Week2: Hadoop MapReduce: How to Build Reliable System from Unreliable Components @Sieun Bae

YANDEX

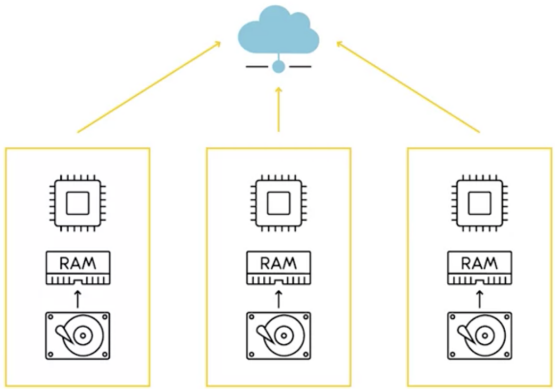
Unreliable Components (Node, Link, Clock)

What do you think is the most popular word in Wikipedia? Which word “a”, “the” or “of” is more popular? How much more times the most popular word occurs than others? What is the most popular word except articles, conjunctions and other “stop” words?

The words “the”, “of” and “a” occur 100 million times, 60 million times and 30 million times correspondingly. While writing Wikipedia articles, people introduce a concept once and then use it 3+ more times (100 / 30 - compare “the” vs “a”). If you discard all the auxiliary and stop words then the most common word is “first” which you can see about 3.2M times, the next one is “references” with 2.9M occurrences.

There is NO Best Model!

- hadoop: fail-recovery + fair-loss link + asynchronous
- super computer (OpenMP, MPI): fail-stop + perfect link + synchronous
- grid computing: byzantine-failure + byzantine link + asynchronous



need first to read data from local disks, to do some computations, and to aggregate results over the network.

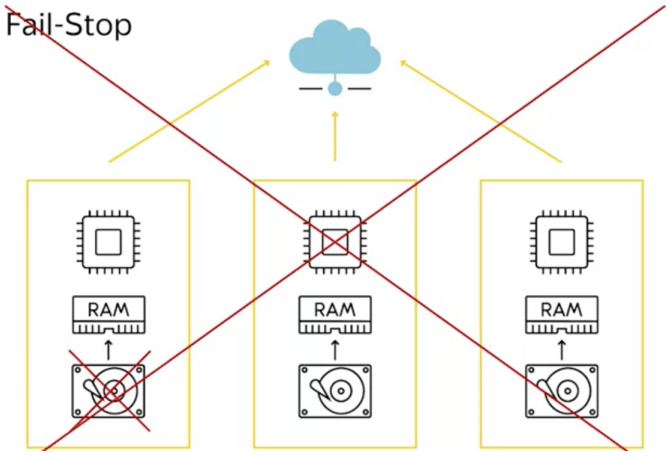
Each and All can break.

cluster nodes can break any time.. power supply, disk damages, overheated CPUs, ..

Node Failure

- **Fail-Stop**

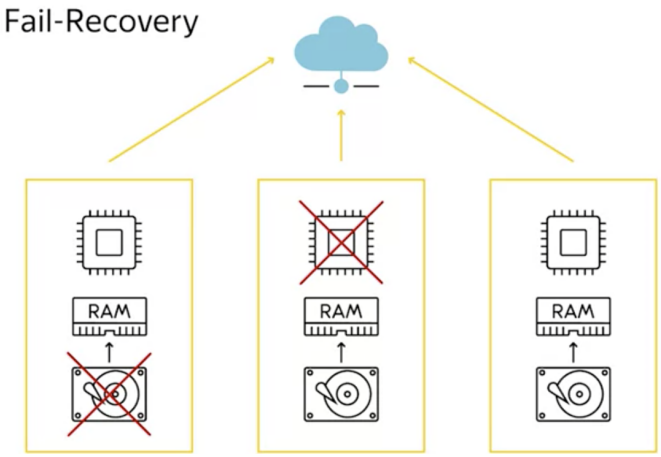
  external impact necessary. 노드에 문제가 생길 경우, 시스템관리자는 노드를 수정하고, 전체/일부 시스템을 재부팅해야함, 혹은 분산시스템을 다시 구성해야함 (reconfigure)



