# Recuperació de la Informació (REIN)

## Grau en Enginyeria Informàtica

Departament de Ciències de la Computació (CS)


UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Escola Politècnica Superior d'Enginyeria
de Vilanova i la Geltrú

# Índex

# Índex

# Google 1998, algunes xifres

- 24 milions de pàgines
- 259 milions d'enllaços
- 147 Gb de text
- 256 Mb de memòria principal per màquina
- 14 milions de termes al diccionari
- 3 rastrejadors (*crawlers*), 300 connexions per rastrejador
- 100 pàgines web rastrejades / segon, 600 Kb/segon
- 41 Gb d'índex invertit
- 55 Gb info per respondre consultes; 7Gb si l'índex de docs. està comprimit

# Google avui?

- Dades actuals = de $\times$ 1 000 a $\times$ 10 000

- 100s petabytes transferits per dia
- 100s exabytes d'emmagatzemament
- Desenes de còpies de la web accessible
- Milions de màquines

# Google el 2003

- Tenia més aplicacions, no només cerques a internet
- Moltes màquines, molts centres de dades, molts programadors
- Quantitats enormes i complexes de dades
- Necessitat de més capes d'abstracció

# Google el 2003

- Tenia més aplicacions, no només cerques a internet
- Moltes màquines, molts centres de dades, molts programadors
- Quantitats enormes i complexes de dades
- Necessitat de més capes d'abstracció

Tres propostes influents:

1. Abstracció de hardware: The Google Cluster
2. Abstracció de dades: The Google File System

   *BigFile* (2003), *BigTable* (2006)
3. Model de programació: *MapReduce*

# El clúster de Google, 2003: criteris de disseny

Usar més màquines barates en lloc de servidors cars

# El clúster de Google, 2003: criteris de disseny

## Usar més màquines barates en lloc de servidors cars

- Elevat paral·lelisme de tasques; baix paral·lelisme d'instruccions

  (p.e., processar llistes de *postings*, resumir documents)
- El rendiment màxim del processador és menys important que la relació preu/rendiment

# El clúster de Google, 2003: criteris de disseny

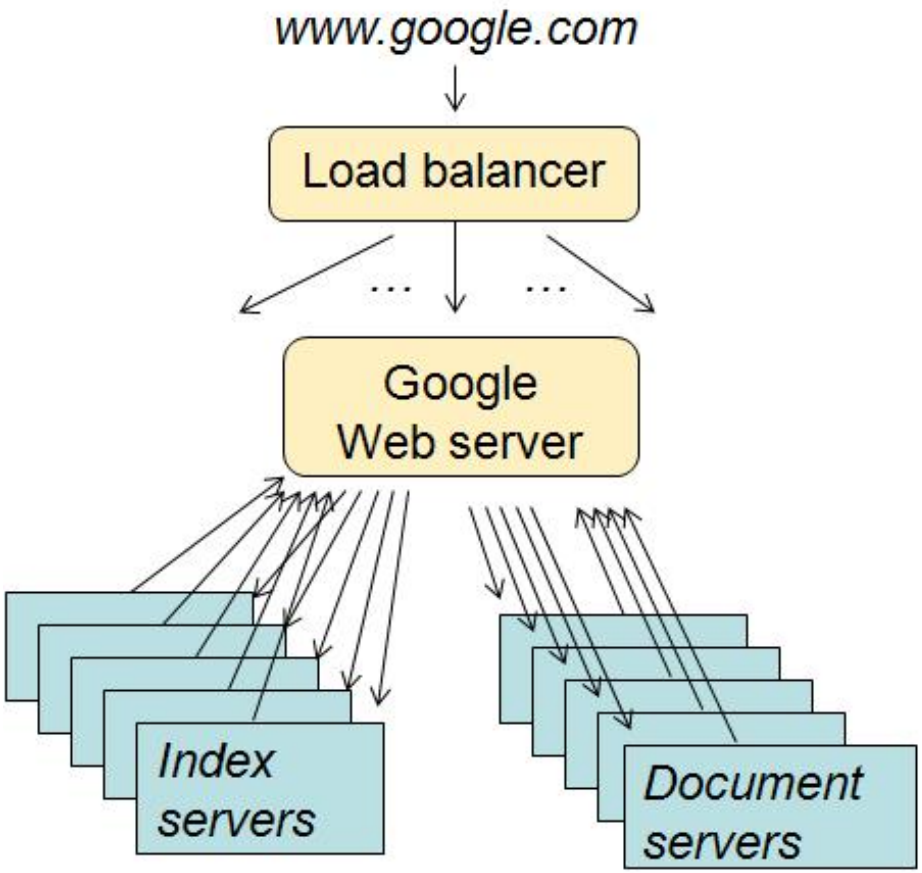
## Usar més màquines barates en lloc de servidors cars

- Elevat paral·lelisme de tasques; baix paral·lelisme d'instruccions

  (p.e., processar llistes de *postings*, resumir documents)
- El rendiment màxim del processador és menys important que la relació preu/rendiment
- PC estàndard, barats i fàcil de fer redundants
- Redundància per obtenir alt rendiment i alta disponibilitat
- Fiabilitat gràcies a la redundància (gestionada per soft)
- No obstant, són de curta durada (< 3 anys)

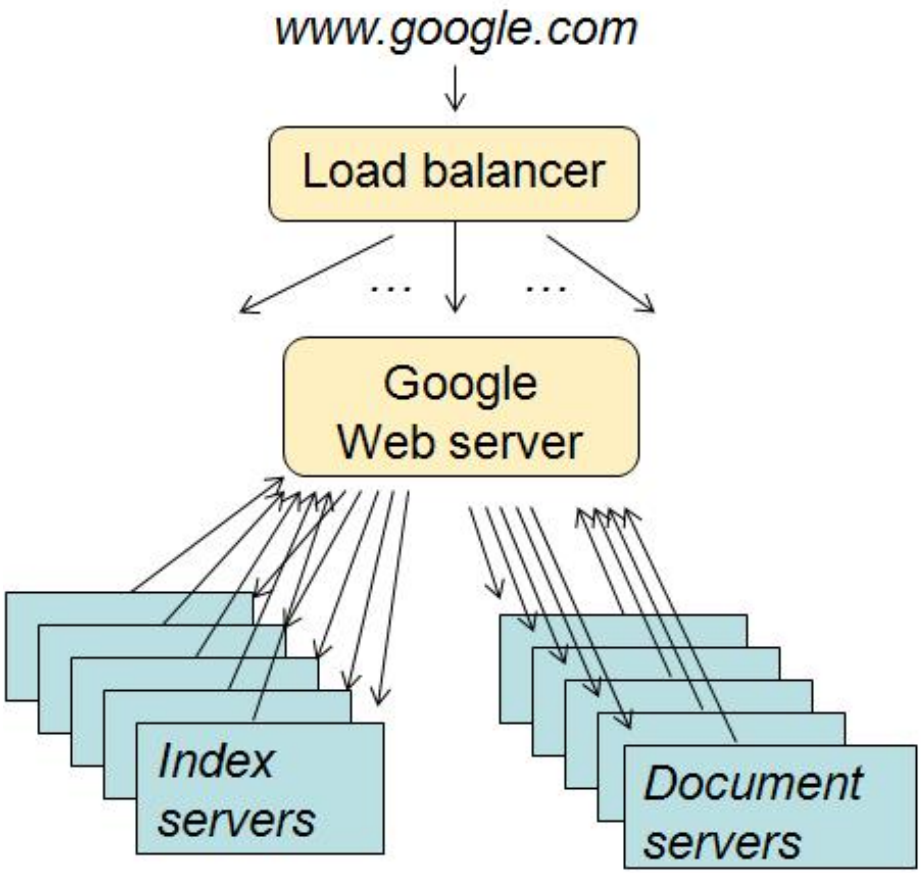
L.A. Barroso, J. Dean, U. Hölzle: "Web Search for a Planet: The Google Cluster Architecture", 2003

# El clúster de Google per fer cerques a internet



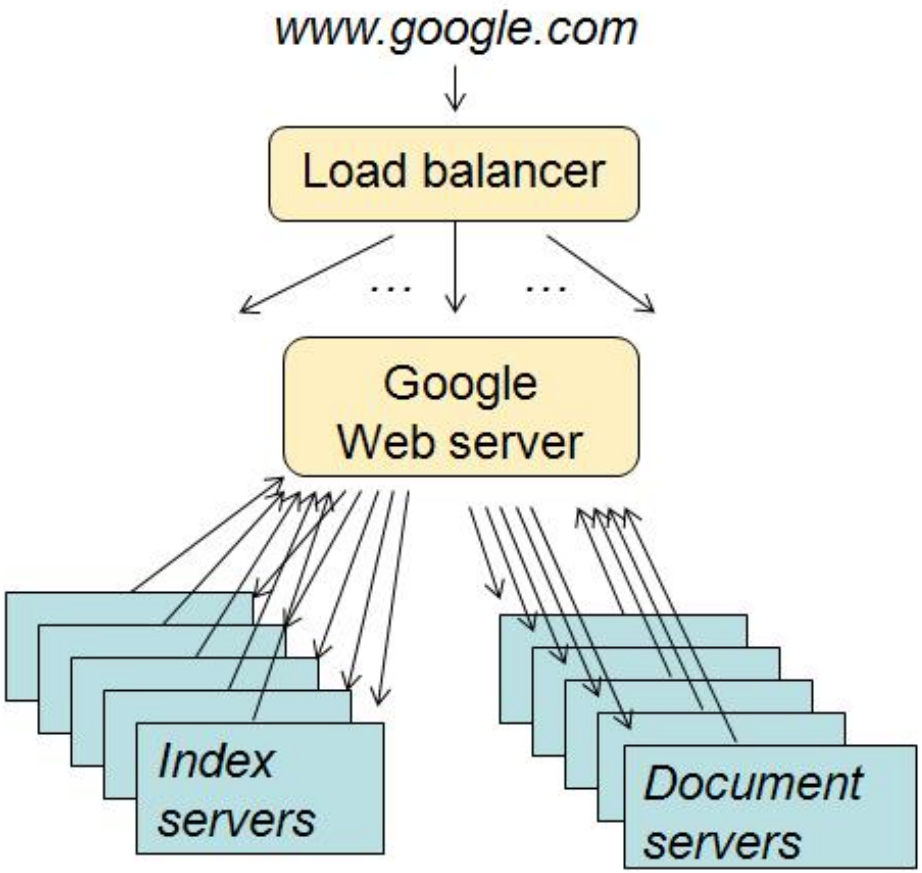

- El balancejador de càrrega tria els GWS més lliures / propers

# El clúster de Google per fer cerques a internet



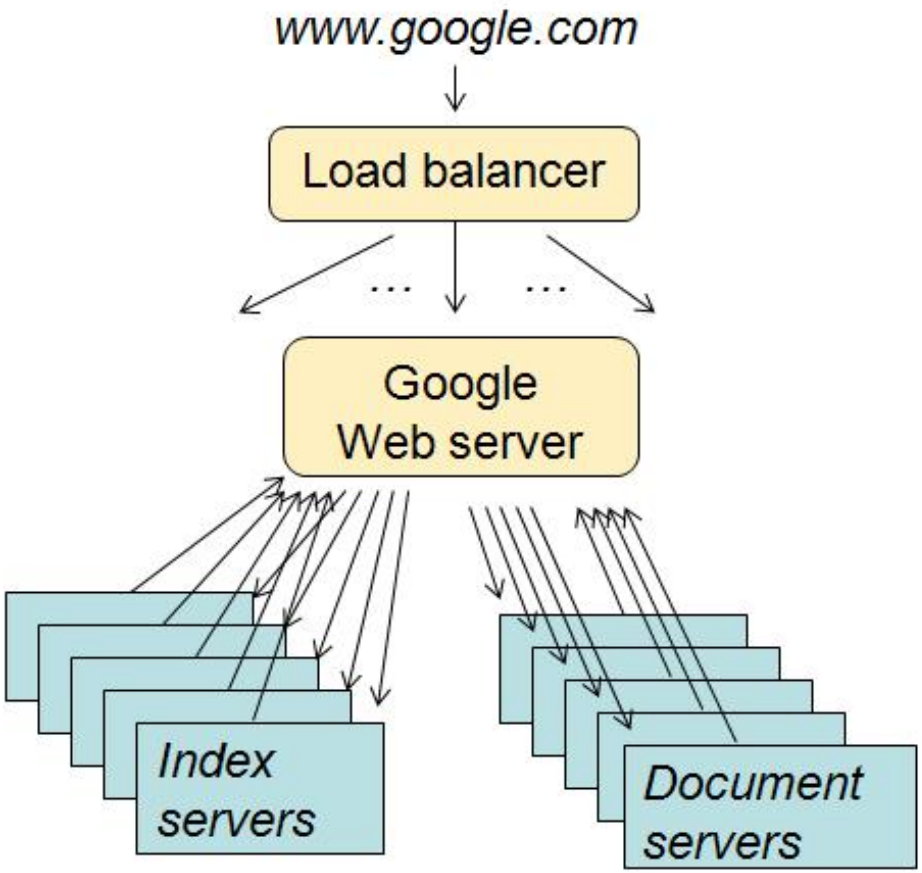

- El balancejador de càrrega tria els GWS més lliures / propers
- El GWS fa la consulta a diversos servidors d'índexs

