# Hadoop: A Distributed Architecture, FileSystem, & MapReduce

컴퓨터AI공학부
천세진

# 빅데이터란?

데이터의 모델과 요약의 **일반화**
**(Generalizations)**

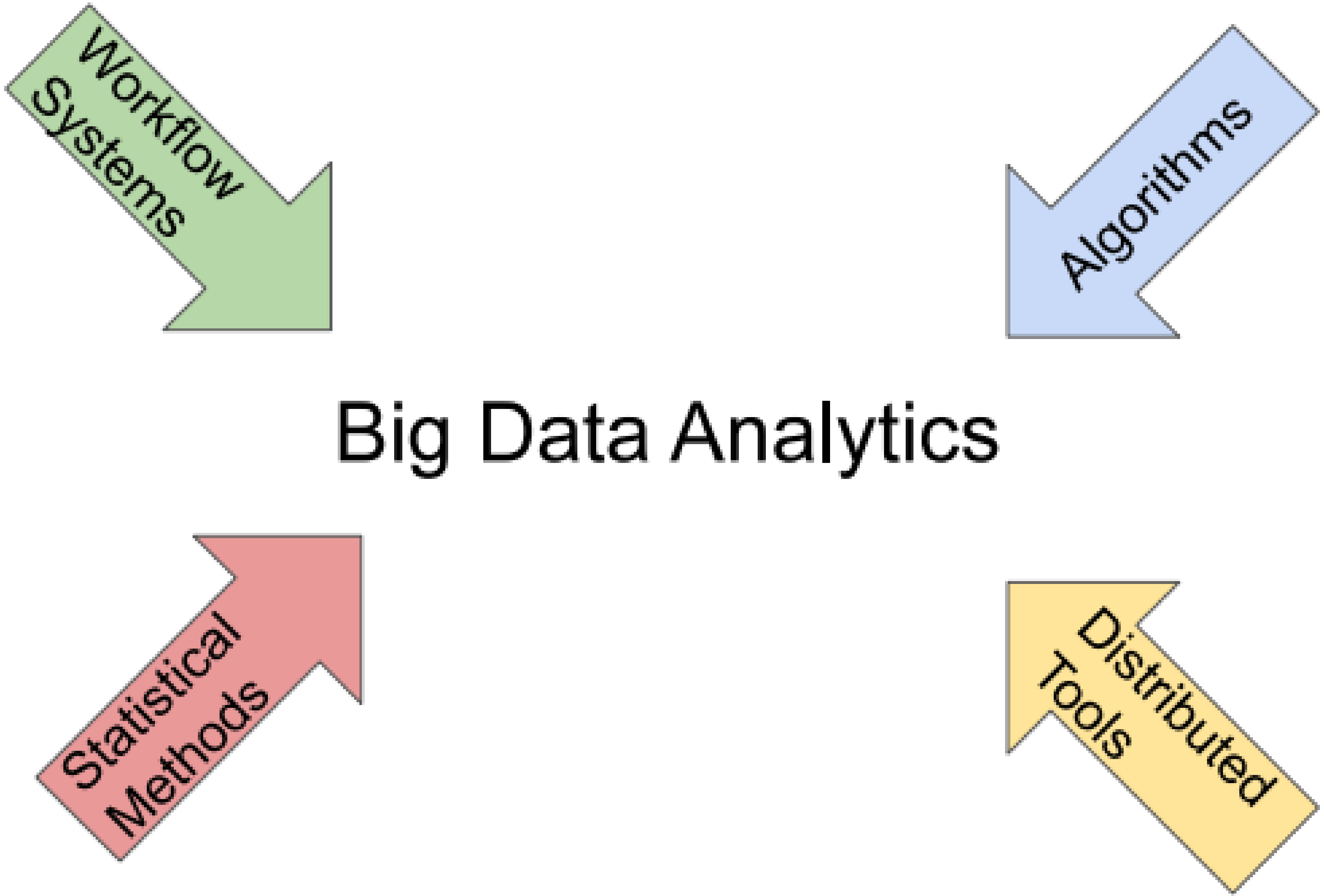
**Data frameworks**

**Hadoop File System**
**Spark**
**Streaming**
**MapReduce**
**Tensorflow**

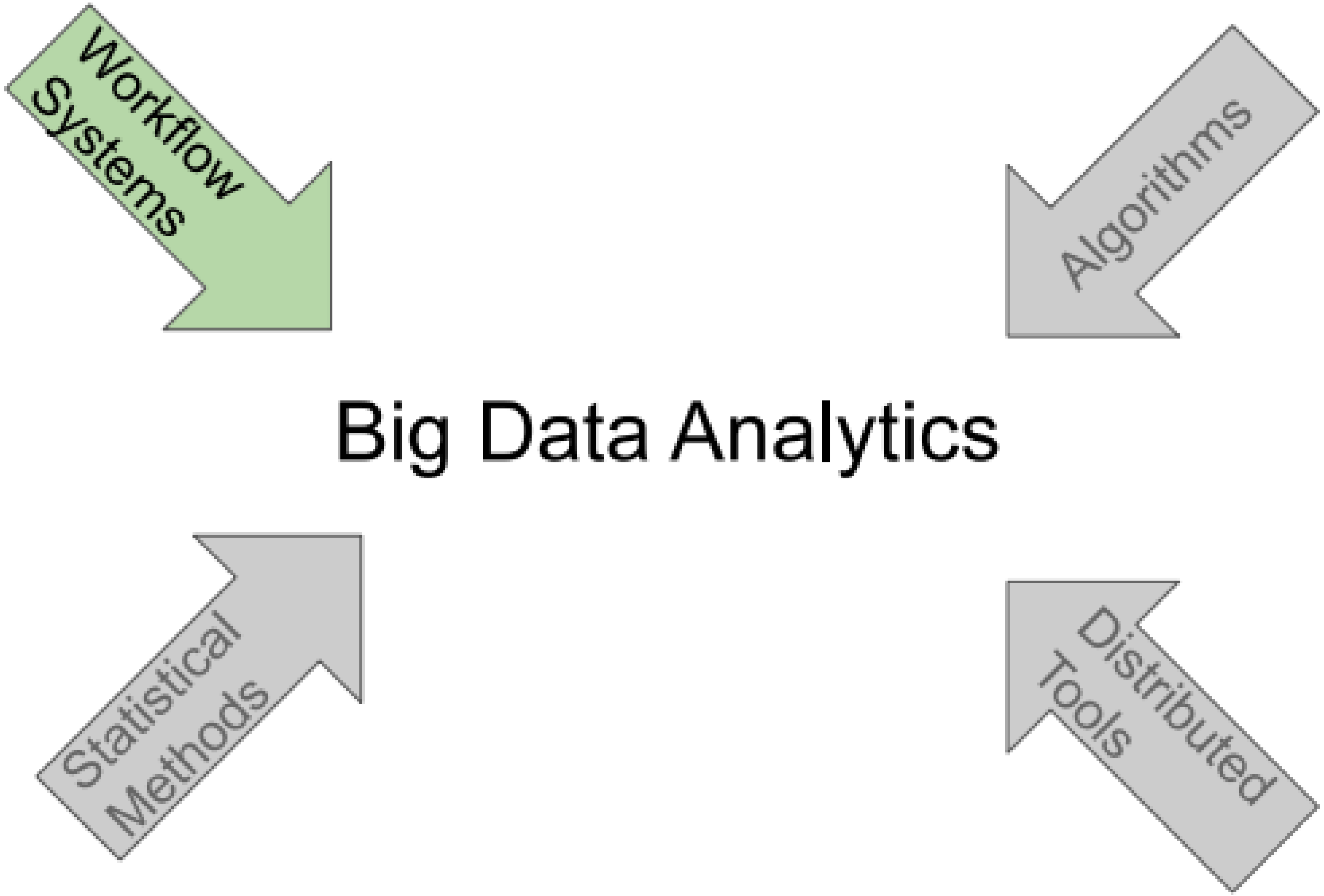
**Algorithms and Analyses**

**Similarity Search**
**Linear Modeling**
**Recommendation Systems**
**Graph Analysis**
**Deep Learning**

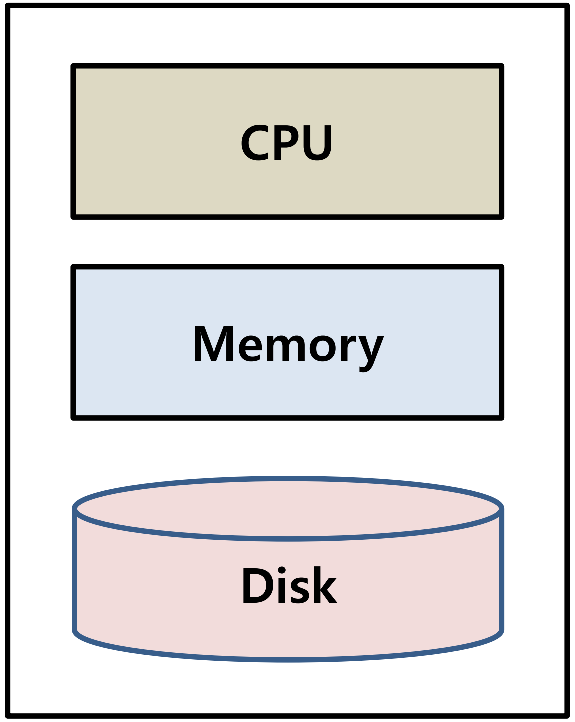
# 빅데이터 분석, 종류(Class)



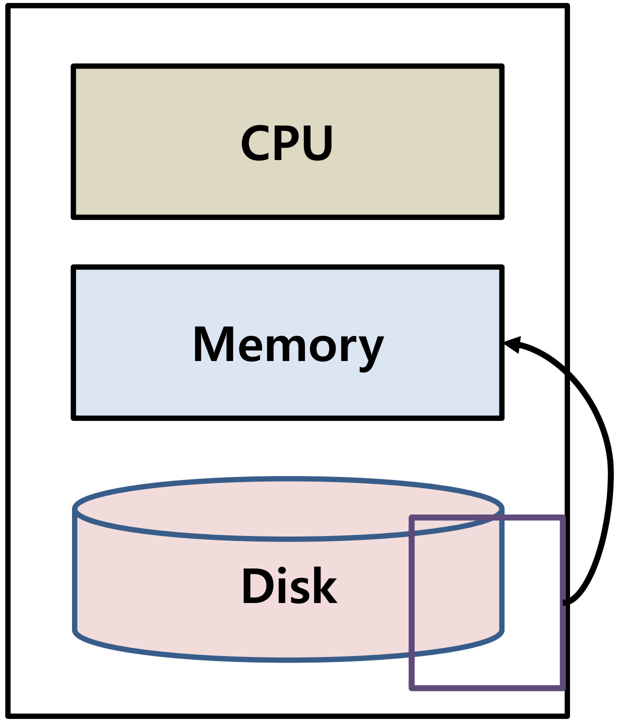
# 빅데이터 분석, 종류(Class)



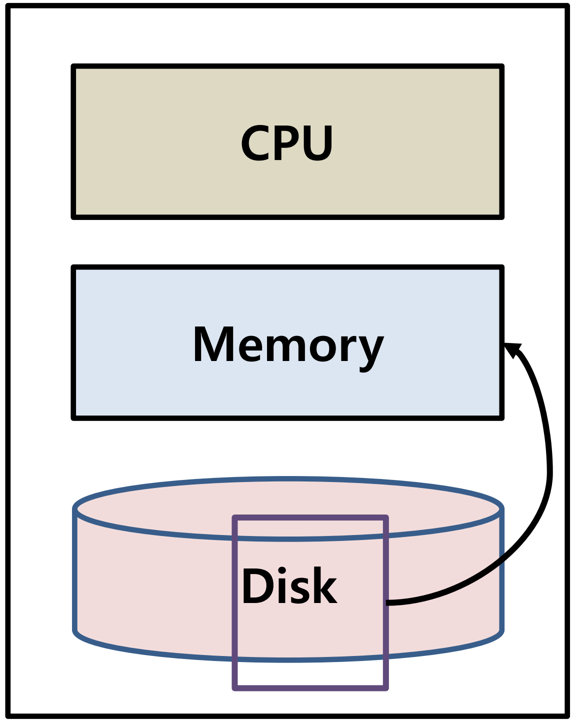
# Classical Data Mining



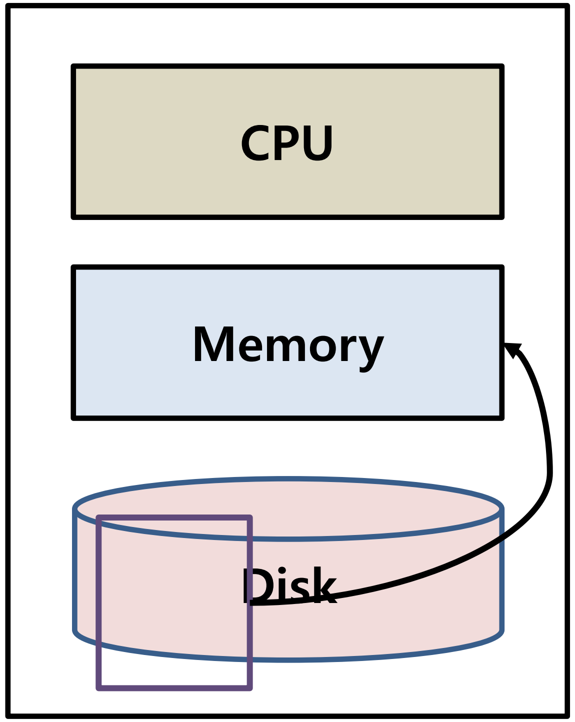
# Classical Data Mining



# Classical Data Mining

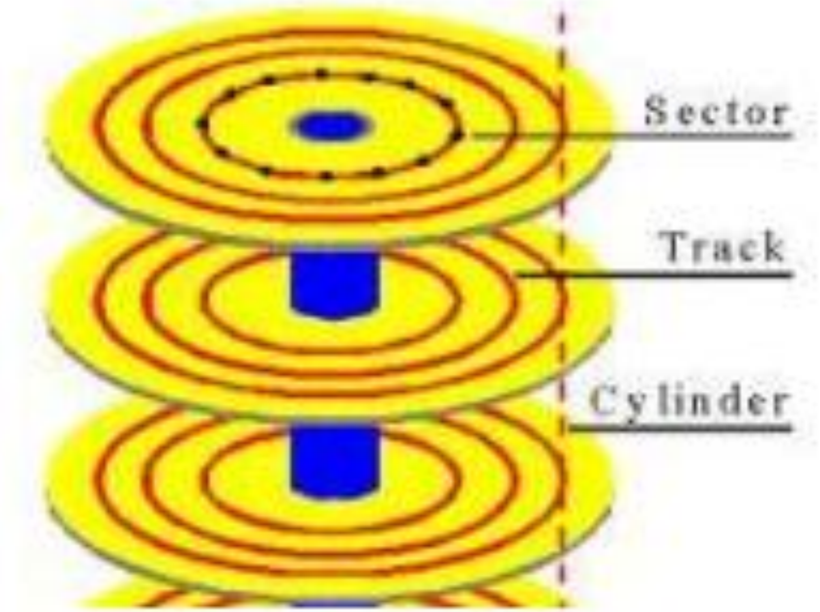


# Classical Data Mining



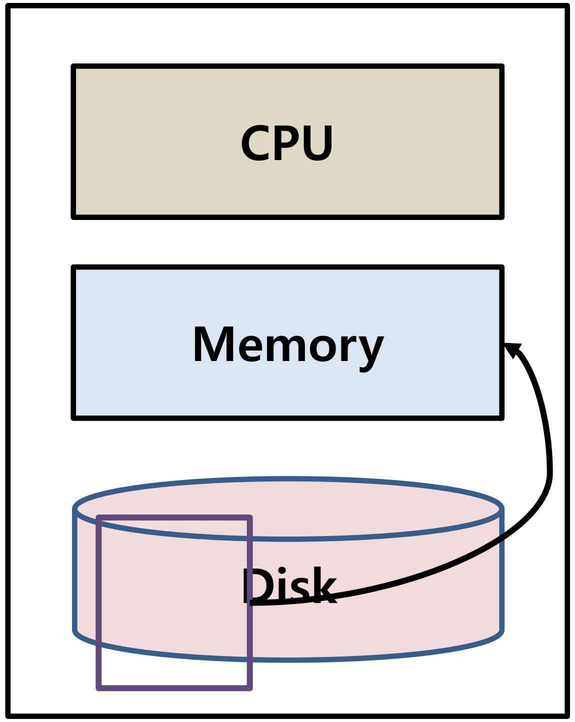
# IO Bounded

- 10^5 차이 단어 읽기: 디스크로부터 vs 메인 메모리 부터
- IO Bound: 디스크에 읽기/쓰기를 진행하는 것은 가장 큰 성능 저하
- IO Bounded: 가장 큰 성능 보틀넥(bottleneck)은 디스크 읽기/쓰기