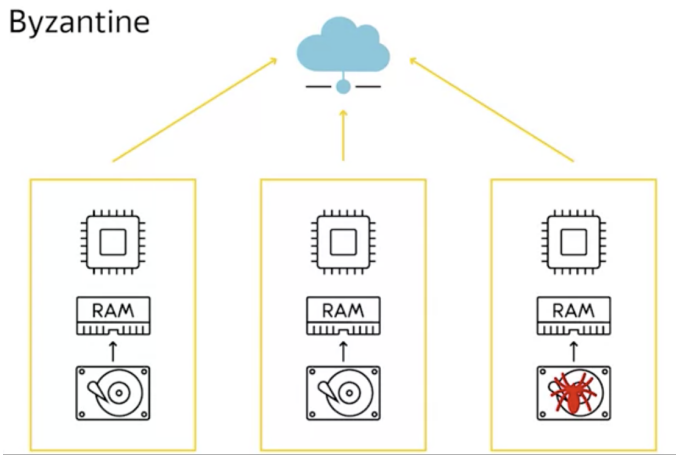
- **Fail-Recovery**

  during computation, notes can arbitrarily crash and return back to servers

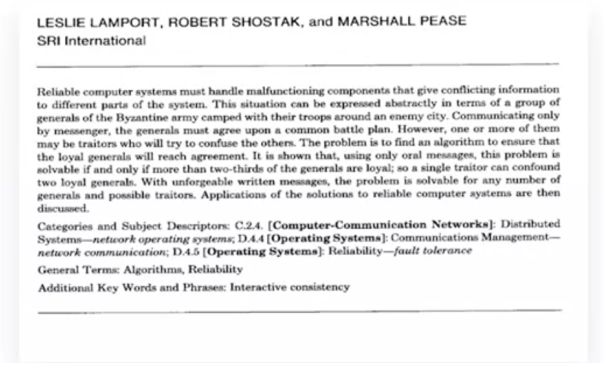
  doesn't influence correctness and succeess of computations, no external impact necessary e.g.) machine의 일부분이 고장나면 시스템이 스스로 해당 machine을 제외하여 서비스를 수행하고 물리적 수리를 끝내면 관리자 관여 없이 시스템이 알아서 서비스에 연결한다



- **Byzantine**



The Byzantine Generals Problem



LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement. It is shown that, using only oral messages, this problem is solvable if and only if more than two-thirds of the generals are loyal; so a single traitor can confound two loyal generals. With unforgeable written messages, the problem is solvable for any number of generals and possible traitors. Applications of the solutions to reliable computer systems are then discussed.

Categories and Subject Descriptors: C.2.4. [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.4 [**Operating Systems**]: Communications Management—*network communication*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Interactive consistency

The Byzantine Generals Problem

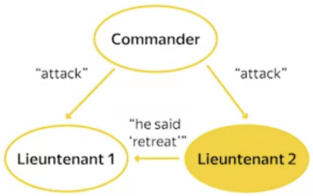


Fig. 1. Lieutenant 2 a traitor.

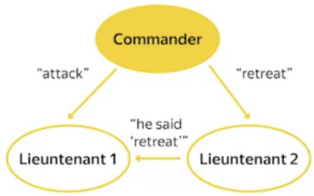
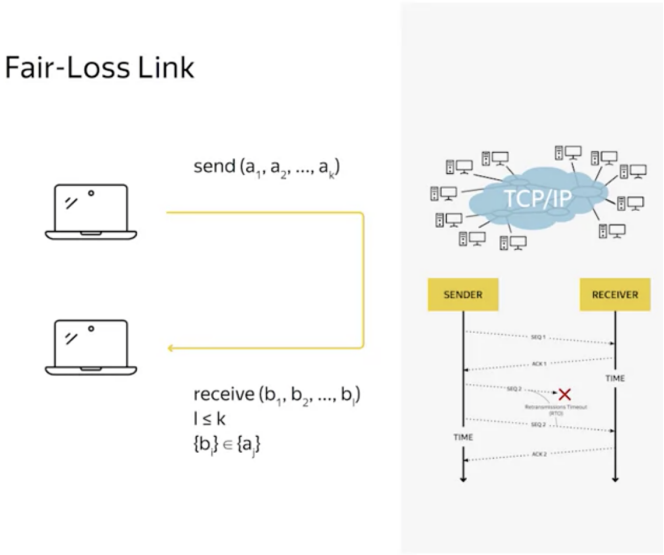


Fig. 2. The commander a traitor.

