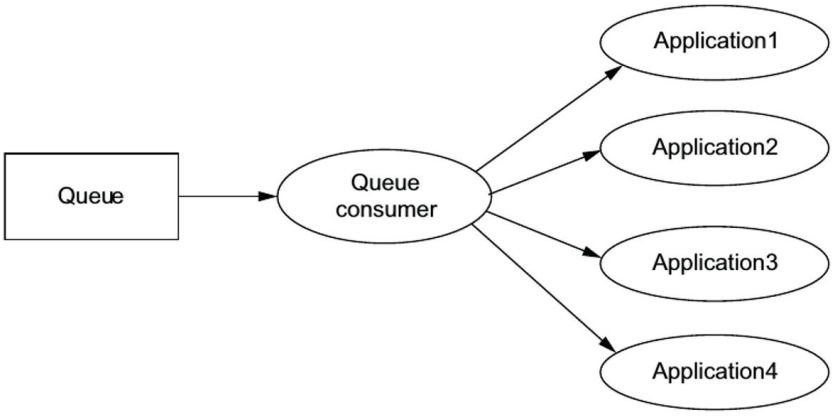


Figure 14.2 **Multiple applications sharing a single queue consumer**

Multi-consumer queues

Pasar el control de los eventos consumidos a la aplicación. Con cola multiconsumidor, las aplicaciones requieren ítems específicos de la cola y es responsable del tracking de los procesos exitosos de cada evento

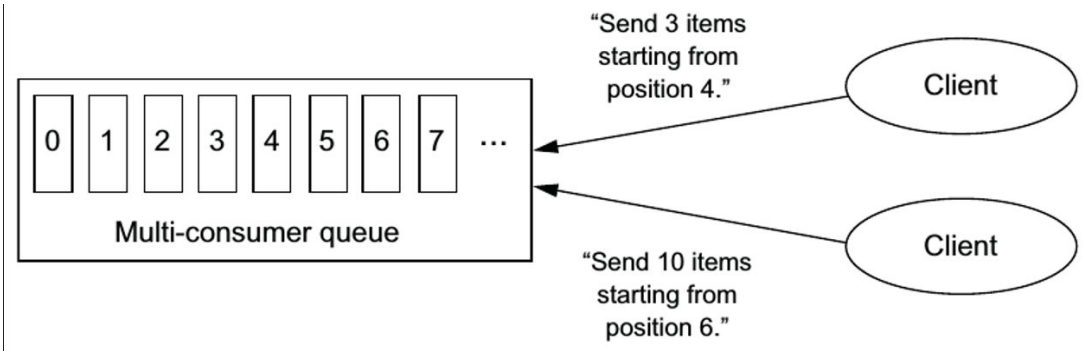


Figure 14.3 **With a multi-consumer queue, applications request specific items from the queue and are responsible for tracking the successful processing of each event.**

Stream processing

- Procesar los eventos y actualizar las vistas en tiempo real
    - Uno a la vez
    - Micro Batches

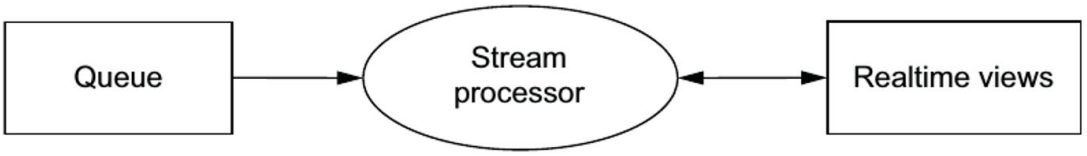


Figure 14.4 **Stream processing**

	One-at-a-time Uno a la vez	Micro-batched Micro-por lotes
Baja latencia	✔	
Alto Rendimiento		✔
Semántica al menos una vez	✔	✔
Semántica exactamente una vez	en algunos casos	✔
Modelo de programación simple	✔	

Colas y Procesadores(Workers)

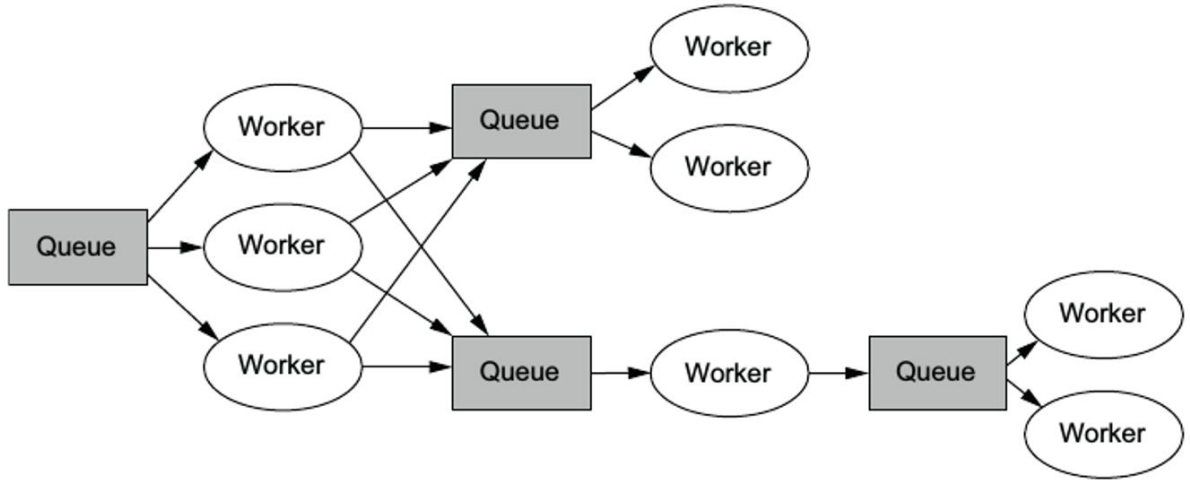


Figure 14.6 A representative system using a queues-and-workers architecture. The queues in the diagram could potentially be distributed queues as well.

Un sistema representativo usado en arquitectura de colas y trabajadores. Las colas en el diagrama pueden potencialmente ser colas distribuidas

Capıtulo 3 - Map Reduce

Big problems

- Facebook con 10 mil millones de fotos (x4: 40 mil millones de archivos), un petabyte en total; Se agregan 2-3 terabytes cada día (2008)
- Bolsa de Nueva York: un terabyte de datos comerciales por día
- El Archivo de Internet: se agregan 100 terabytes por mes; 3 petabytes en total (2009)
- El Gran Colisionador de Hadrones en Ginebra, Suiza produce ~ 15 petabytes por año
- La Web: 100 mil millones de páginas web → 400-500 terabytes comprimidos (duplicados en varios clústeres)
- eBay tiene 6,5 PB de datos de usuario + 50 TB / día (2009)

Parallel Algorithm

Un algoritmo que

- se puede ejecutar una pieza a la vez
- en muchos dispositivos de procesamiento diferentes
- luego volver a armar
- para obtener el resultado correcto

Un problema que demora 4 meses en una máquina, solo tomará 3 horas en 1000 máquinas

Challenges

- Identificar el trabajo que se puede realizar al mismo tiempo
- Sin dependencia de datos entre subproblemas
- Mapeo del trabajo a las unidades de procesamiento
- Distribuyendo el trabajo
- Administrar el acceso a los datos compartidos
- Sincronizar varias etapas de ejecución
- Recopilación de resultados Tolerancia a fallos

Programming Efforts

Trabajo por hacer:

- Comunicación y coordinación
- Recuperarse de un fallo de la máquina
- Informe de estado
- Depuración
- Mejoramiento
- Localidad

Esto debe repetirse para cada problema distribuido que desee resolver.

Prelude to MapReduce

MapReduce es un paradigma diseñado por Google para hacer que un subconjunto (grande) de problemas distribuidos sea más fácil de codificar
Automatiza la distribución de datos y la agregación de resultados
Restringe las formas en que los datos pueden interactuar para eliminar bloqueos.
Proporciona una interfaz genérica que oculta la distribución y la complejidad de los cálculos a gran escala.

Map and Reduce

Dos funciones principales:

- Map: toma datos y crea registros de datos interesantes
- Reduce: toma datos interesantes del mapeador y los resume

El esquema permanece igual, mapee y reduzca el cambio para adaptarse al problema

History

Los conceptos de MAP y REDUCE provienen de la programación funcional (Lisp, ML - Metalanguage)
Framework lanzado internamente en Google en 2003.
En 2004, Jeffrey Dean y Sanjay Ghemawat publicaron un artículo "MapReduce: procesamiento de datos simplificado en grandes clústeres"
En 2007, primera implementación de código abierto (Hadoop)
En 2010, el primer taller internacional sobre MapReduce y sus aplicaciones (MAPREDUCE'10)

MapReduce Provides

Distribución y paralelización automática

- Reduce la complejidad de la sincronización
- Divide datos automáticamente
- Maneja el equilibrio de carga
- Optimización de la transferencia de red y disco

Tolerancia a fallos

- Proporciona transparencia de fallas

Herramientas de estado y monitoreo

Abstracción limpia para programadores

Elimina muchas preocupaciones de confiabilidad de la lógica de aplicación

Parallel Computing at Google

Otros sistemas de cosecha propia

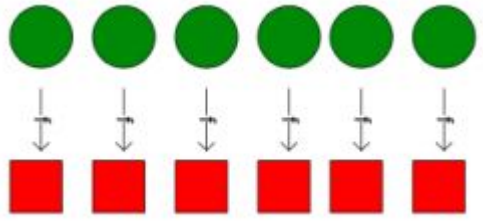
- Google File System (GFS): un sistema de archivos distribuido tolerante a fallas
- BigTable: una base de datos distribuida tolerante a fallos

Alternativa de Hadoop:

- HDFS
- HBase

Functional Programming - Map

map function applies a function f1 to each value of a sequence



map input:

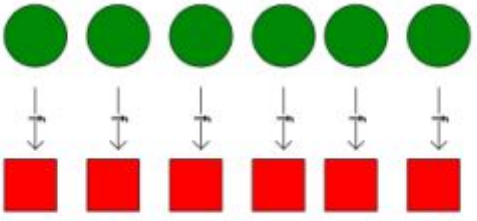
Unary function with parameter of type T1 that produces output of type T2