- 100GBs: ~10분 정도
- 200TBs: ~20,000 분 = 13일

# Classical Big Data



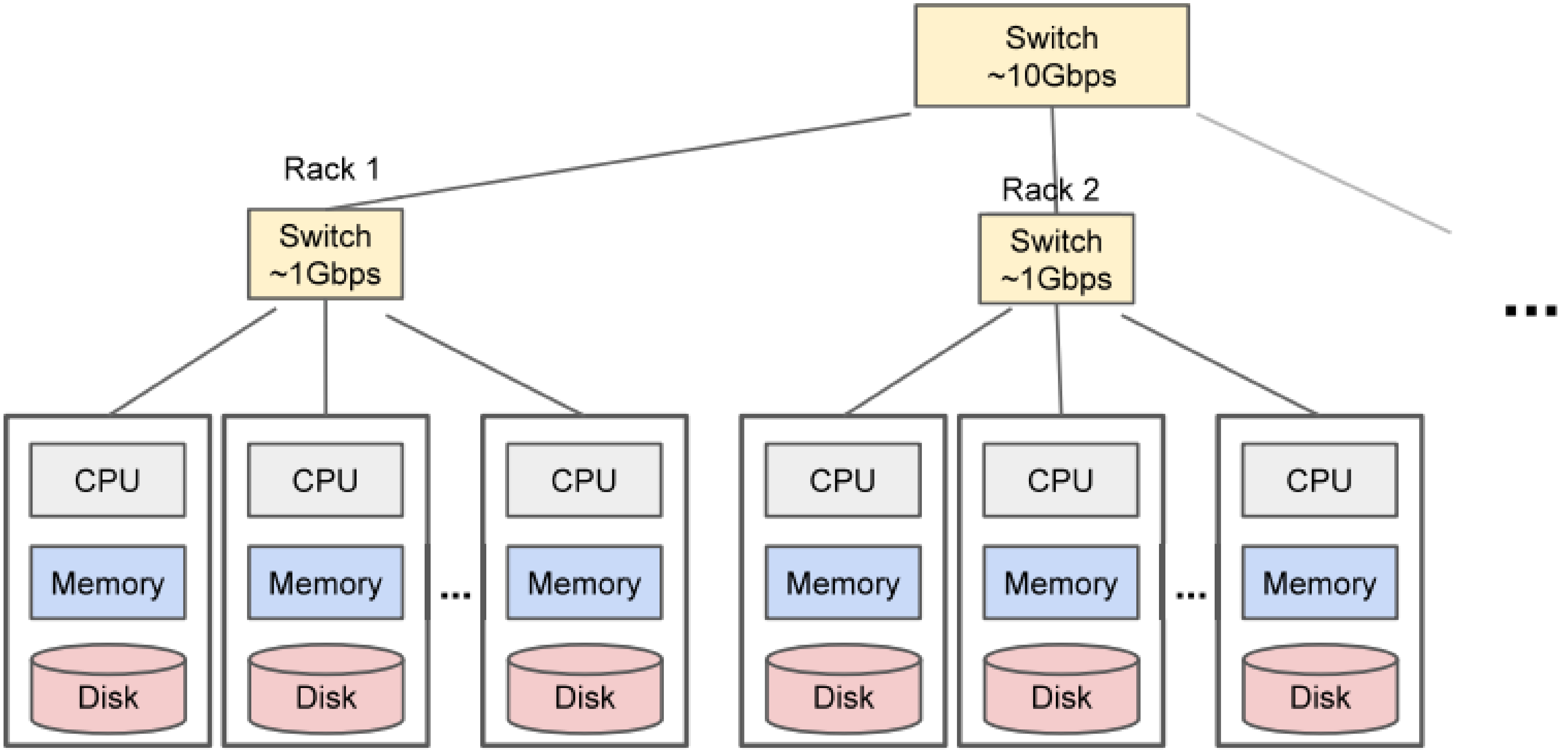

**Classical focus: 디스크의 효율적인 사용,**
**Eg. Apache Lucene / Solr**

**Classical limitation:**
**매우 큰 파일의 전부를 처리할 때**
**여전히 bounded 된다**

**How to solve?**

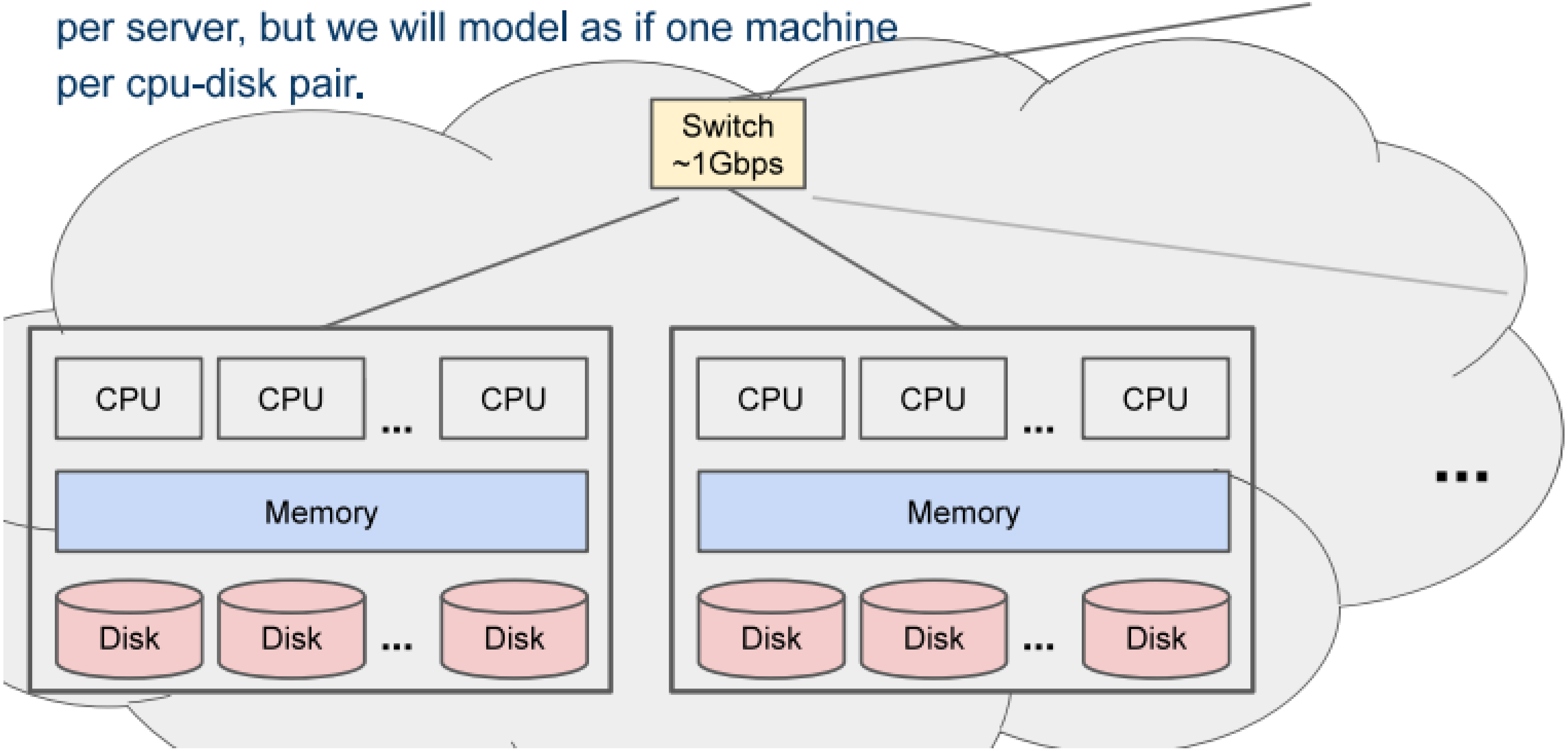
**Classical limitation:**
**매우 큰 파일의 전부를 처리할 때**
**여전히 bounded 된다**

# Distributed Architecture

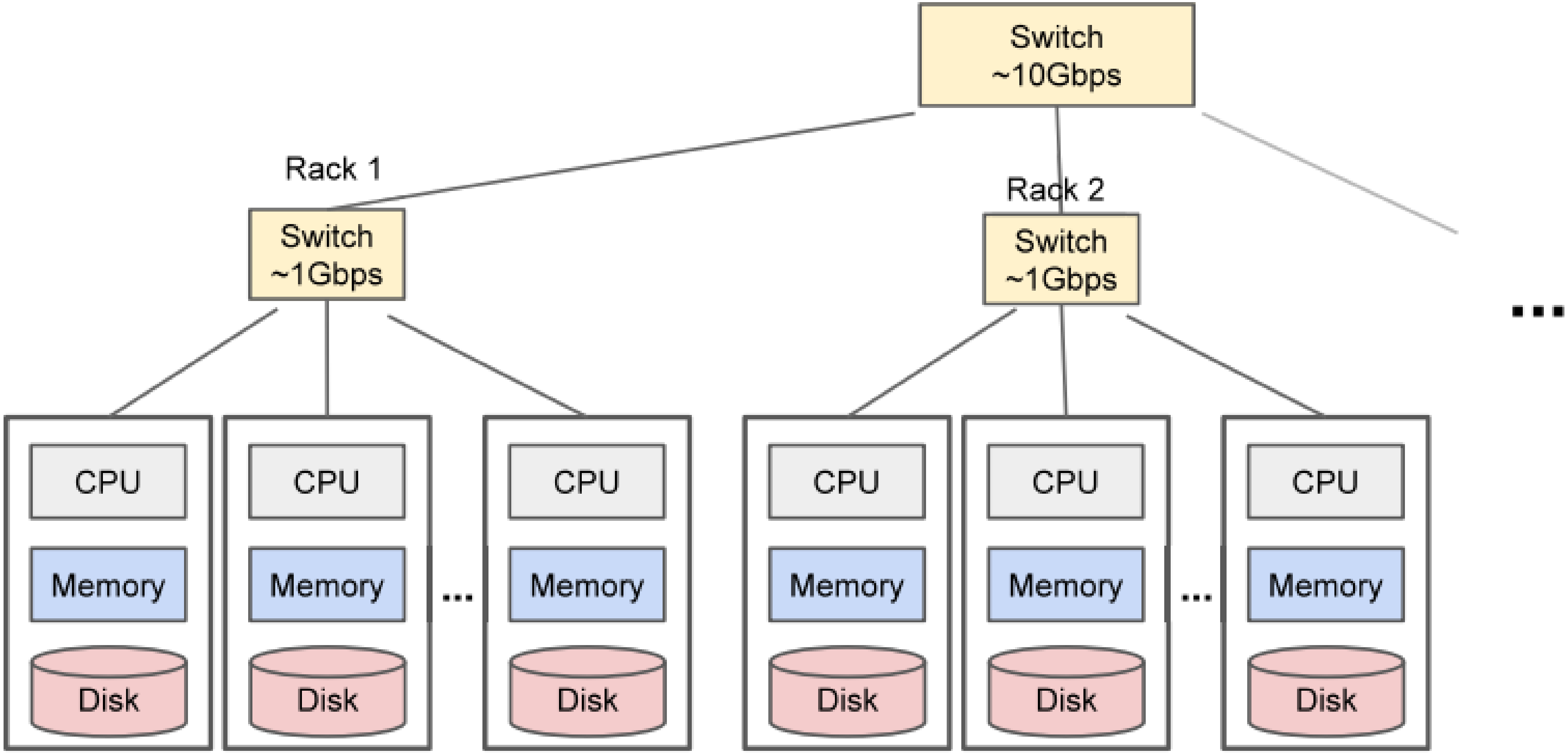


# Distributed Architecture

In reality, modern setups often have multiple cpus and disks per server, but we will model as if one machine per cpu-disk pair.



