MapReduce Tutorial

김동원@동원리더스아카데미
포항공대 프로그래밍 언어 연구실

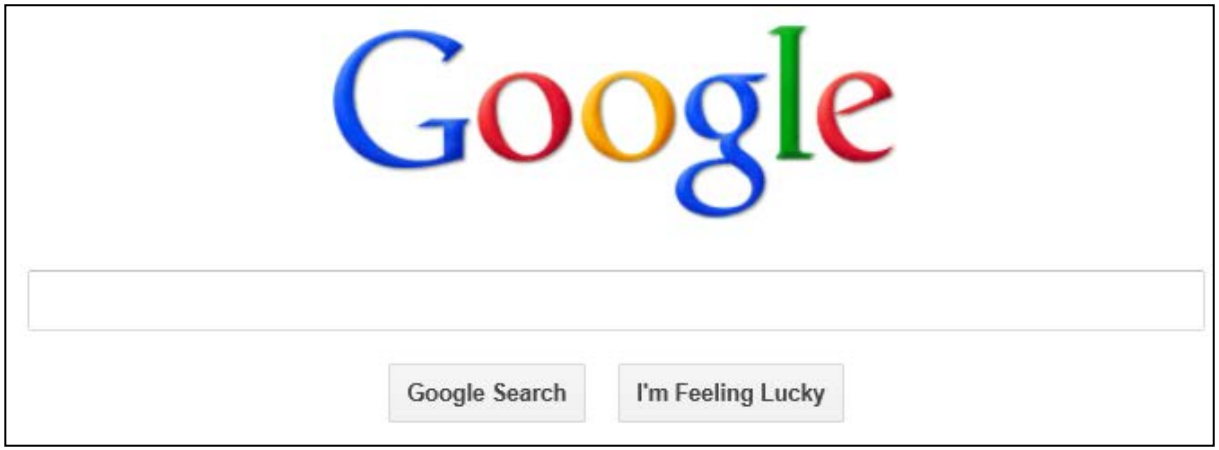
Contents

Google introduces MapReduce

<Google apps>

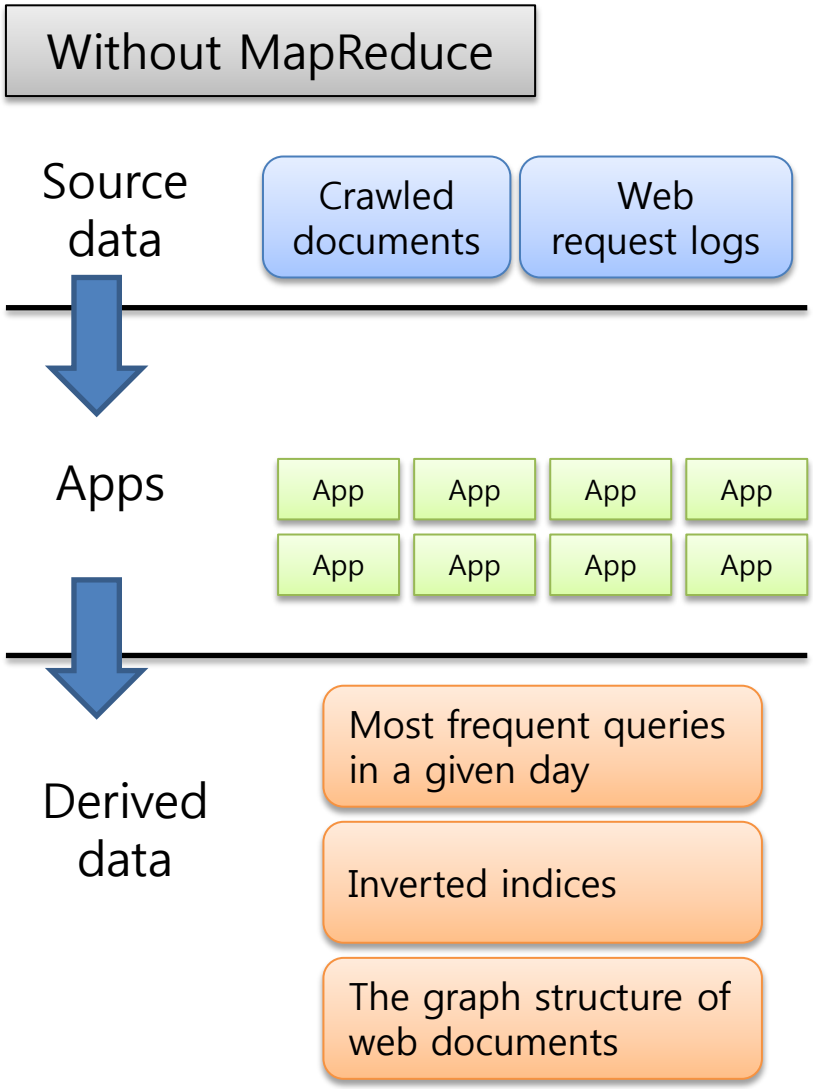
<Android>

<Google TV>



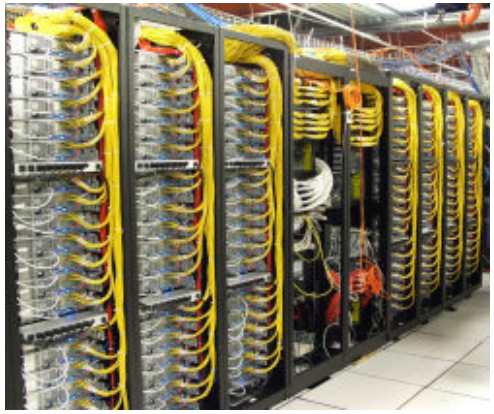
<Google search>

Why Google introduces MapReduce?



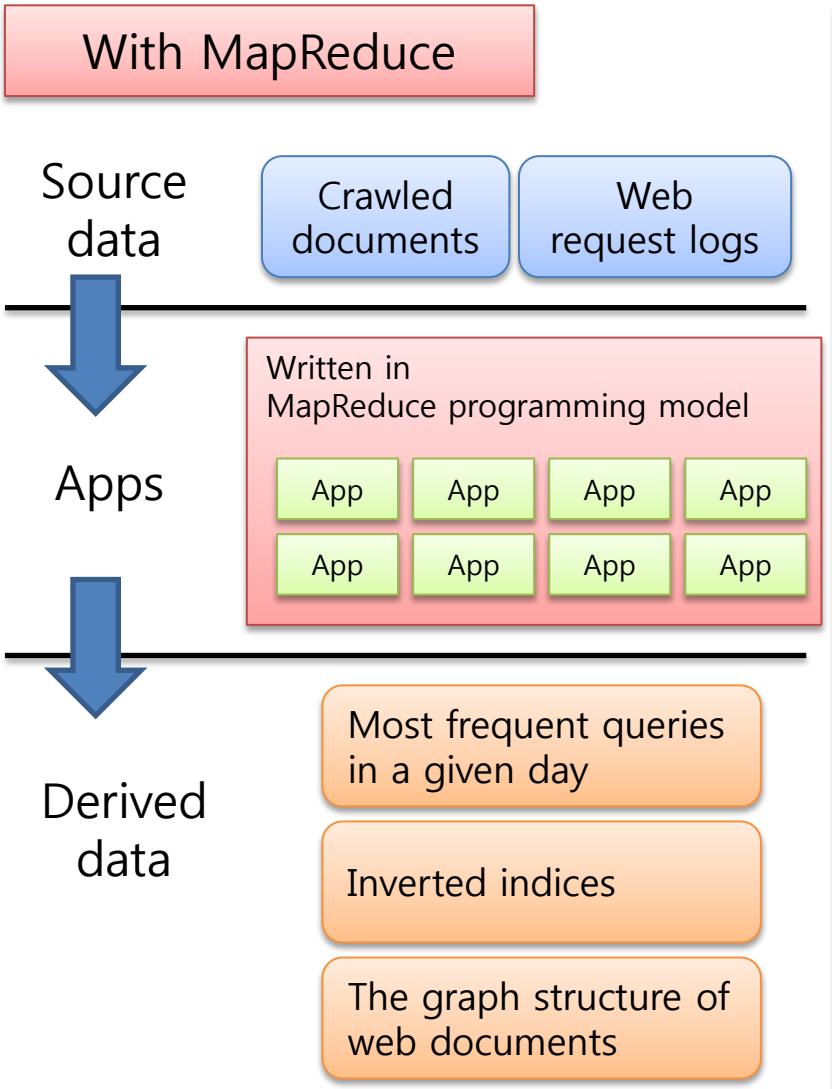
- The common characteristics of the applications
  - Conceptually straightforward
  - Large input
  - Parallelization
  - Fault-tolerance

Make code complex

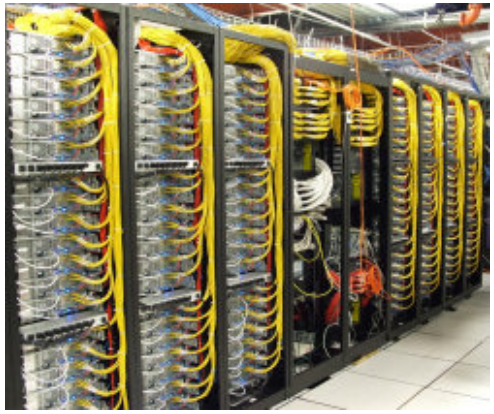


<Google datacenter>

Why Google introduces MapReduce?