# El clúster de Google per fer cerques a internet



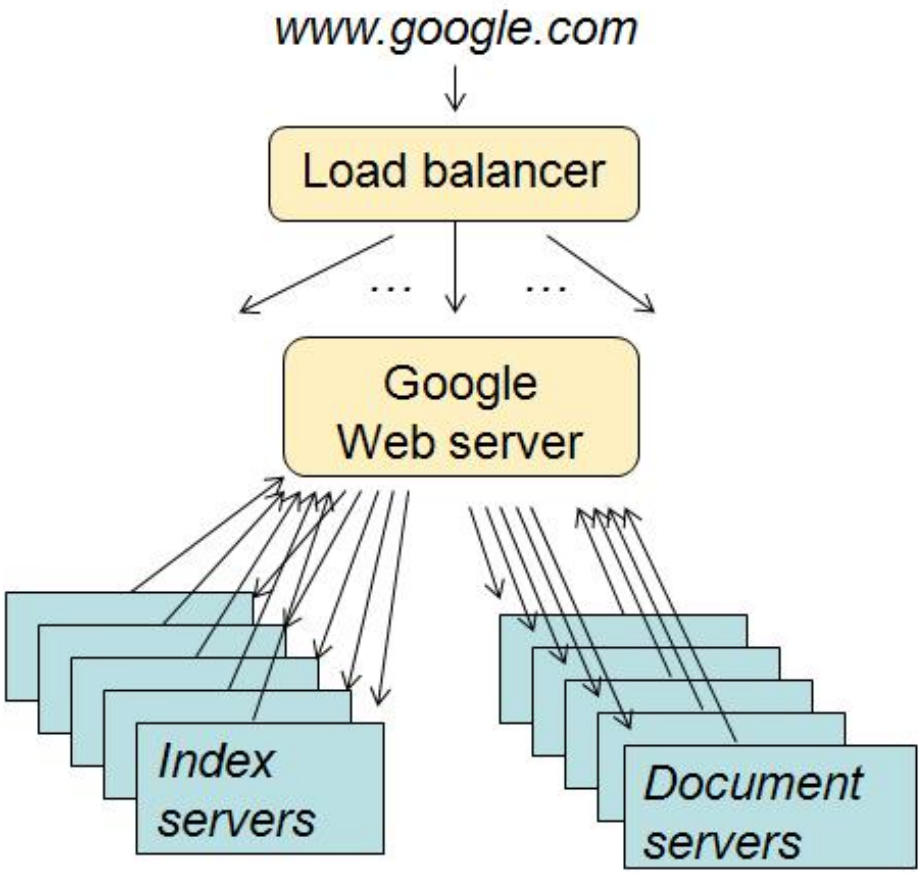

- El balancejador de càrrega tria els GWS més lliures / propers
- El GWS fa la consulta a diversos servidors d'índexs
- Aquests calculen les *hit lists* pels termes de la consulta, fan la intersecció i els ordenen per rellevància

# El clúster de Google per fer cerques a internet



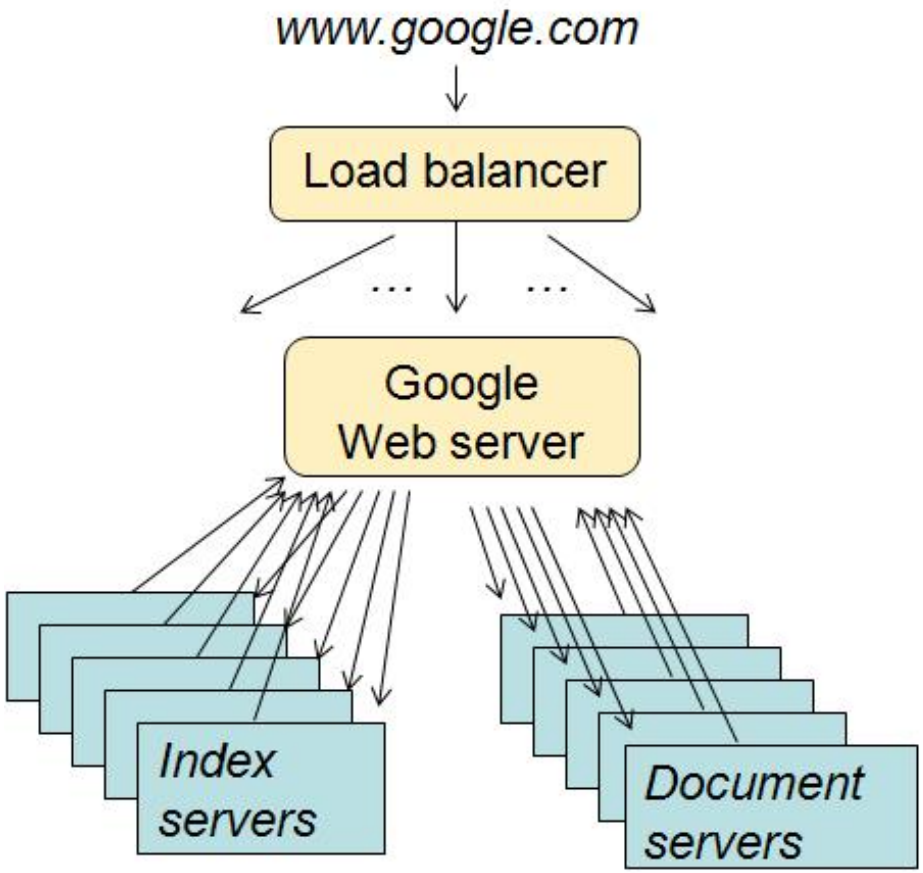

- El balancejador de càrrega tria els GWS més lliures / propers
- El GWS fa la consulta a diversos servidors d'índexs
- Aquests calculen les *hit lists* pels termes de la consulta, fan la intersecció i els ordenen per rellevància
- La resposta (llista de *docid*) és retornada al GWS

# El clúster de Google per fer cerques a internet



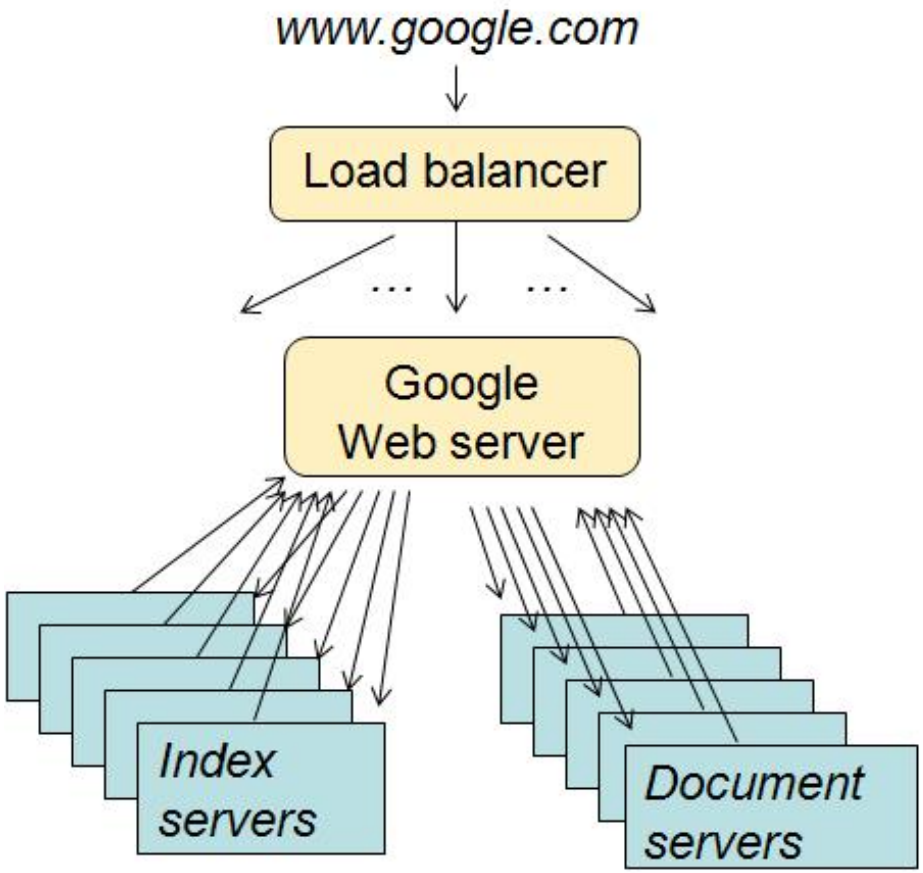

- El balancejador de càrrega tria els GWS més lliures / propers
- El GWS fa la consulta a diversos servidors d'índexs
- Aquests calculen les *hit lists* pels termes de la consulta, fan la intersecció i els ordenen per rellevància
- La resposta (llista de *docid*) és retornada al GWS
- El GWS els demana a diversos servidors de documents

# El clúster de Google per fer cerques a internet




- El balancejador de càrrega tria els GWS més lliures / propers
- El GWS fa la consulta a diversos servidors d'índexs
- Aquests calculen les *hit lists* pels termes de la consulta, fan la intersecció i els ordenen per rellevància
- La resposta (llista de *docid*) és retornada al GWS
- El GWS els demana a diversos servidors de documents
- Ells donen títol de la pàgina, url, text rellevant de la consulta en el doc., etc.

# El clúster de Google per fer cerques a internet




- El balancejador de càrrega tria els GWS més lliures / propers
- El GWS fa la consulta a diversos servidors d'índexs
- Aquests calculen les *hit lists* pels termes de la consulta, fan la intersecció i els ordenen per rellevància
- La resposta (llista de *docid*) és retornada al GWS
- El GWS els demana a diversos servidors de documents
- Ells donen títol de la pàgina, url, text rellevant de la consulta en el doc., etc.
- El GWS crea una pàgina html i la retorna a l'usuari

# Els *index shards* o fragments d'índex

Objectiu: paral·lelitzar la cerca

- Un *index shard* conté un subconjunt aleatori de documents de l'índex complet.
- De cada *index shard* hi ha diverses rèpliques (servidors d'índexs).
- Les consultes es redirigeixen a través del balancejador de càrrega local.
- Millora en velocitat i en tolerància a fallades.

# Els *index shards* o fragments d'índex

Objectiu: paral·lelitzar la cerca

- Un *index shard* conté un subconjunt aleatori de documents de l'índex complet.
- De cada *index shard* hi ha diverses rèpliques (servidors d'índexs).
- Les consultes es redirigeixen a través del balancejador de càrrega local.
- Millora en velocitat i en tolerància a fallades.
- Les actualitzacions són poc freqüents, a diferència de les BD tradicionals.
- Un servidor es pot desconnectar temporalment mentre fa l'actualització.