Array of elements of type T1

map output:

Array of the same size of elements of type T2

map function applies a function f1 to each value of a sequence



Lisp:

```
(define (square n)
      (* n n))

(map square '(1 2 3 4 5))
```

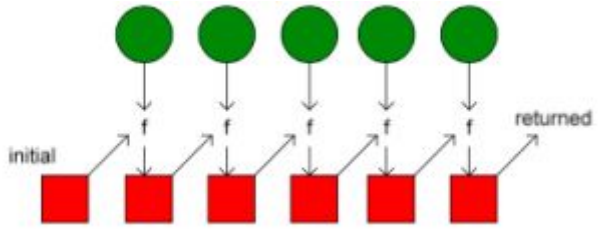
Scala:

```
def square(n:Int):Int = n*n

(1 to 5).map(x => square(x))
```

```
Output:(1 2 3 4 5) => (1 4 9 16 25)
```

Functional Programming - Reduce

reduce combines all elements of a sequence using a binary operator f2



reduce input:

Binary function with parameters of type (T1, T2) that produces output of type T2

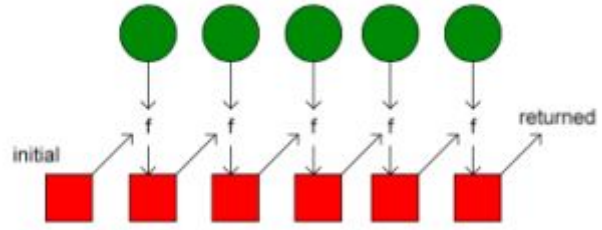
Initial element of type T2

Array of elements of type T1

reduce output:

Element of type T2

reduce combines all elements of a sequence using a binary operator f2



Lisp:

```
(define (plus a b)
      (+ a b))

(reduce plus 0 '(1 2 3 4 5))
```

Scala:

```
def plus(n1:Int, n2:Int):Int
                = n1+n2

(1 to 5).foldLeft(0)(
        (x,y) => plus(x,y))
```

```
Output:(1 2 3 4 5) => (((((0+1)+2)+3)+4)+5) => 15
```

Data Flow in MapReduce

Leer datos de entrada

Map:

- Extraiga algo que le interese de cada registro
- Particione la salida: qué teclas van a qué reductor

Shuffle and Sort: el reductor espera que sus claves estén ordenadas y para cada clave: lista de todos los valores

Reducir:

- Aggregate, summarize, filter, or transform

Escribe los resultados

Key / Value Pairs

```
map (in_key, in_value) ->
  list (out_key, intermediate_value)
```

Processes input key/value pair

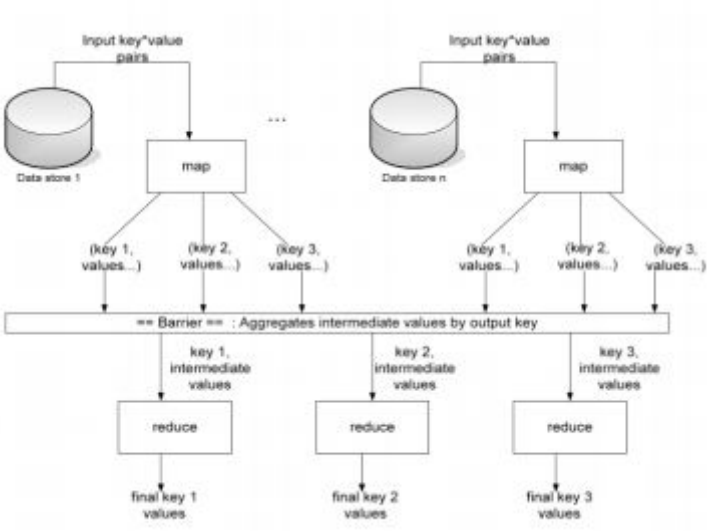
Produces set of intermediate pairs

```
reduce (out_key,list<intermediate_value>)
  -> (out_key, out_value)
```

Combina todos los valores intermedios para una clave en particular
Produce un valor de salida combinado (también puede ser una lista)

Map
Los registros de la fuente de datos (líneas de archivos, filas de una base de datos, etc.) se introducen en la función de mapa como key*value pairs: e.g., (filename, line).
Map produce uno o más valores intermedios junto con una clave de salida de la entrada.

Reduce
Una vez finalizada la fase de mapa, todos los valores intermedios para una clave de salida determinada se combinan en una lista.
reduce combina esos valores intermedios en uno o más valores finales para esa misma clave de salida (en la práctica, generalmente solo un valor final por clave)



Parallelism
funciones map se ejecutan en paralelo, creando diferentes valores intermedios a partir de diferentes conjuntos de datos de entrada
funciones reduce también se ejecutan en paralelo, cada una trabajando en una tecla de salida diferente
Todos los valores se procesan de forma independiente
Bottleneck: la fase de reducción no puede comenzar hasta que la fase del mapa haya finalizado por completo.

Example: Count Word Occurrences

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values)
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
 Emit(AsString(result));
```

```
O my Love's like a red, red rose
That's newly sprung in June;
O my Love's like the melodie
That's sweetly played in tune.
                (R. Burns, 1794)

MAP 1:
<a, 1>
<like, 1>
<Love, 1>
<my, 1>
<O, 1>
<red, 1>
<red, 1>
<rose, 1>
<'s, 1>

MAP 2:
<in, 1>
<June, 1>
<newly, 1>
<'s, 1>
<sprung, 1>
<That, 1>

MAP 3:
<like, 1>
<Love, 1>
<melodie, 1>
<my, 1>
<O, 1>
<'s, 1>
<the, 1>

MAP 4:
<in, 1>
<played, 1>
<'s, 1>
<sweetly, 1>
<That, 1>
<tune, 1>
```

```
REDUCE 1 (a..l):
<a, 1>          => <a, 1>
<in, (1,1)>     => <in, 2>
<June, 1>       => <June, 1>
<like, (1,1)>   => <like, 2>
<Love, (1,1)>   => <Love, 2>

REDUCE 2 (m..r):
<melodie, 1>    => <melodie, 1>
<my, (1,1)>     => <my, 2>
<newly, 1>      => <newly, 1>
<O, (1,1)>      => <O, 2>
<played, 1>     => <played, 1>
<red, (1,1)>    => <red, 2>
<rose, 1>       => <rose, 1>

REDUCE 3 (s..z):
<'s, (1,1,1,1)> => <'s, 4>
<sprung, 1>     => <sprung, 1>
<sweetly, 1>    => <sweetly, 1>
<That, (1,1)>   => <That, 2>
<the, 1>        => <the, 1>
<tune, 1>       => <tune, 1>
```

Example 2: Inverted Web Graph

Para cada página, genere una lista de enlaces entrantes.

¿Por qué querrías tener un gráfico así?

Input: Web documents

Map: For each link L in document D emit <href(L), D>

Reduce: Combinar todos los documentos en una lista

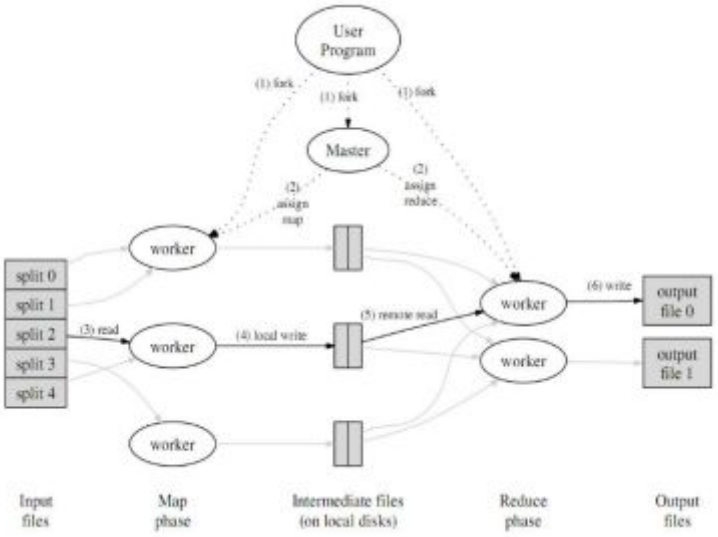
```
techcrunch.com -> apple.com, microsoft.com, ubuntu.com, google.com
reddit.com -> en.wikipedia.org, techcrunch.com
digg.com -> techcrunch.com, microsoft.com, ubuntu.com
```

```
MAP 1:
<apple.com, techcrunch.com>
<microsoft.com, techcrunch.com>
<ubuntu.com, techcrunch.com>
<google.com, techcrunch.com>

MAP 2:
<en.wikipedia.org, reddit.com>
<techcrunch.com, reddit.com>

MAP 3:
<techcrunch.com, digg.com>
<microsoft.com, digg.com>
<ubuntu.com, digg.com>
```