# Distributed Architecture (Cluster)



# Distributed Architecture (Cluster)

- Challenges for IO Cluster Computing

1. Nodes fail
   1 in 1000 nodes fail a day

2. Network is a bottleneck
   Typically 1-10 Gb/s throughput

3. Traditional distributed programming is often ad-hoc and complicated

# Distributed Architecture (Cluster)

## Challenges for IO Cluster Computing

1. Nodes fail
   1 in 1000 nodes fail a day
   1. Duplicate Data
2. Network is a bottleneck
   Typically 1-10 Gb/s throughput
   1. Bring computation to nodes, rather than data to nodes
3. Traditional distributed programming is often ad-hoc and complicated
   1. Stipulate a programming system that can easily be distributed

# Distributed Architecture (Cluster)

- Challenges for IO Cluster Computing

1. Nodes fail
   1 in 1000 nodes fail a day
   1. Duplicate Data
2. Network is a bottleneck
   Typically 1-10 Gb/s throughput
   1. Bring computation to nodes, rather than data to nodes
3. Traditional distributed programming is often ad-hoc and complicated
   1. Stipulate a programming system that can easily be distributed

**HDFS/ MapReduce Accomplishes**

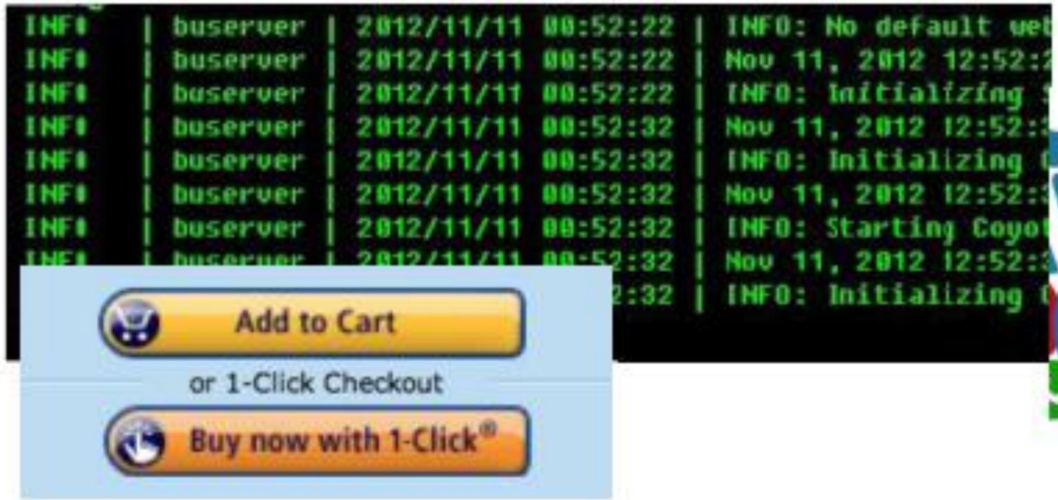
# Distributed Filesystem

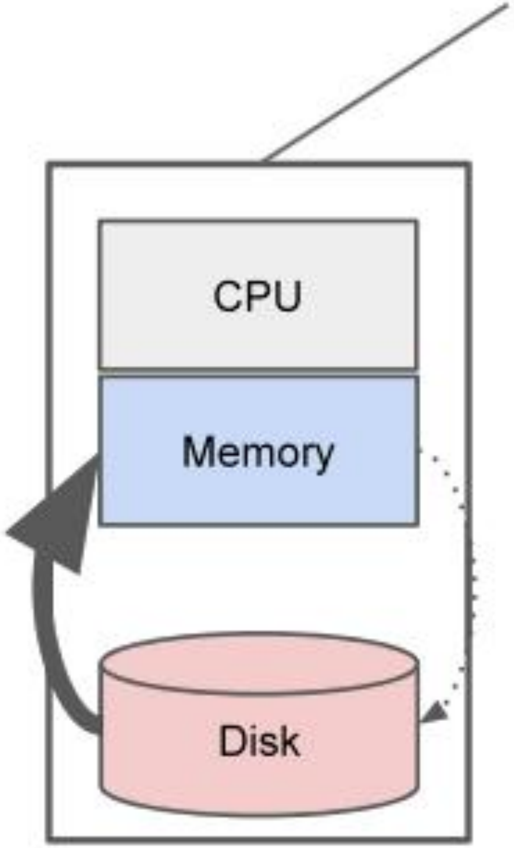
- MapReduce의 효율성은 분산 파일시스템의 특징을 사용함으로써 간단하게 성취된다

# Distributed Filesystem

## Characteristics for Big Data Tasks

- Large files (i.e. >100 GB to TBs)
- Reads are most common
- No need to update in place (append preferred)

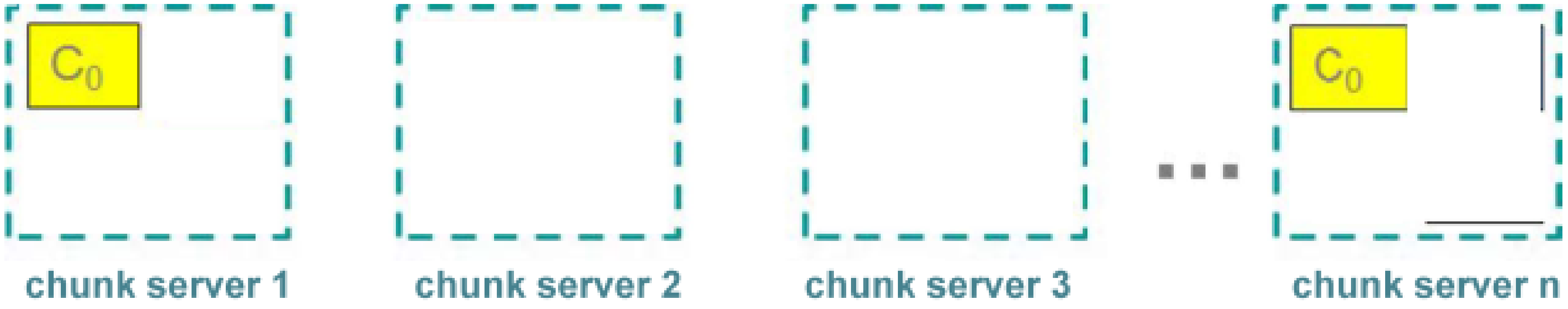


# Distributed Filesystem

(e.g. Apache HadoopDFS, GoogleFS, EMRFS)

C, D: Two different files

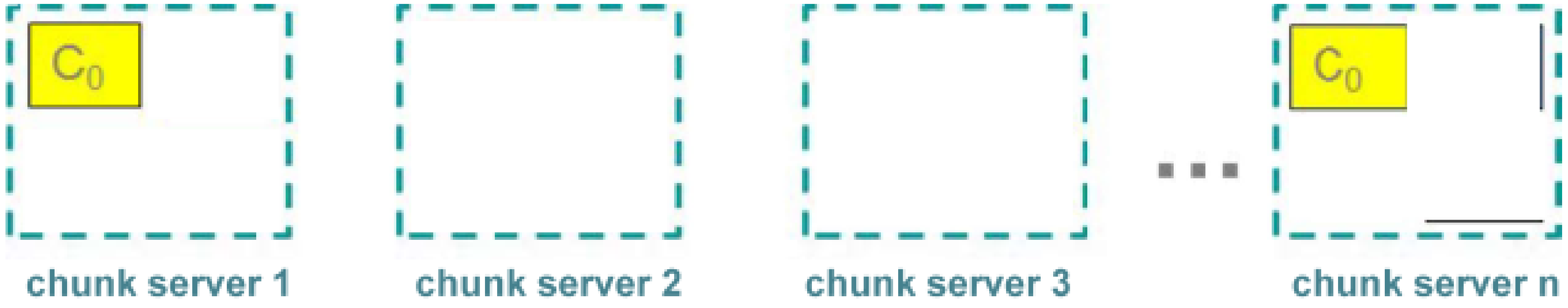



(Leskovec at al., 2014; http://www.mmds.org/)

# Distributed Filesystem

(e.g. Apache HadoopDFS, GoogleFS, EMRFS)

C, D: Two different files

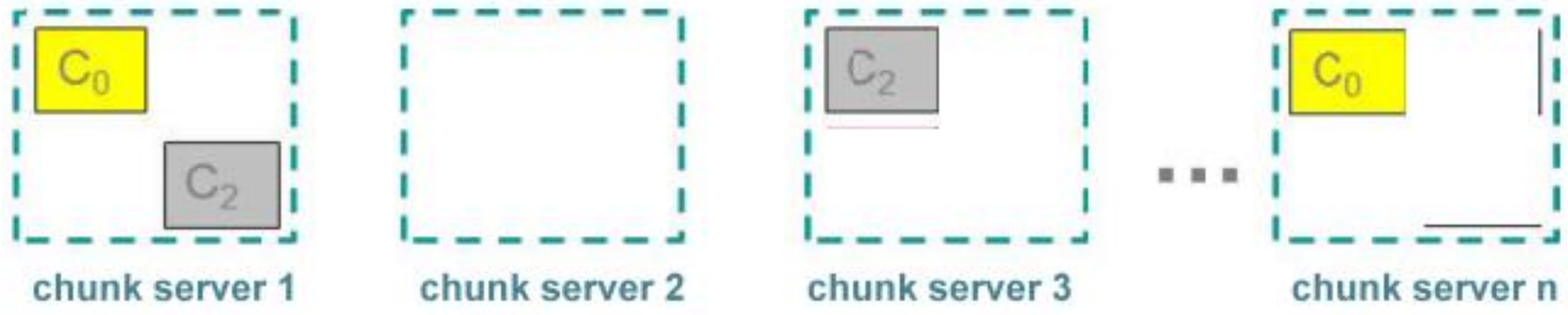



(Leskovec at al., 2014; http://www.mmds.org/)

# Distributed Filesystem

(e.g. Apache HadoopDFS, GoogleFS, EMRFS)

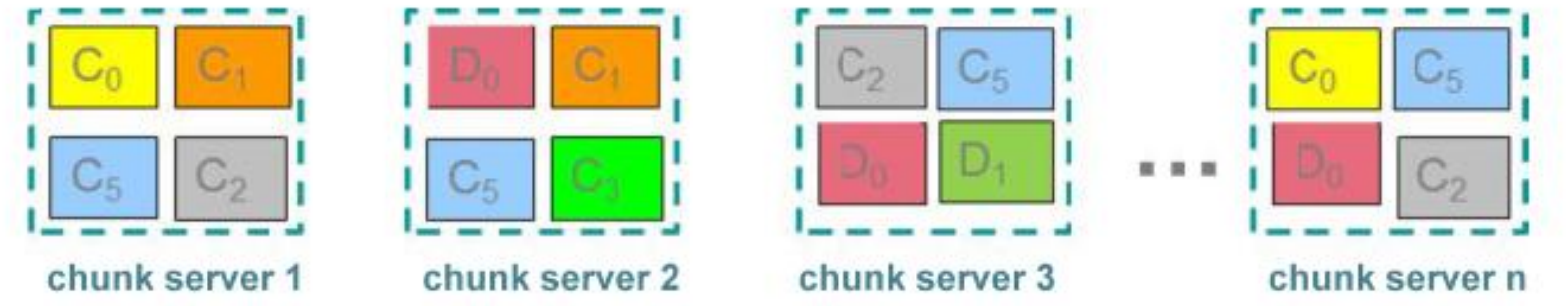
C, D: Two different files



# Distributed Filesystem

(e.g. Apache HadoopDFS, GoogleFS, EMRFS)

C, D: Two different files



# Components of a Distributed Filesystem

## Chunk servers (on Data Nodes)

File is split into contiguous chunks

Typically each chunk is 16-64MB

Each chunk replicated (usually 2x or 3x)

Try to keep replicas in different racks

# Components of a Distributed Filesystem

## Chunk servers (on Data Nodes)

- File is split into contiguous chunks
- Typically each chunk is 16-64MB
- Each chunk replicated (usually 2x or 3x)
- Try to keep replicas in different racks

## Name node (aka master node)

- Stores metadata about where files are stored
- Might be replicated or distributed across data nodes.

# Components of a Distributed Filesystem

## Chunk servers (on Data Nodes)

File is split into contiguous chunks

Typically each chunk is 16-64MB

Each chunk replicated (usually 2x or 3x)

Try to keep replicas in different racks

## Name node (aka master node)

Stores metadata about where files are stored

Might be replicated or distributed across data nodes.

## Client library for file access

Talks to master to find chunk servers

Connects directly to chunk servers to access data

# Distributed Architecture (Cluster)

## Challenges for IO Cluster Computing

1. Nodes fail
   1 in 1000 nodes fail a day
   Duplicate Data **(Distributed FS)**
2. Network is a bottleneck
   Typically 1-10 Gb/s throughput
   Bring computation to nodes, rather than data to nodes.
3. Traditional distributed programming is often ad-hoc and complicated
   Stipulate a programming system that can easily be distributed

# What is MapReduce

# What is MapReduce

***noun.1 - A style of programming***

input chunks => map tasks | group_by keys | reduce tasks => output

"|" is the linux "pipe" symbol: passes stdout from first process to stdin of next.

# What is MapReduce

***noun.1 - A style of programming***

input chunks => map tasks | group_by keys | reduce tasks => output

"|" is the linux "pipe" symbol: passes stdout from first process to stdin of next.

E.g. counting words:

```
tokenize(document) | sort | uniq -c
```

# What is MapReduce

***noun.1* - *A style of programming***

input chunks => map tasks | group_by keys | reduce tasks => output

"|" is the linux "pipe" symbol: passes stdout from first process to stdin of next.

E.g. counting words:

```
tokenize(document) | sort | uniq -c
```

***noun.2* - A *system* that distributes MapReduce style programs across a distributed file-system.**

(e.g. Google's internal "MapReduce" or apache.hadoop.mapreduce with hdfs)

# What is MapReduce

***noun.1* - *A style of programming***

input chunks => map tasks | group_by keys | reduce tasks => output

"|" is the linux "pipe" symbol: passes stdout from first process to stdin of next.

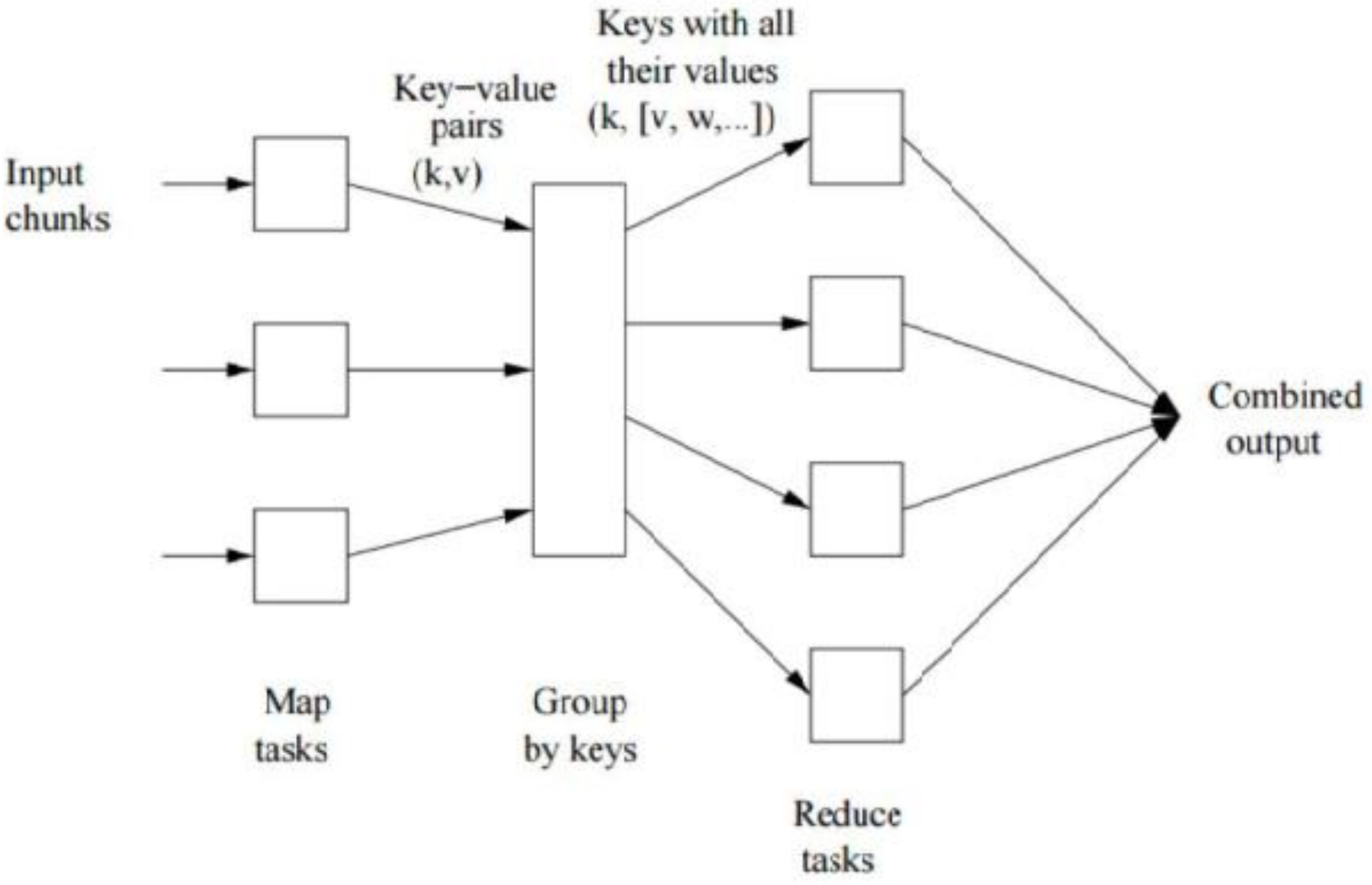
E.g. counting words:

```
tokenize(document) | sort | uniq -c
```