# El sistema de fitxers de Google (*GFS*), 2003

- Sistema format per PC estàndard barats que fallen sovint.
- Cal una monitorització constant i recuperar les fallades de forma transparent i rutinària.
  - s'han de complir els requirements rendiment, escalabilitat, fiabilitat i disponibilitat

# El sistema de fitxers de Google (*GFS*), 2003

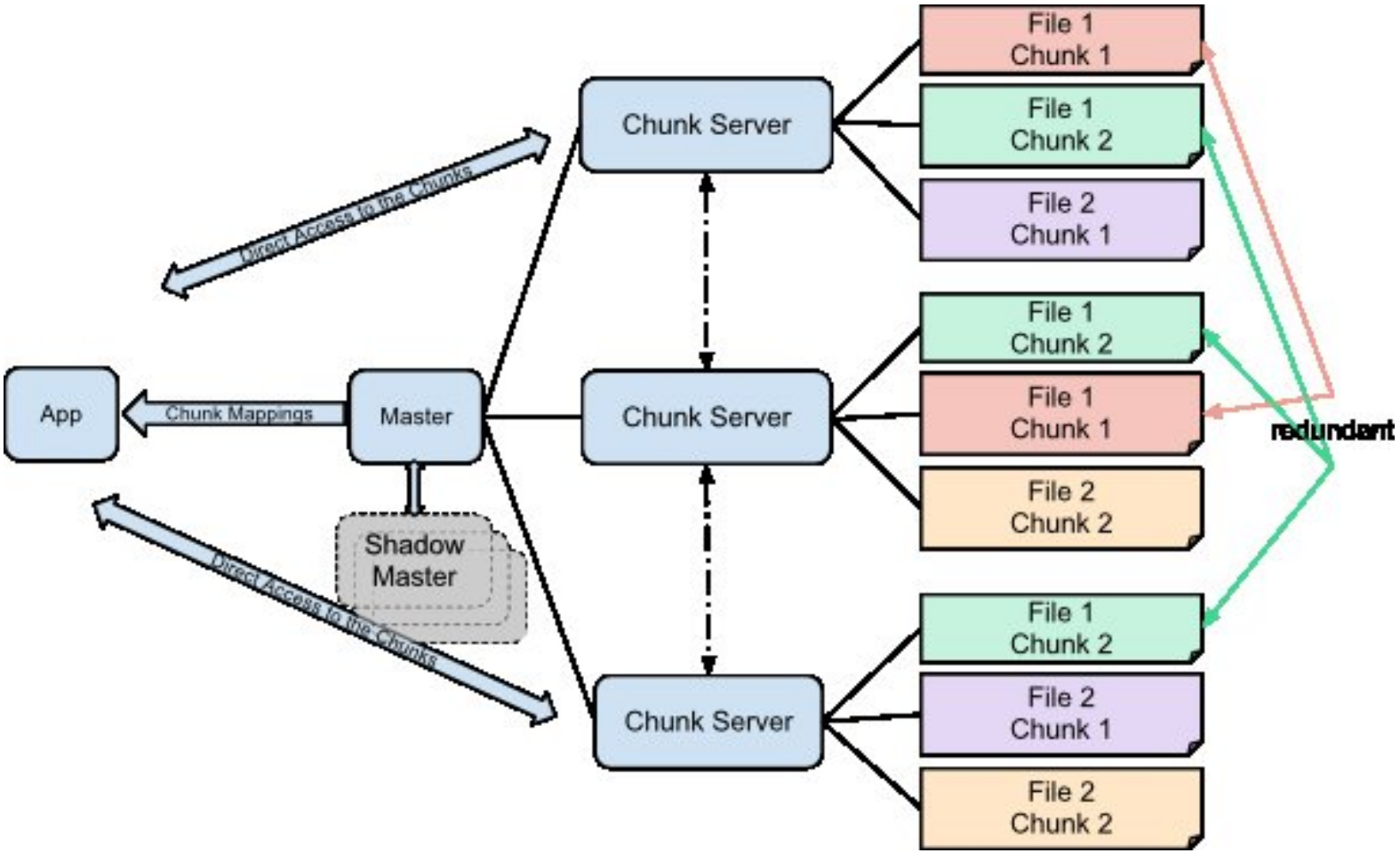
- Sistema format per PC estàndard barats que fallen sovint.
- Cal una monitorització constant i recuperar les fallades de forma transparent i rutinària.
  - s'han de complir els requirements rendiment, escalabilitat, fiabilitat i disponibilitat
- Nombre moderat de fitxers grans (milions de fitxers de 100MB o més).
- Tracta fitxers petits però no està optimitzat per això.
- Barreja lectures llargues (*streaming*) amb lectures curtes aleatòries.
- De tant en tant, llargues escriptures continuades per afegir dades als fitxers.
- Concurrència molt elevada (en els mateixos fitxers).

S. Ghemawat, H. Gobioff, Sh.-T. Leung: "The Google File System", 2003

# El sistema de fitxers de Google (*GFS*), 2003

- Un clúster GFS = 1 procés *master* + diversos *chunkservers* i és accedit per múltiples clients.
- Els fitxers estan fragmentats en (*chunks*) de mida fixa. Cada *chunk* té un identificador únic de 64 bits.
- Cada *chunk* és replicat (en diferents *racks*, per seguretat).
- El *master* coneix el mapeig entre *chunks* $\rightarrow$ *chunkservers*
- El *master* no serveix dades: dirigeix als clients cap al *chunkserver* correcte.
- Els *chunkservers* no tenen estat propi; repliquen l'estat del *master*.
- Nucli de l'algorisme: detectar i deixar de banda els *chunkservers* que fallin.

# El sistema de fitxers de Google (*GFS*), 2003




(Font: Wikipedia)

# Índex

# *MapReduce* i *Hadoop*

- *MapReduce*: Model de programació paral·lela per treballar en entorns de clústers grans. Desenvolupat per Google (2004).
  - Implementació propietària
  - Implementa idees antigues de la programació funcional, sistemes distribuïts, BD...

# *MapReduce* i *Hadoop*

- *MapReduce*: Model de programació paral·lela per treballar en entorns de clústers grans. Desenvolupat per Google (2004).
  - Implementació propietària
  - Implementa idees antigues de la programació funcional, sistemes distribuïts, BD...


- *Hadoop*: Implementació més popular del paradigma MapReduce.
  - Desenvolupada a Yahoo (2006 i posterior)
  - Implementació de codi obert (projecte Apache)
  - Components bàsics:
    - HDFS *Hadoop Distributed File System*: codi obert; anàleg a GFS
    - MapReduce Software Framework
  - L'ecosistema Hadoop conté altres components
    - Pig: Llenguatge tipus script de Yahoo! per a les tasques d'anàlisi de dades a Hadoop
    - Hive: Llenguatge tipus SQL / emmagatzematge de dades a Hadoop
    - HBase, Flume, OOzie, Sqoop, Mahout...

Objectius de disseny:

- Escalabilitat a grans volums de dades i a gran quantitat de màquines.
  - 1 000 (milers de) màquines, 10 000 (desenes de milers de) discs
  - Abstracció de hardware i de distribució de les dades
- Cost-eficiència:
  - PC estàndard (barats però poc fiables)
  - Xarxa senzilla (amplada de banda petita)
  - Tolerància a fallades i sintonització automàtica (pocs administradors)
  - Fàcil d'usar (pocs programadors)

# *Hadoop Distributed File System (HDFS)*

- Optimitzat per arxius de grans dimensions, lectures seqüencials llargues
- Optimitzat per “escriu un cop, llegeix molts”
- Blocs grans (64MB). Pocs accessos (*seek*), transferències llargues
- Gestiona les rèpliques i les fallades
- Ús de *racks* (per localitat, per rèpliques tolerants a fallades)
- Tipus propis (`IntWritable`, `LongWritable`, `Text`, ...)
  - Serialitzat per transmissions a la xarxa i per les operacions entre llenguatge i sistema

# El model de programació *MapReduce*

- Tipus de dades: parells (clau, valor)
- El programador especifica dues funcions:

# El model de programació *MapReduce*

- Tipus de dades: parells (clau, valor)
- El programador especifica dues funcions:
- Funció `Map`:

$$(K_{ini}, V_{ini}) \rightarrow \text{list}\langle(K_{inter}, V_{inter})\rangle$$

  - Processa un parell (clau, valor) d'entrada
  - Produeix un conjunt de parells intermedis

# El model de programació *MapReduce*

- Tipus de dades: parells (clau, valor)
- El programador especifica dues funcions:
- Funció `Map`:

$$(K_{ini}, V_{ini}) \rightarrow \text{list}\langle(K_{inter}, V_{inter})\rangle$$

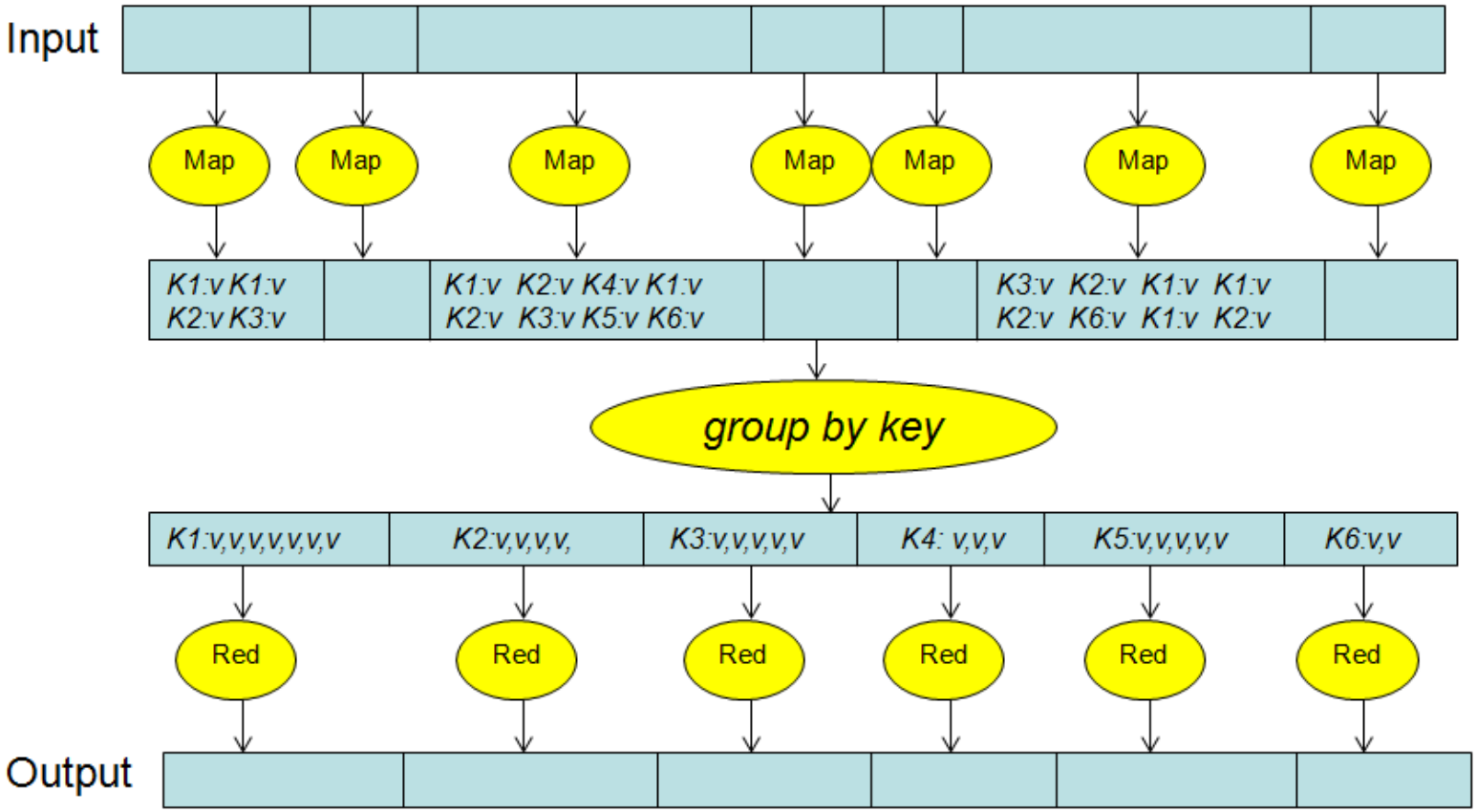
  - Processa un parell (clau, valor) d'entrada
  - Produeix un conjunt de parells intermedis
- Funció `Reduce`:

$$(K_{inter}, \text{list}\langle V_{inter}\rangle) \rightarrow \text{list}\langle(K_{out}, V_{out})\rangle$$

  - Combina tots els valors intermedis per una determinada clau
  - Produeix un conjunt de valors fusionats de sortida (normalment, només 1)

# Semàntica

Pas clau, gestionat per la plataforma: group by o shuffle per clau



# Exemple 1: Comptador de paraules, I

Entrada: Un fitxer enorme amb moltes línies de text.