```
REDUCE 1:
<apple.com, techcrunch.com>

REDUCE 2:
<microsoft.com,
   (techcrunch.com, digg.com)>

REDUCE 3:
<ubuntu.com, (techcrunch.com,
   digg.com)>

REDUCE 4:
<google.com, techcrunch.com>

REDUCE 5:
<en.wikipedia.org, reddit.com>

REDUCE 6:
<techcrunch.com, (reddit.com,
   digg.com)>
```

More Examples

Coincidencia de patrones distribuidos
Orden distribuido
La probabilidad de que una palabra esté en mayúscula
Count of URL Access Frequency Term-Vector per Host
Agrupación de documentos
Traducción automática estadística



Functionating

Scheduling

Una maestra, muchas trabajadoras

- Input data split into M map tasks (typically 64 MB in size)
- Reduce phase partitioned into R reduce tasks
- Tasks are assigned to workers dynamically

Maestra asigna cada tarea de mapa a una trabajadora libre

- Considers locality of data to worker when assigning task
- Worker reads task input (often from local disk!)

- Worker produces R local files containing intermediate k/v pairs

Master assigns each map task to a free worker

- Worker reads intermediate k/v pairs from map workers
- Worker sorts & applies user's Reduce operation to produce the output

Localidad

Master program divides up tasks based on location of data: tries to have map tasks on same machine as physical file data, or at least same rack

map task inputs are divided into 64 MB blocks: same size as Google File System chunks

Fault Tolerance

Workers send heartbeats to master

If worker fails

Re-execute completed and in-progress map tasks

Re-execute in-progress reduce tasks

Master notices particular input key/values cause crashes in map, and skips those values on re-execution.

Effect: Can work around bugs in third-party libraries

Master writes checkpoints periodically to database. If master fails a new master is started.

Master is a single machine only, so failing is unlikely, easier just to restart the whole MapReduce task

Task Granularity

Map phase is split on M tasks

Reduce phase is split on R tasks

Practical bounds on how large M and R can be:

Scheduling decisions to make? O(M+R)

States in memory? O(M*R)

Separate output files? R

Example from Google:

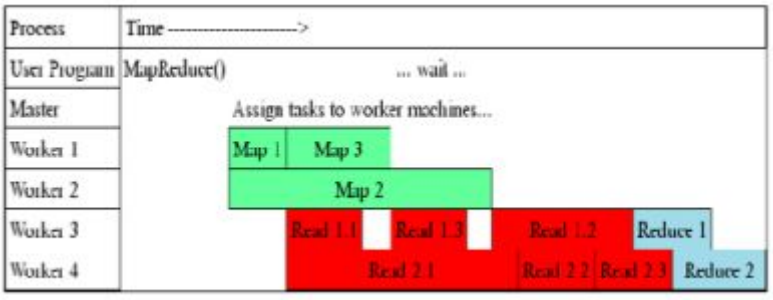
M = 200 k; R = 5 k; workers = 2 k

Fine granularity tasks: many more map tasks than machines

Minimizes time for fault recovery

Can pipeline shuffling with map execution

Better dynamic load balancing



Backup Task

The worst case is not when machine is dead, but when it is barely working, really slow.

Slow workers significantly lengthen completion time

Other jobs consuming resources on machine

Bad disks with soft errors transfer data very slowly

Solution: Near end of phase, spawn backup copies of tasks

Whichever one finishes first "wins"

Effect: Dramatically shortens job completion time

Partitioning

R tasks of reducer should be approximately evenly loaded.

How to ensure this?

Default function uses hashing:

hash(key) mod R

Programmer can override partitioning function

Combiner

"Combiner" function can run on same machine as a mapper

Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth

Typically has the same code as the reducer

Applicable if a reducer function is

commutative

associative

Chaining MapReduce Executions

Output of a MapReduce task can be processed further as input to another MapReduce task

Examples?

Results from the first reducer may either be written to the permanent storage, or left on reducer machines as temporary files.

Mappers of the second MapReduce job start on the same machines, where reducers of the first one were.

MapReduce Implementations

Patented by Google

Google implementation in C++ with bindings to Python and Java via interfaces, not available publicly

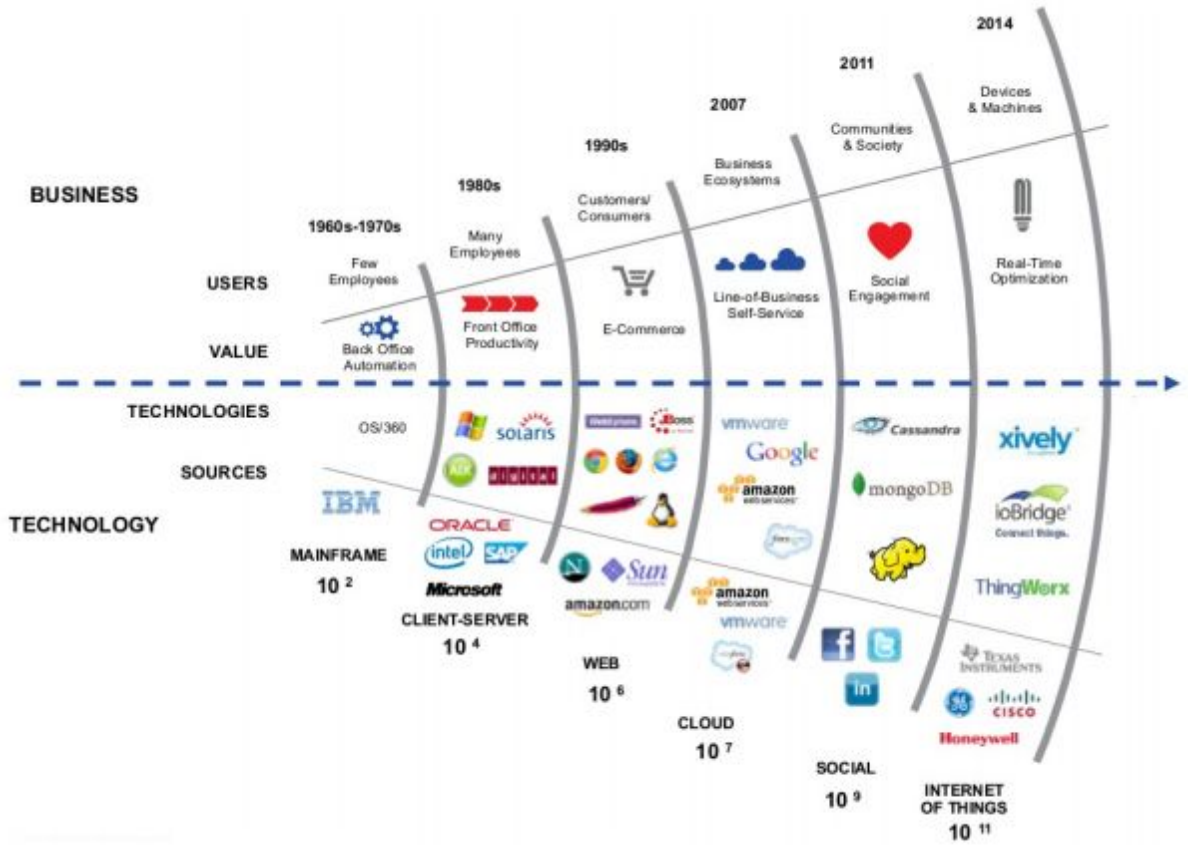
Hadoop project has free open-source implementation in Java

MongoDB

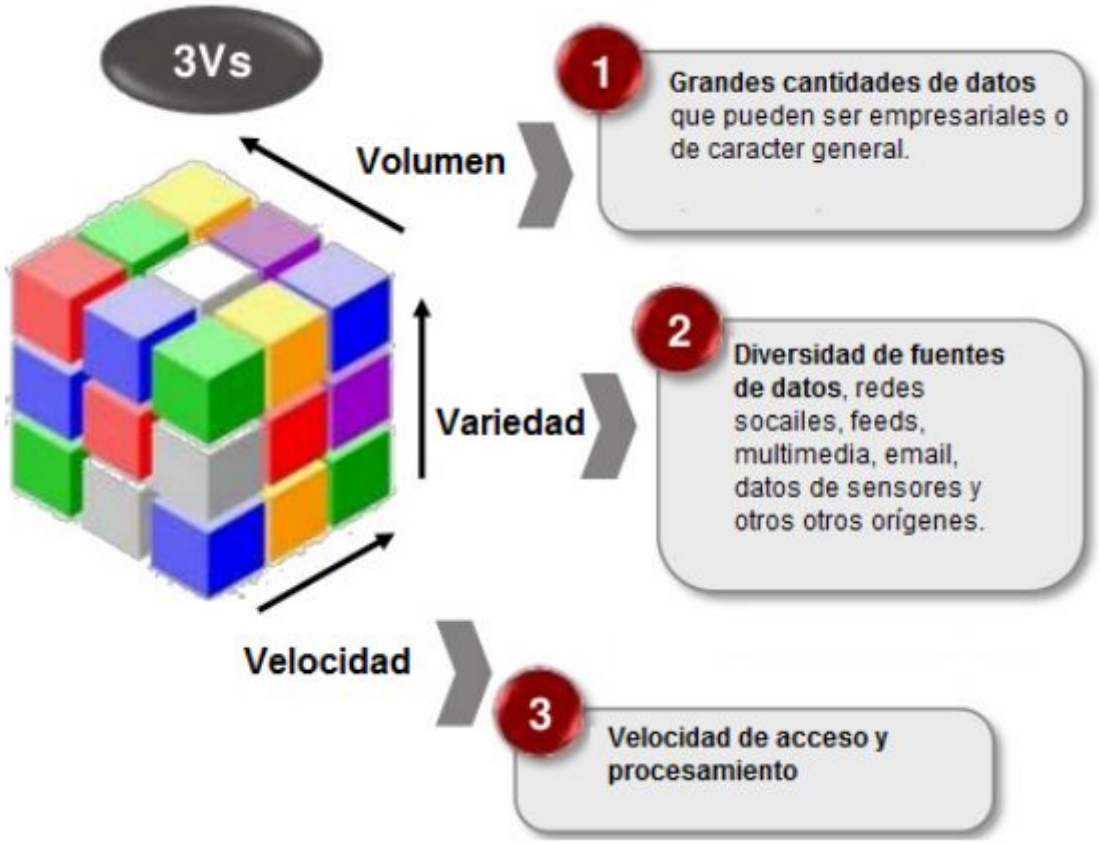
CouchDB

Diapos Hadoop

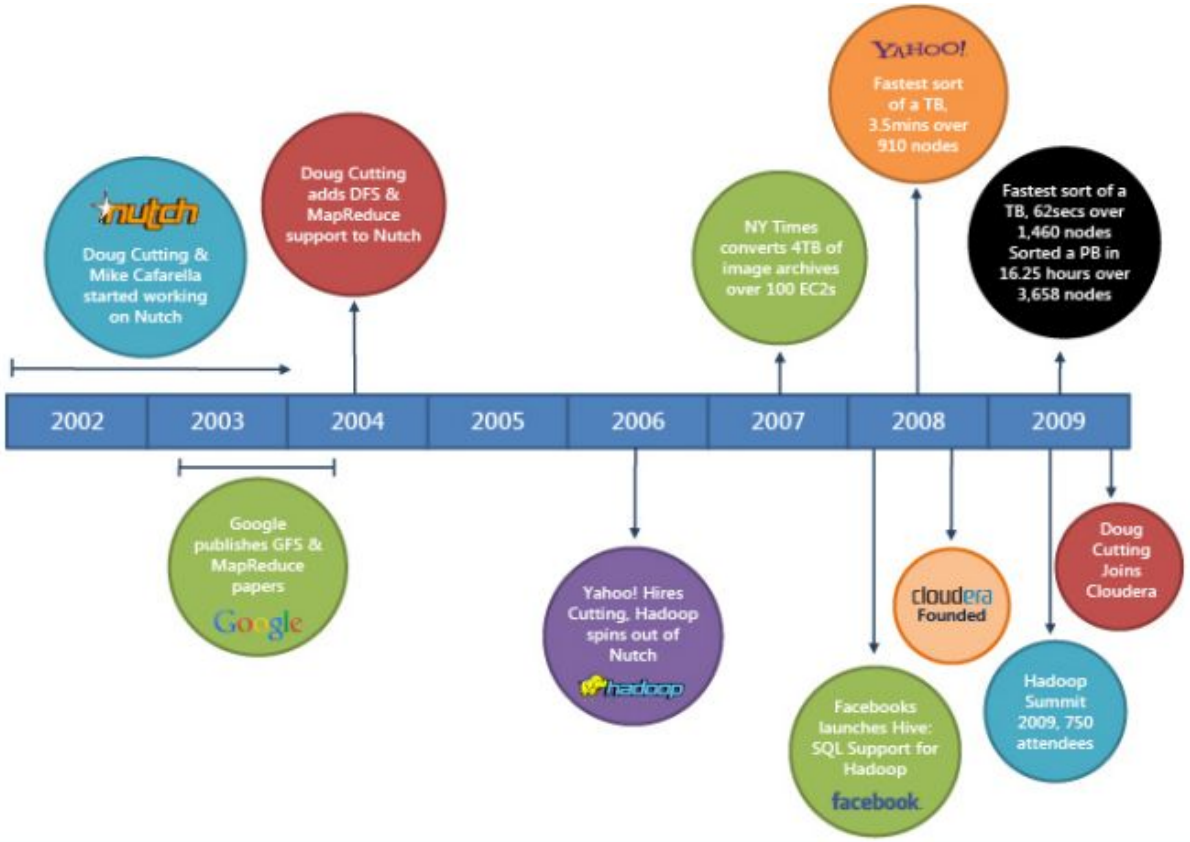
Contexto histórico


BUSINESS
USERS
VALUE
TECHNOLOGIES
SOURCES
TECHNOLOGY
1960s-1970s
Few Employees
Back Office Automation
OS/360
IBM
MAINFRAME
10 2
1980s
Many Employees
Front Office Productivity
ORACLE
Microsoft
CLIENT-SERVER
10 4
1990s
Customers/ Consumers
E-Commerce
amazon.com
WEB
10 6
2007
Business Ecosystems
Line-of-Business Self-Service
vmware
Google
amazon
CLOUD
10 7
2011
Communities & Society
Social Engagement
Cassandra
mongoDB
SOCIAL
10 9
2014
Devices & Machines
Real-Time Optimization
xively
ioBridge
ThingWorx
Honeywell
INTERNET OF THINGS
10 11

3Vs

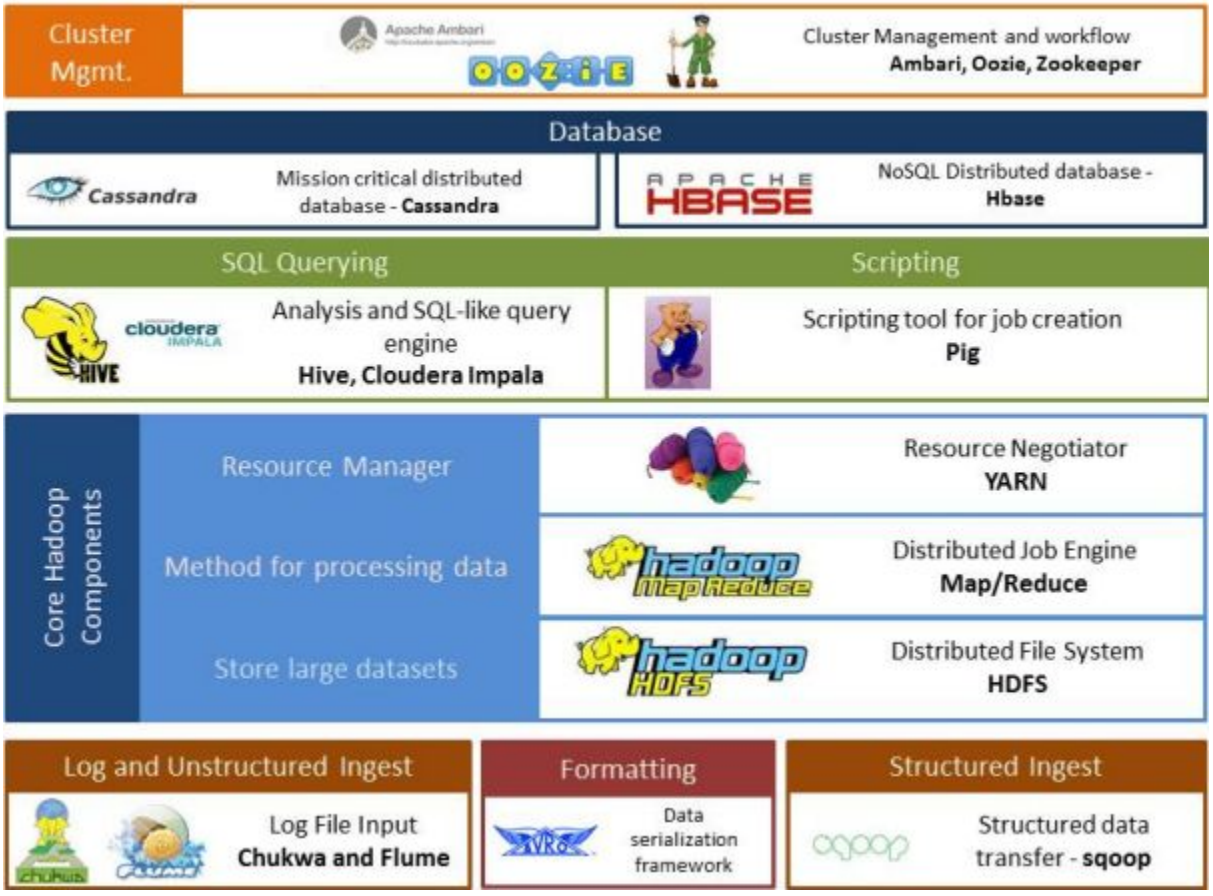

3Vs
Volumen
1
Grandes cantidades de datos que pueden ser empresariales o de caracter general.
Variedad
2
Diversidad de fuentes de datos, redes socailes, feeds, multimedia, email, datos de sensores y otros otros orígenes.
Velocidad
3
Velocidad de acceso y procesamiento

Historia



Ecosistema Hadoop

- Ambari: Simplificación de la gestión de Hadoop (Clústeres).
- Avro: Serialización de datos.
- Cassandra: Base de datos NoSQL distribuida.
- Chukwa: Análisis y recolección de Logs.
- HBase: Base de datos NoSQL Distribuida.
- Hive: Lenguaje de alto nivel similar a SQL que se traduce en tareas MapReduce.
- Mahout: Librería de aprendizaje automático y minería de datos.
- Pig: Lenguaje de alto nivel para generación de tareas MapReduce.
- Spark: Motor de cálculo general.
- Tez: Framework de programación de flujos de datos de carácter general.
- ZooKeeper: Servicio de coordinación distribuido de Hadoop.
- Oozie: Flujo de trabajo para tareas MapReduce
- Sqoop: Conectividad entre bases de datos tradicionales y Hadoop.
- Flume: Transferencia de datos entre los sistemas de empresa y Hadoop



Core Hadoop

Sistema de ficheros distribuido auto gestionado + Sistema de computación distribuido con tolerancia a errores y abstracción de computación paralela

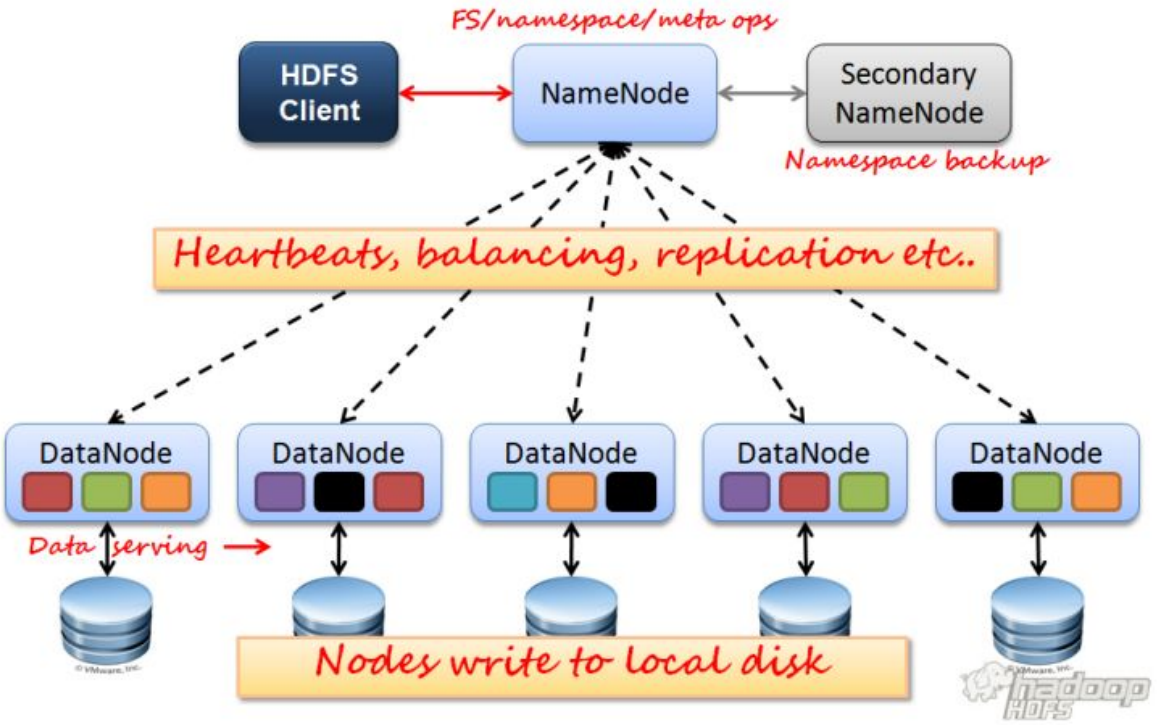
HDFS

HDFS (Hadoop Distributed File System) es una evolución de GFS (Google File System). Es un sistema de archivos distribuido diseñada para contener gran cantidad de datos y el acceso concurrente a los mismos.