Link Failures

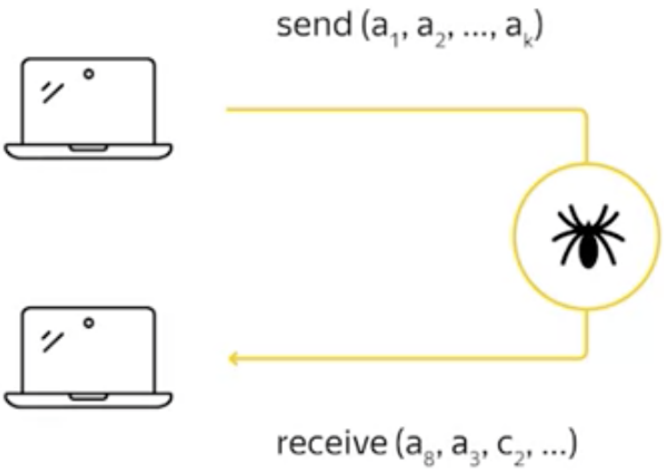
- **Perfect link**
- **Fair-loss link**

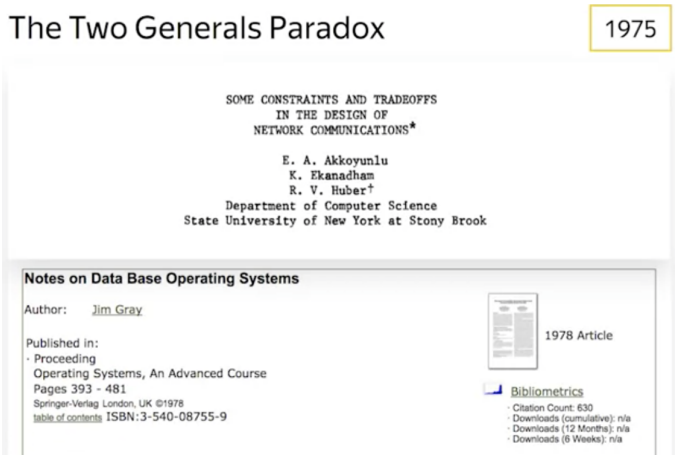
packet loss 발생 가능, 재전송하여 해결

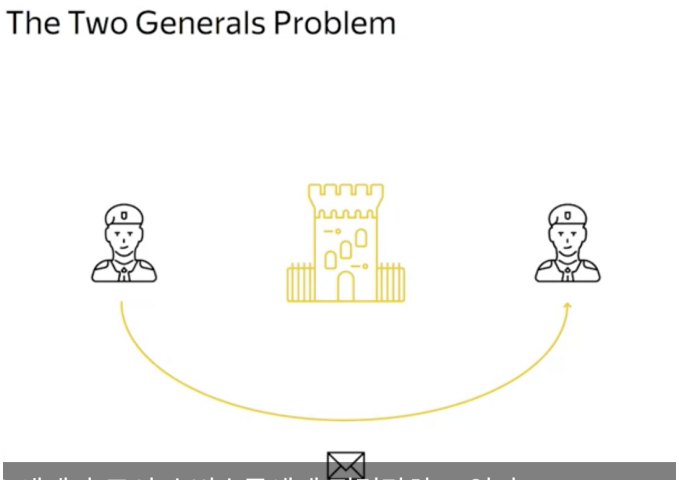


- **Byzantine**

  수정되어 보내질지도. send, ACK.. 계속 반복

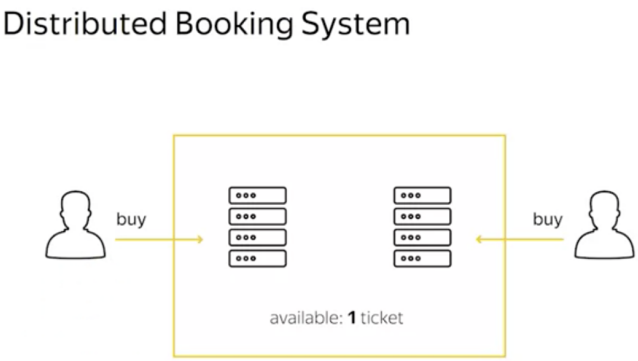


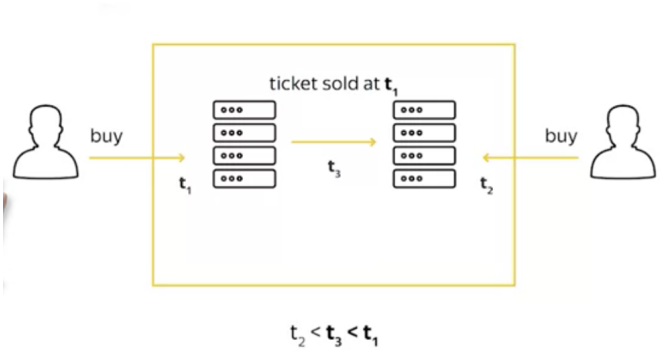




Clock

- Distributed Booking System 동시에





- **clock synchronization problem**

1 clock skew

2 clock drift

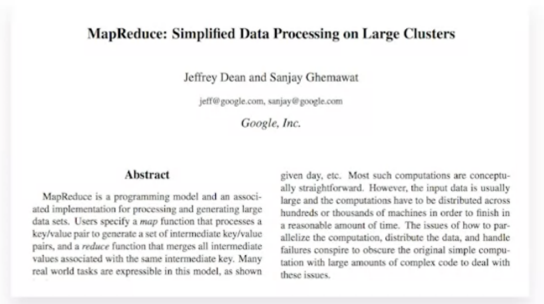
- **clock skew**: 다른 machine 다른 time
- **clock drift**: different clock rate → use Lamport logical clocks, help to track happened before events and therefore, order events to build reliable protocols

[A] Synchronous Systems

› Every message between nodes is delivered within limited time;
› Clock drift is limited;
› Each instruction execution is also limited.

MapReduce

MapReduce



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

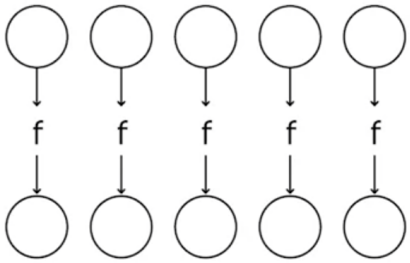
Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

Map: 모든 컬렉션에 동일한 함수 적용