Sortida: Per cada paraula, indicar quantes vegades apareix en el fitxer.

# Exemple 1: Comptador de paraules, I

Entrada: Un fitxer enorme amb moltes línies de text.

Sortida: Per cada paraula, indicar quantes vegades apareix en el fitxer.

```
map(docid a, doc d):
    for word in d:
        output (word, 1)
```

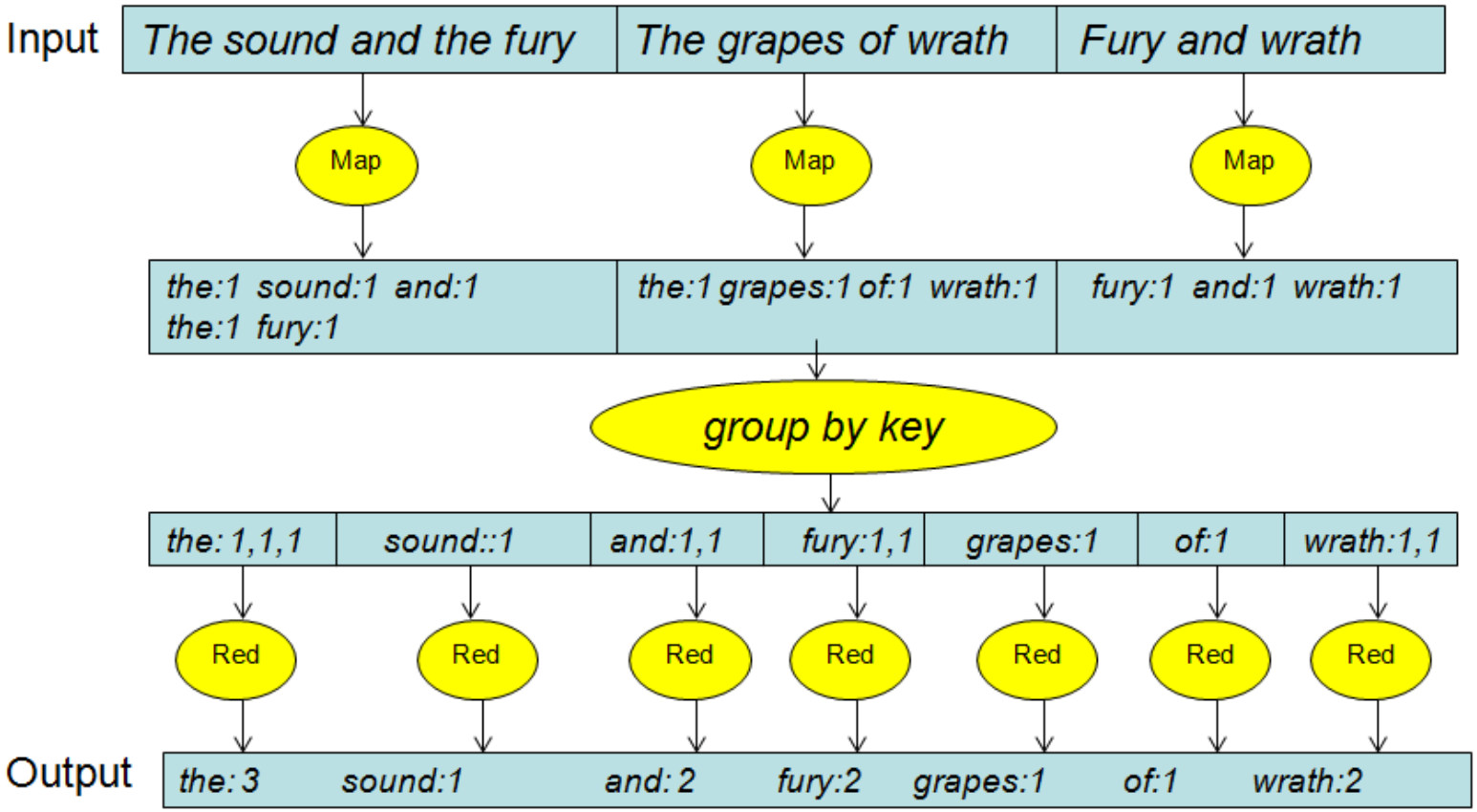
# Exemple 1: Comptador de paraules, I

Entrada: Un fitxer enorme amb moltes línies de text.

Sortida: Per cada paraula, indicar quantes vegades apareix en el fitxer.

```
map(docid a, doc d):
    for word in d:
        output (word, 1)

reduce(word w, list L):
    output (w, sum(L))
```

# Exemple 1: Comptador de paraules, II



# Exemple 2: Estadístiques de temperatures

Entrada: Conjunt de fitxers amb registres `(hora, lloc, temperatura)`.

Sortida: Per cada lloc, donar temperatures màxima, mínima i mitjana.

# Exemple 2: Estadístiques de temperatures

Entrada: Conjunt de fitxers amb registres `(hora, lloc, temperatura)`.

Sortida: Per cada lloc, donar temperatures màxima, mínima i mitjana.

```
map(docid a, file b):
    for record (hora, lloc, temp) in b:
        output (lloc, temp)
```

# Exemple 2: Estadístiques de temperatures

Entrada: Conjunt de fitxers amb registres `(hora, lloc, temperatura)`.

Sortida: Per cada lloc, donar temperatures màxima, mínima i mitjana.

```
map(docid a, file b):
    for record (hora, lloc, temp) in b:
        output (lloc, temp)

reduce(lloc l, list L):
    output (l,(max(L), min(L), sum(L)/length(L)))
```

# Exemple 3: Integració numèrica

Entrada: Registres $(start, end)$ pels intervals a integrar.

Sortida: Una aproximació de la integral de $f$ en l'interval $[a, b]$.

# Exemple 3: Integració numèrica

Entrada: Registres $(start, end)$ pels intervals a integrar.

Sortida: Una aproximació de la integral de $f$ en l'interval $[a, b]$.

```
map(start, end):
    sum = 0;
    for (x = start; x < end; x += step)
        sum += f(x)*step;
    output (0, sum)

reduce(key, L) :
    output (0, sum(L))
```

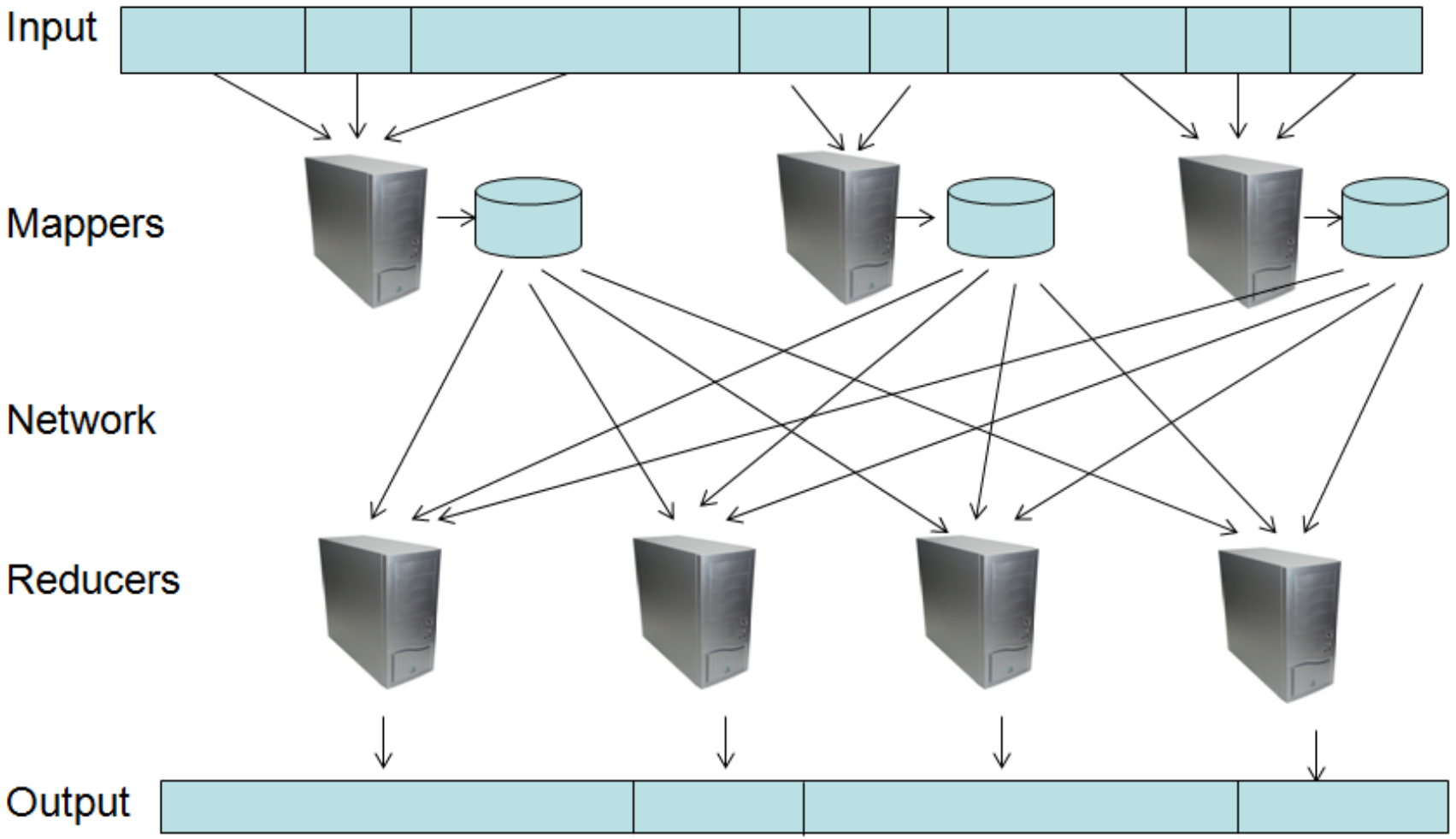
# Implementació

- Algunes màquines *mapper*, algunes *reducer*
- Instàncies de la funció *map* distribuïdes als *mapper*
- Instàncies de la funció *reduce* distribuïdes als *reducer*
- La plataforma s'encarrega de distribuir-ho per la xarxa
- Fa un balanç dinàmic de la càrrega

# Implementació

- Algunes màquines *mapper*, algunes *reducer*
- Instàncies de la funció *map* distribuïdes als *mapper*
- Instàncies de la funció *reduce* distribuïdes als *reducer*
- La plataforma s'encarrega de distribuir-ho per la xarxa
- Fa un balanç dinàmic de la càrrega
- Els *mappers* escriuen la seva sortida al disc local (no al HDFS)
- Si una instància de *map* o *reduce* falla, és automàticament re-executada
- La informació ha d'enviar-se de forma comprimida

# Implementació



# Una optimització: el *Combiner*

- `map` torna parells `(clau, valor)`
  sovint, torna més d'un parell amb la mateixa clau
- `reduce` rep un parell `(clau, llista-de-valors)`

# Una optimització: el *Combiner*

- `map` torna parells `(clau, valor)`
  sovint, torna més d'un parell amb la mateixa clau
- `reduce` rep un parell `(clau, llista-de-valors)`
- `combiner(clau, llista-de-valors)` s'aplica a la sortida d'un *mapper*, **abans** d'agrupar
- pot ajudar a enviar molta menys informació
- només funciona si és associativa i commutativa (p.e., sum, max, count)