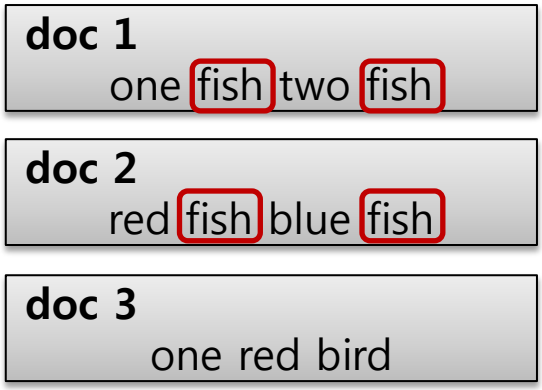


- MapReduce
  - Programming model
    - Expressive
  - Runtime system
    - Scalable
    - Fault-tolerant



<Google datacenter>

Example – Inverted index



<Input>

Term	[<doc id, # of occurrences>]	
Bird	<doc 3, 1>	
Blue	<doc 2, 1>	
Fish	<doc 1, 2>	<doc 2, 2>
One	<doc 1, 1>	<doc 3, 1>
Red	<doc 2, 1>	<doc 3, 1>
Two	<doc 1, 1>	

<Inverted index>

- A very basic search engine to find "red fish"
  1) Retrieve entries for "red" and "fish"

Red	<doc 2, 1>	<doc 3, 1>
Fish	<doc 1, 2>	<doc 2, 2>

  2) Return documents that appears in both entries
     - ➢ doc 2

Building an inverted index using **a small dataset** on **a node**

doc 1
one fish two fish

doc 2
red fish blue fish

doc 3
one red bird

<Input>

1
```
intermediate_pairs = []
for docid in ['doc1', 'doc2', 'doc3']:
    f = open(docid)
    line = f.read()
    for word in line.split():
        intermediate_pairs.append( (word, docid)  )
    f.close()
```

2
```
sorted_pairs = sorted(intermediate_pairs, key=itemgetter(0))
grouped_pairs = groupby(sorted_pairs, itemgetter(0))
```

3
```
for word, iterator in grouped_pairs:
    dict = {}
    for (_, docid) in iterator:
        if docid not in dict:
            dict[docid] = 1
        else:
            dict[docid] += 1
    print word, dict
```

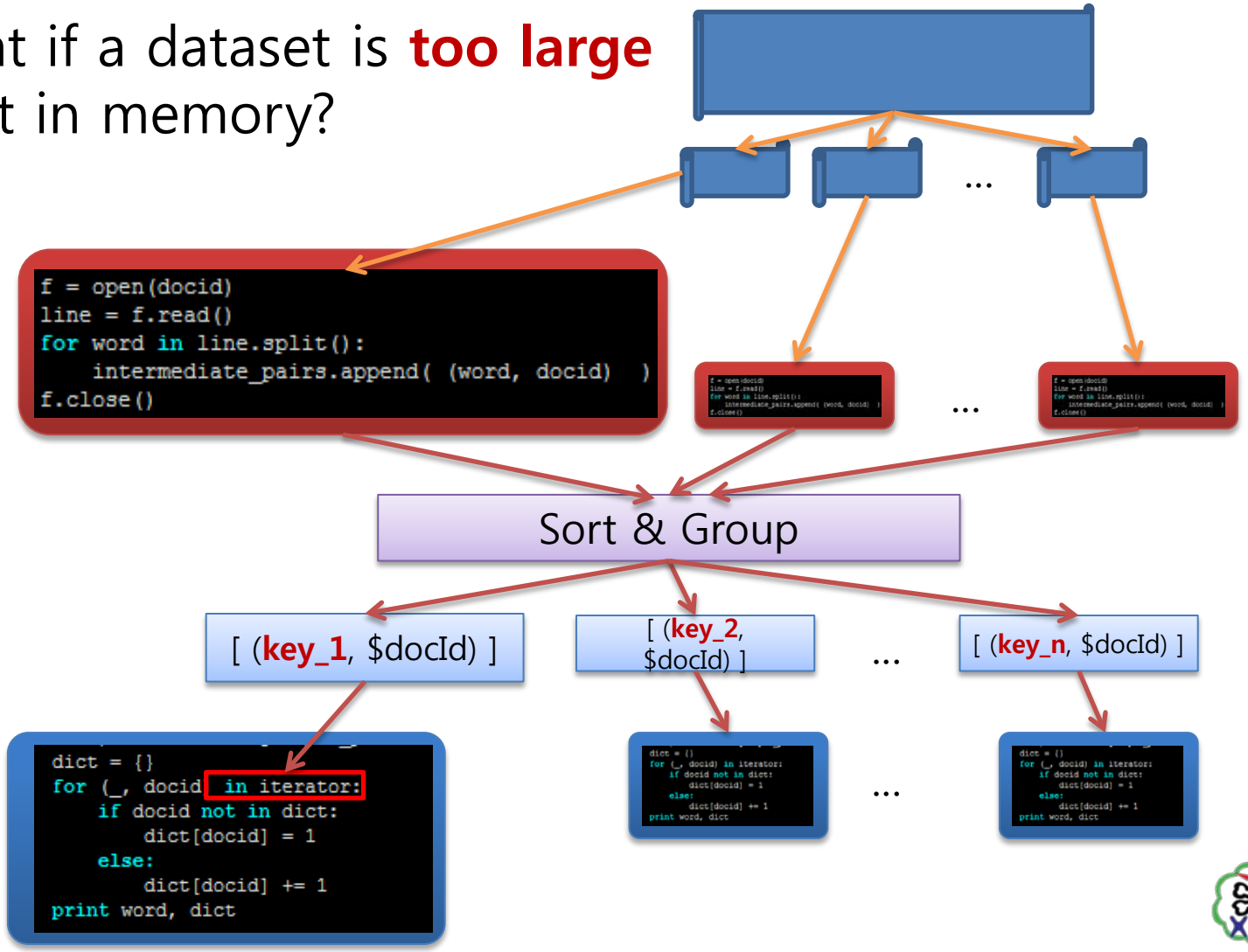
<Python program>

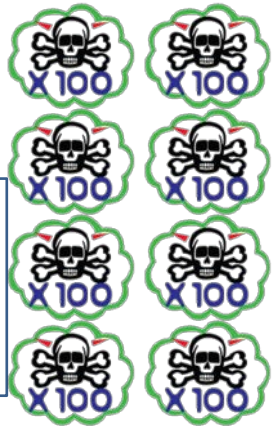
1 **intermediate_pairs**	(one, doc1) (fish, doc1) (two, doc1) (fish, doc1)	← from doc1
	(red, doc2) (fish, doc2) (blue, doc2) (fish, doc2)	← from doc2
	(one, doc3) (red, doc3) (bird, doc3)	← from doc3
2 **sorted_pairs**	(bird, doc3) (blue, doc2) (fish, doc1) (fish, doc1) (fish, doc2) (fish, doc2)	
grouped_pairs	(one, doc1) (one, doc3) (red, doc2) (red, doc3) (two, doc1)	

3
```
bird {'doc3': 1}
blue {'doc2': 1}
fish {'doc2': 2, 'doc1': 2}
one {'doc3': 1, 'doc1': 1}
red {'doc2': 1, 'doc3': 1}
two {'doc1': 1}
```

<Result>

What if a dataset is **too large** to fit in memory?





- **How to** parallelize the computation across many machines?
- **How to** sort & group intermediate pairs?
- **How to** handle machine failures?