HDFS - Características

- Capacidad de almacenaje de grandes cantidades de datos (Terabytes o Petabytes)
- Fiabilidad en el almacenamiento en Cluster
- Posibilidad de leer los ficheros localmente
- Diseñada para lecturas intensas (Penalización sobre la búsqueda aleatoria)
- Operaciones
    - Lectura
    - Escritura
    - Borrado
    - Creación

HDFS - Funcionamiento

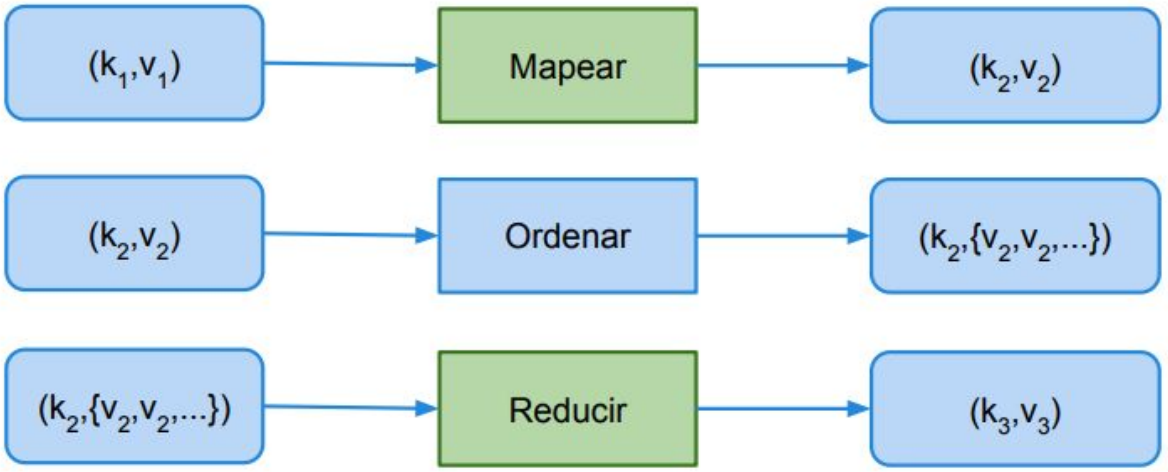


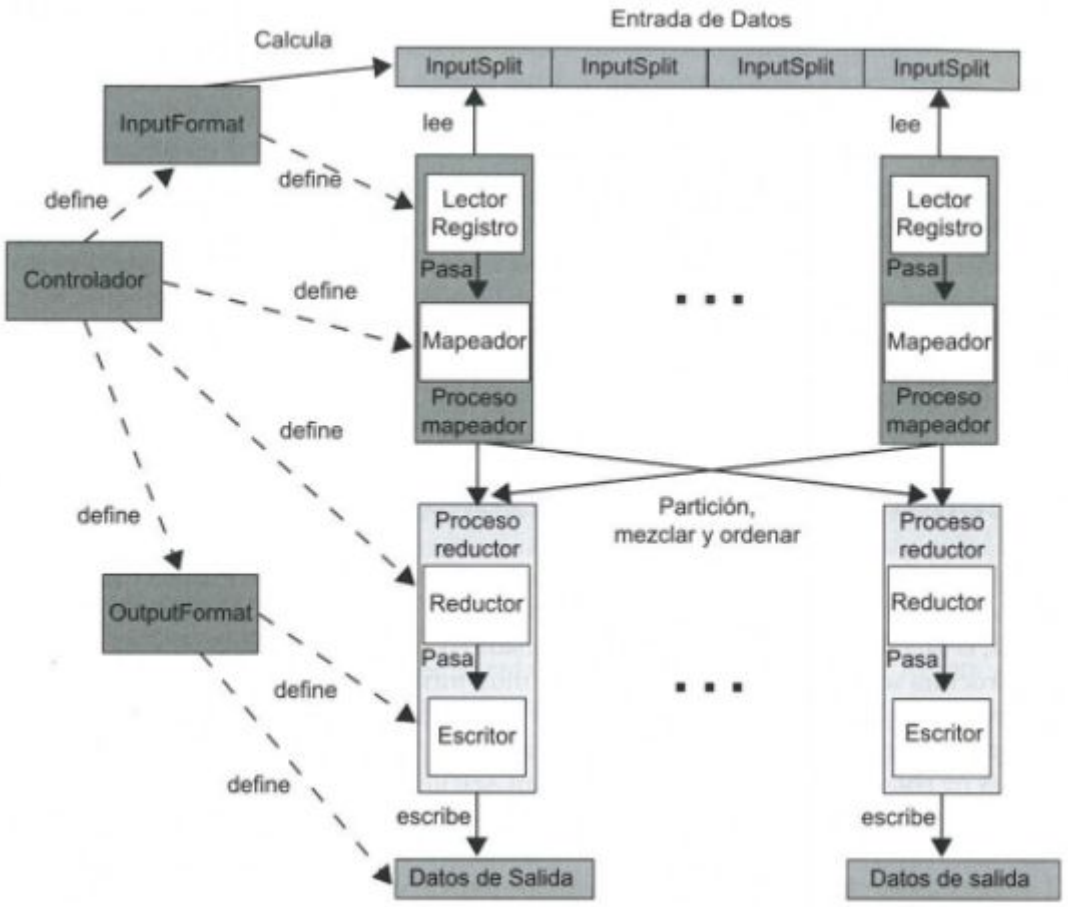
MapReduce

Estructura para ejecutar algoritmos altamente paralelizables y distribuidos utilizando ordenadores "básicos"

Origen del concepto de combinador Map y Reduce de lenguajes de programación funcional como Lisp.

Divide y vencerás





Controlador

- Inicializa el trabajo MapReduce
- Define la configuración
- Enumera los componentes
- Monitoriza la ejecución

Datos de entrada

- Ubicación de los datos para el trabajo
- HDSF
- HBase
- Otros

InputFormat

- Definición de cómo se leen y dividen los datos.
- Definición de InputSplit
- Definición de número de tareas Map

InputSplit

- Define la unidad de trabajo para una tarea Map única

RecordReader

- Define un subconjunto de datos para una tarea map
- Lee los datos desde el origen
- Convierte los datos en pares clave / valor

Mapeador

- Ejecuta un algoritmo definido por el usuario.
- Instancias JVM por tarea Map.
- Aislamiento de tareas
- Fiabilidad del resultado dependiente únicamente de la máquina local.

Partición

- Elección de dónde reducir cada par clave / valor

Mezclar

- Las tareas Reduce trabajan sobre la misma clave
- Aquí se desplazar las salidas de las tareas Map a donde se necesite

Ordenar

- Se reciben los pares clave / valor Mezclados anteriormente y se ordenan para pasarlos al reductor

Reductor

- Ejecuta un algoritmo definido por el usuario
- Recibe una clave y todos los valores asociados

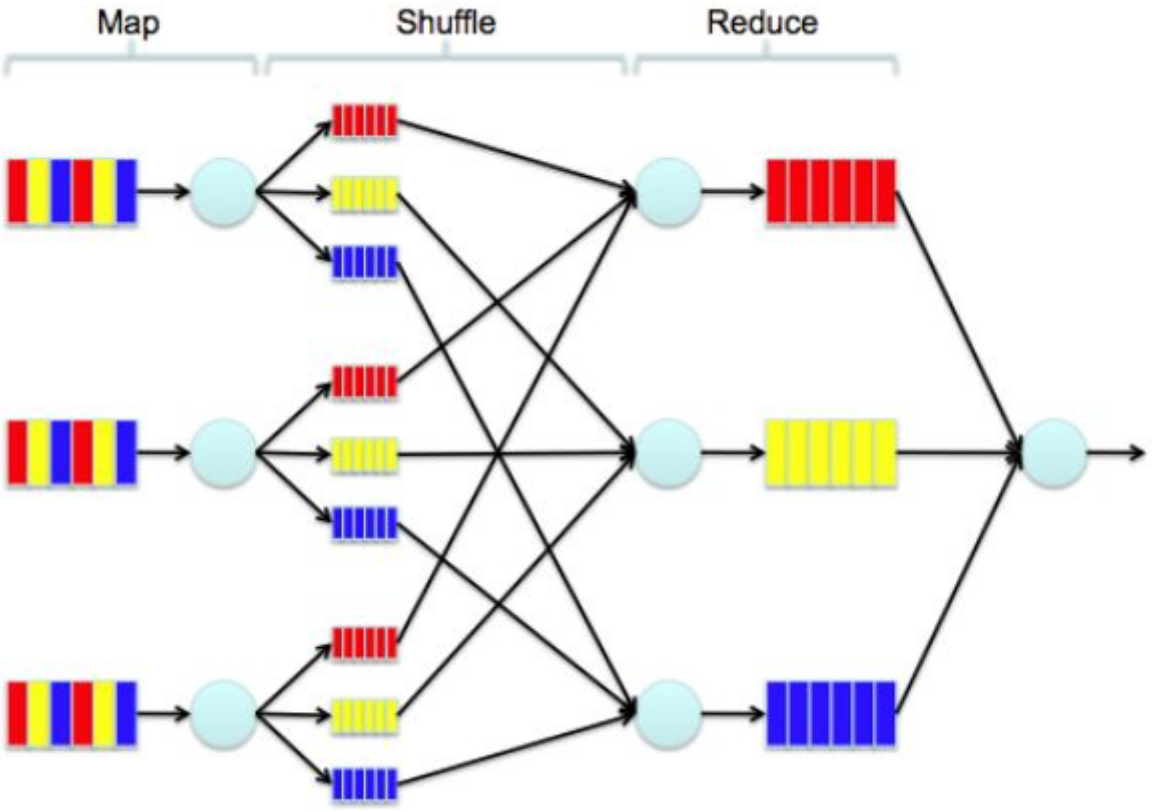
OutputFormat

- Define cómo se escribe la salida del proceso MapReduce.
- Define la ubicación del RecordWriter
- Definir los datos a devolver

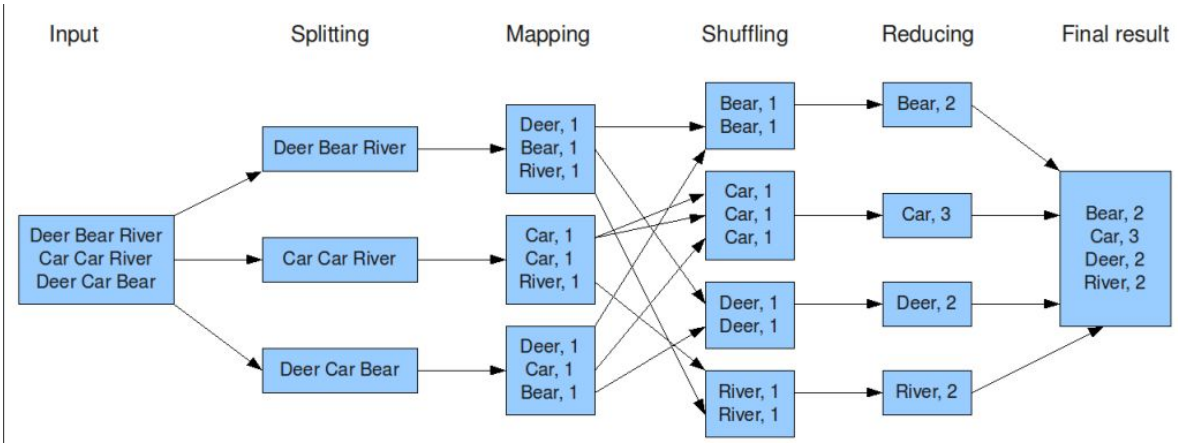
RecordWriter

- Define cómo se escriben los registros individuales de salida

MapReduce: Diagrama de Nodos



MapReduce - Ilustración

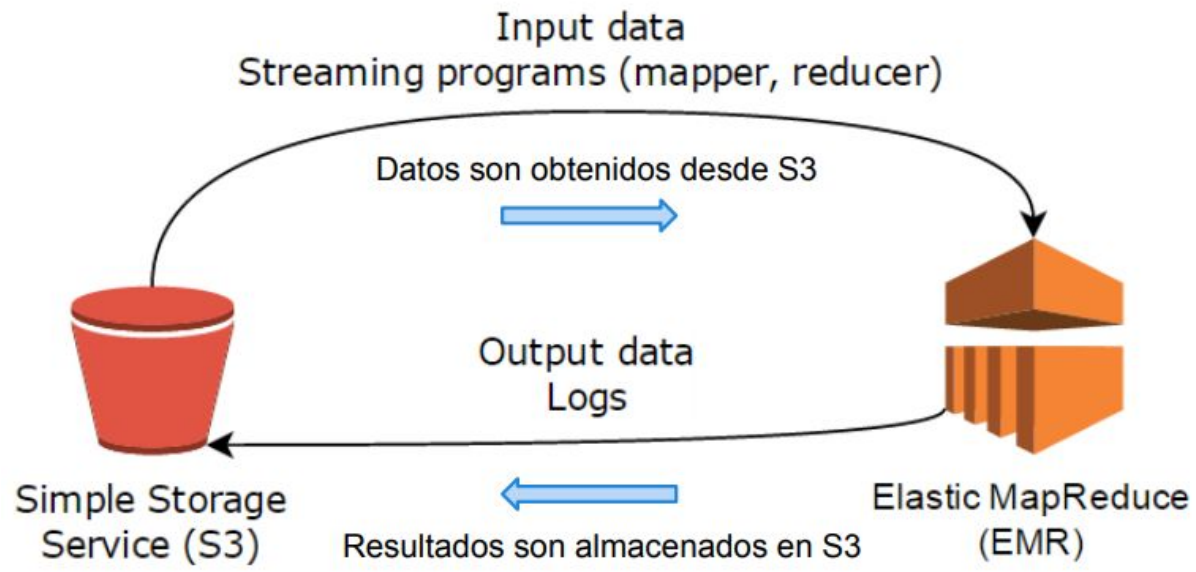


Elastic Map Reduce

Servicio Web de AWS (Amazon Web Services) para implementar el MapReduce mediante Apache Hadoop.

Permite crear trabajos en Apache Hadoop que operan sobre datos en Amazon S3 en instancias de Amazon EC2

Elastic Map Reduce - Arquitectura



Elastic Map Reduce - Streaming

Streaming consiste en analizar y aplicar MapReduce mediante el uso de un Mapper (cuya función es Organizar) y un Reducer (cuya función es Agregar).

Un Streaming job (tarea) es una tarea de Hadoop que consiste de un Mapper y Reducer. Los Mapper y Reducer pueden ser escritos en: Java, Ruby, PHP, Perl, Python, R, Bash, C++..

1) Crear el stream job en la Consola de AWS.
2) Indicar parámetros del stream job
3) Configurar instancias EC2
4) Lanzar el job
5) Revisar ejecución
6) Revisar resultados

“Mapper”