# Exemple 1: Comptador de paraules revisat, I

```
map(docid a, doc d):
    for word in d:
        output (word, 1)
```

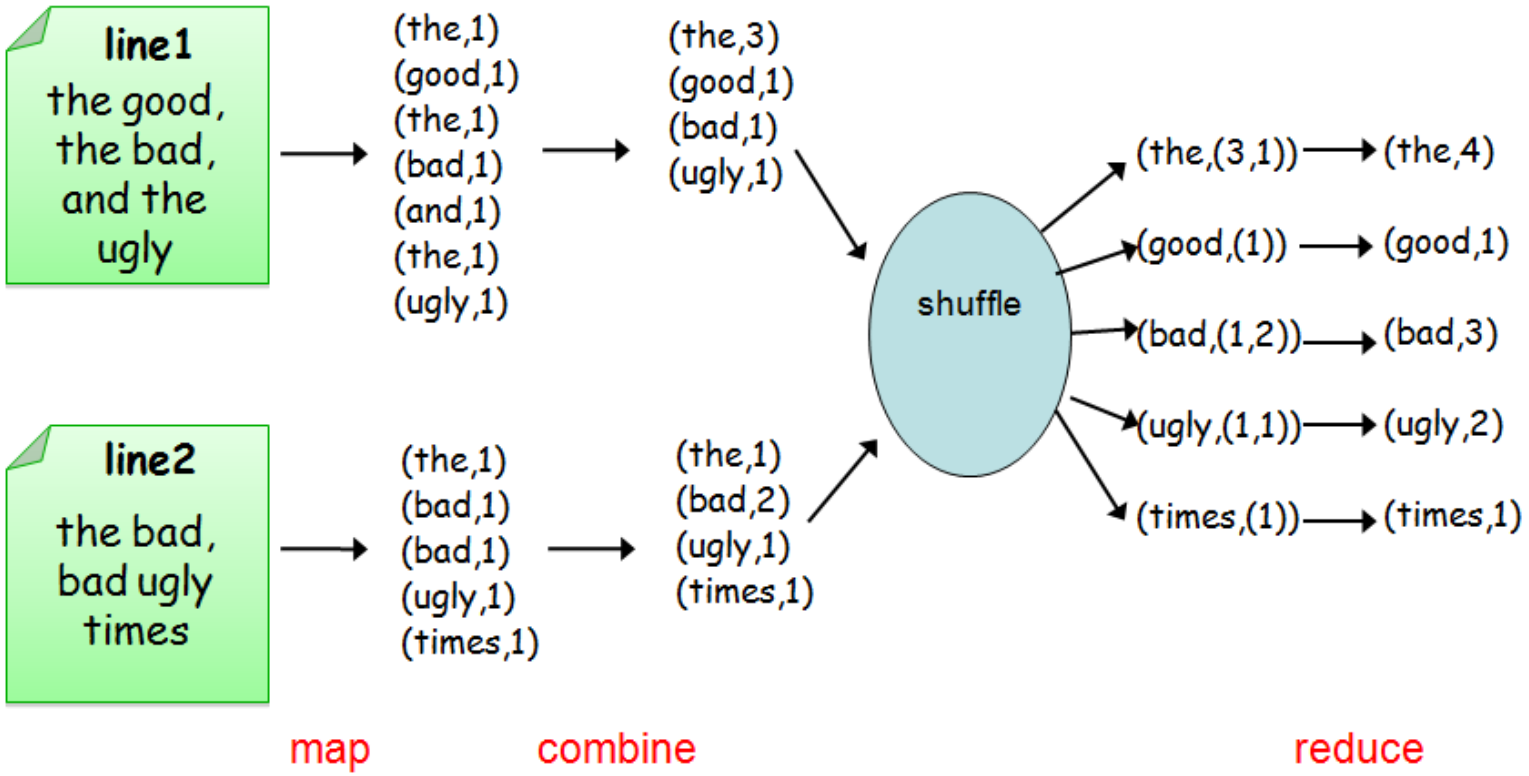
# Exemple 1: Comptador de paraules revisat, I

```
map(docid a, doc d):
    for word in d:
        output (word, 1)

combine(word w, list L): // L només conté valors 1
    output (w, sum(L))
```

# Exemple 1: Comptador de paraules revisat, I

```
map(docid a, doc d):
    for word in d:
        output (word, 1)

combine(word w, list L): // L només conté valors 1
    output (w, sum(L))

reduce(word w, L):
    output (w, sum(L))
```

# Exemple 1: Comptador de paraules revisat, II



# Exemple 4: Índex invertit

Entrada: Un conjunt de fitxers de text.

Sortida: Per cada paraula, la llista de fitxers que la contenen.

# Exemple 4: Índex invertit

Entrada: Un conjunt de fitxers de text.

Sortida: Per cada paraula, la llista de fitxers que la contenen.

```
map(docid a, doc d):
    for word in d:
        output (word, a)
```

# Exemple 4: Índex invertit

Entrada: Un conjunt de fitxers de text.

Sortida: Per cada paraula, la llista de fitxers que la contenen.

```
map(docid a, doc d):
    for word in d:
        output (word, a)

combine(word w, list L):
    remove duplicates in L;
    output (w, L)
```

# Exemple 4: Índex invertit

Entrada: Un conjunt de fitxers de text.

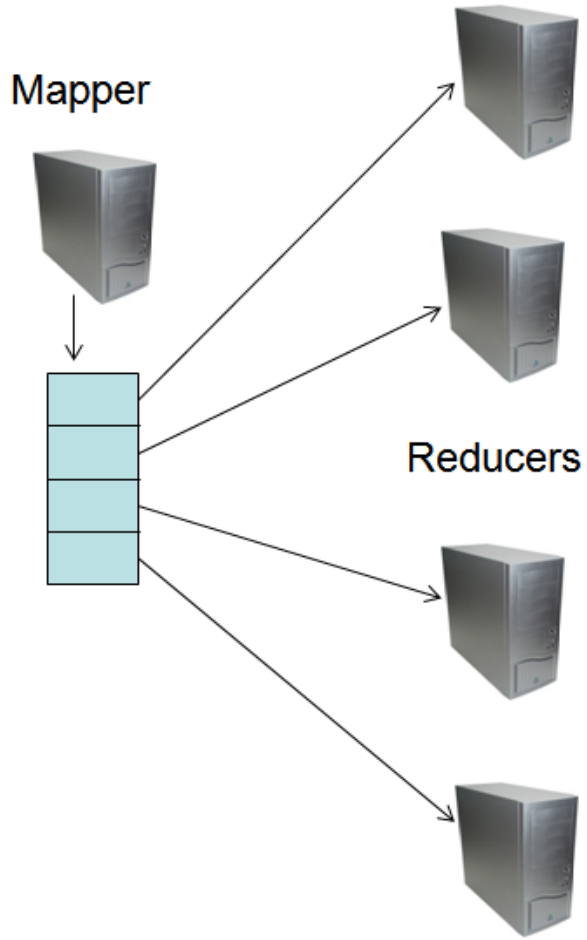
Sortida: Per cada paraula, la llista de fitxers que la contenen.

```
map(docid a, doc d):
    for word in d:
        output (word, a)

combine(word w, list L):
    remove duplicates in L;
    output (w, L)

reduce(word w, list L):
    #volem posting lists ordenades
    output (w, sort(L))
```

També es poden mantenir parells `(a, frequency)`.

# Més sobre la implementació, I

- Un *mapper* escriu al disc local
- Fa tantes particions com *reducers*
- Les claus són distribuïdes en les particions per mitjà de la funció `Partition`
- Per defecte, una funció *hash*
- També pot ser definida per l'usuari



# Exemple 5: Ordenació

Entrada: Un conjunt $S$ d'elements del tipus $T$ amb la relació $<$.

Sortida: El conjunt $S$, ordenat.

1. `map(x): output x`
2. `Partition:` qualsevol tal que `k` $<$ `k'` $\rightarrow$ `Partition(k) < Partition(k')`
3. Ara, cada *reducer* rep un interval de $T$ d'acord amb $<$ (p.e., 'A'..'F', 'G'..'M', 'N'..'S','T'..'Z')
4. Cada *reducer* ordena la seva llista

Nota: De fet, Hadoop ja garanteix que la llista enviada a cada *reducer* està ordenada per clau, per tant, el pas 4 potser no és necessari.

# Més sobre la implementació, II

- Un usuari envia un `job` o un seqüència de `job`.
- L'usuari envia les classes que implementen les operacions `map`, `reduce`, `combiner`, `partitioner`...
- ...a més de diversos fitxers de configuració (màquines i rols, clústers, sistema de fitxers, permisos, etc)
- Es parteix l'entrada en particions de la mateixa mida, una per *mapper*.
- En l'execució d'un `job` hi intervenen un procés *jobtracker* i diversos processos *tasktracker*.
- El *jobtracker* s'encarrega de gestionar tot el `job`.
- Els *tasktrackers* executen instàncies o bé de `map` o bé de `reduce`.
- L'operació *map* s'executa sobre cada un dels registres de cada partició.
- El nombre de *reducers* l'especifica l'usuari.

# Més sobre la implementació, III

```
public class C {

  static class CMapper
     extends Mapper<KeyType,ValueType> {
     ....
     public void map(KeyType k, ValueType v, Context context) {
         .... code of map function ...
           ... context.write(k',v');
    }

  static class CReducer
    extends Reducer<KeyType,ValueType> {
    ....
    public void reduce(KeyType k, Iterable<ValueType> values,
                Context context) {
         .... code of reduce function ...
           .... context.write(k',v');
    }
  }

}
```

# Exemple 6: Entropia d'una distribució

Entrada: Un *multiset* $S$ (conjunt on els elements poden estar repetits).

Sortida: L'entropia de $S$

$$H(S) = \sum_i -p_i \log(p_i), \text{ on } p_i = \#(S,i)/\#S$$

# Exemple 6: Entropia d'una distribució

Entrada: Un *multiset* $S$ (conjunt on els elements poden estar repetits).

Sortida: L'entropia de $S$

$$H(S) = \sum_i -p_i \log(p_i), \text{ on } p_i = \#(S,i)/\#S$$

Job 1: Per cada $i$, calcular $p_i$:

- `map(i): output (i, 1)`
- `combiner(i, L) = reduce(i, L):`

  `output (i, sum(L))`

# Exemple 6: Entropia d'una distribució

Entrada: Un *multiset* $S$ (conjunt on els elements poden estar repetits).

Sortida: L'entropia de $S$

$$H(S) = \sum_i -p_i \log(p_i), \text{ on } p_i = \#(S,i)/\#S$$

Job 1: Per cada $i$, calcular $p_i$:

- `map(i): output (i, 1)`
- `combiner(i, L) = reduce(i, L):`

  `output (i, sum(L))`

Job 2: Donat un vector $p$, calcular $H(p)$:

- `map(p(i)): output (0, p(i))`
- `reduce(k, L) :`

  `output sum( -p(i)*log(p(i)) )`

# *MapReduce*/*Hadoop*: Conclusions

Avantatges *MapReduce*:

- Una de les bases de la revolució del Big Data / NoSQL.
- Ha estat, durant una dècada, l'estàndard pel processament distribuït de dades massives (*big data*) de codi obert.
- L'abstracció dels detalls del clúster.
- Permet afegir més opcions externes:
  - Components per l'emmagatzematge i recuperació de dades (p.e. HDFS a Hadoop), llenguatges tipus script o tipus SQL...

Inconvenients *MapReduce*:

- Configuració complexa, programació extensa.
- L'E/S de cada job es fa de/a disc (p.e. HDFS); és lent.
- Orientat a conjunts de dades per lots (*batch*), no per manegar dades en *streaming*.
- Sovint, presenta colls d'ampolla en el rendiment; no sempre és la millor solució.