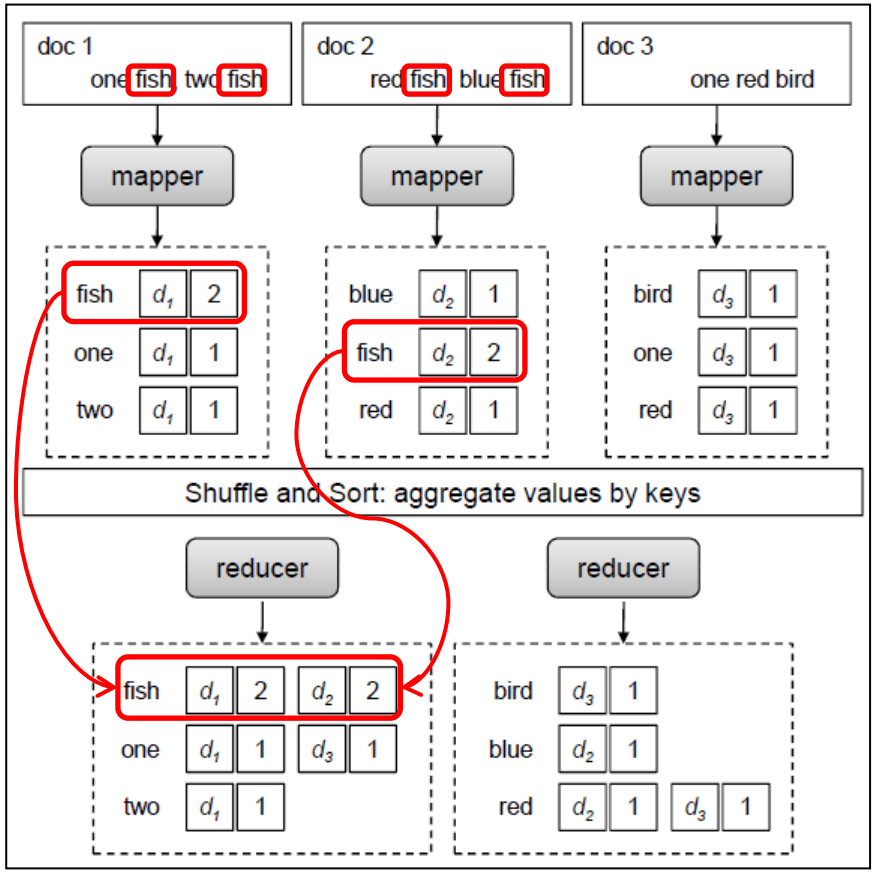
Building an inverted index using MapReduce

```
procedure MAP(docid n, doc d)
    H ← new ASSOCIATIVEARRAY
    for all term t ∈ doc d do
        H{t} ← H{t} + 1
    for all term t ∈ H do
        EMIT(term t, posting ⟨n, H{t}⟩)
```

Make an entry(posting) for each word(term) in the document

fish <d_1, 2> fish <d_2, 2>

```
procedure REDUCE(term t, postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...])
    P ← new LIST
    for all posting ⟨a, f⟩ ∈ postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...] do
        APPEND(P, ⟨a, f⟩)
    SORT(P)
    EMIT(term t, postings P)
```

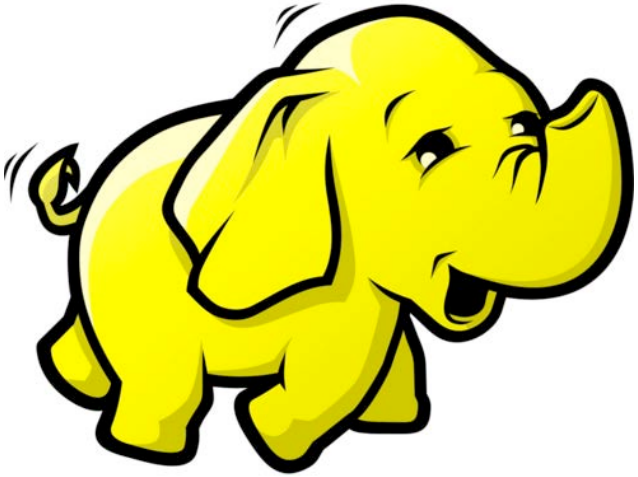


- Users specify the computation in terms of a **map** and a **reduce** function
- MapReduce runtime system automatically
  - + parallelizes the computation across large-scale clusters of machines
  - + sort & group intermediate pairs
  - + handles machine failures

Contents

1) MapReduce
   - What is MapReduce?
     - Programming model
     - Runtime system
   - Example – building an inverted index
2) Hadoop, a representative MapReduce runtime implementation
   - How to parallelize computations
   - How to sort & group intermediate pairs
   - How to handle machine failures

Hadoop



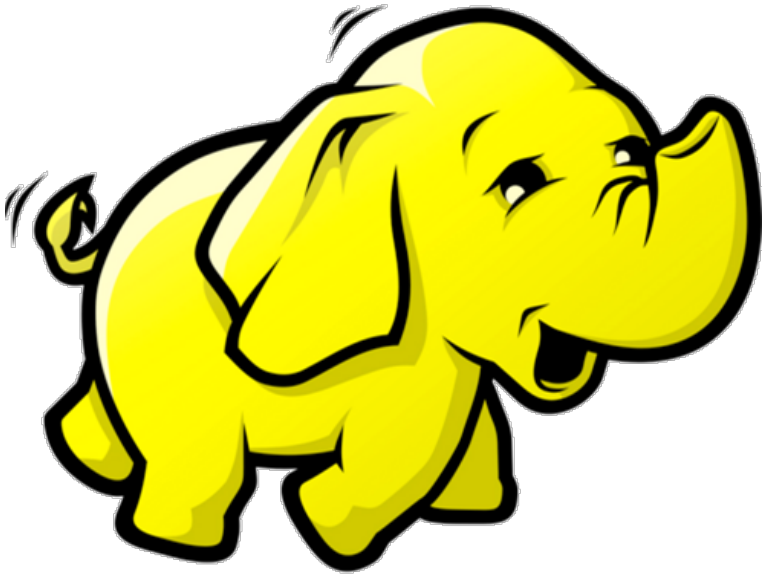
<Hadoop>

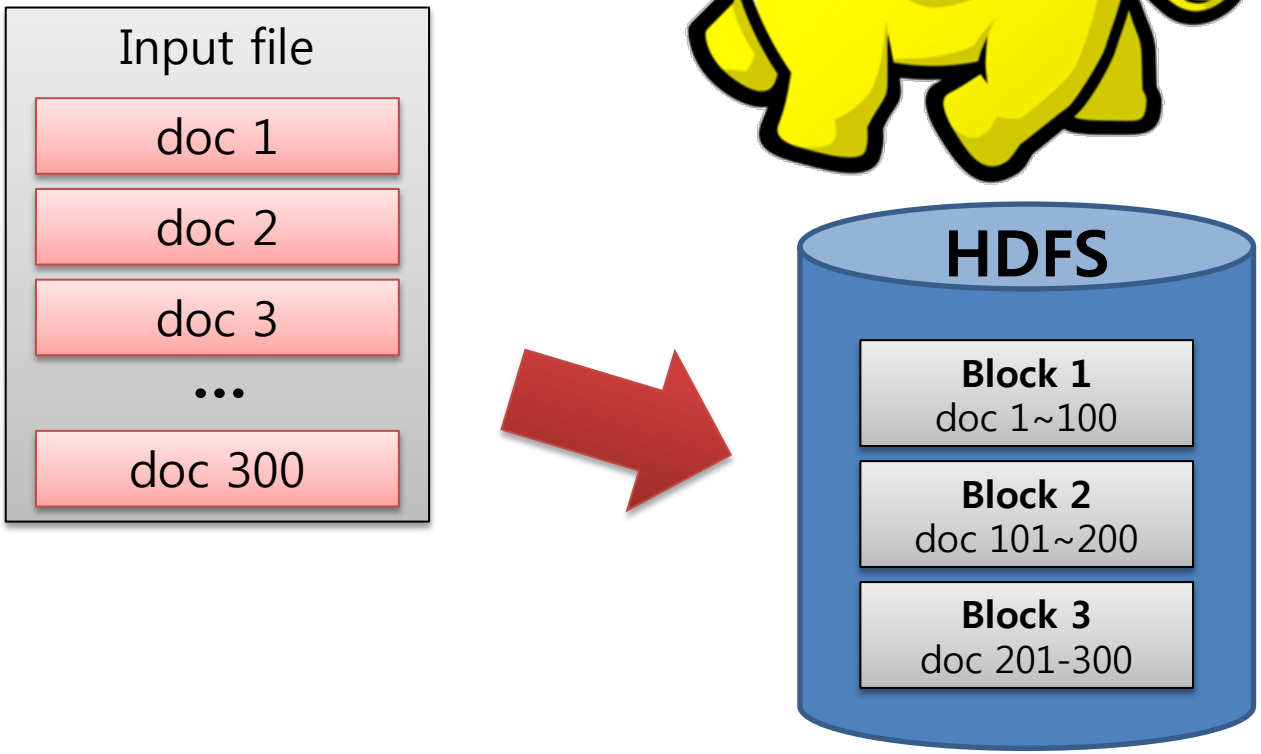
<Doug Cutting>

- A top-level **Apache** project
- Derived from **Google's MapReduce**
- Written in **Java**
- Supported by **Yahoo**!