```
#!/usr/bin/php
<?php
//sample mapper for hadoop streaming job
$word2count = array();

// input comes from STDIN (standard input)
while (($line = fgets(STDIN)) !== false) {
  // remove leading and trailing whitespace and lowercase
  $line = strtolower(trim($line));
  // split the line into words while removing any empty string
  $words = preg_split('/\W/', $line, 0, PREG_SPLIT_NO_EMPTY);
  // increase counters
  foreach ($words as $word) {
     $word2count[$word] += 1;
  }
}

// write the results to STDOUT (standard output)

foreach ($word2count as $word => $count) {
  // tab-delimited
  echo "$word\t$count\n";
}

?>
```

“Reducer”

```
#!/usr/bin/php
<?php
//reducer script for sample hadoop job
$word2count = array();

// input comes from STDIN
while (($line = fgets(STDIN)) !== false) {
   // remove leading and trailing whitespace
   $line = trim($line);
   // parse the input we got from mapper.php
   list($word, $count) = explode("\t", $line);
   // convert count (currently a string) to int
   $count = intval($count);
   // sum counts
   if ($count > 0) $word2count[$word] += $count;
}

ksort($word2count);  // sort the words alphabetically

// write the results to STDOUT (standard output)
foreach ($word2count as $word => $count) {
   echo "$word\t$count\n";
}