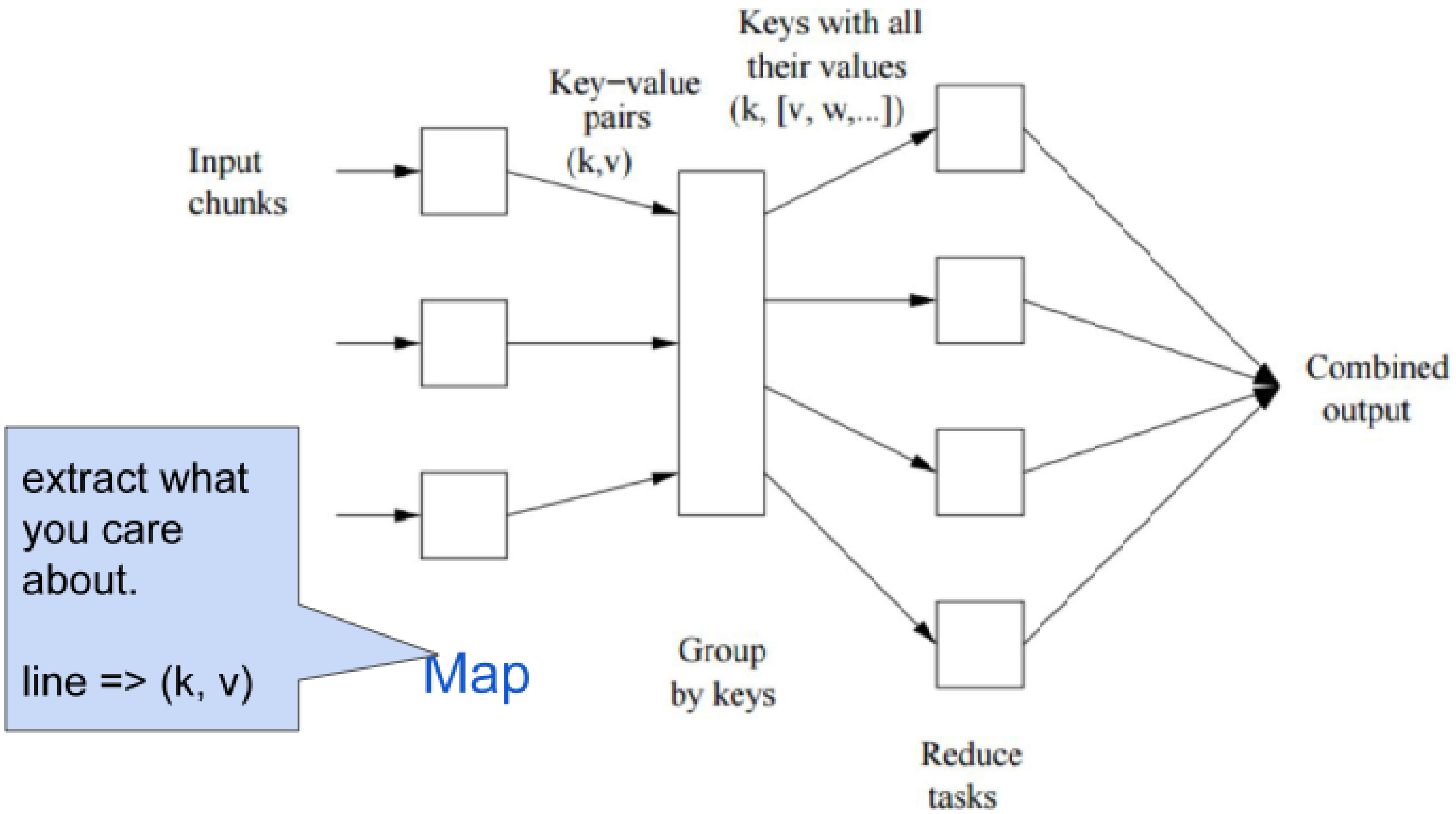
***noun.2* - A *system* that distributes MapReduce style programs across a distributed file-system.**

(e.g. Google's internal "MapReduce" or apache.hadoop.mapreduce with hdfs)