# Índex

# Big Data

- Conjunts de dades amb una mida que sobrepassa el que les eines d'emmagatzematge poden gestionar normalment.
- Les 3 V: Volum, Velocitat, Varietat, etc.
- Les xifres creixen paral·lelament amb la tecnologia.

# Big Data

- Conjunts de dades amb una mida que sobrepassa el que les eines d'emmagatzematge poden gestionar normalment.
- Les 3 V: Volum, Velocitat, Varietat, etc.
- Les xifres creixen paral·lelament amb la tecnologia.
- El problema ha existit sempre.
- De fet, és el que ha guiat la innovació.

# Big Data

I segueix:

- 5 bilions de telèfons mòbils
- internet de les coses, entorns sensoritzats
- iniciatives *Open Data* (ciència, govern)
- informàtica en núvol (*the Cloud*)
- aplicacions a escala planetària
- ...

cada vegada emmagatzemem més dades i hi ha més usuaris que accedeixen a aquestes dades al mateix temps

# Big Data

- Problema tecnològic: com emmagatzemar, usar i analitzar les dades.

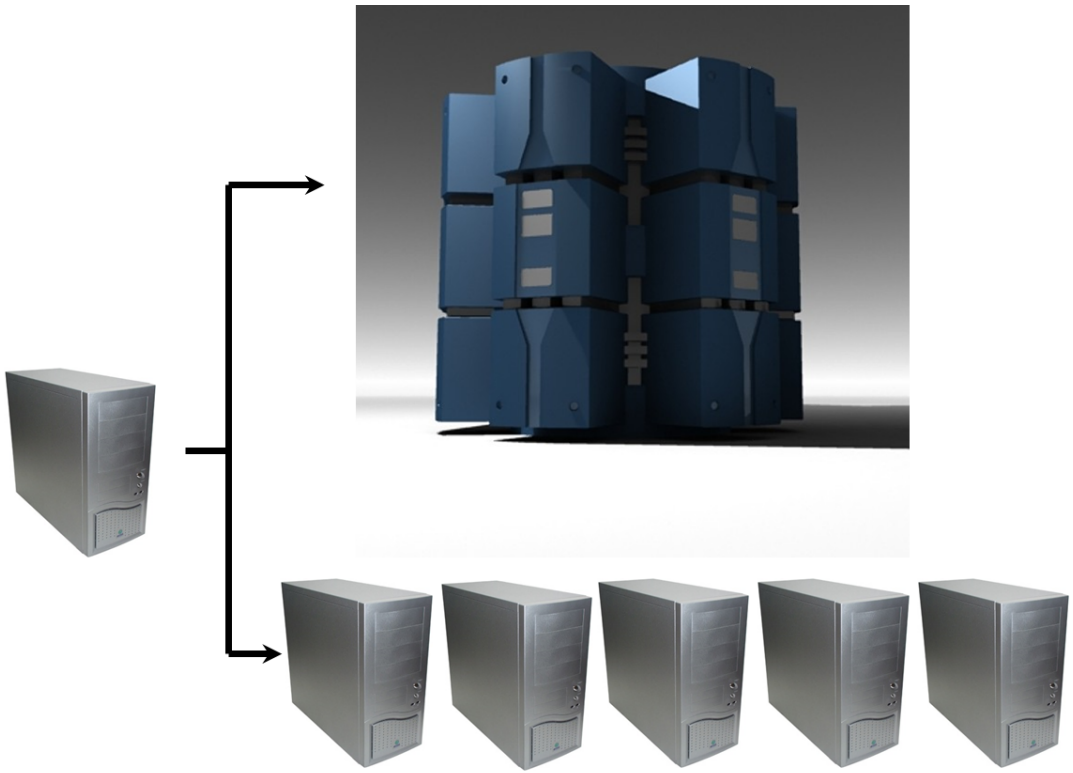
# Big Data

- Problema tecnològic: com emmagatzemar, usar i analitzar les dades.
- Problema de gestió:
  - com modelar les dades?
  - com consultar les dades?
  - per on començar?

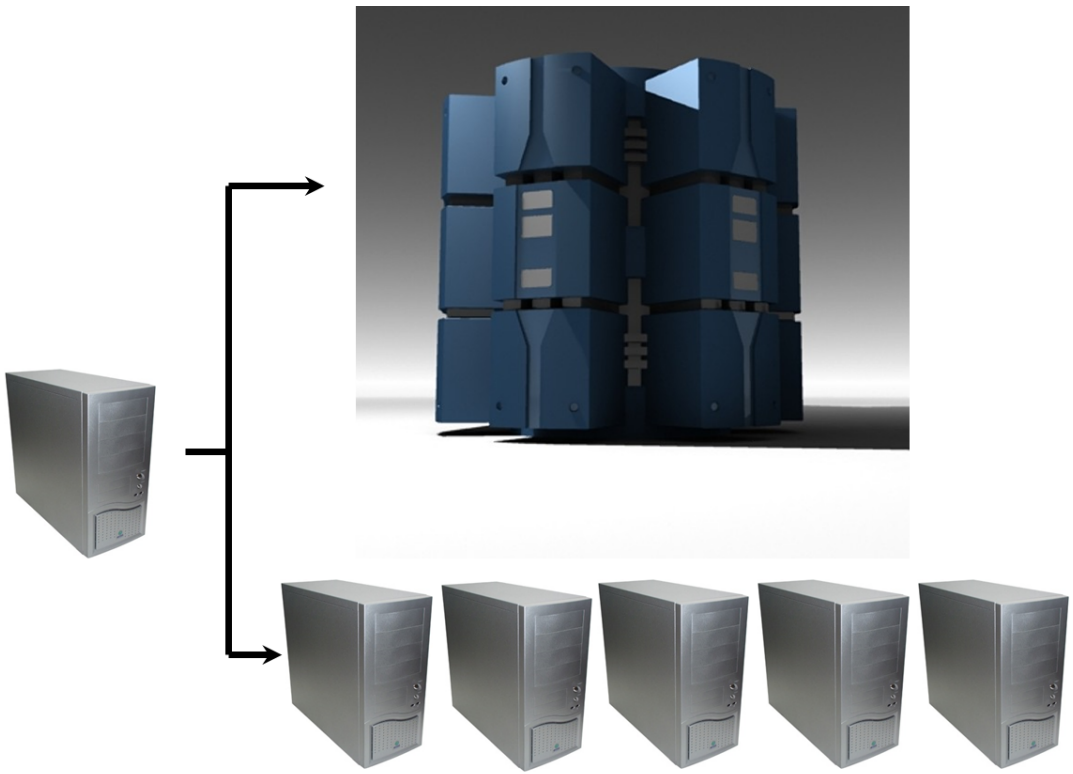
# Problemes amb les BD relacionals, I

- Les BD relacionals han funcionat durant 2-3 dècades.
- Tenen capacitats magnífiques i existeixen implementacions excel·lents.
- Un dels ingredients de la revolució a la xarxa:
  - LAMP = Linux + servidor Apache HTTP + MySQL + PHP
- Problema principal: escalabilitat.

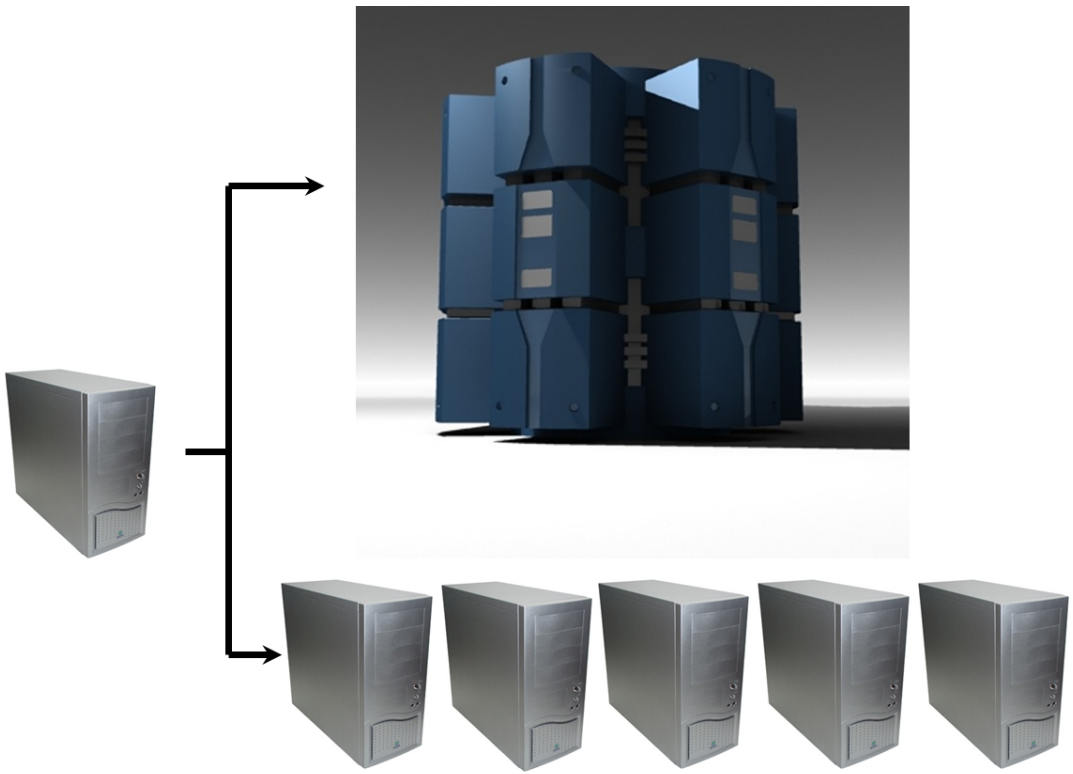
# Escalabilitat vertical (millora dels components)



## Escalabilitat vertical (millora dels components)



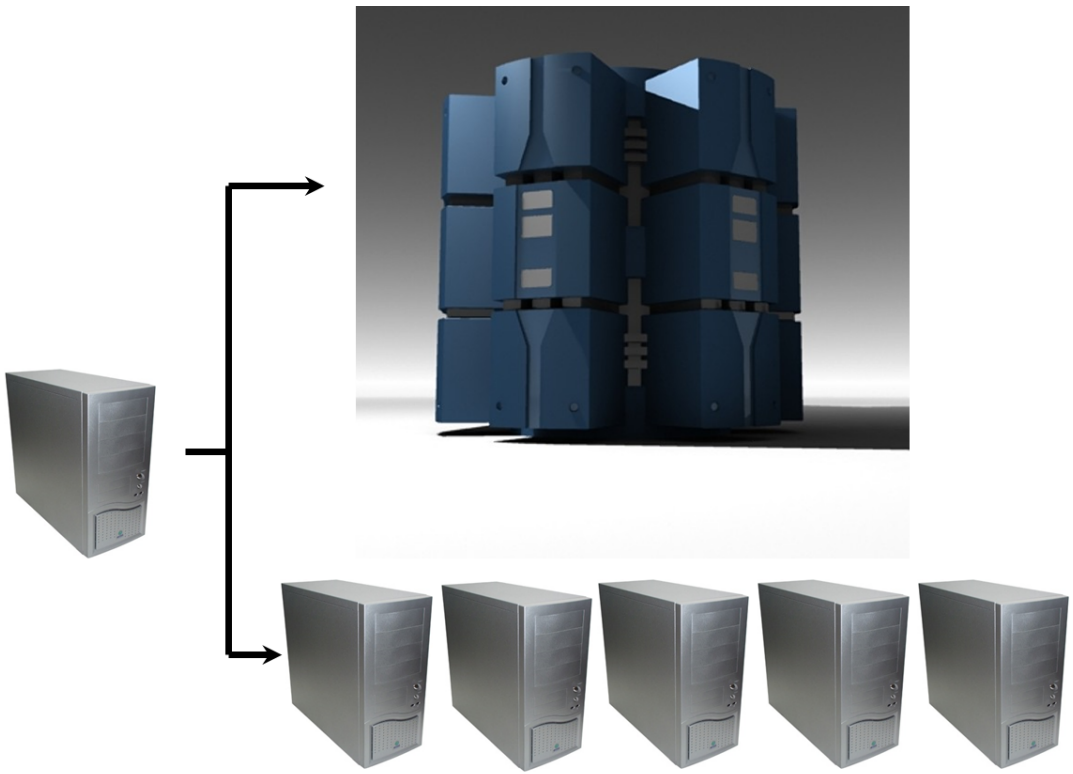
- El preu és superlineal en el rendiment
- Sostre de rendiment



## Escalabilitat vertical (millora dels components)

- El preu és superlineal en el rendiment
- Sostre de rendiment

## Escalabilitat horitzontal (afegir més components)



## Escalabilitat vertical (millora dels components)

- El preu és superlineal en el rendiment
- Sostre de rendiment

## Escalabilitat horitzontal (afegir més components)

- No hi ha sostre de rendiment, però
- La gestió és més complexa
- La programació és més complexa
- Mantenir les propietats ACID és problemàtic

# Problemes amb les BD relacionals, II

- Els RDBMS escalen bé verticalment (un sol node). No escalen bé horitzontalment.
- Fragmentació vertical: columnes d'una mateixa taula en servidors diferents.
- Fragmentació horizontal: files d'una mateixa taula en servidors diferents.

