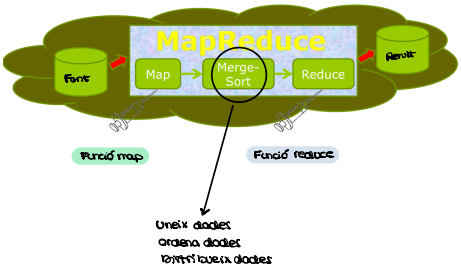
mapReduce

- Treballa sobre Google File System (ara en més)

- Requeriments
  - Tu treballes, ho envies al mapReduce, i ells es distribueixen
  - En comptes de per Data Shipping, envies Query Shipping ⇝ s'executa en local on són les dades
  - Paral·lelisme:
    - Problema anterior → claus foranes (per per validacions, joins, ...) → dades estan lluny
    - Volien treballar amb petabytes.
  - Escalable
  - Si una màquina falla ⇝ sistema es recupera

- Basat en functional programming

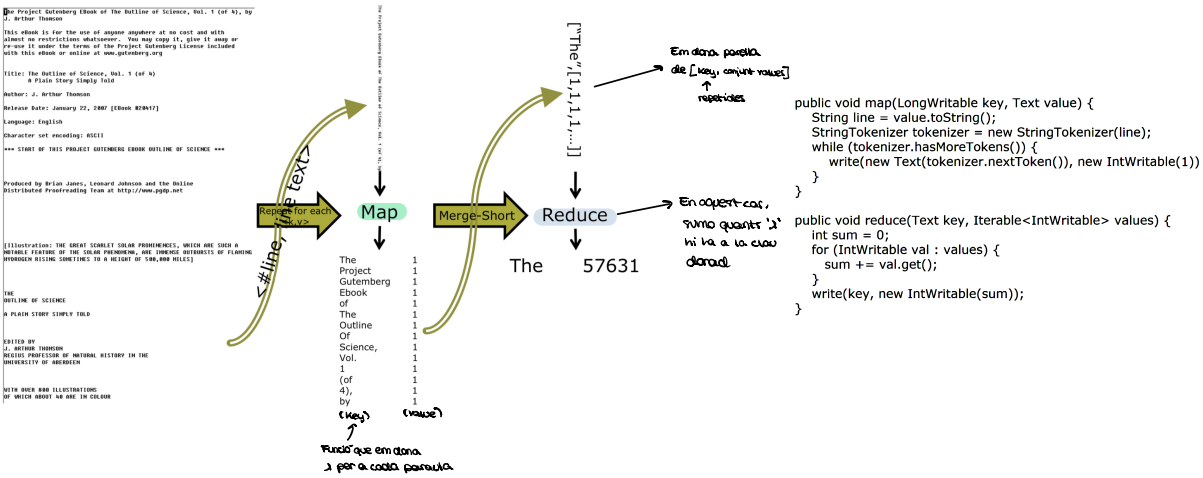


- Processen parelles [key, value] (Es pot repetir)
- Signatura ⇝ el que hem de passar nosaltres

map (key k, value v) → [(ik1, iv1), ..., (ikr(k,v), ivm(k,v))]

reduce (key ik, vset) → [ov1, ..., ovr (ikr, ivr)]
(vset: set de values)

Exemple: word count




```
public void map(LongWritable key, Text value) {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
        write(new Text(tokenizer.nextToken()), new IntWritable(1))
    }
}

public void reduce(Text key, Iterable<IntWritable> values) {
    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }
    write(key, new IntWritable(sum));
}
```

```
public void map (key, value) {
    whatever
    key, value
}
```

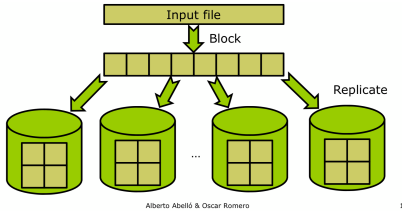
```
public void reduce (key, value) {
    whatever
    key, value
}
```

PROBLEMA.

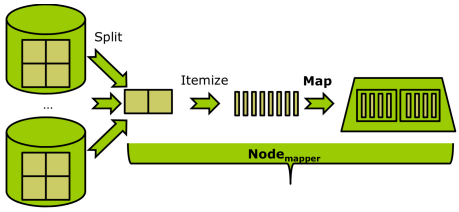
Com que no sap què hi ha al "whatever", no pot optimitzar res.

Versió 2 ⇝ Spark: intenta mirar el "whatever" per optimitzar una mica.