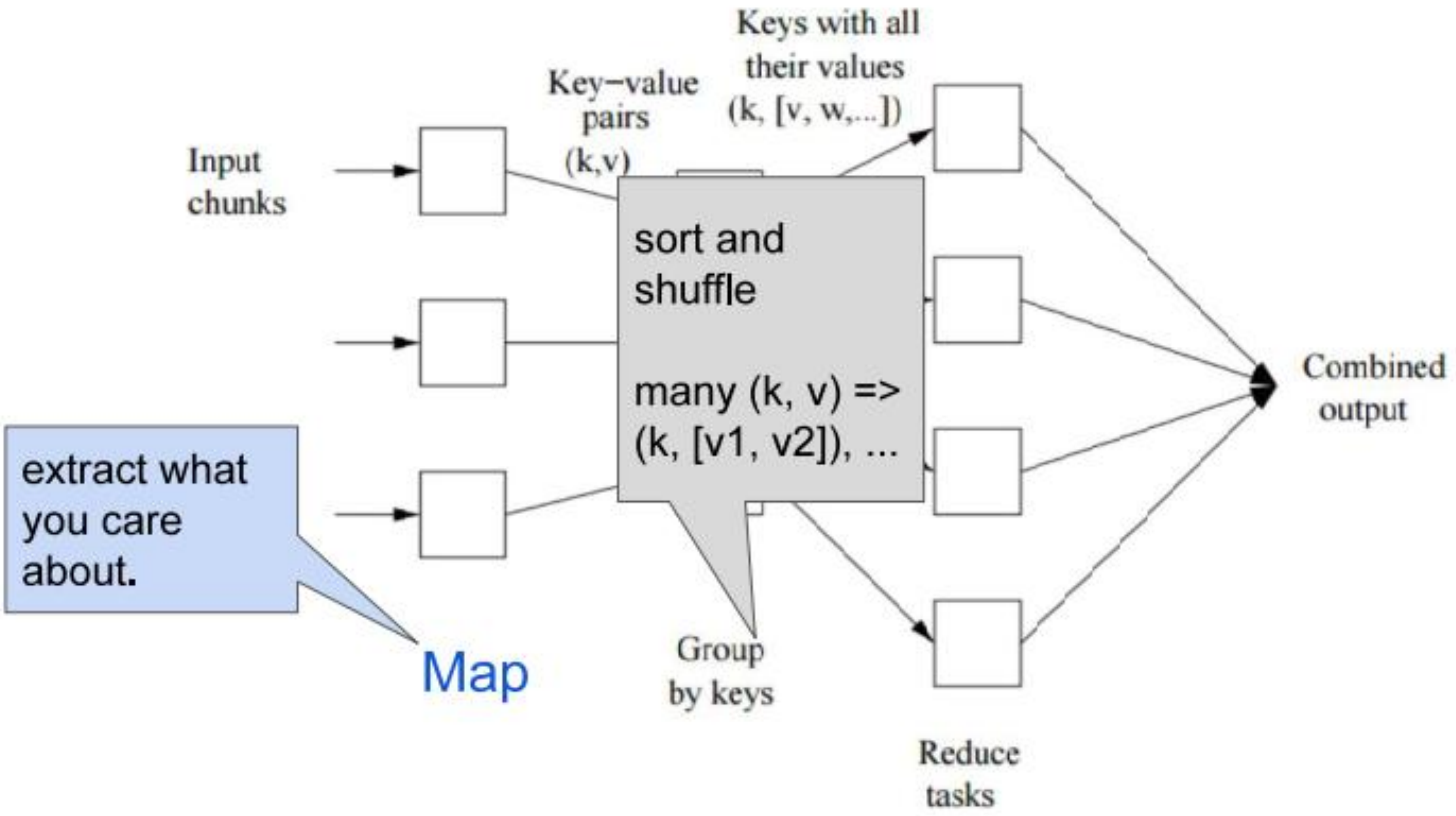
# What is MapReduce



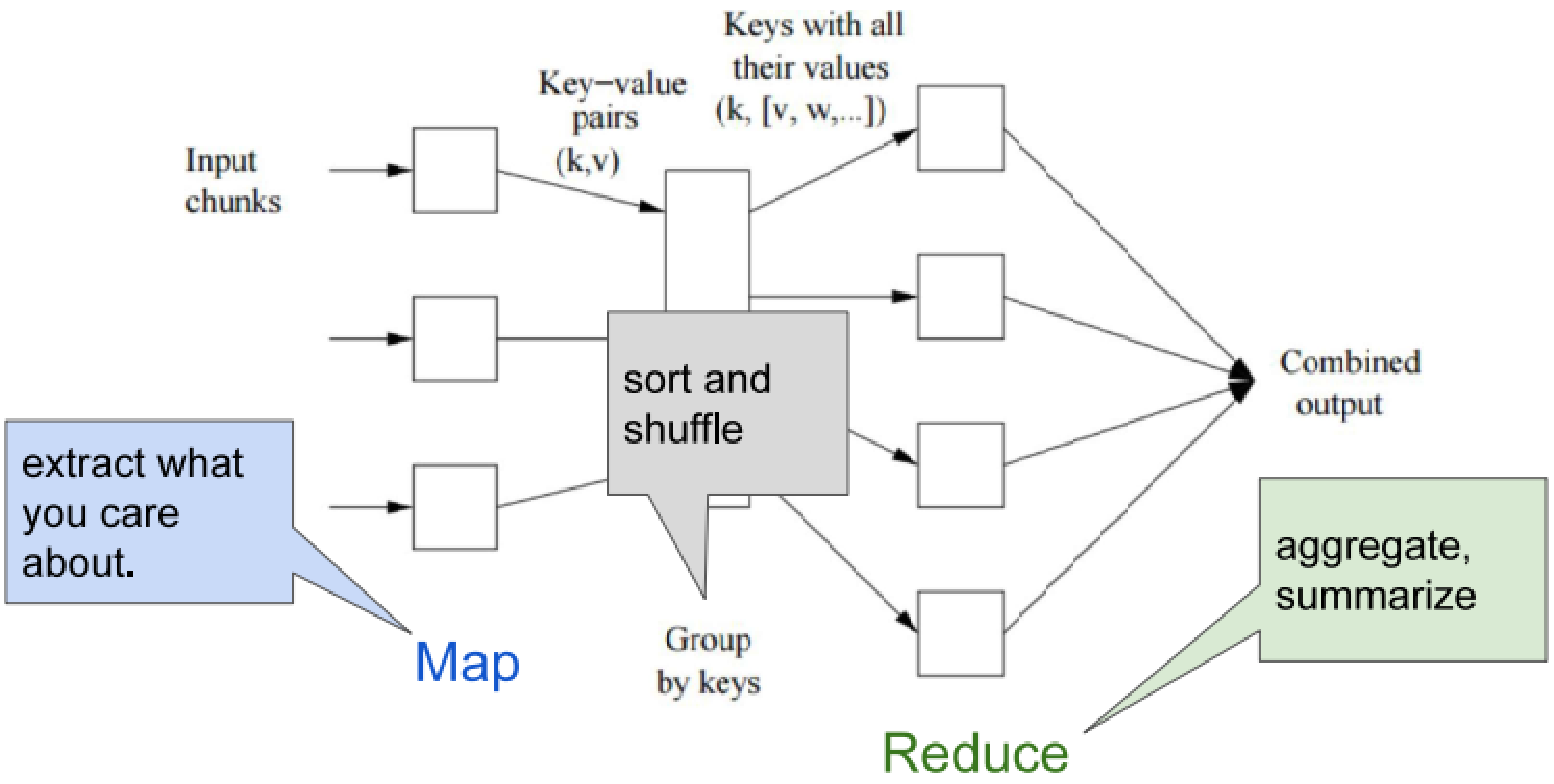
# What is MapReduce



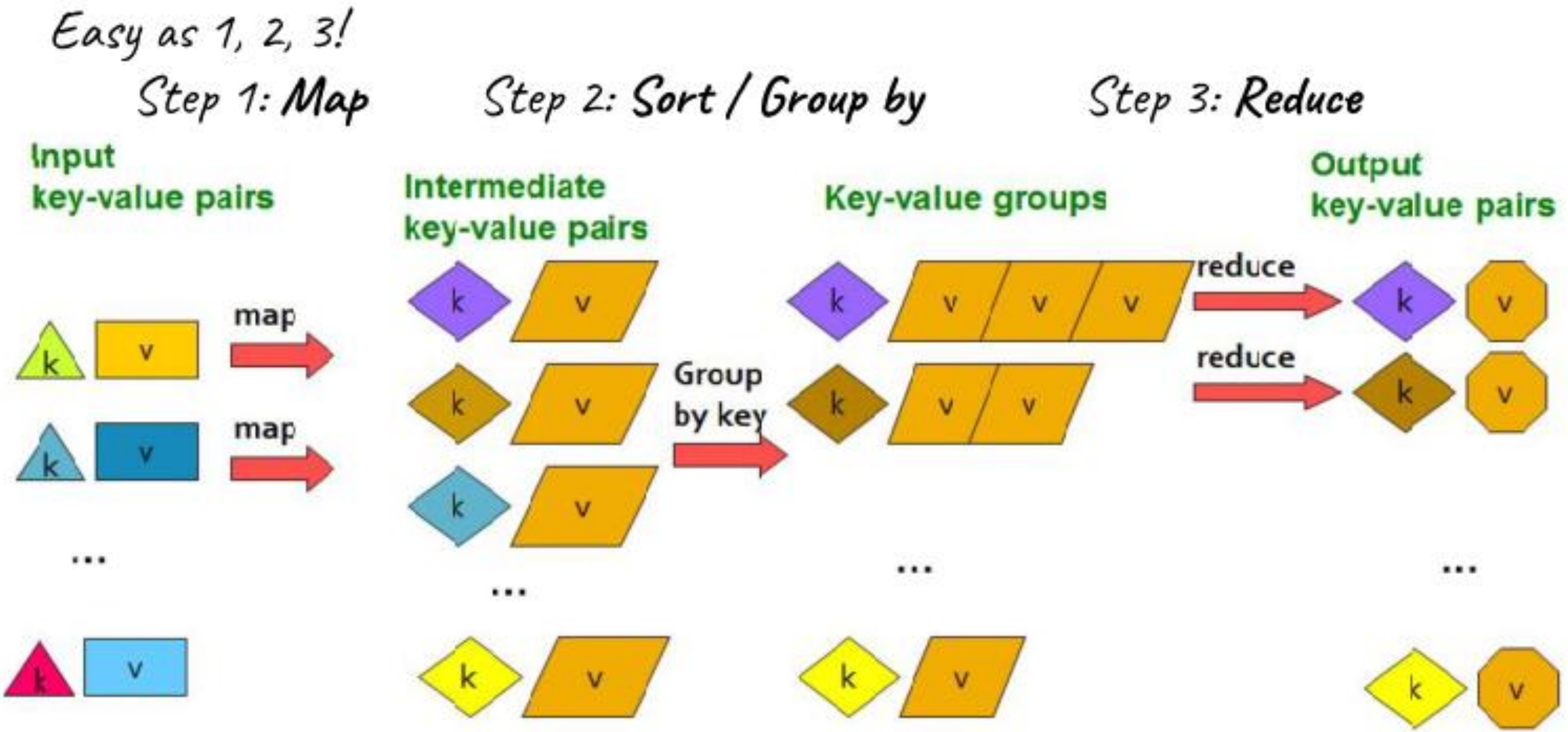
# What is MapReduce



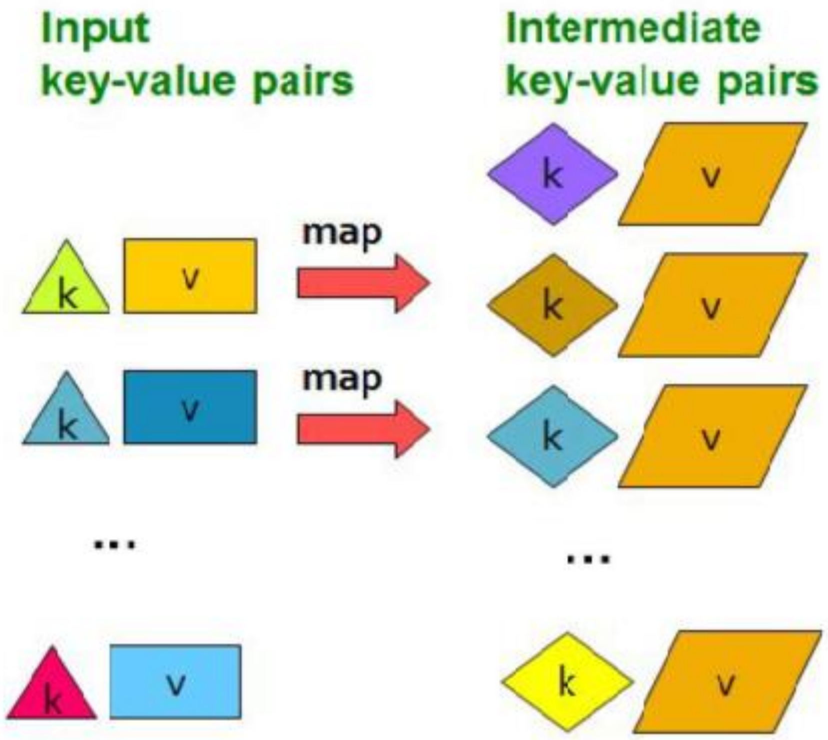
# What is MapReduce



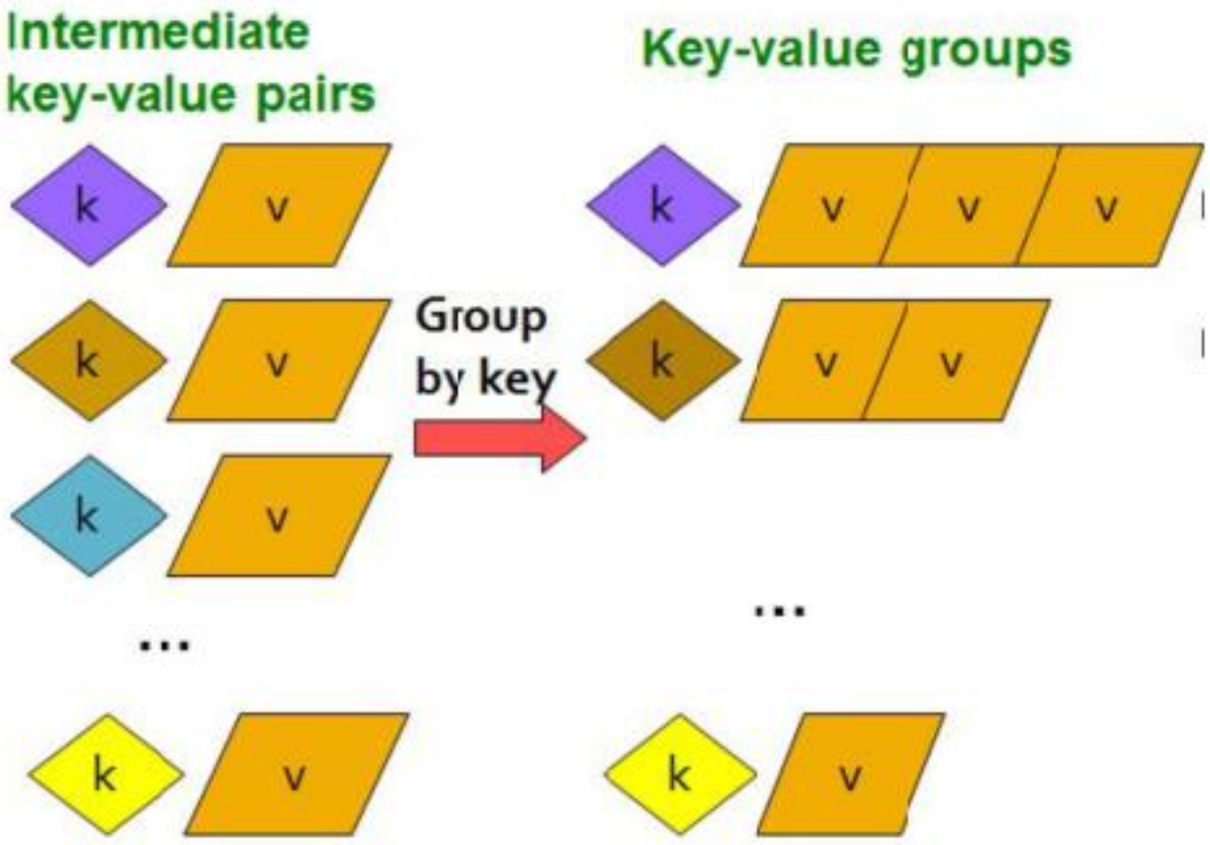
# What is MapReduce



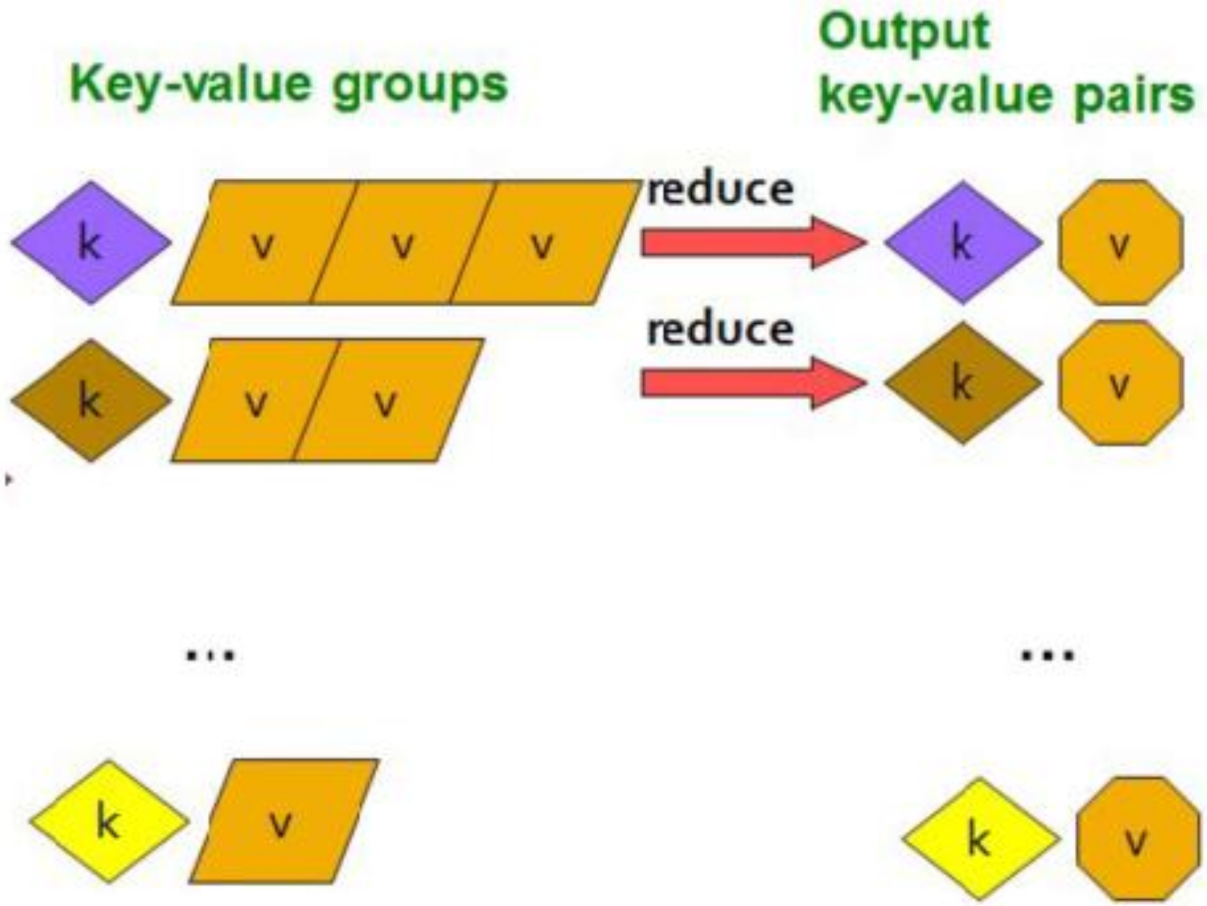
# (1) The Map Step



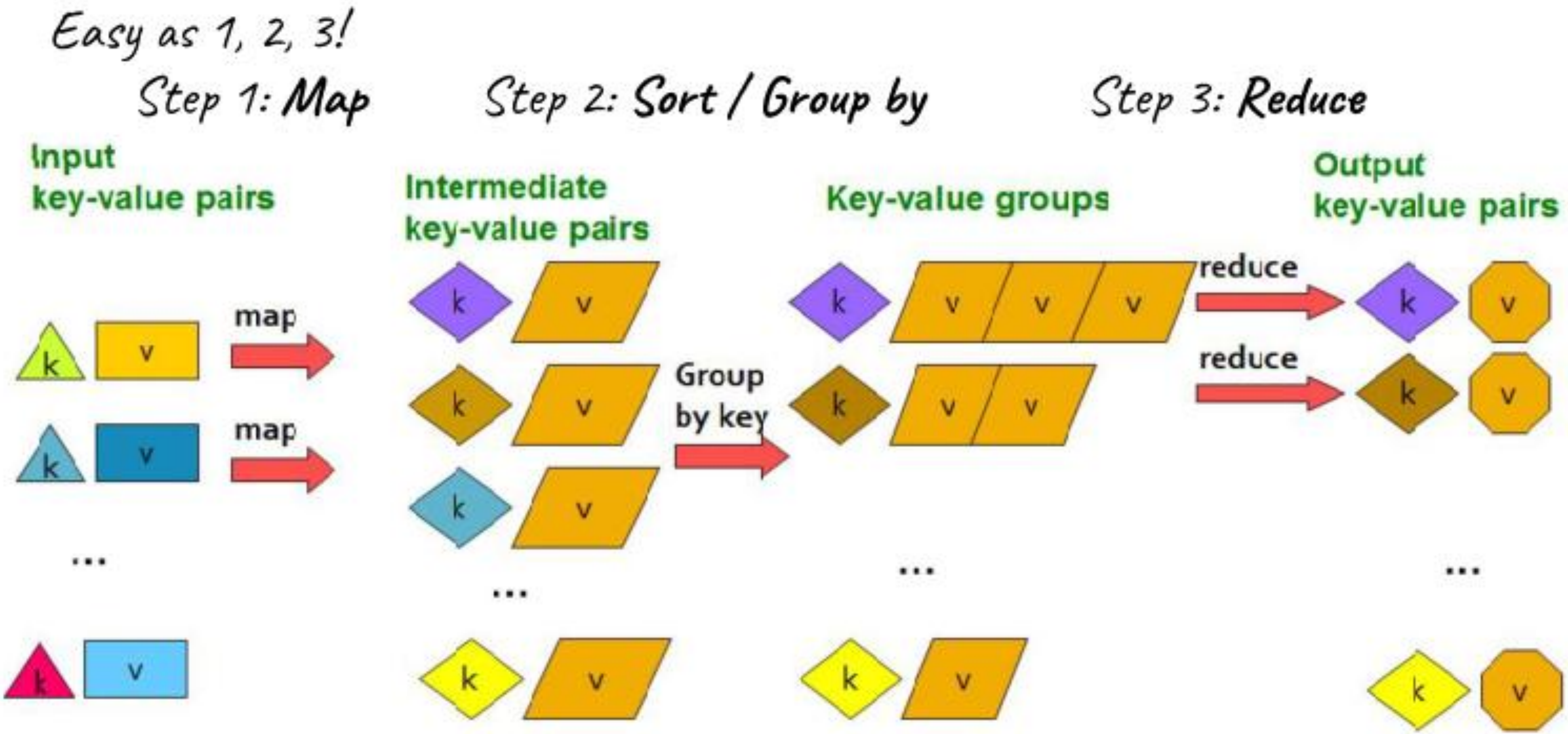
# (2) The Sort / Group-by Step



# (2) Reduce Step



# What is MapReduce



# What is MapReduce

Map: $(k,v) \rightarrow (k', v')^*$

(Written by programmer)

Group by key: $(k_1', v_1'), (k_2', v_2'), \ldots \rightarrow (k_1', (v_1', v', \ldots)),$

(system handles) $(k_2', (v_1', v', \ldots)), \ldots$

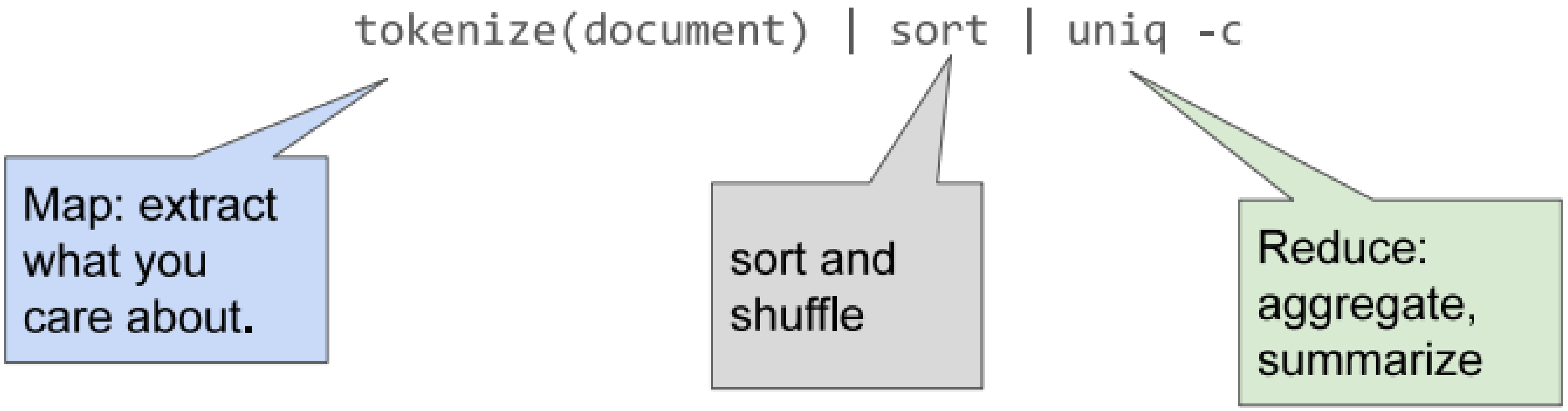
Reduce: $(k', (v_1', v', \ldots)) \rightarrow (k', v'')^*$

(Written by programmer)

# Example: Word Count

```
tokenize(document) | sort | uniq -c
```

# Example: Word Count



# Example: Word Count


The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/mache partnership. "The work we're doing now -- the robotics we're doing -- is what we're going to need ......................