?>
```

Showtime

Vamos a lanzar 2 ejecuciones reales de EMR: ● Con 1 Nodo ● Con 2 Nodos

Aplicaciones de Hadoop

● Detección de fraude bancario.
● Análisis de Marketing en redes sociales
    ○ Twitter genera 12 TB de información al día
● Análisis de patrones de compra
    ○ Wallmart utilizaba en 2012 30.000 millones de sensores RFID
● Reconocimiento de patrones de tráfico para el desarrollo urbano
● Previsión de averías en aviones ○ El airbus A380 genera 640 TB de información por vuelo
● Transformación de datos grandes
● Aplicación de algoritmos de reconocimiento facial ○ En 2012, facebook publicó que se suben 250 millones de fotos al día

Que no es hadoop

Una base de datos
La solución a todos los problemas
HDFS no es un sistema de archivos POSIX completo

¿Donde aplicaríais Hadoop?

● Procesamiento paralelo
● Sistema de ficheros distribuido
● Heterogeneidad de fuentes
● Tamaño de las fuentes
● Dimensionamiento de las infraestructuras

Casos de éxito - Análisis de Riesgos

Reto
Con la crisis de 2008, una importante entidad financiera quedó expuesta a la morosidad de los clientes, era vital mejorar los análisis de riesgo.
Solución
● Creación de un cluster único con Hadoop (Con petabytes de información)
● Cargo la información de todos los almacenes de datos de la entidad que disponían de una visión específica del cliente
● Cargo información no estructurada
    ○ Chats
    ○ Correos al servicio de atención al cliente
    ○ Registros de los Call Center
    ○ Otros orígenes
● Capacidad para realizar un análisis completo de la situación de los clientes

Casos de éxito - Fuga de clientes

Reto
Una importante compañía de telecomunicaciones necesitaba entender porque perdía clientes, para ello, necesitaba dar respuesta a las siguientes preguntas: ¿Eran clientes que se iban o simplemente estaban negociando las condiciones? ¿Se iban a la competencia? ¿Se iban por problemas en la cobertura? ¿Por los precios? ¿Por incidencias en los dispositivos? ¿Por otros motivos?
Solución
● Creación de un cluster único con Hadoop
● Se combinaron las fuentes transaccionales tradicionales y las redes sociales
● Analizaron los registros de llamadas y crearon una red de contactos de los clientes
● Cruzaron esta información con los contactos de las fugas en las redes sociales y concluyeron que cuando un cliente se iba de la compañía, sus contactos eran más proclives a abandonar también.

● Cruzaron los mapas de cobertura con la ubicación de los cliente y dimensionan el impacto de las incidencias de cobertura en la fuga de clientes
● Optimizaron la inversión en infraestructuras y el desarrollo de nuevos productos

Casos de éxito - Puntos de venta

Reto

Una importante empresa de Retail quería incorporar las nuevas fuentes de información disponibles en los análisis (Tiendas Online, Tiendas Offline, Redes Sociales...). Los sistemas tradicionales son muy caros para almacenar datos complejos.

Solución

● Creación de un cluster único con Hadoop
● Cargaron 20 años de transacciones
● Utilizaron Hive para realizar algunos de los análisis que ejecutaban realizaban en el almacén de datos, aunque extendiéndose a periodos mucho mayores.
● Redujeron los costes de infraestructura
● Incluyeron nuevos orígenes de datos (Canales de noticias, Redes Sociales, Twitter...)

Casos de éxito - Datos en bruto

Reto

Una importante agencia de viajes genera volúmenes de datos en bruto enormes que solo podían almacenar unos días por el coste del almacén de datos.

Solución

● Creación de un cluster único con Hadoop
● Almacenarón toda la información a un coste muy inferior al guardarlas en el formato original y comprimida
● Utilizaron Hive para analizar el comportamiento en la web de reservas.
● Consiguieron un mejor entendimiento de sus clientes y pudieron ofrecen mejores productos aumentando la rentabilidad.

CAPITUO 4 Arquitectura Lambda

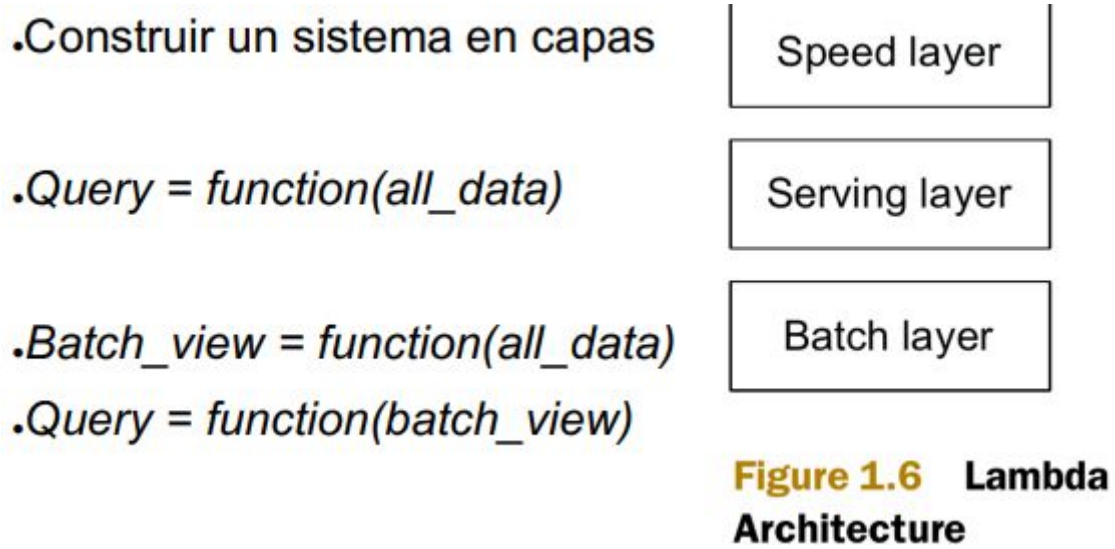


Figure 1.6 Lambda Architecture

●Número de visitas a una página web en un rango de fechas

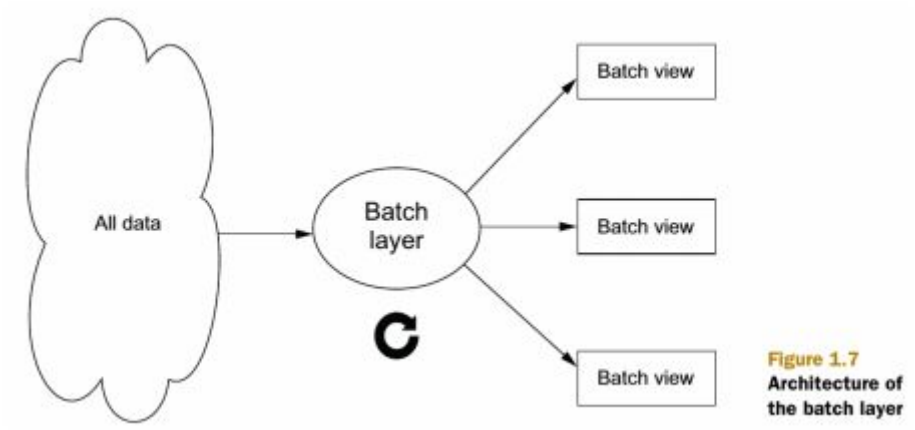


Figure 1.7 Architecture of the batch layer

Batch Layer

- Batch_view = function(all_data)
- Copia maestra del dataset (todos los datos)
- Precalcula las vistas arbitrarias sobre ese dataset

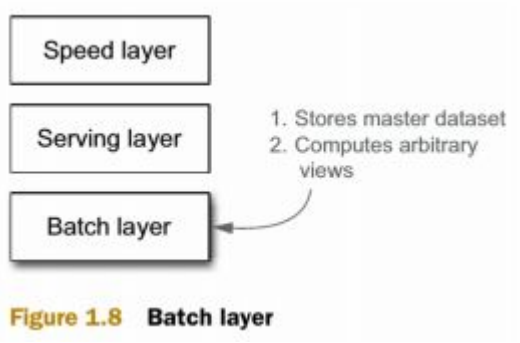


Figure 1.8 Batch layer

Serving Layer

- La capa Batch emite vistas como resultado
- El objetivo es almacenar esas vistas en algún lugar para que puedan ser consultadas
- Capa de servicios
    - Actualizaciones desde la capa batch.
    - Lecturas randómicas
- No necesita soportar escrituras randómicas
- La escrituras randómicas aumentan la complejidad
- Simplicidad: robusta, predecible, fácil de configurar y fácil de operar

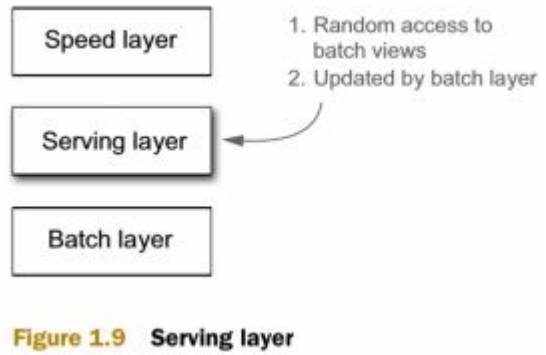


Figure 1.9 Serving layer

Batch and Serving Layer

- Dan soporte a queries arbitrarios sobre un cualquier dataset.
- Tienen un desface de unas cuantas horas

●Satisfacen todas las propiedades deseadas de un sistema de Big Data.
- ● Robustes y tolerancia a fallos:
  - ● Hadoop maneja la recuperación de errores cuando se cae algún servidor
  - ● La capa de servicios usa replicación para asegurar la disponibildad.
  - ● Permite corregir errores humanos ya que se pueden recalcular las vistas desde cero

●Satisfacen todas las propiedades deseadas de un sistema de Big Data.
- ● Escalabilidad:
  - ● Ambas se manejan con sistemas distribuidos fáciles de escalar tan solo añadiendo nuevos servidores al cluster.
- ● Generalización:
  - ● La arquitectura descrita permite calcular vistas arbirarias sobre cualquier dataset.

Batch and Serving Layer

●Satisfacen todas las propiedades deseadas de un sistema de Big Data.
- ● Extensibilidad:
  - ● Agregar una nueva vista es simplemente añadir una nueva función sobre el dataset maestro.
- ● Consultas Adhoc
  - ● Toda la data está disponible en una sola ubicación, haciendo sencillo ejecutar cualquier query sobre dicha data.

●Satisfacen todas las propiedades deseadas de un sistema de Big Data.
- ● Mantenimiento mínimo:
  - ● Hadoop requiere cierto conocimiento especializado pero es bastante simple de configurar y operar.
  - ● La capa de servicios al no requerir escrituras randómicas se vuelve mucho más simple de mantener.
- ● Lo único que falta es cómo manejar la baja latencia de los updates.