How to parallelize computations

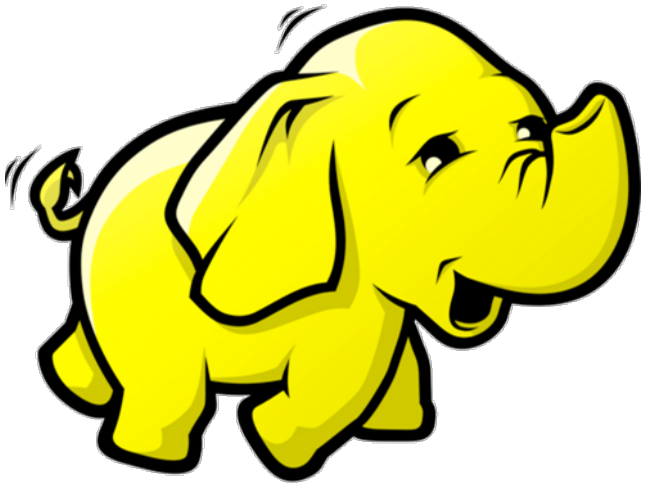
(1) **Split** the **file** into multiple smaller **blocks**
and **store** the **blocks** in a **Hadoop distributed file system (HDFS)**

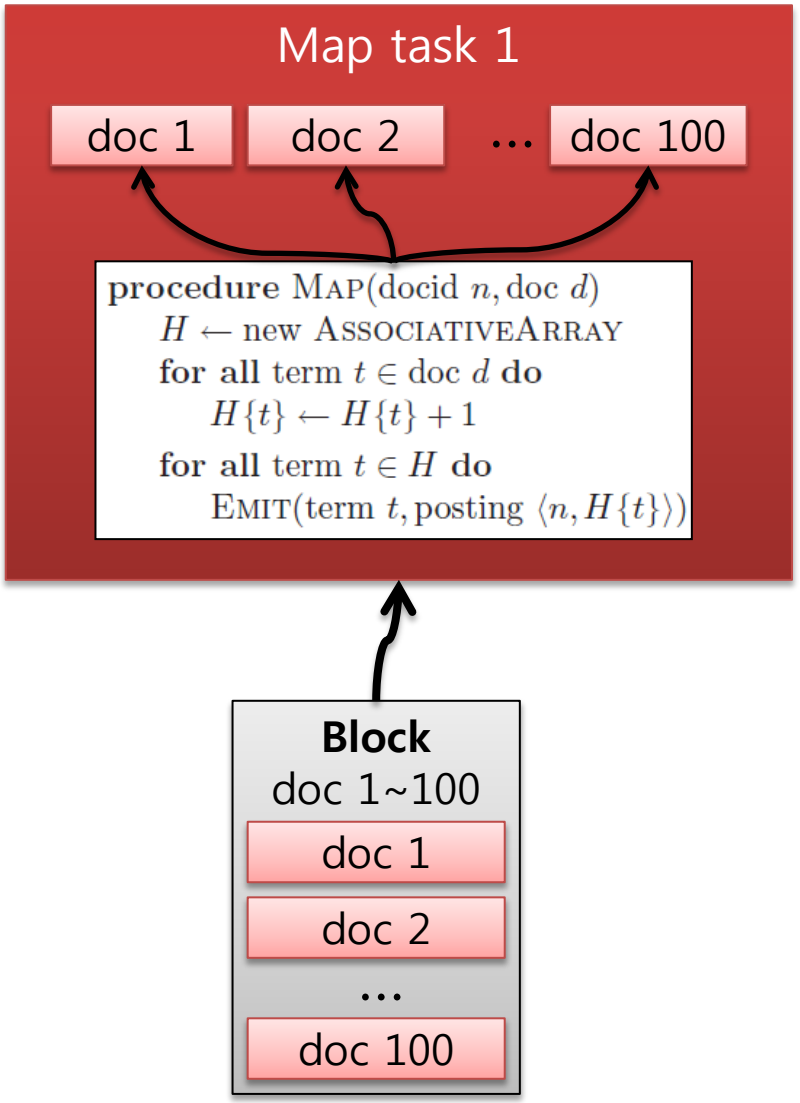




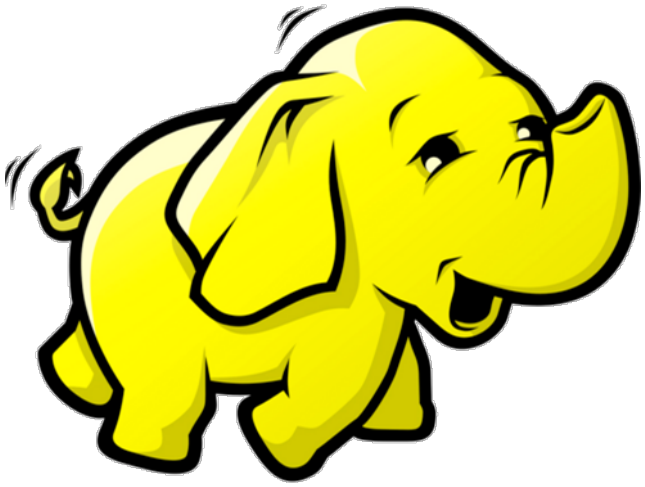
How to parallelize computations

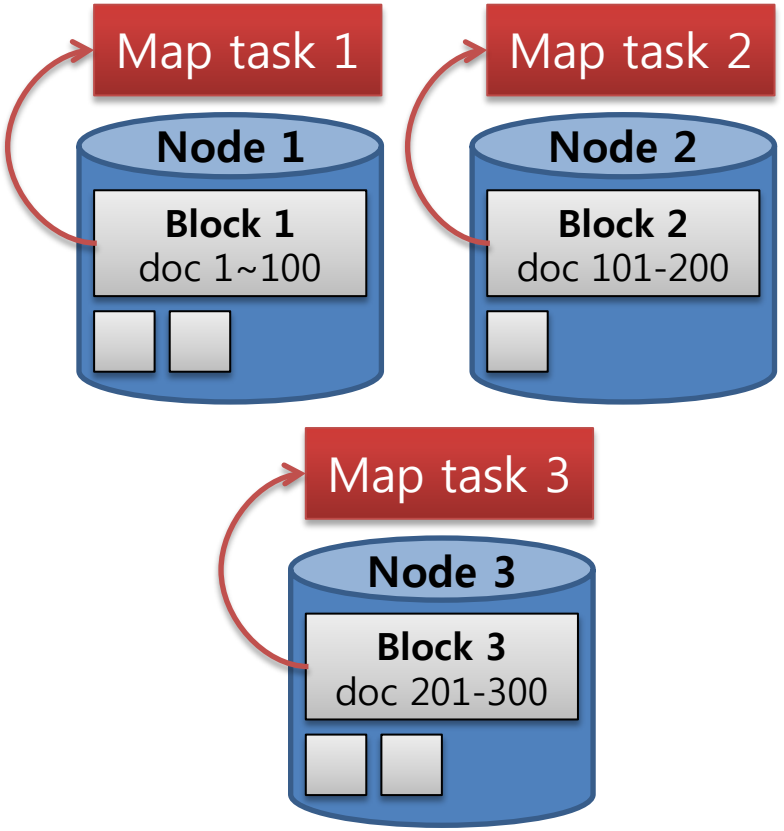
(2) # of map tasks = # of data blocks





How to parallelize computations
(2) # of map tasks = # of data blocks

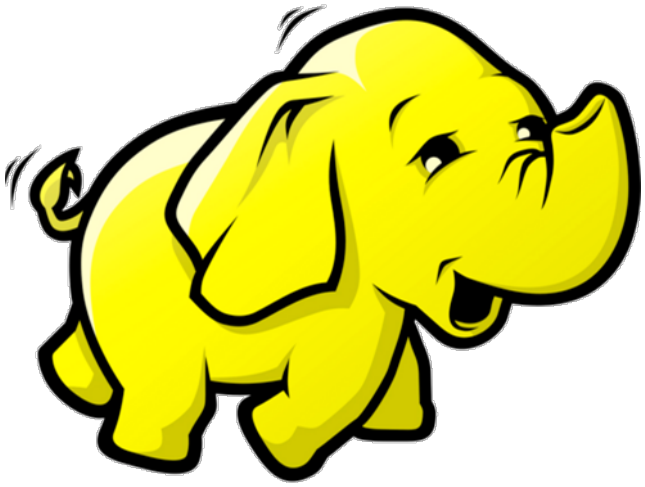


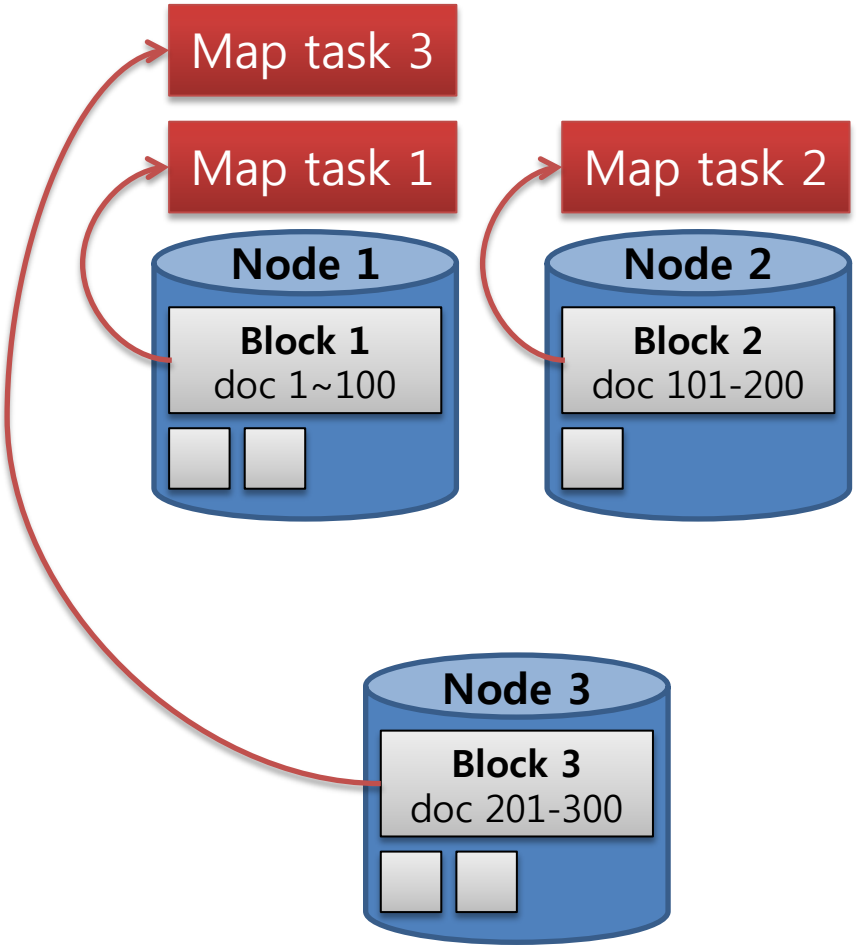


- Data-local = 3
- Non data-local = 0

How to parallelize computations

(2) # of map tasks = # of data blocks



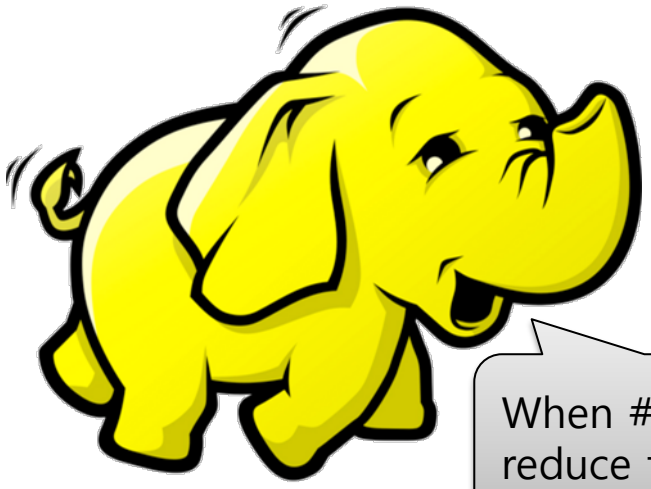


- Data-local = 2
- **Non data-local = 1**

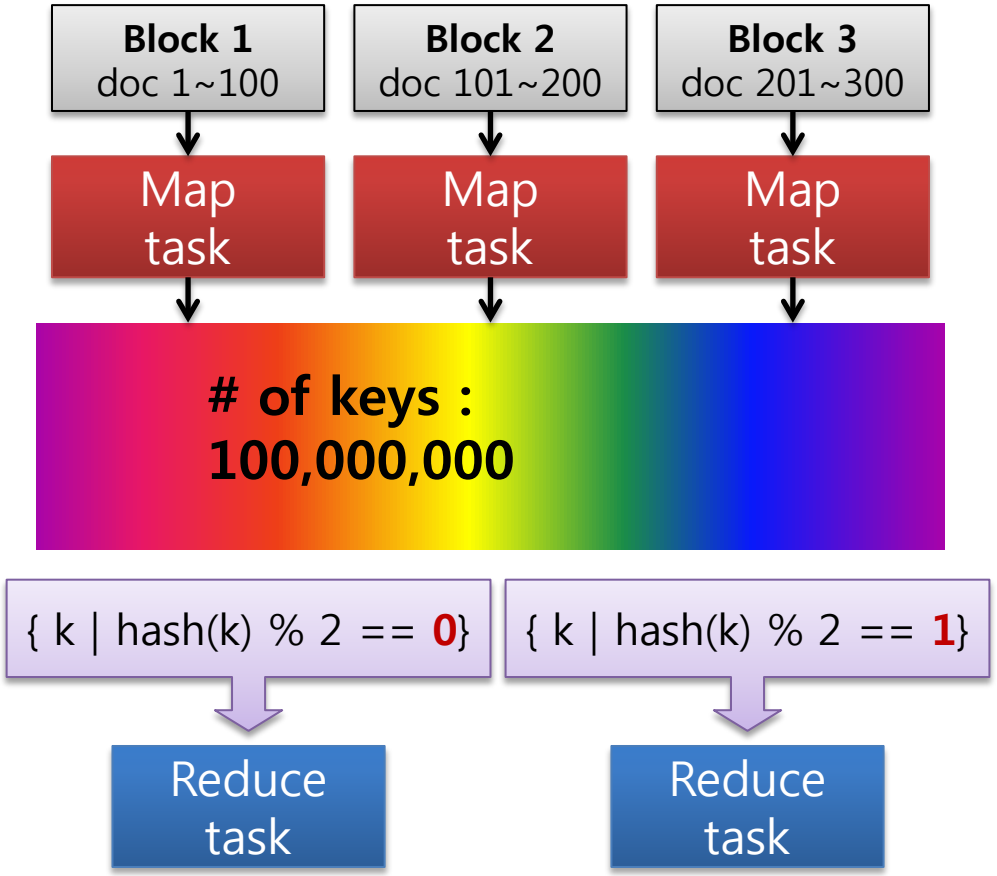