Big document

# Example: Word Count

**Provided by the programmer**

**MAP:**
Read input and produces a set of key-value pairs

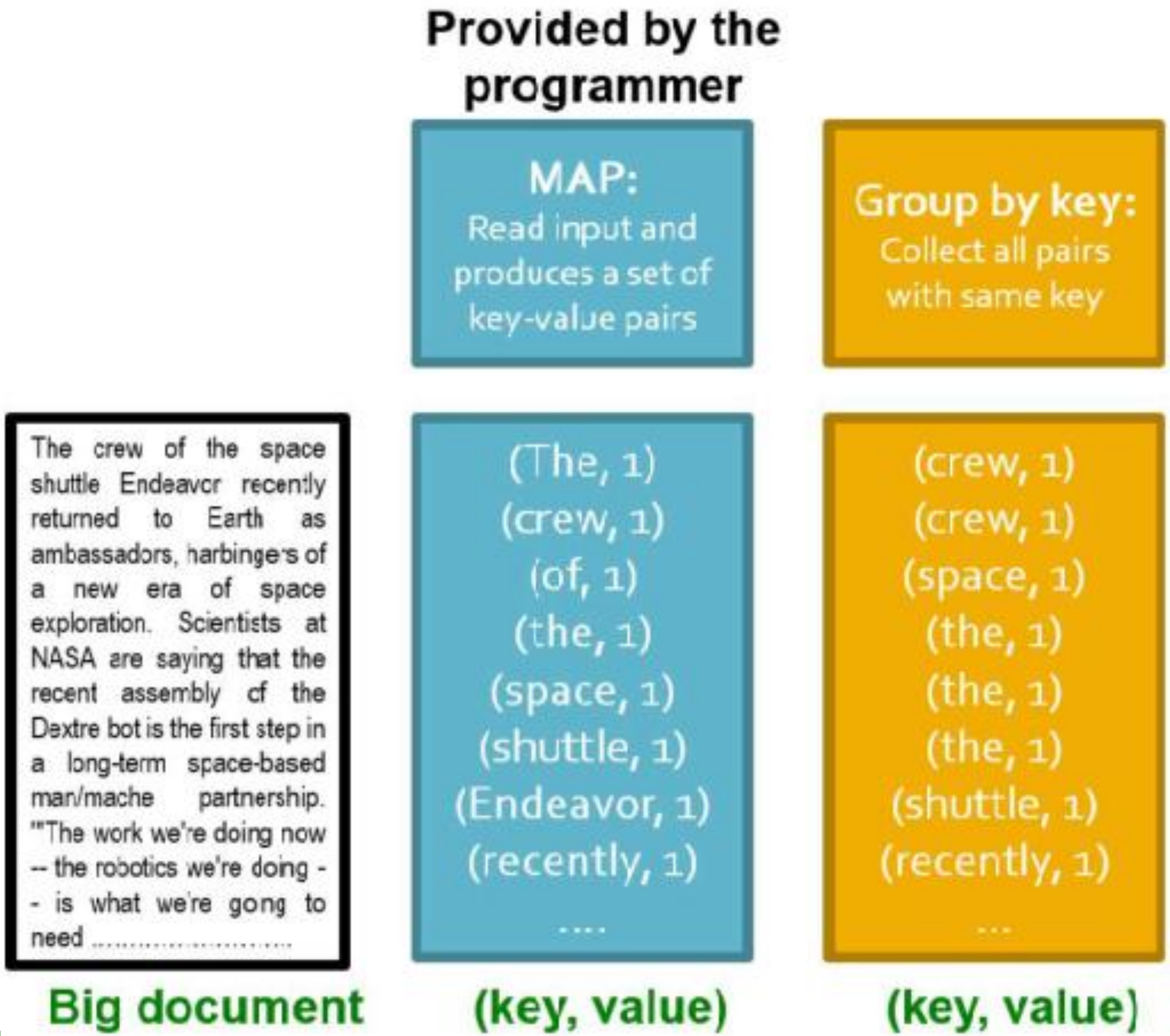
The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/mache partnership. "The work we're doing now -- the robotics we're doing -- is what we're going to need ……………………

**Big document**

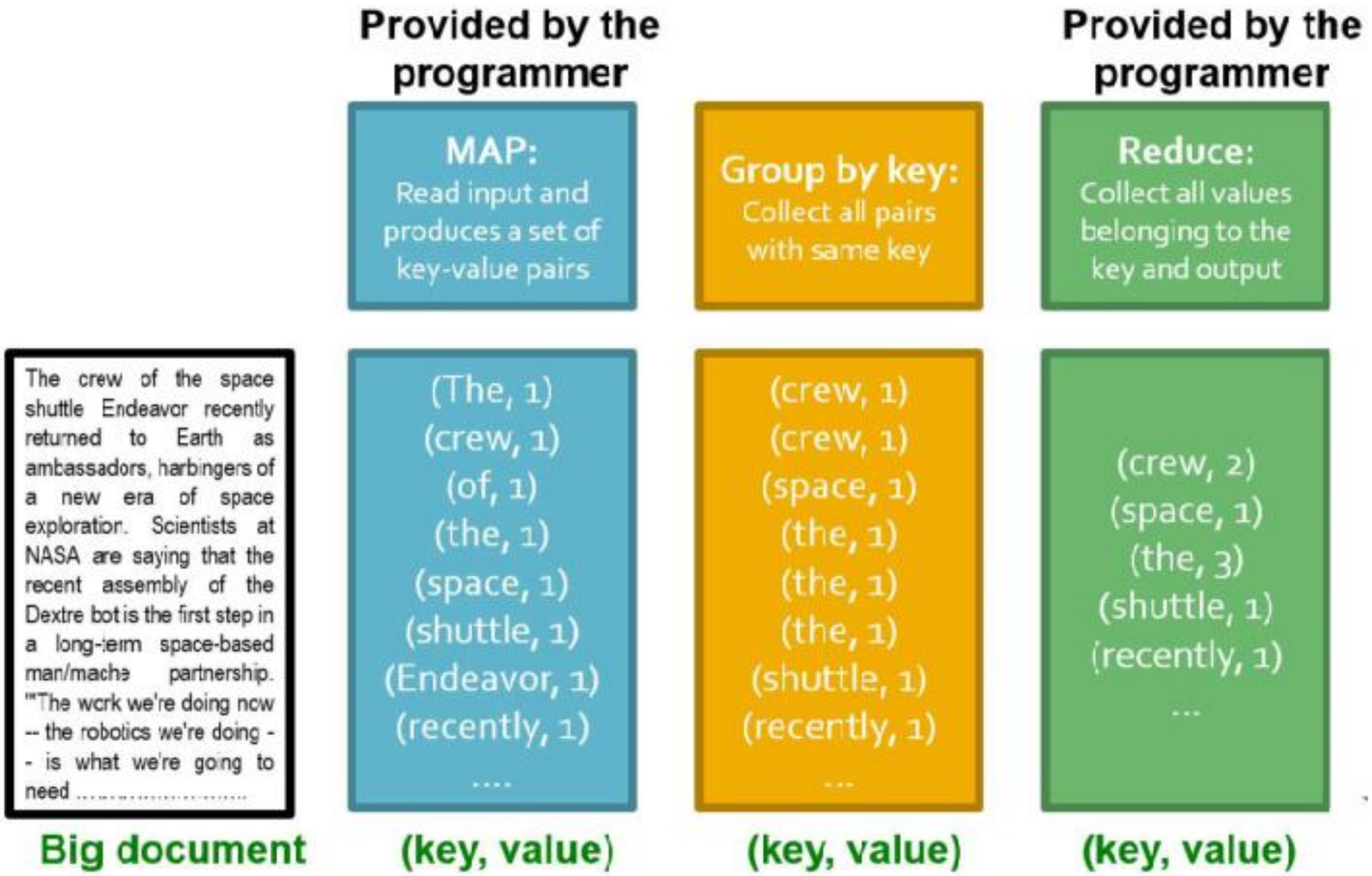
(The, 1)
(crew, 1)
(of, 1)
(the, 1)
(space, 1)
(shuttle, 1)
(Endeavor, 1)
(recently, 1)
….

**(key, value)**

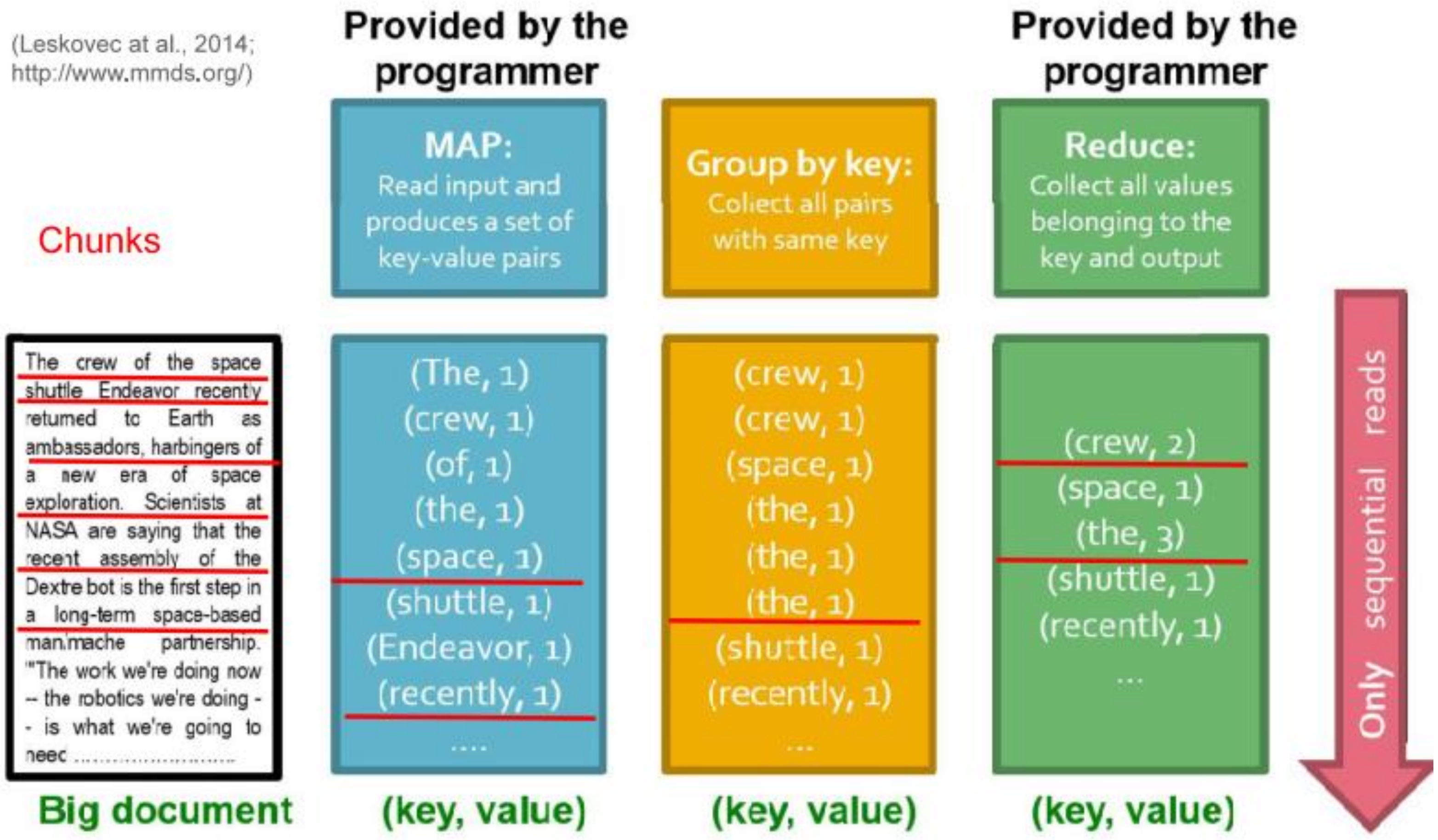
# Example: Word Count



# Example: Word Count



# Example: Word Count



# Generator vs. Iterator
# Python MapReduce

# Iterator vs Iteration

- Iterable
  - Iterator를 반환
  - Index를 가지며, getItem을 통해 데이터를 가져옴
- Iterator
  - Next 메소드를 사용하여, 데이터를 순차적으로 가져옴
- Iteration
  - 특정 collection위에서 아이템을 가져오는 과정

# Generator

- Iterators 지만 한번만 순회(iterate) 가능
  - 모든 값을 메모리에 저장하지 않음.
- Use **yield**, instead of **return**
- 결과가 Large sets 일때 매우 유용함

# Example: Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

| | |
|---|---|
| 1+1=2 | 13+21=34 |
| 1+2=3 | 21+34=55 |
| 2+3=5 | 34+55=89 |
| 3+5=8 | 55+89=144 |
| 5+8=13 | 89+144=233 |
| 8+13=21 | 144+233=377 |

# Examples

```python
# generator version
def fibon(n):
    a = b = 1
    for i in range(n):
        yield a
        a, b = b, a + b
```

```python
for x in fibon(1000000):
    print(x)
```

# Examples

```python
def fibon(n):
    a = b = 1
    result = []
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result
```

```python
for x in fibon(1000000):
    print(x)
```

# Map, Filter, Reduce in Python

- Functional approach to a programming
  - On-the-fly manner
- Map
  - Apply a function to all the items in an input list
- Filter
  - Create a list of elements for which a function returns true.
- Reduce
  - Perform a computation on a list and return the result
  - A rolling computation to sequential pairs of values in a list

# Examples: Map in Python

- How to use: map ( fun, iter )
  - fun: Function
  - iter: Iterable
- **Add two lists of x, y** using map and lambda

```python
# Return double of n
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

# Examples: Filter in Python

- How to use: filter ( fun, iter )
  - fun: Function that check if each element of “iter” is true or not.
  - iter: Iterable
- **Filter some elements of x** using filter and lambda

```
# a list contains both even and odd numbers.
seq = [0, 1, 2, 3, 5, 8, 13]

# result contains odd numbers of the list
result = filter(lambda x: x % 2 != 0, seq)
print(list(result))
```

# Examples: Reduce in Python

- How to use: reduce ( fun, iter )
  - fun: a function is applied to all elements in "iter"
  - iter: Iterable
  - **import functools**
- **Sum x, y** using reduce, zip and lambda

```
# importing functools for reduce()
import functools

# initializing list
lis = [1, 3, 5, 6, 2]

# using reduce to compute sum of list
print("The sum of the list elements is : ", end="")
print(functools.reduce(lambda a, b: a+b, lis))
```

# Example: Word Count

```
@abstractmethod
def map(k, v):
    pass
```

```
@abstractmethod
def reduce(k, vs):
    pass
```

# Example: Word Count (v1)

```
def map(k, v):
    for w in tokenize(v):
        yield (w,1)
```

```
def reduce(k, vs):
    return len(vs)
```

# Example: Word Count (v1)

```
def map(k, v):
    for w in tokenize(v):
        yield (w,1)
```

```
def tokenize(s):
    #simple version
    return s.split(' ')
```

```
def reduce(k, vs):
    return len(vs)
```

# Example: Word Count (v2)

```
def map(k, v):
    counts = dict()
    for w in tokenize(v):
```

counts each word within the chunk
(try/except is faster than
“if w in counts”)

# Example: Word Count (v2)