Algoritme

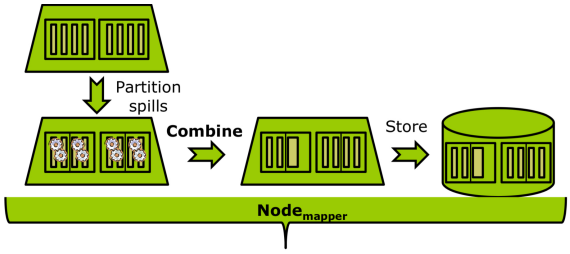


1. Pujar fitxer a HDFS (cloud)
   - HDFS ho divideix en chunks
   - By default, ho replica a 3 mateges.

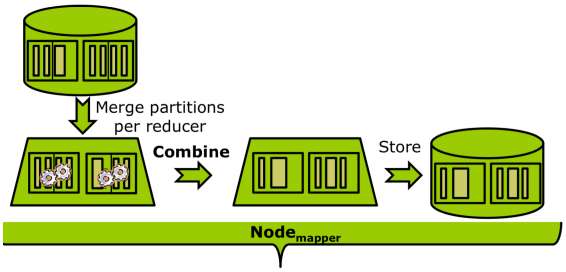


2. A cadascun dels workers, se li assigna un split (subset de blocks)
3. Worker el divideix en records
4. Executa la funció del map i els posa a memòria (spill)
   ↳ conté resultats d'haver executat 1 o més cops "map"

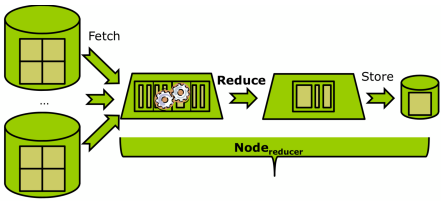
5. Per cada spill, el particiona amb tants trossos com reducers tenim
6. Cada partició de l'spill s'ordena de forma independent.
7. Cada cop que ordenem, s'executa "combine"
   ↗ reduce parcial → pel merge.



8. Guardem les particions de spill a disc (massive writing)



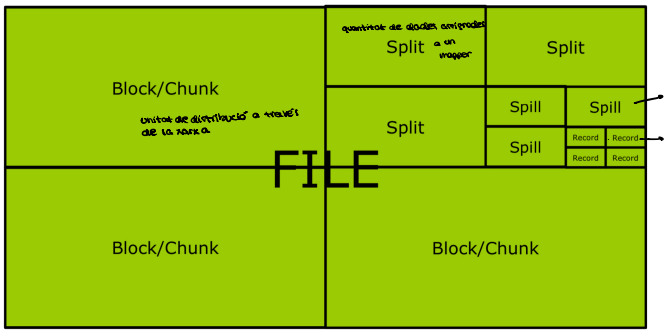
9. Agafo les particions de spill ordenades i les combino i ordeno
10. Guardem el resultat al disc.



11. Reducer fa fetch de les particions que hi ha al mapper
12. L'output del mapper es torna a fer merge
13. S'executa el reduce
14. El guarda al disc un altre cop.

M → R-B, R-B, R-B

Objectes



Record=Key-Value pair

Combine

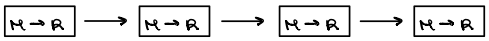
- Permet optimitzar (↓ lectures de disc)
- Redueix transit de xarxa
- Important → commutativa
  → associativa
  → igual que el reducer
- $\frac{|input|}{|output|}$ >> # CPU

Problemes mapReduce

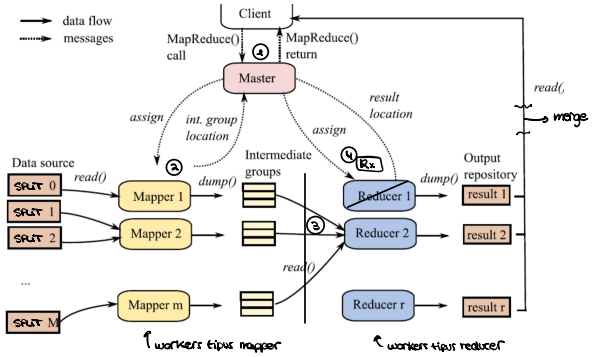
- Massa baix nivell (java) → tant eficient com tu el facis

Avantatges mapReduce

- Pots fer el que vulguis
- Tancat: pots fer cicles entre maps-reduce



Tasques i dataflow



Sempre que s'escriu, es fa en local
És el "reader" qui va a buscar les dades
→ no perdem fitxer mai

Costos

1. Start-up time
2. Writing intermediate results (necessari per no perdre dades)
3. Transferència de dades (xarxa)
4. Fer pings a master per dir que estan vius (necessari per tenir availability)

PROBLEMES → Reassignments needed (4)
→ Single-point failure (1 master)

Què fa Spark ?