How to parallelize computations

(3) # of reduce tasks = <user parameter> = # of output files

```
procedure REDUCE(term t, postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...])
    P ← new LIST
    for all posting ⟨a, f⟩ ∈ postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...] do
        APPEND(P, ⟨a, f⟩)
    SORT(P)
    EMIT(term t, postings P)
```



When # of reduce tasks = 2



Contents

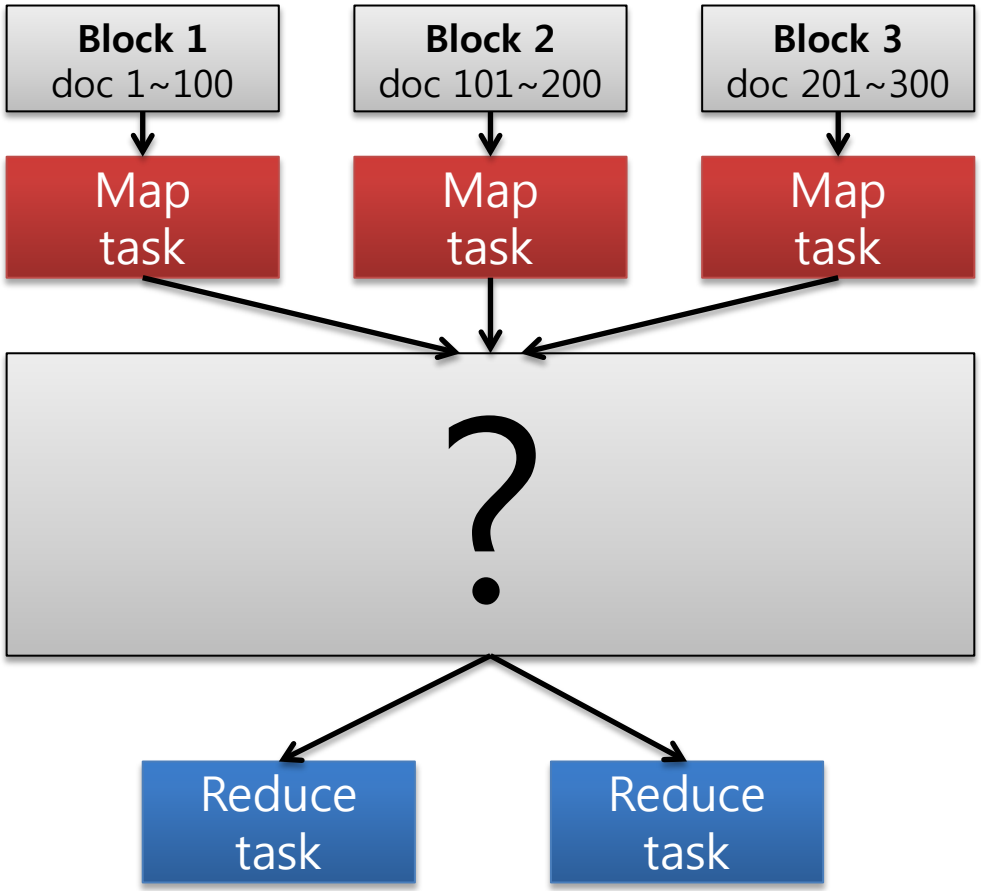
1) MapReduce
   - What is MapReduce?
     - Programming model
     - Runtime system
   - Example – building an inverted index
2) Hadoop, a representative MapReduce runtime implementation
   - How to parallelize computations
   - How to sort & group intermediate pairs
   - How to handle machine failures

How to sort & group intermediate pairs?