# Problemes amb les BD relacionals, II

- Els RDBMS escalen bé verticalment (un sol node). No escalen bé horitzontalment.
- Fragmentació vertical: columnes d'una mateixa taula en servidors diferents.
- Fragmentació horizontal: files d'una mateixa taula en servidors diferents.

Possible solució: rèpliques i *caches*

- Bo per la tolerància a fallades, per seguretat.
- Bo per a diverses lectures concurrents.

# Problemes amb les BD relacionals, II

- Els RDBMS escalen bé verticalment (un sol node). No escalen bé horitzontalment.
- Fragmentació vertical: columnes d'una mateixa taula en servidors diferents.
- Fragmentació horizontal: files d'una mateixa taula en servidors diferents.

Possible solució: rèpliques i *caches*

- Bo per la tolerància a fallades, per seguretat.
- Bo per a diverses lectures concurrents.
- No ajuda gaire en les escriptures, si volem mantenir les propietats ACID (Atomicitat, Coherència, Isolament, Durabilitat).

# El teorema CAP (per a sistemes distribuïts), I

Tres requeriments desitjables:

- *Consistency*: Després d'actualitzar un objecte, qualsevol accés a l'objecte tornarà el valor actualitzat.
- *Availability*: En qualsevol moment, tots els clients de la BD han de tenir accés a *alguna* versió de les dades. És a dir, cada consulta rep una resposta.
- *Partition tolerance*: La BD es distribueix en diversos servidors que es comuniquen per xarxa. El sistema continua funcionant encara que es perdi algun missatge entre els nodes o hi hagi una caiguda parcial del sistema.

# El teorema CAP (per a sistemes distribuïts), I

Tres requeriments desitjables:

- *Consistency*: Després d'actualitzar un objecte, qualsevol accés a l'objecte tornarà el valor actualitzat.
- *Availability*: En qualsevol moment, tots els clients de la BD han de tenir accés a *alguna* versió de les dades. És a dir, cada consulta rep una resposta.
- *Partition tolerance*: La BD es distribueix en diversos servidors que es comuniquen per xarxa. El sistema continua funcionant encara que es perdi algun missatge entre els nodes o hi hagi una caiguda parcial del sistema.

El teorema CAP [Brewer 00, Gilbert-Lynch 02] diu:

Cap sistema distribuït pot complir els tres requeriments alhora.

O bé: En un sistema compost per nodes no fiables i una xarxa, és impossible implementar lectures i escriptures atòmiques, i garantir que tota consulta tindrà una resposta.

# El teorema CAP (per a sistemes distribuïts), II

## Demostració

- Dos nodes, A, B,
- A rep la consulta “read(x)”,
- Per ser coherent, A ha de comprovar si s’ha fet algun “write(x, value)” a B
- . . . per tant, envia un missatge a B,
- Si A no rep resposta de B, o bé A respon (de forma incoherent)
- o bé A no respon (no està disponible).

# Problemes amb les BD relacionals, III

- Un DBMS realment distribuït i realment relacional hauria de complir els tres requeriments: *Consistency*, *Availability* i *Partition Tolerance*
- . . . però és impossible.

# Problemes amb les BD relacionals, III

- Un DBMS realment distribuït i realment relacional hauria de complir els tres requeriments: *Consistency*, *Availability* i *Partition Tolerance*
- . . . però és impossible.

- Relacional, compleix del tot C+A però a canvi de P.

- Les tecnologies NoSQL obtenen escalabilitat *sense l'objectiu* de complir els tres C, A, P.
- sovint, intenten aconseguir A+P o C+P
- . . . i tant com puguin de la tercera propietat.

# NoSQL: Generalitats, I

NoSQL: significa “Not Only SQL”.

Propietats de la majoria de BD NoSQL:

1. BASE en lloc de ACID.

# BASE, *eventual consistency*

- *Basically Available, Soft state, Eventual consistency*
  Una aplicació bàsicament s'ha poder executar tot el temps (*basically available*) i no cal que sigui sempre coherent (*soft state*) però sí ho serà amb el temps en algun moment conegut (*eventually consistent*).
- ACID és pessimista. BASE és optimista. Accepta que la coherència de la BD es produirà en un estat de flux.
- Sorprenentment, funciona bé amb moltes aplicacions
- i permet *molta* més escalabilitat que ACID.

# NoSQL: Generalitats, I

NoSQL: significa “Not Only SQL”.

Propietats de la majoria de BD NoSQL:

1. BASE (*Basically Available, Soft state, eventual Consistency*) en lloc de ACID.
2. Consultes simples. No *joins*.
3. No hi ha esquemes fixos ni relacions entre taules.
4. Descentralitzat, particionat (fins i tot en múltiples centres de dades).
5. Linealment escalable usant PC estàndard.
6. Tolerància a fallades.
7. No per al processament de transaccions (complexes) online.
8. No per fer “datawarehousing”.

# Alguns noms, segons el model de dades

Clau-valor, Orientat-a-columnes, Document, Orientat a grafs

- Per a dades semiestructurades o no estructurades

  Clau-valor: DynamoDB, Riak, Voldemort, Cassandra, Redis, Memcached, BigTable

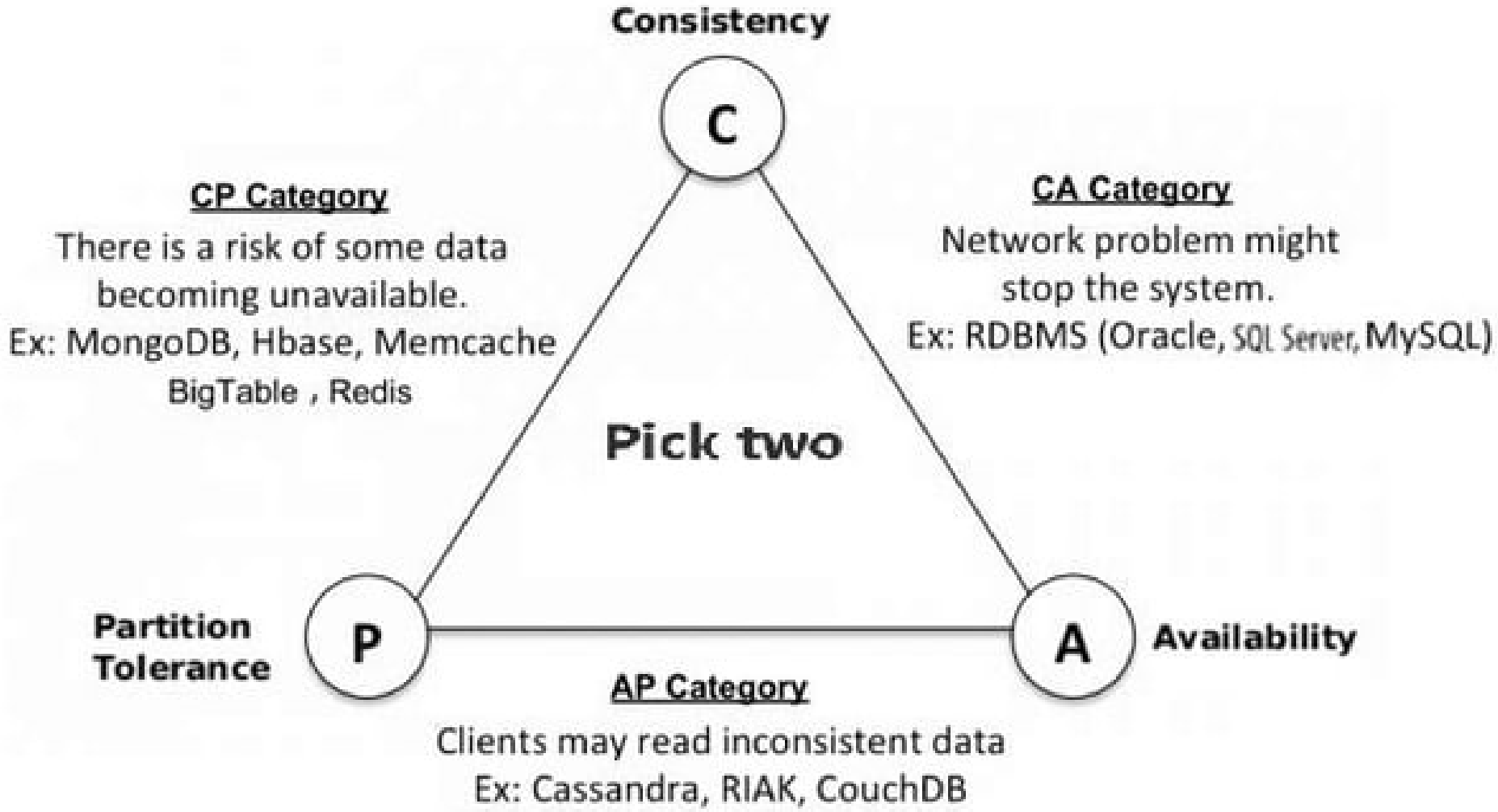
  Orientat-a-columnes: Cassandra, Hbase, Hypertable

  Document: MongoDB, CouchDB, Firebase Realtime, Google Cloud Datastore

- Per a dades amb relacions múltiples i complexes

  Orientat a grafs: Neo4j, Giraph, Sparksee (abans DEX), Pregel, FlockDB

# Alguns noms, segons les seves propietats CAP



# Alguns noms, pels seus usuaris

Voldemort: LinkedIn

Cassandra: Facebook, Digg, Reddit, Twitter, Netflix, Instagram

Riak: Facebook chat, Tuenti chat, Yammer, GitHub

Hbase: Twitter, Facebook, Mendeley

# BD relacionals o BD NoSQL

- La solució més òptima pot ser una barreja d'ambdues
  - Per treballar amb dades estructurades o funcionalitats on calgui integritat referencial: BD relacional.
  - Per funcionalitats on necessitem un alt rendiment o una solució distribuïda sense haver de garantir consistència total: BD NoSQL

# BD relacionals o BD NoSQL

- La solució més òptima pot ser una barreja d'ambdues
  - Per treballar amb dades estructurades o funcionalitats on calgui integritat referencial: BD relacional.
  - Per funcionalitats on necessitem un alt rendiment o una solució distribuïda sense haver de garantir consistència total: BD NoSQL
- Exemple: l'arquitectura de dades de Twitter

| | |
|---|---|
| rel | MySQL per emmagatzemar usuaris i tuits |
| nosql | FlockDB per emmagatzemar grafs socials (*followers*) |
| nosql | SnowFlake (Cassandra) per generar ID únics |
| nosql | Redis per generar els timelines |
| nosql | Hadoop, Pig, Hive i HBase |
| | Gizzard: entorn per treballar amb info distribuïda |
| | Apache Lucene: API per fer cerques |

# Índex

# Apache Spark: motivació

- Alternativa pel processament de dades massives en casos d'ús en els que MapReduce no es mostra del tot eficient. Exemples:
  - Processos iteratius
  - Processos intensius en consultes
- L'alternativa es basa en l'ús de memòria principal en lloc de l'emmagatzemament a disc.
- Spark ofereix una abstracció anomenada *Resilient Distributed Datasets* (RDD).