- menys escriptures a disc : només quan ho veure necessari
- permet més operacions
- et facilita controlar les particions

Exercici mapReduce

Assumim que tenim el següent codi:

```
public void map(LongWritable key, Text value) {
  String line = value.toString();
  StringTokenizer tokenizer = new StringTokenizer(line);
  while (tokenizer.hasMoreTokens()) {
    write(new Text(tokenizer.nextToken()), new IntWritable(1));
  }
}

public void reduce(Text key, Iterable<IntWritable> values) {
  int sum = 0;
  for (IntWritable val : values) {
    sum += val.get();
  }
  write(key, new IntWritable(sum));
}
```

ex Set:

Ex: Block0: "a b b a c | c d c a e"

Block1: "a b a d a | b b c c f"

configuration

- combine = reduce
- 1 split = 1 block
- '1' divideix els records del block → 2 records/block
- dfs.replication = 1 (hadoop)
- 1 spill = 4 [key, value]
- 2 mappers, 2 reducers
  - machine0 → block0 → mapper0 → reducer0 , machine1 → block1 → mapper1 → reducer1
- hash for reducers → {b, d, f} → 0 ; {a, c, e} → 1

#split ← #màquines virtuals

Fill the gaps:

1) Machine0 contains 1 blocks. (block0)
   Machine1 contains 1 blocks. (block1)
2) We keep 2 replicas (including the master copy) per block.
3) We have 2 splits per machine. (1 split = 1 block)

4) Mapper0 reads 2 records.
   Mapper1 reads 2 records.

5) Spills in Machine0:

Spill1	Spill2	Spill3	Spill4
[a,1] [b,1]	[c,1] [c,2]	[a,1] [e,1]	[,] [,]
[b,1] [a,1]	[d,1] [c,2]	[,] [,]	[,] [,]

Spills in Machine1:

Spill1	Spill2	Spill3	Spill4
[a,1] [b,1]	[a,1] [b,1]	[c,1] [f,1]	[,] [,]
[d,1] [d,1]	[b,1] [c,1]	[,] [,]	[,] [,]

amics com.

6) Partitions in machine0:

Spill 1		Spill 2		Spill 3	
Partition 0	Partition 1	Partition 0	Partition 1	Partition 0	Partition 1
[b,1] [b,1]	[a,1] [a,1]	[d,1] [,]	[c,1] [c,1]	[,] [,]	[a,1] [e,1]
[,] [,]	[,] [,]	[,] [,]	[c,1] [,]	[,] [,]	[,] [,]

Partitions in Machine1:

Spill 1		Spill 2		Spill 3	
Partition 0	Partition 1	Partition 0	Partition 1	Partition 0	Partition 1
[b,1] [d,1]	[a,1] [,]	[b,1] [b,1]	[a,1] [c,1]	[f,2] [,]	[c,1] [,]
[d,1] [,]	[,] [,]	[,] [,]	[,] [,]	[,] [,]	[,] [,]

7) Partitions in machine0:

Spill 1		Spill 2		Spill 3	
Partition 0	Partition 1	Partition 0	Partition 1	Partition 0	Partition 1
[b, 2] [,] [,] [,]	[a, 2] [,] [,] [,]	[d, 1] [,] [,] [,]	[c, 2] [,] [,] [,]	[,] [,] [,] [,]	[a,1] [c,1] [,] [,]

Partitions in Machine1:

Spill 1		Spill 2		Spill 3	
Partition 0	Partition 1	Partition 0	Partition 1	Partition 0	Partition 1
[b,1] [d,2] [,] [,]	[a, 1] [,] [,] [,]	[b 2] [,] [,] [,]	[a,1] [e,1] [,] [,]	[g,1] [,] [,] [,]	[e,1] [,] [,] [,]

8) Files in machine0:

File0	File1	File2
[b, 2] [,] [,] [,]	[a,2] [,] [,] [,]	[d,1] [,] [,] [,]
File3	File4	File5
[c, 3] [,] [,] [,]	[,] [,] [,] [,]	[a,1] [e,1] [,] [,]

Files in Machine1:

File0	File1	File2
[b,1] [d, 2] [,] [,]	[a,1] [,] [,] [,]	[b,2] [,] [,] [,]
File3	File4	File 5
[a,1] [c,1] [,] [,]	[g,1] [,] [,] [,]	[e,1] [,] [,] [,]