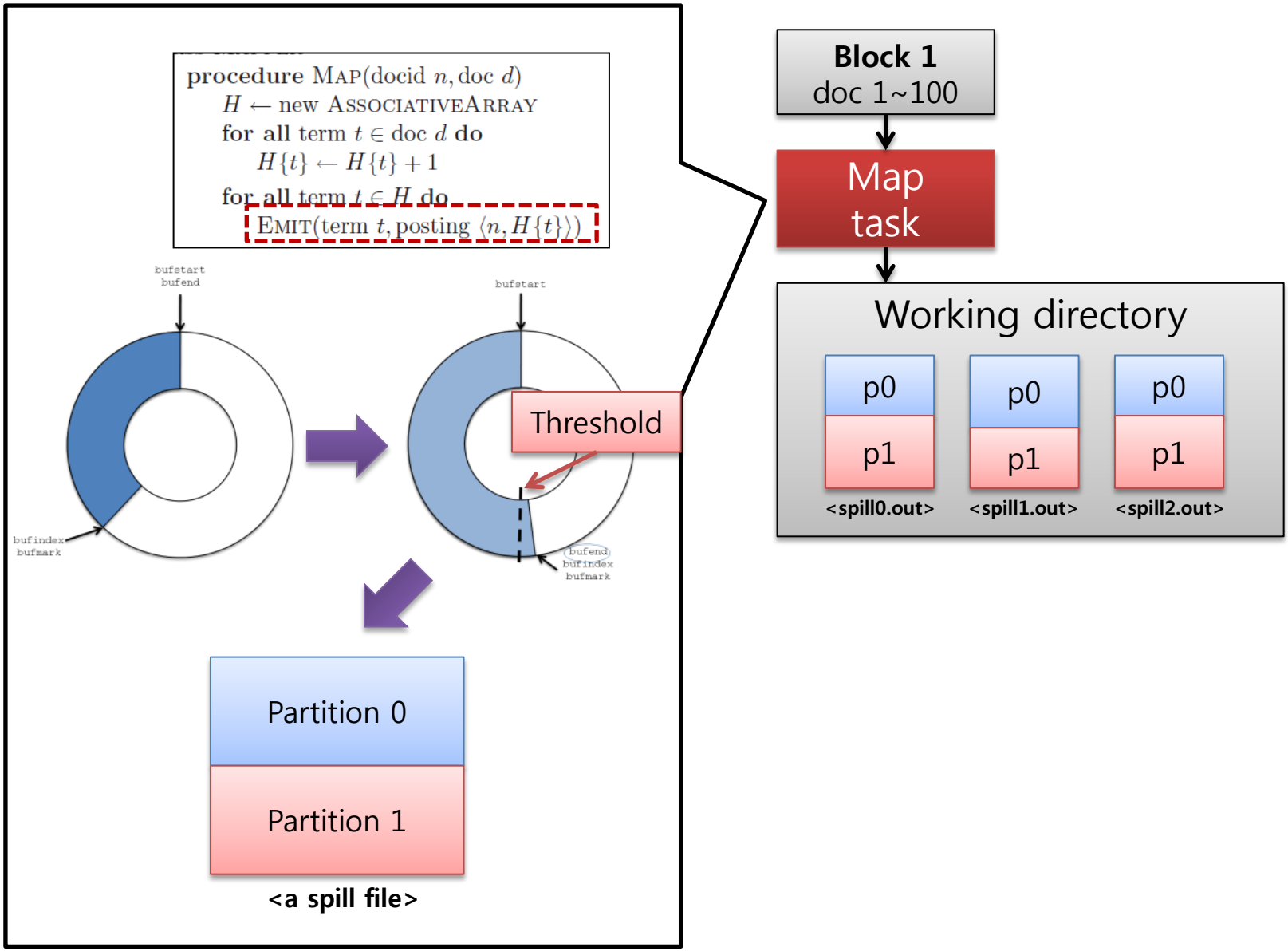
```
procedure MAP(docid n, doc d)
    H ← new ASSOCIATIVEARRAY
    for all term t ∈ doc d do
        H{t} ← H{t} + 1
    for all term t ∈ H do
        EMIT(term t, posting ⟨n, H{t}⟩)
```

??

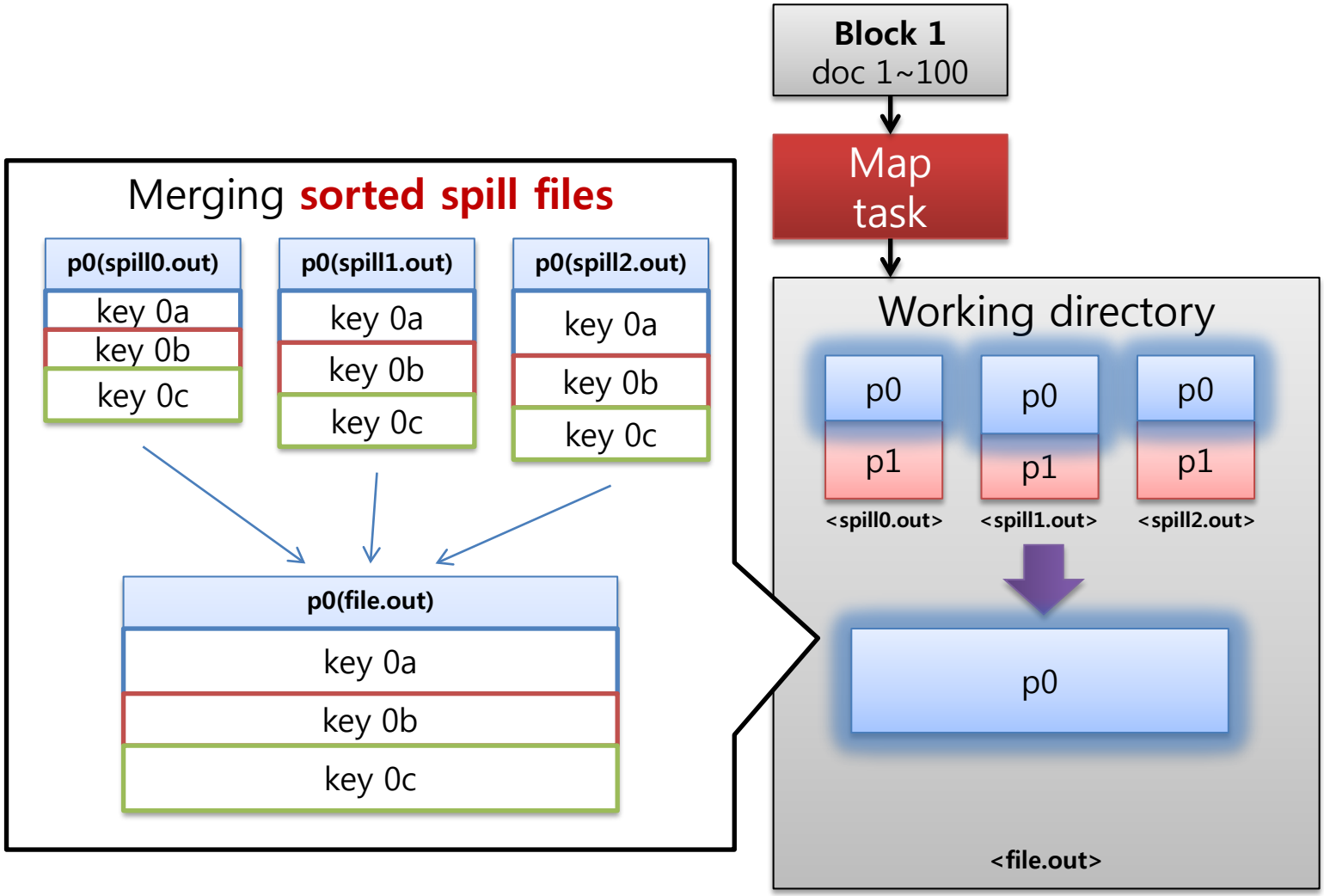
```
procedure REDUCE(term t, postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...])
    P ← new LIST
    for all posting ⟨a, f⟩ ∈ postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...] do
        APPEND(P, ⟨a, f⟩)
    SORT(P)
    EMIT(term t, postings P)
```