```
def map(k, v):
    counts = dict()
    for w in tokenize(v):
        try:
            counts[w] += 1
        except KeyError:
            counts[w] = 1
    for item in counts.iteritems():
        yield item
```

counts each word within the chunk
(try/except is faster than
"if w in counts")

# Example: Word Count (v2)

```
def map(k, v):
    counts = dict()
    for w in tokenize(v):
        try:
            counts[w] += 1
        except KeyError:
            counts[w] = 1
    for item in counts.iteritems():
        yield item
```

counts each word within the chunk
(try/except is faster than
"if w in counts")

```
def reduce(k, vs):
    return sum(vs)
```

sum of counts from different chunks

# Distributed Architecture (Cluster)

## Challenges for IO Cluster Computing

1. Nodes fail
   1 in 1000 nodes fail a day
   Duplicate Data **(Distributed FS)**
2. Network is a bottleneck
   Typically 1-10 Gb/s throughput
   Bring computation to nodes, rather than data to nodes.
3. Traditional distributed programming is often ad-hoc and complicated
   Stipulate a programming system that can easily be distributed

# Distributed Architecture (Cluster)

## Challenges for IO Cluster Computing

1. Nodes fail
   1 in 1000 nodes fail a day
   Duplicate Data **(Distributed FS)**
2. Network is a bottleneck
   Typically 1-10 Gb/s throughput
   Bring computation to nodes, rather than data to nodes. **(Sort and Shuffle)**
3. Traditional distributed programming is often ad-hoc and complicated
   Stipulate a programming system that can easily be distributed

# Distributed Architecture (Cluster)

## Challenges for IO Cluster Computing

1. Nodes fail
   1 in 1000 nodes fail a day
   Duplicate Data **(Distributed FS)**
2. Network is a bottleneck
   Typically 1-10 Gb/s throughput
   Bring computation to nodes, rather than data to nodes. **(Sort and Shuffle)**
3. Traditional distributed programming is often ad-hoc and complicated **(Simply define a map and reduce)**
   Stipulate a programming system that can easily be distributed

# Example: Relational Algebra

Select

Project

Union, Intersection, Difference

Natural Join

Grouping

# Example: Relational Algebra

**Select**

Project

Union, Intersection, Difference

**Natural Join**

Grouping

# Example: Relational Algebra

## Select

$R(A_1, A_2, A_3, \ldots)$, Relation $R$, Attributes $A_*$

return only those attribute tuples where condition $C$ is true

# Example: Relational Algebra

## Select

$R(A_1, A_2, A_3, \ldots)$, Relation $R$, Attributes $A_*$

return only those attribute tuples where condition $C$ is true

```
def map(k, v): #v is list of attribute tuples
    for t in v:
        if t satisfies C:
            yield (t, t)

def reduce(k, vs):
    For each v in vs:
        yield  (k, v)
```

# Example: Relational Algebra

## Select

$R(A_1, A_2, A_3, \ldots)$, Relation $R$, Attributes $A_*$

return only those attribute tuples where condition $C$ is true

```
def map(k, v): #v is list of attribute tuples
    for t in v:
        if t satisfies C:
            yield (t, t)

def reduce(k, vs):
    For each v in vs:
        yield  (k, v)
```

# Example: Relational Algebra

## Natural Join

Given $R_1$ and $R_2$ return $R_{join}$ -- union of all pairs of tuples that match given attributes.

# Example: Relational Algebra

## Natural Join

Given $R_1$ and $R_2$ return $R_{join}$ -- union of all pairs of tuples that match given attributes.

```
def map(k, v): #k \in {R1, R2}, v is (R1=(A, B), R2=(B, C));B are matched attributes
    if k==”R1”:
        (a, b) = v
        yield (b,(R1,a))
    if k==”R2”:
        (b,c) = v
        yield (b,(R2,c))
```

# Example: Relational Algebra

## Natural Join

Given $R_1$ and $R_2$ return $R_{join}$ -- union of all pairs of tuples that match given attributes.