9) Merges in machine0:

Merge0	Merge1
[b,2] [d,1] [,] [,] [,] [,]	[a,3] [c,3] [e,1] [,] [,] [,]

Merges in Machine1:

Merge0	Merge1
[b,3] [d, 2] [g,1] [,] [,] [,]	[a, 2] [c,2] [,] [,] [,] [,]

10) Files in machine0:

File0	File1
[b,2] [d,1] [,] [,] [,] [,]	[a,3] [c,3] [e,1] [,] [,] [,]

Files in Machine1:

File0	File1
[b, 3] [d, 2] [g,1] [,] [,] [,]	[a, 2] [c, 2] [,] [,] [,] [,]

11) Reducer0 reads .....File 0..... Files from machine0
and .....File 0..... Files from machine1. (answer which Files)

Reducer1 reads .....File 1..... Files from machine0
and .....File 1..... Files from machine1. (answer which Files)

12) Merge in machine0:

File0	File1
[b, 12,34] [d, 11,21] [g, 11]	[,] [,] [,]

Merge in Machine1:

File0	File1
[a, 13,11] [e, 13,21] [e, 11]	[,] [,] [,]

13) Reduce function is executed ...3.... times in machine0.
Reduce function is executed ...3.. times in machine1.

14) Files in machine0:

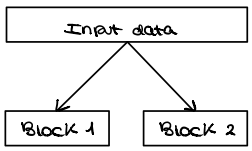
File0	File1
[b, 5] [d, 3] [g,1] [,] [,] [,]	[,] [,] [,] [,] [,] [,]

Files in Machine1:

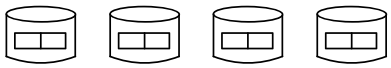
File0	File1
[a, 4] [c, 5] [e,1] [,] [,] [,]	[,] [,] [,] [,] [,] [,]

Algorithme pas a pas.

① Input data partioned into blocks Fusion les blocks au cloud par blocks ⇝ Block 0, Block 1



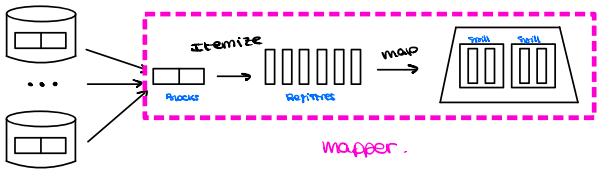
② Replicate them into different nodes 2 réplica chaque block



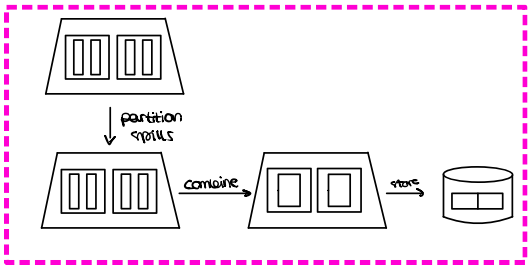
③ Each map worker reads a subset of blocks (split)

2 mapper ⇝ 2 block = 1 split

④ Divides into records. ⇝ 2 records / block (split)



⑤ Executes the map for each record ⇝ 4 records ⇝ 4 map() ⊢ on stock sous map
and leaves them in memory devided into spills



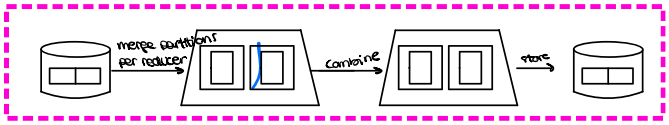
⑥ Each spill is partitioned per reducers ⇝ 2 reducers ⇝ 2 spills

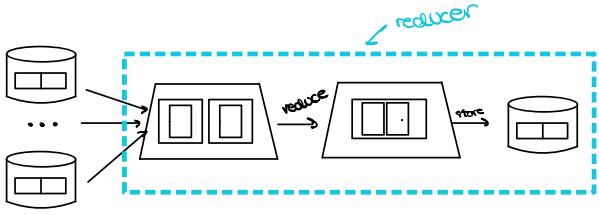
⑦ Each spill is stored independently (+combine)

⑧ Store spill partitions into disk

⑨ Spill partitions are merged and stored independently (+combine)

⑩ Store into disk.





⑪ Reducers fetch data from mappers

⑫ Mappers output is merged and stored

⑬ Reduce function is executed per key

⑭ Store the result into disk.