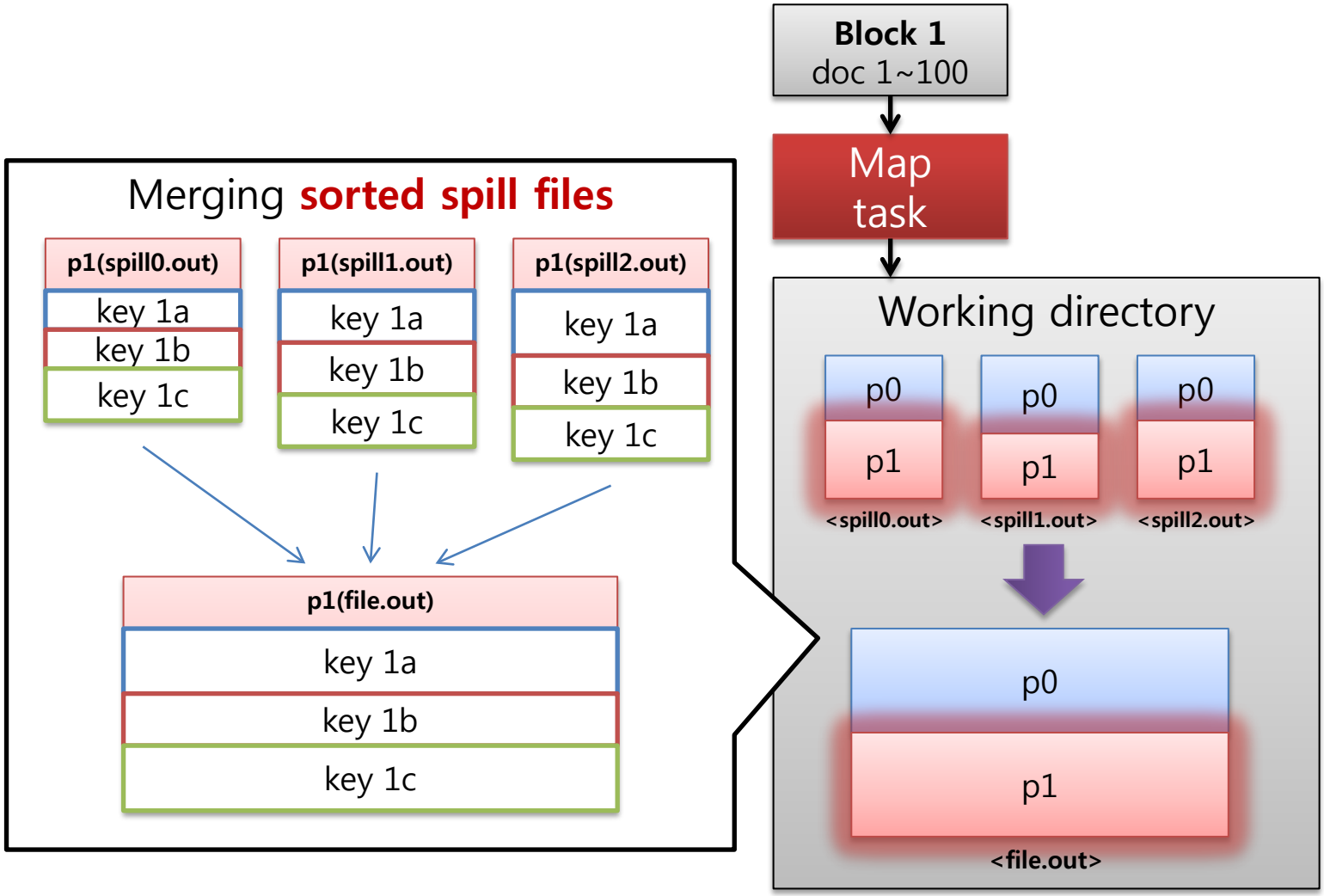
How to sort & group intermediate pairs?
(1) **Sort** and **spill** intermediate pairs in a buffer



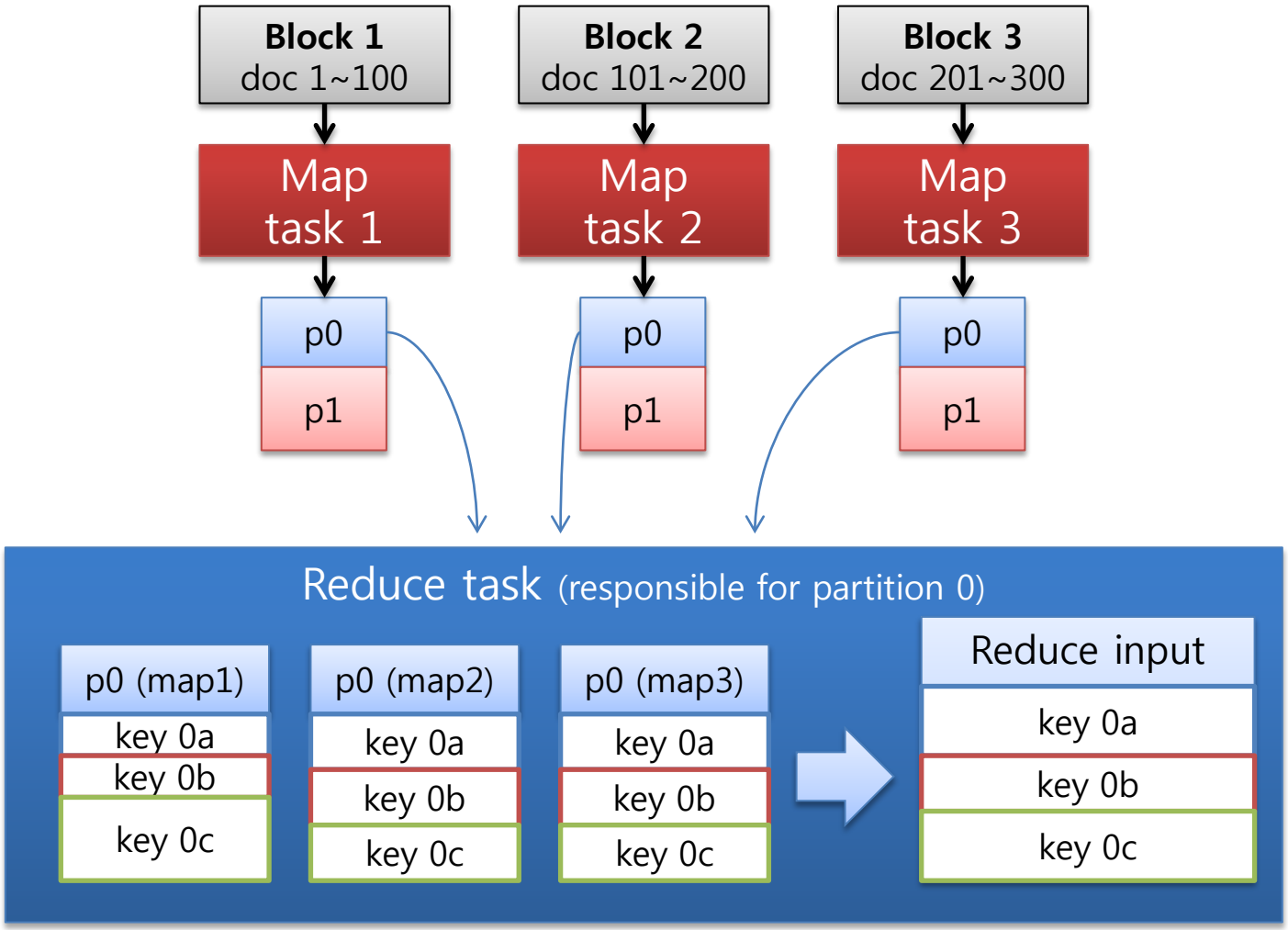
How to sort & group intermediate pairs?
(2) **Merge** local spill files of a map task



How to sort & group intermediate pairs?
(2) **Merge** local spill files of a map task