```
def map(k, v): #k \in {R1, R2}, v is (R1=(A, B), R2=(B, C));B are matched attributes
    if k=="R1":
        (a, b) = v
        yield (b,(R1,a))
    if k=="R2":
        (b,c) = v
        yield (b,(R2,c))
```

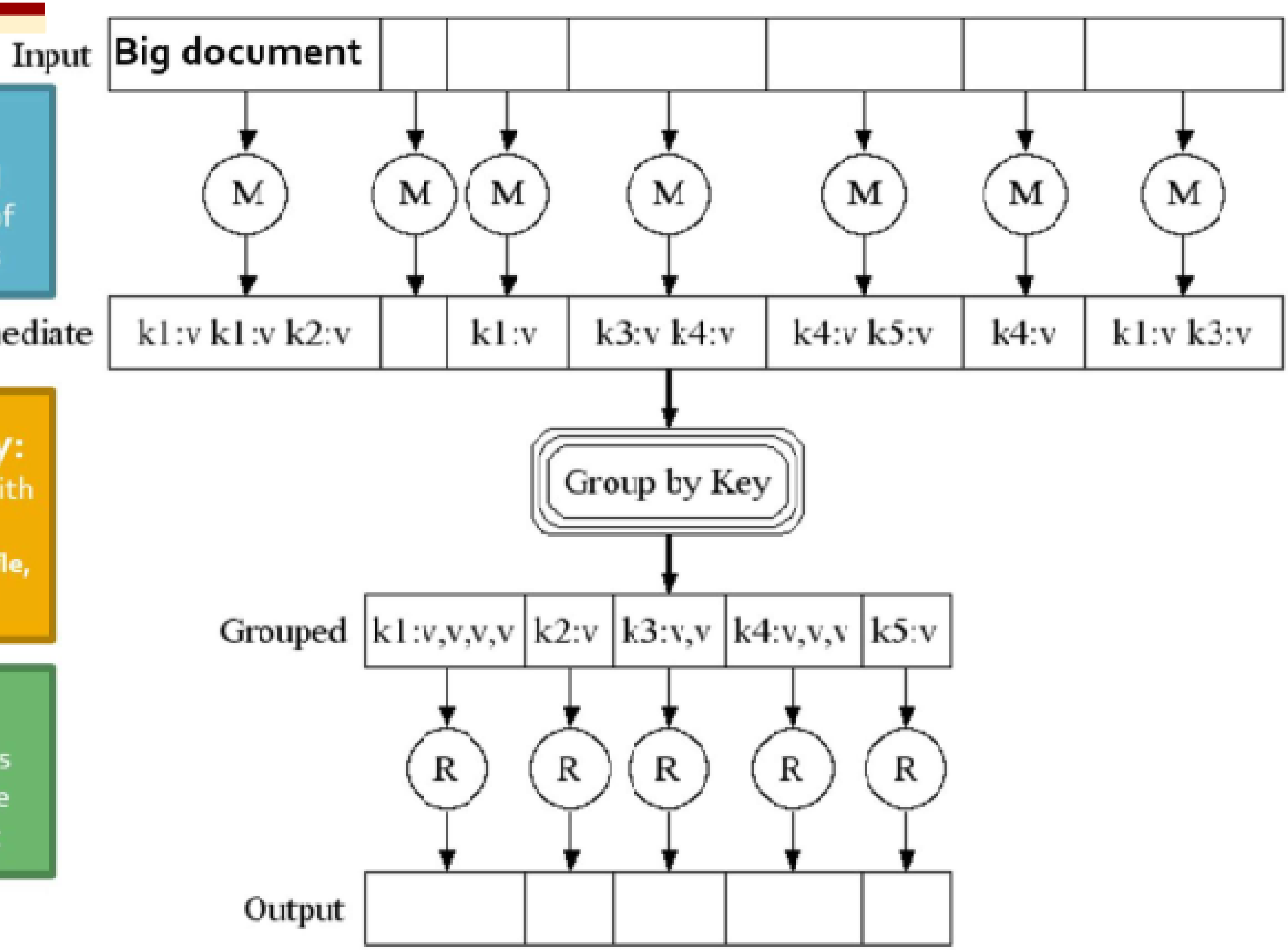
```
def reduce(k, vs):
    r1, r2 = [], []
    for (S, x) in vs: #separate rs
        if S == r1: r1.append(x)
        else: r2.append(x)
    for a in r1: #join as tuple
        for each c in r2:
            yield (Rjoin', (a, k, c)) #k is
```

# Data Flow

**MAP:**
Read input and produces a set of key-value pairs

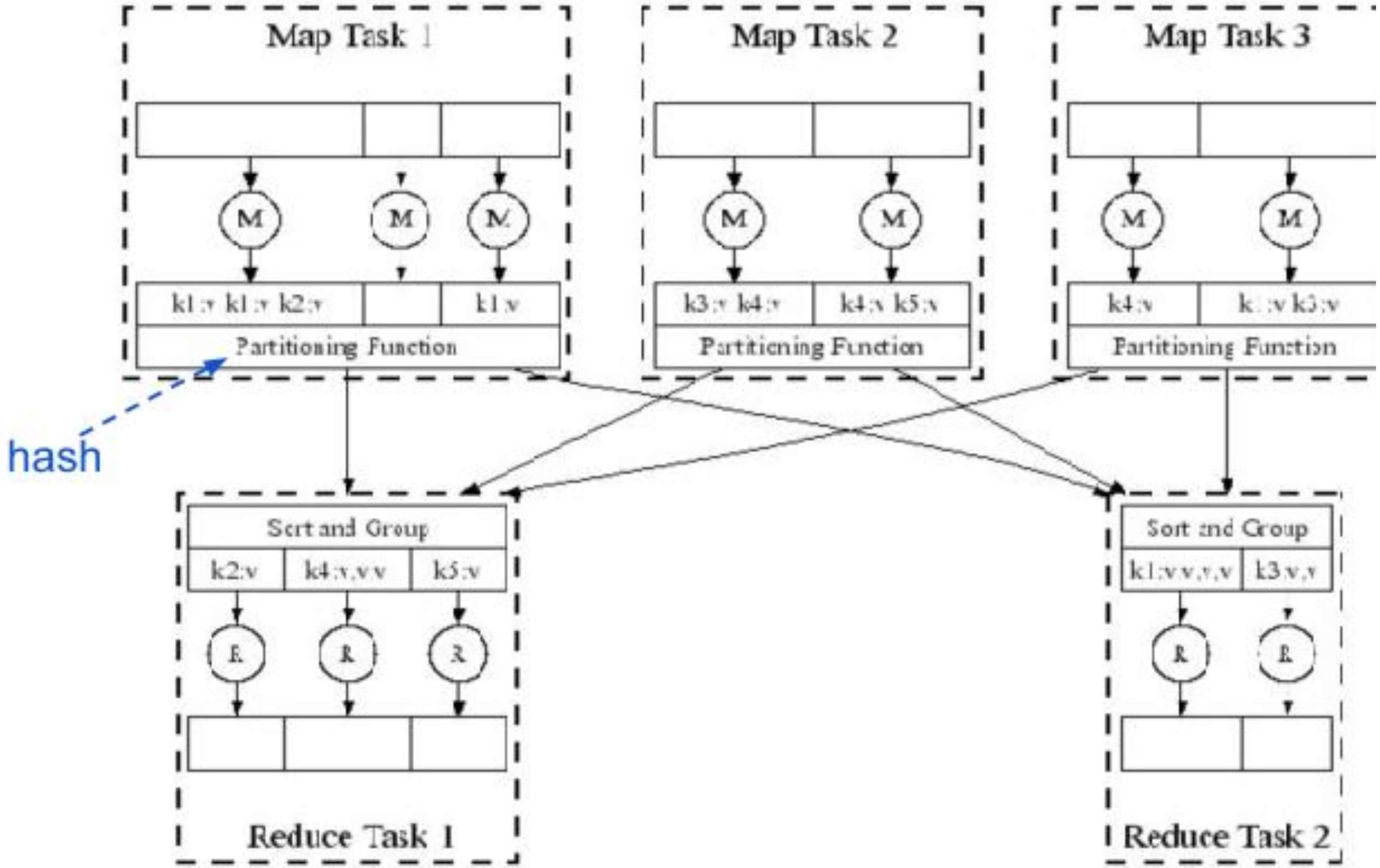
**Group by key:**
Collect all pairs with same key
**(Hash merge, Shuffle, Sort, Partition)**

**Reduce:**
Collect all values belonging to the key and output



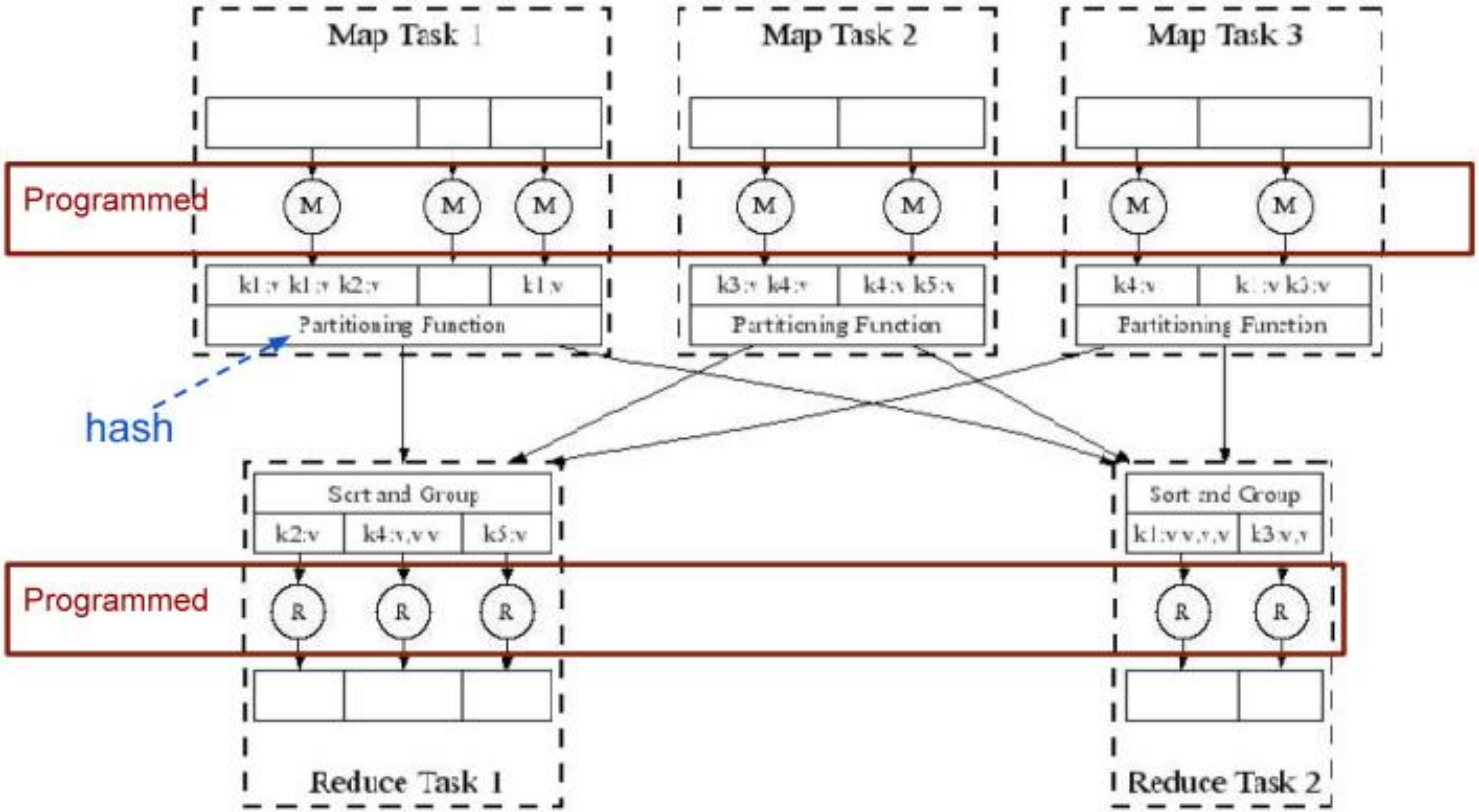

J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org

# Data Flow



Map Task 1
Map Task 2
Map Task 3
M
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Partitioning Function
hash
Sort and Group
k2:v
k4:v,v,v
k5:v
k1:v,v,v,v
k3:v,v
R
Reduce Task 1
Reduce Task 2

# Data Flow


Map Task 1
Map Task 2
Map Task 3
Programmed
M
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Partitioning Function
hash
Sort and Group
k2:v
k4:v,v,v
k5:v
k1:v,v,v,v
k3:v,v
Programmed
R
Reduce Task 1
Reduce Task 2

# Data Flow

DFS ⇨ Map ⇨ Map's Local FS ⇨ Reduce ⇨ DFS

# Data Flow

MapReduce system handles:

- Partitioning
- Scheduling map / reducer execution
- Group by key

- Restarts from node failures
- Inter-machine communication

# Data Flow

DFS ⇨ MapReduce ⇨ DFS

- Schedule map tasks near physical storage of chunk
- Intermediate results stored locally
- Master / Name Node coordinates

# Data Flow

DFS ⇨ MapReduce ⇨ DFS

- Schedule map tasks near physical storage of chunk
- Intermediate results stored locally
- Master / Name Node coordinates
  - Task status: idle, in-progress, complete
  - Receives location of intermediate results and schedules with reducer
  - Checks nodes for failures and restarts when necessary
    - All map tasks on nodes must be completely restarted
    - Reduce tasks can pickup with reduce task failed

# Data Flow

DFS ⇨ MapReduce ⇨ DFS

- Schedule map tasks near physical storage of chunk
- Intermediate results stored locally
- Master / Name Node coordinates
  - Task status: idle, in-progress, complete
  - Receives location of intermediate results and schedules with reducer
  - Checks nodes for failures and restarts when necessary
    - All map tasks on nodes must be completely restarted
    - Reduce tasks can pickup with reduce task failed

DFS ⇨ MapReduce ⇨ DFS ⇨ MapReduce ⇨ DFS

# Data Flow

Skew: The degree to which certain tasks end up taking much longer than others.

Handled with:

- More reducers than reduce tasks
- More reduce tasks than nodes

# Data Flow

**Key Question:** *How many Map and Reduce jobs?*

# Data Flow

**Key Question:** *How many Map and Reduce jobs?*

*M*: map tasks, *R*: reducer tasks

**A:** If possible, one chunk per map task

and $M \gg |\text{nodes}| \approx\approx |\text{cores}|$

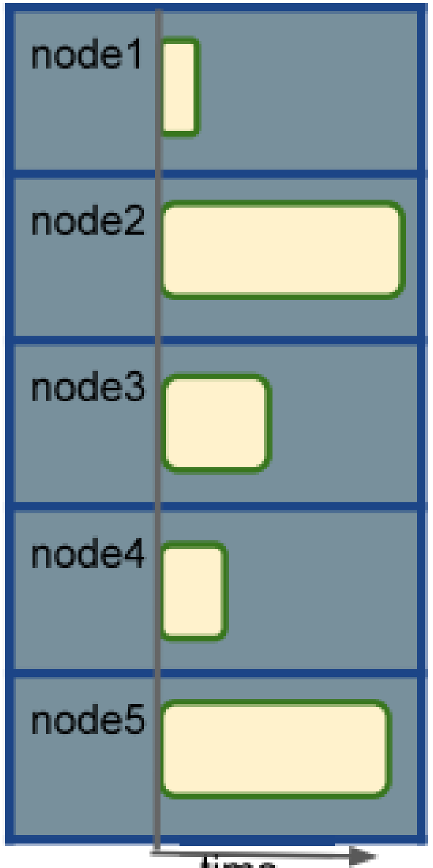
(better handling of node failures, better load balancing)

$R < M$

(reduces number of parts stored in DFS)

# Data Flow

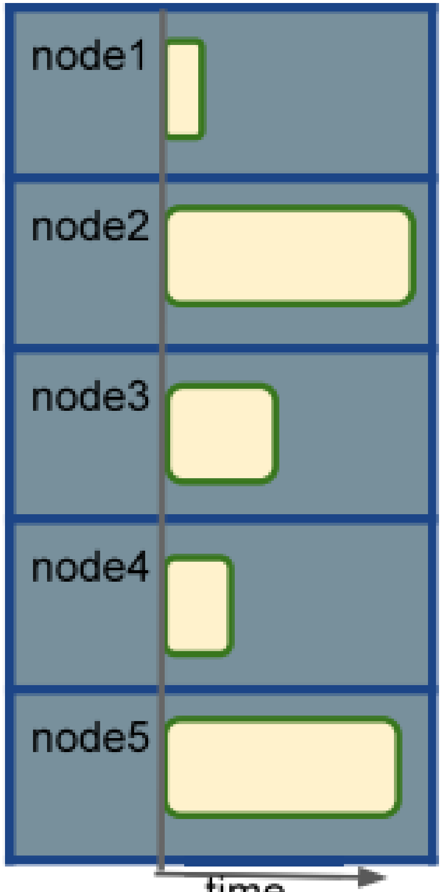
version 1: few reduce tasks
(same number of reduce tasks as nodes)




Reduce tasks represented by
**time to complete task**
(some tasks take much longer)

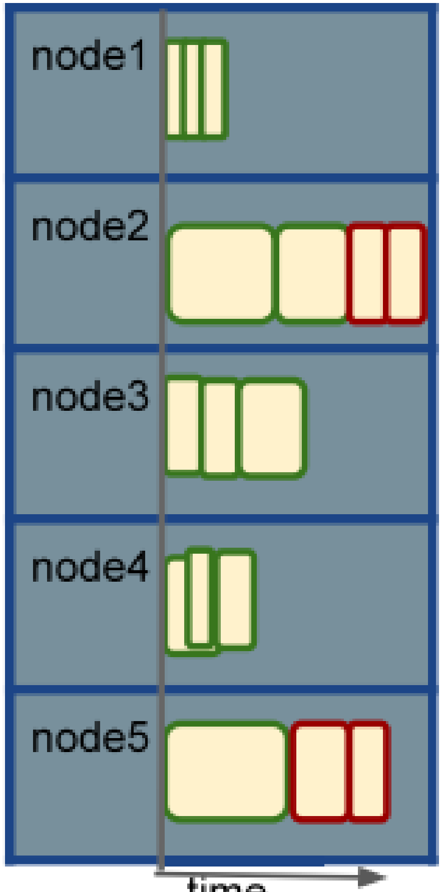
# Data Flow

version 1: few reduce tasks
(same number of reduce tasks as nodes)



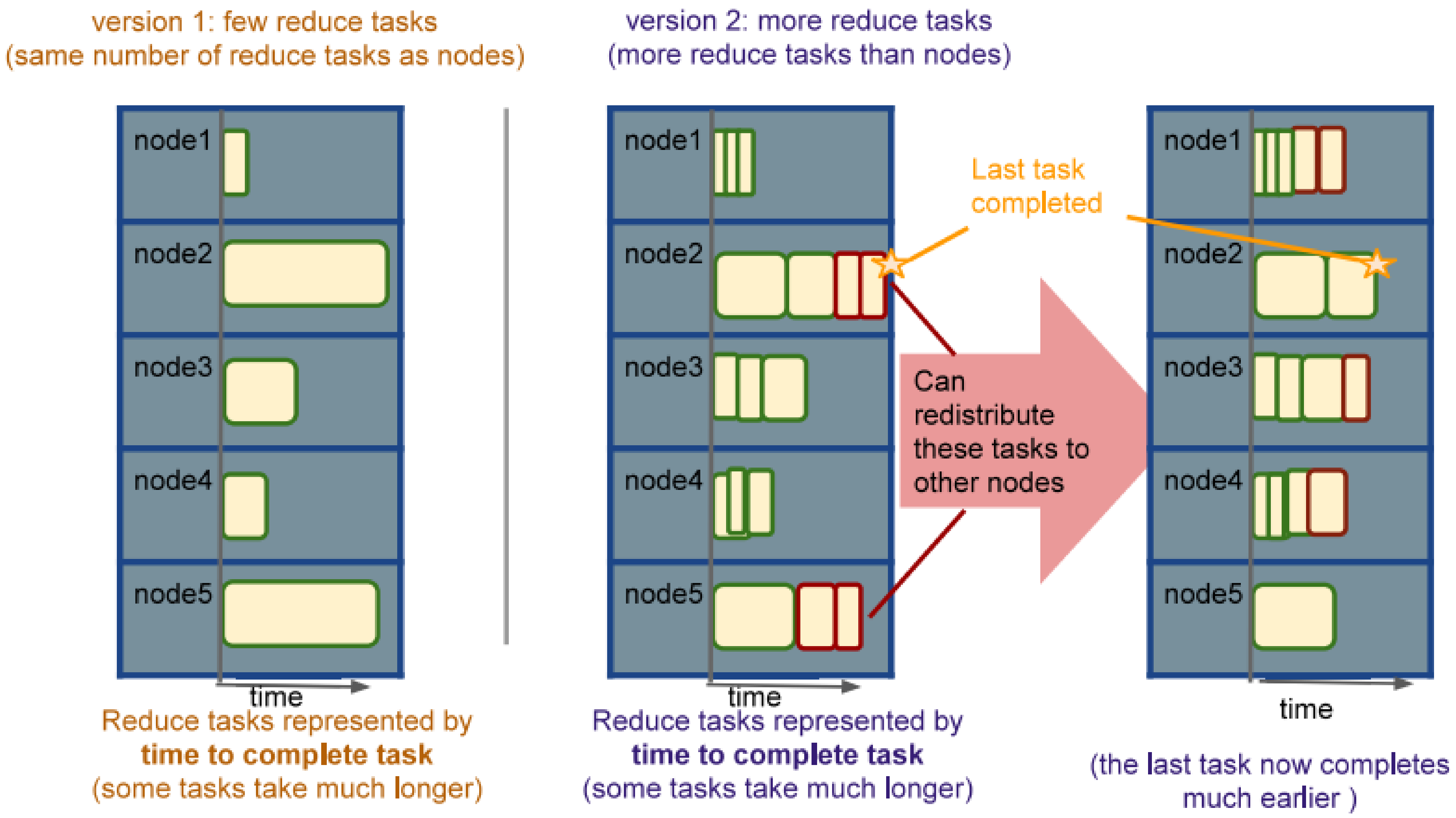

Reduce tasks represented by
**time to complete task**
(some tasks take much longer)

version 2: more reduce tasks
(more reduce tasks than nodes)




Reduce tasks represented by
**time to complete task**
(some tasks take much longer)

# Data Flow



version 1: few reduce tasks
(same number of reduce tasks as nodes)
version 2: more reduce tasks
(more reduce tasks than nodes)
node1
node2
node3
node4
node5
time
Last task
completed
Can
redistribute
these tasks to
other nodes
Reduce tasks represented by
**time to complete task**
(some tasks take much longer)
Reduce tasks represented by
**time to complete task**
(some tasks take much longer)
(the last task now completes
much earlier )

# Communication Cost Model

How to assess performance?

(1) Computation: Map + Reduce + System Tasks

(2) Communication: Moving (key, value) pairs

# Communication Cost Model

## How to assess performance?

(1) Computation: Map + Reduce + System Tasks

(2) Communication: Moving (key, value) pairs

Ultimate Goal: wall-clock Time.

# Communication Cost Model

## How to assess performance?

**(1) Computation: Map + Reduce + System Tasks**

- Mappers and reducers often single pass O(n) within node
- System: sort the keys is usually most expensive
- Even if map executes on same node, disk read usually dominates
- In any case, can add more nodes

(2) Communication: Moving key-value pairs

Ultimate Goal: Wall-clock Time.

# Communication Cost Model

## How to assess performance?

(1) Computation: Map + Reduce + System Tasks

(2) **Communication: Moving key, value pairs**

Ultimate Goal: wall-clock Time

Often dominates computation.

- Connection speeds: 1-10 giga**bits** per sec;
- HD read: 50-150 giga**bytes** per sec
- Even reading from disk to memory typically takes longer than operating on the data.

# Communication Cost Model

## How to assess performance?

**Communication Cost =** input size +
(sum of size of all map-to-reducer files)

## (2) Communication: Moving key, value pairs

Often dominates computation.

- Connection speeds: 1-10 giga**bits** per sec;
  HD read: 50-150 giga**bytes** per sec
- Even reading from disk to memory typically takes longer than operating on the data.

# Communication Cost Model

## How to assess performance?

**Communication Cost =** input size + (sum of size of all map-to-reducer files)

## (2) Communication: Moving key, value pairs

Often dominates computation.

- Connection speeds: 1-10 giga**bits** per sec;
  HD read: 50-150 giga**bytes** per sec
- Even reading from disk to memory typically takes longer than operating on the data.
- Output from reducer ignored because it's either small (finished summarizing data) or being passed to another mapreduce job.

# Communication Cost: Natural Join

R, S: Relations (Tables) $R(A, B) \bowtie S(B, C)$

**Communication Cost =** input size +
(sum of size of all map-to-reducer files)

DFS ⇨ Map ⇨ LocalFS ⇨ Network ⇨ Reduce ⇨ DFS ⇨ ?

# Communication Cost: Natural Join

R, S: Relations (Tables)  *R(A, B)* ⋈ *S(B, C)*

**Communication Cost =** input size + (sum of size of all map-to-reducer files)

```
def map(k, v):
    if k=="R1":
        (a, b) = v
        yield (b,(R1,a))
    if k=="R2":
        (b,c) = v
        yield (b,(R2,c))
```

```
def reduce(k, vs):
    r1, r2 = [], []
    for (rel, x) in vs: #separate rs
        if rel == 'R': r1.append(x)
        else: r2.append(x)
    for a in r1: #join as tuple
        for each c in r2:
            yield (Rjoin', (a, k, c)) #k is
```

# Communication Cost: Natural Join

R, S: Relations (Tables)    $R(A, B) \bowtie S(B, C)$

**Communication Cost =** input size + (sum of size of all map-to-reducer files)

= |R1| + |R2| + (|R1| + |R2|)

= $O(|R1| + |R2|)$

```
def map(k, v):
    if k=="R1":
        (a, b) = v
        yield (b,(R1,a))
    if k=="R2":
        (b,c) = v
        yield (b,(R2,c))
```

```
def reduce(k, vs):
    r1, r2 = [], []
    for (rel, x) in vs: #separate rs
        if rel == 'R': r1.append(x)
        else: r2.append(x)
    for a in r1: #join as tuple
        for each c in r2:
            yield (Rjoin', (a, k, c)) #k is
```

# MapReduce: Final Considerations

- Performance Refinements:
  - Combiners (like word count version 2 but done via reduce)
    - Run reduce right after map from same node before passing to reduce (MapTask can execute)
    - Reduces communication cost

  - Backup tasks (aka speculative tasks)
    - Schedule multiple copies of tasks when close to the end to mitigate certain nodes running slow.

  - Override partition hash function to organize data

    E.g. instead of `hash(url)` use `hash(hostname(url))`