Speed Layer

●En esta capa se procesan los datos nuevos que ingresan al sistema cuando se están calculando las vistas.

●El procesamiento es en tiempo real.

●Realtime View = function(realtime_view, new data)

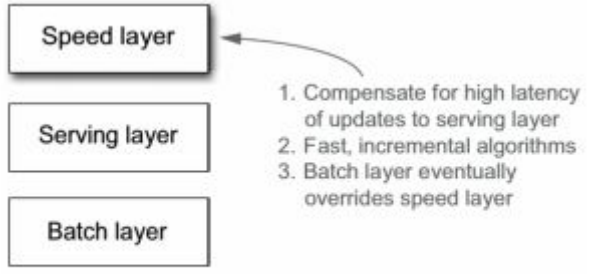


Figure 1.10 **Speed layer**

Sistema de Big Data

●Batch View = function(all data)

●Realtime View = function(realtime_view, new data)

●Query = function(batch view, realtime view)

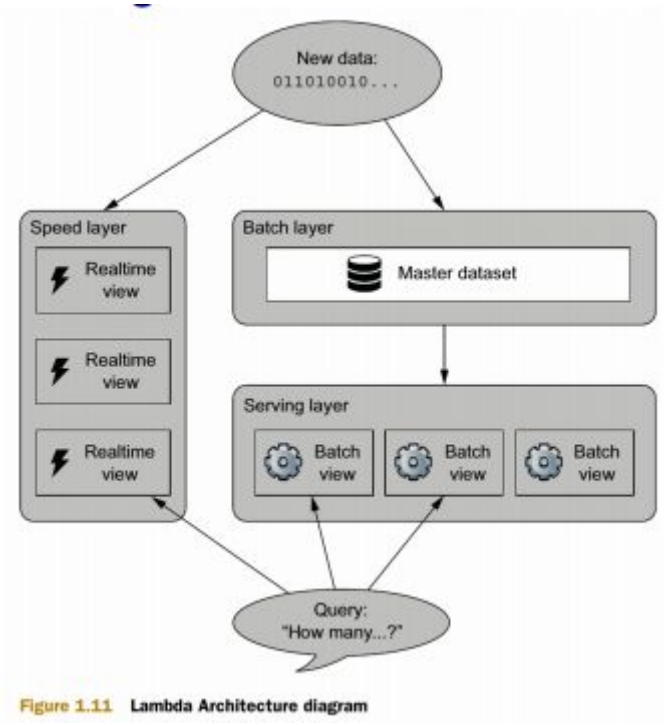


Figure 1.11 **Lambda Architecture diagram**

- Propiedades de los queries
  - Latencia: El tiempo que le toma ejecutarse
  - Timeliness: Que tan actualizados son los resultados de la consulta
  - Exactitud: Aproximaciones a los resultados de las consultas para ganar rendimiento o escalabilid

Detalles de la capa Batch

- Una buena métrica es cuánto tiempo toma actualizar las vistas
- Mientrás más tiempo le tome a la capa Batch precalcular las vistas
  - Más grande debe ser la capa en tiempo real.
  - Más tiempo toma la recuperación ante errores de programación (bugs)
- Procesamiento en Batch Incremental

Procesamiento en Batch Incremental

- Algoritmos Incrementales
- Algoritmos de re-cálculo
- Contar el número total de registros del master dataset

.Algoritmos de re-cálculo

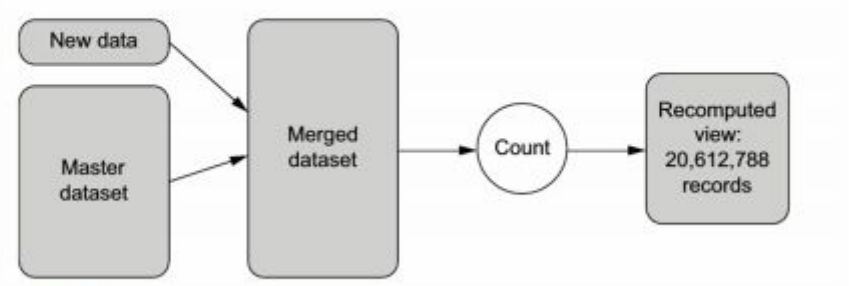


Figure 6.5 **A recomputing algorithm to update the number of records in the master dataset. New data is appended to the master dataset, and then all records are counted.**

.Algoritmos Incrementales

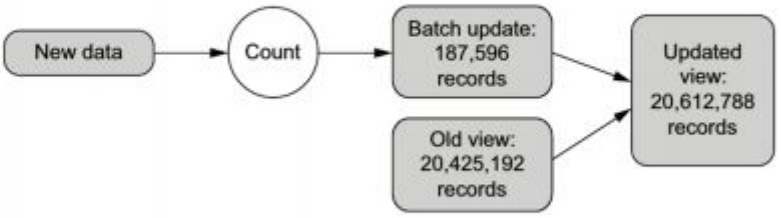


Figure 6.6 An incremental algorithm to update the number of records in the master dataset. Only the new dataset is counted, with the total used to update the batch view directly.

●Algoritmos Incrementales vs. re-cálculo

Table 6.1 Comparing recomputation and incremental algorithms

	Recomputation algorithms	Incremental algorithms
Performance	Requires computational effort to process the entire master dataset	Requires less computational resources but may generate much larger batch views
Human-fault tolerance	Extremely tolerant of human errors because the batch views are continually rebuilt	Doesn't facilitate repairing errors in the batch views; repairs are ad hoc and may require estimates
Generality	Complexity of the algorithm is addressed during precomputation, resulting in simple batch views and low-latency, on-the-fly processing	Requires special tailoring; may shift complexity to on-the-fly query processing
Conclusion	Essential to supporting a robust data-processing system	Can increase the efficiency of your system, but only as a supplement to recomputation algorithms

Medir y optimizar el rendimiento en la capa Batch

●Después de duplicar el tamaño del cluster la latencia bajó de 30 horas a 6 horas. (80% de ganancia)

●Después de reconfigurar mal un cluster de Hadoop, se tenía 10% más de fallas de los servidores. Esto incrementó la latencia de 8 horas a 72 horas.

●T = O + PH

T: es el tiempo de ejecución en horas

O – (Sobrecarga): es el tiempo independiente de los datos a procesar en horas. Configurar los procesos, copiar los datos en el cluster, etc.

H: es el número de horas de datos procesadas en esa iteración

P: El tiempo de procesamiento dinámico. El número de horas que cada hora de datos agrega al tiempo total. Si cada hora de datos agrega media hora al total, P¿0.5

$$T = O + PT$$

$$T = \frac{O}{(1-P)}$$

●H variará en cada iteración dependiendo si la iteración anterior tomo más o menos tiempo.

●Para determinar cuando el tiempo de procesamiento se estabiliza se debe considerar cuando el tiempo total (T) es igual al número de horas de datos que procesa (H)

●El tiempo total es directamente proporcional a la sobrecarga

●El tiempo total no es directamente proporcional al tiempo de procesamiento dinámico.

$$T = O + PT$$

$$T = \frac{O}{(1-P)}$$

●Qué pasa si P es mayor o igual que 1?

Cada iteración tendrá más datos que la anterior.

El procesamiento estará retrasado siempre.

●Al incrementar el tamaño del cluster al doble, P se reduce en aproximadamente la mita

$$T_1 = \frac{O}{(1-P)}$$

$$T_2 = \frac{O}{\left(1-\frac{P}{2}\right)}$$

●6 min, no se gana demasiado

●54 min, la ganancia es bastante considerable

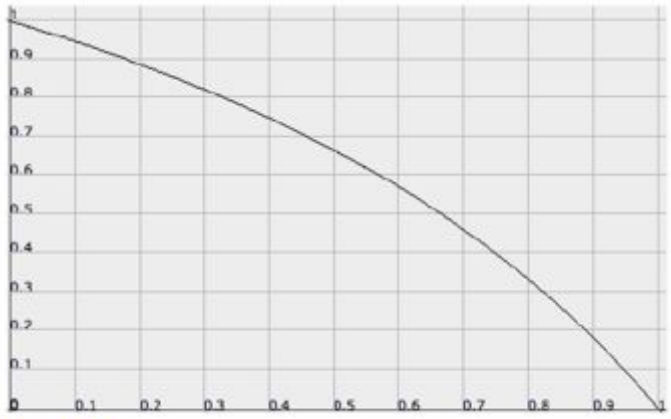


Figure 18.3 **Performance effect of doubling cluster size**

●10% de fallos

– Si se tienen 100 tareas 10 fallan y se tienen que reintentar

– De las 10 fallará 1, que también se reintentará

– P aumentará en un 11%

$$T_1 = \frac{O}{(1-P)}$$

$$T_2 = \frac{O}{(1-1.11P)}$$

●P debajo de 0.7

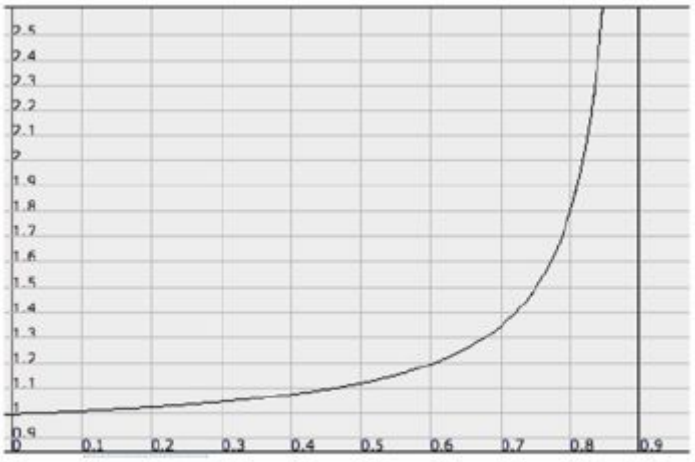


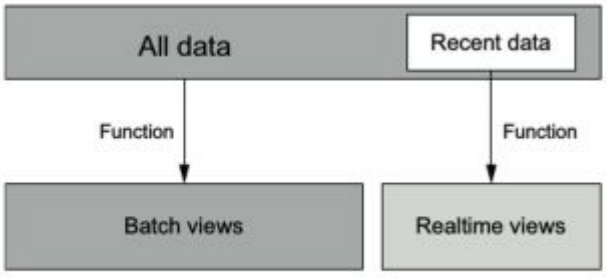
Figure 18.4 **Performance effect of 10% increase in error rates**

- Si P es mayor a 0.5 añadiendo 1% de máquinas reducirá la latencia en más de 1%
- Si P es menor a 0.5 añadiendo 1% de máquinas reducirá la latencia en menos de 1%

Figure 18.4 **Performance effect of 10% increase in error rates**

Detalles de la capa en tiempo real (Speed Layer)

- La capa de servicios se actualiza con una latencia alta.
  - Siempre está desactualizada unas horas
  - Las vistas representan la mayoría de nuestros datos
  - Los únicos datos faltantes son los que ingresaron después de la última actualización de las vistas
- Calcular queries en tiempo real para compensar esa pocas horas de datos.
  - La capa en tiempo real (Speed Layer)
- En esta capa es donde optamos por rendimiento
  - Algoritmos incrementales en lugar de recalculo
  - Bases de datos mutables (lecturas / escrituras)
- Se necesita latencia baja
- Los errores humanos no son un problema La capa de servicios sobreescribe esta capa



A simple strategy mirrors the batch/serving layer and computes the realtime views using all recent data as input.

Figure 12.2 **Strategy: realtime view = function(recent data)**

- Este esquema sirve si la aplicación acepta una latencia de unos cuantos minutos

- Usualmente la latencia en esta capa debe estar en el orden de los milisegundos

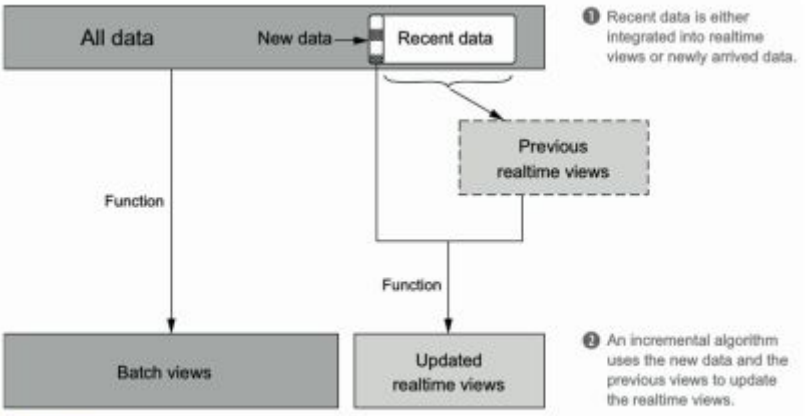


Figure 12.3 **Incremental strategy: realtime view = function(new data, previous realtime view)**

Capa de consultas

- Batch y Speed layer para responder consultas
- Que usar de cada capa y cómo unir todo para mostrar los resultados correctos