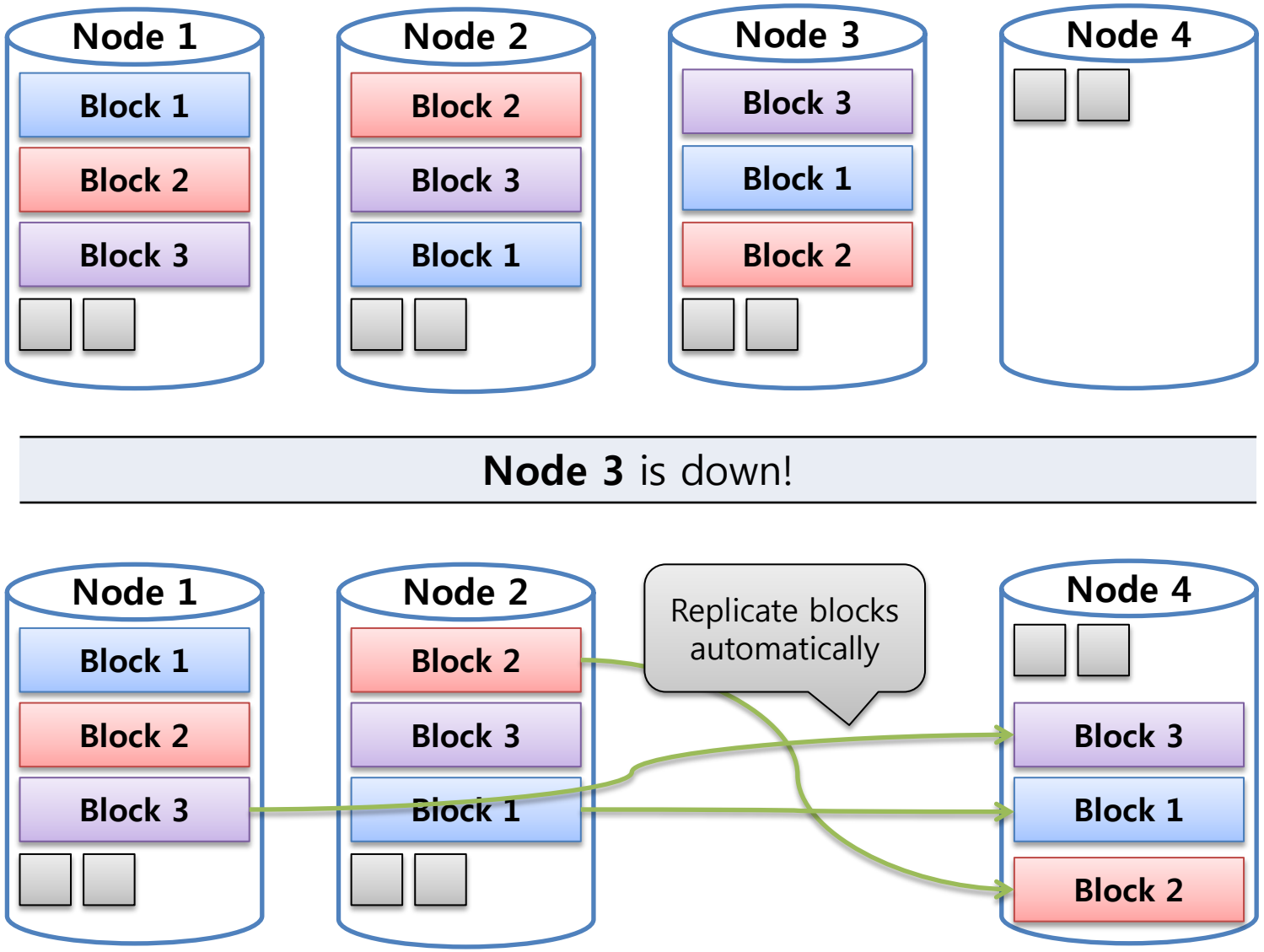
How to sort & group intermediate pairs?
(3) Reduce tasks **ask** map tasks **for their portions**



Contents

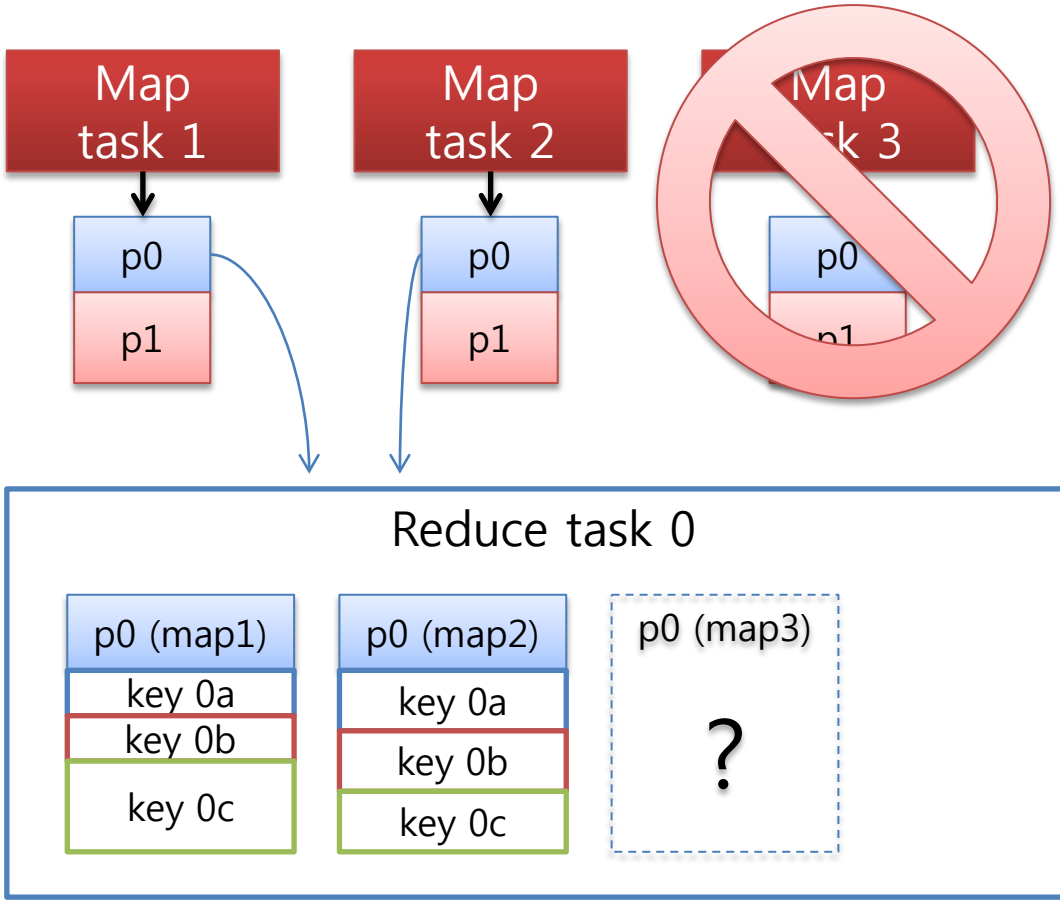
1) MapReduce
   - What is MapReduce?
     - Programming model
     - Runtime system
   - Example – building an inverted index
2) Hadoop, a representative MapReduce runtime implementation
   - How to parallelize computations
   - How to sort & group intermediate pairs
   - How to handle machine failures

How to handle machine failures?
(a) Block replication (default : 3)

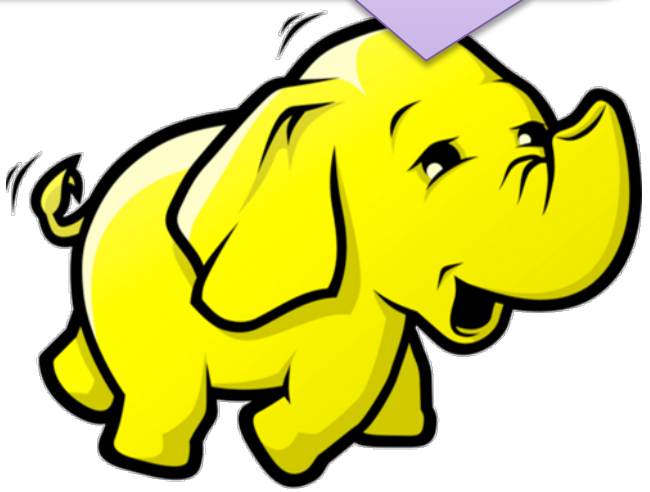


How to handle machine failures?
(b) Task rescheduling

- **A reduce task** should download its input from **all map output files**
  - before applying a user-defined reduce function
- What if **an map output file is unavailable** due to **machine failure**?



- Reschedule **map task 3** on another node!
- Don't worry about **block 3**!
- Do not need to restart the job from scratch!



Summary

- Users specify the computation in terms of a **map** and a **reduce** function
- **MapReduce** runtime system automatically
  - + **parallelizes** the computation across large-scale clusters of machines
  - + **sort & group** intermediate pairs
  - + handles machine **failures**
- Hadoop is a representative MapReduce runtime system

```
procedure MAP(docid n, doc d)
    H ← new ASSOCIATIVEARRAY
    for all term t ∈ doc d do
        H{t} ← H{t} + 1
    for all term t ∈ H do
        EMIT(term t, posting ⟨n, H{t}⟩)
```

```
procedure REDUCE(term t, postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...])
    P ← new LIST
    for all posting ⟨a, f⟩ ∈ postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...] do
        APPEND(P, ⟨a, f⟩)
    SORT(P)
    EMIT(term t, postings P)
```