# Hadoop vs Spark. Disc vs memòria

(Font: https://www.tutorialspoint.com/apache_spark/apache_spark_tutorial.pdf)

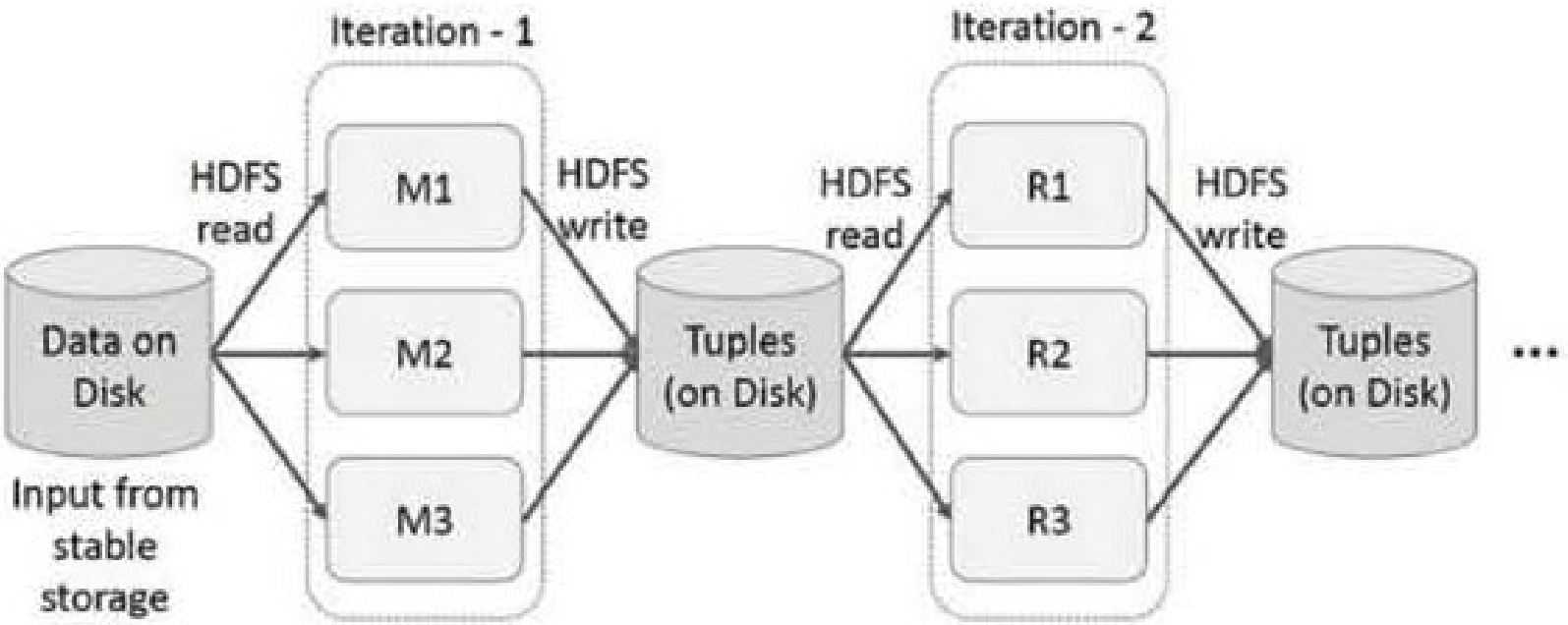



**Figure**: Iterative operations on MapReduce

# Hadoop vs Spark. Disc vs memòria

(Font: https://www.tutorialspoint.com/apache_spark/apache_spark_tutorial.pdf)

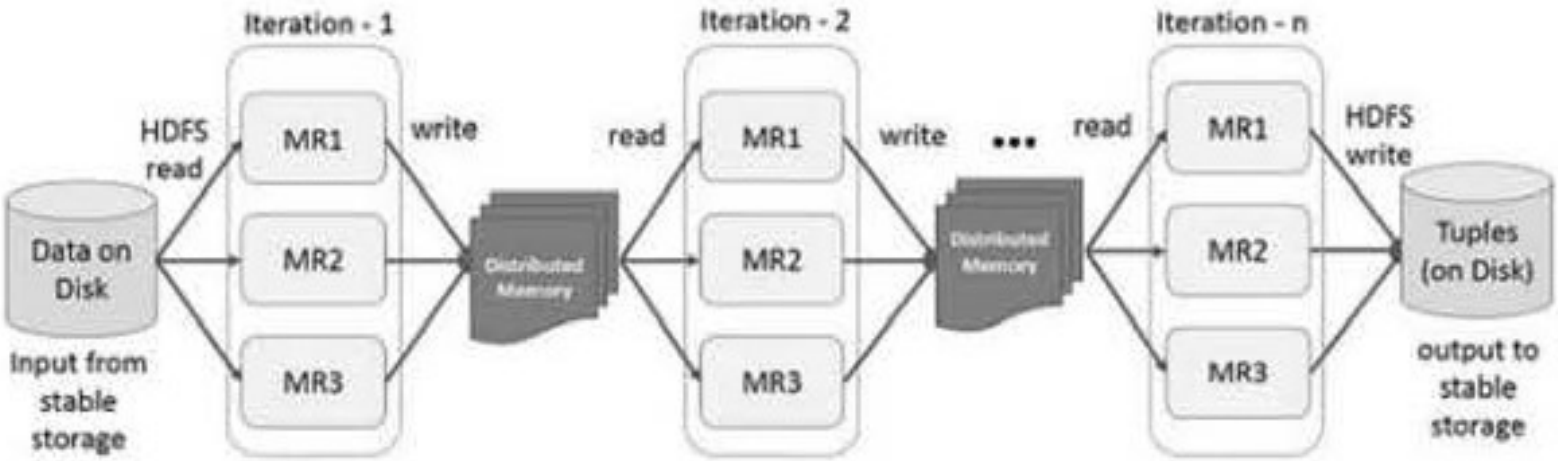



**Figure**: Iterative operations on Spark RDD

# Hadoop vs Spark. Disc vs memòria

(Font: https://www.tutorialspoint.com/apache_spark/apache_spark_tutorial.pdf)

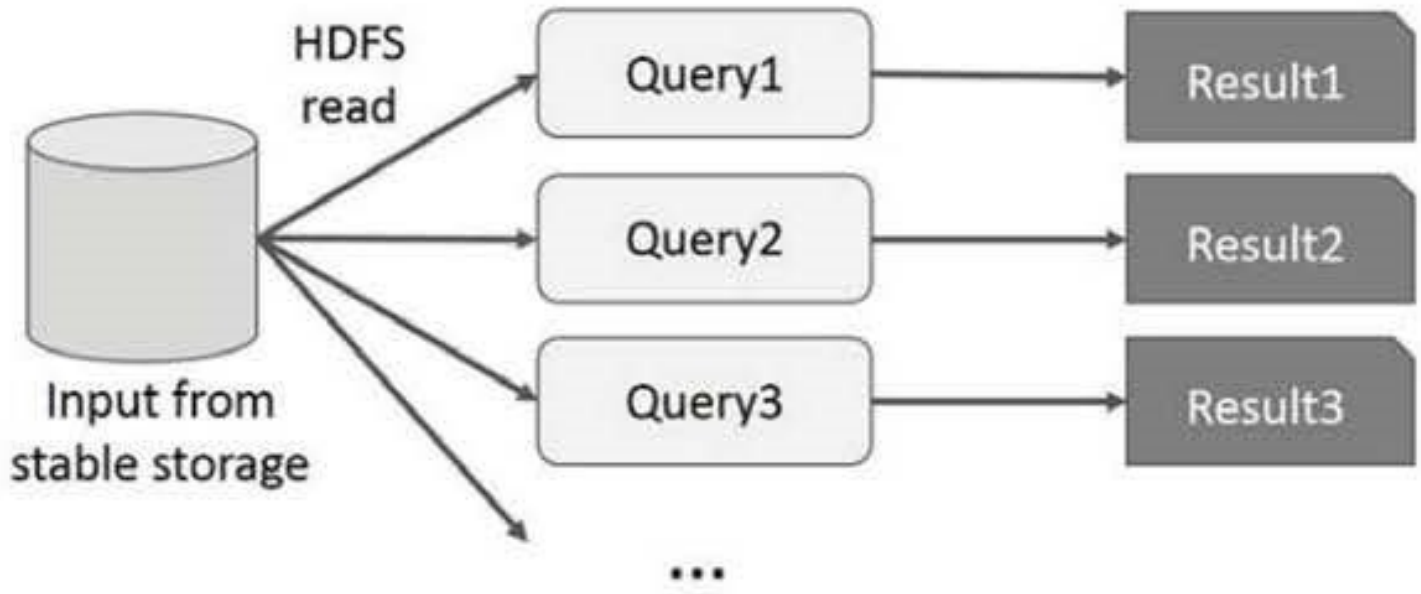



**Figure**: Interactive operations on MapReduce

# Hadoop vs Spark. Disc vs memòria

(Font: https://www.tutorialspoint.com/apache_spark/apache_spark_tutorial.pdf)

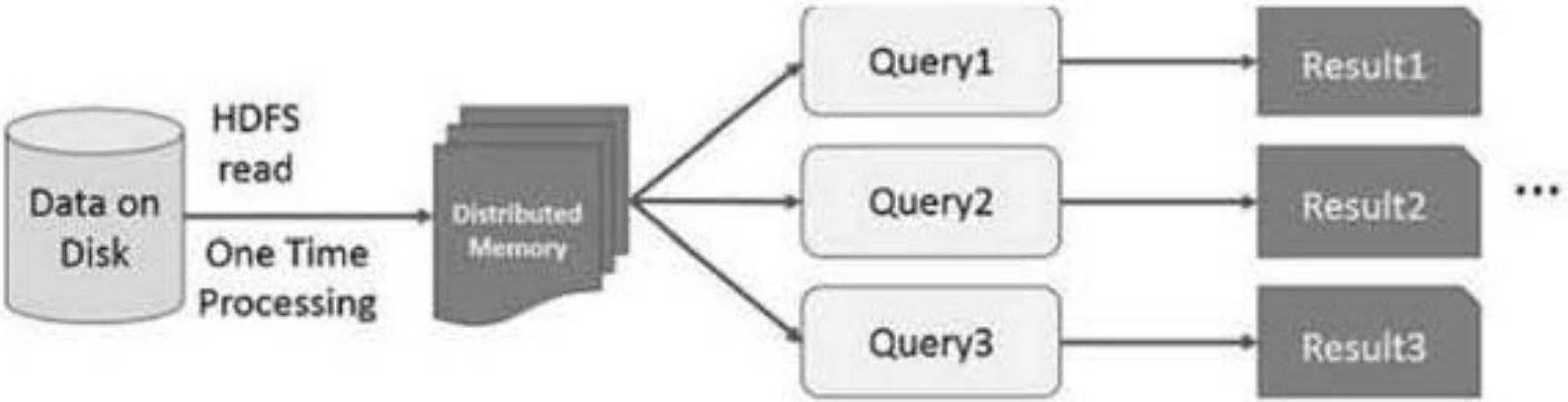



**Figure:** Interactive operations on Spark RDD

# Spark: dos conceptes clau

1. *Resilient Distributed Datasets* (RDD)
   - El conjunt de dades es particiona entre *worker nodes*; es poden emmagatzemar en memòria (o a disc, si cal)
   - Es poden crear a partir de fitxers HDFS
   - Són immutables
2. *Directed Acyclic Graph* (DAG)
   - Especifica el conjunt de transformacions que s'han de fer a les dades (traça)
   - Les dades passen d'un estat a un altre

- Eviten un dels colls d'ampolla de Hadoop: les escriptures a disc
- La traçabilitat proporciona la tolerància a fallades (recuperar i regenerar l'estat de les dades, *resilient*)
- Permet el processament continu (*Spark Streaming*)

Més informació sobre RDD.