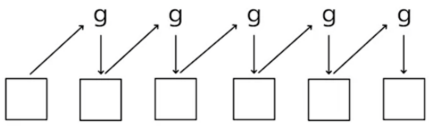



```
>>> map(lambda x: x*x, [1,2,3,4])
[1, 4, 9, 16]
```

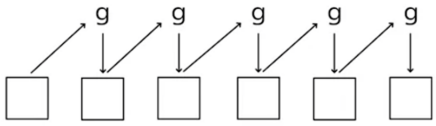
Reduce: 컬렉션 왼쪽에서 오른쪽으로 iterative하게 값 계산, 결국 30

changing the order of the atoms does not change the sum, but changing the application order definitely effects the result.

Fold / Reduce / Aggregate




```
>>> reduce(operator.sum, [1, 4, 9, 16])
>>> reduce(operator.sum, [5 = 1 + 4, 9, 16])
```

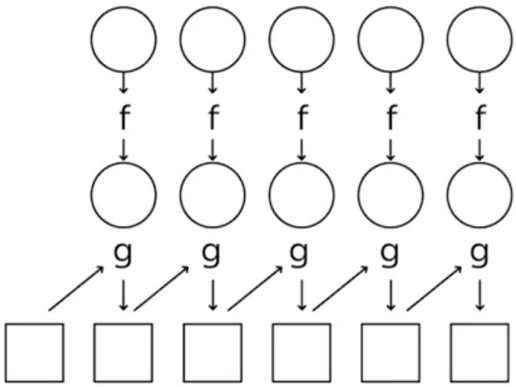



```
>>> average = lambda x, y: (x + y) / 2.
>>> reduce(average, [1, 2, 3])
>>> reduce(average, [1.5, 3])
2.25
```

```
>>> from functools import reduce
>>> import operator
>>> reduce(operator.add, [1,4,9,16])
30
```

```
from functools import reduce
import operator

reduce (operator.add, [1,4,9,16])
reduce (operator.add, [5,9,16])
reduce (operator.add, [14,16])
```




```
>>> reduce(operator.add, map(lambda x: x*x,
[1, 2, 3, 4]))
```

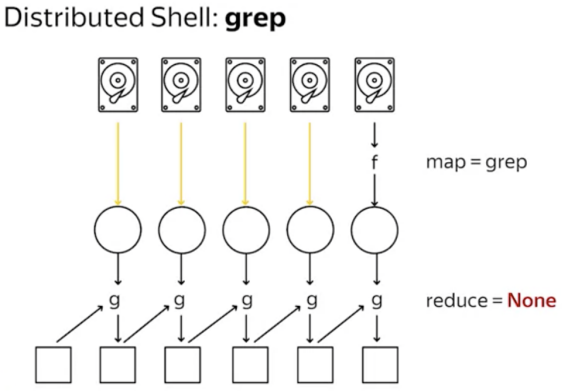
MapReduce

```
>>> reduce(operator.add, map(lambda x: x*x, [1,2,3,4]))
30
```

Distributed Shell

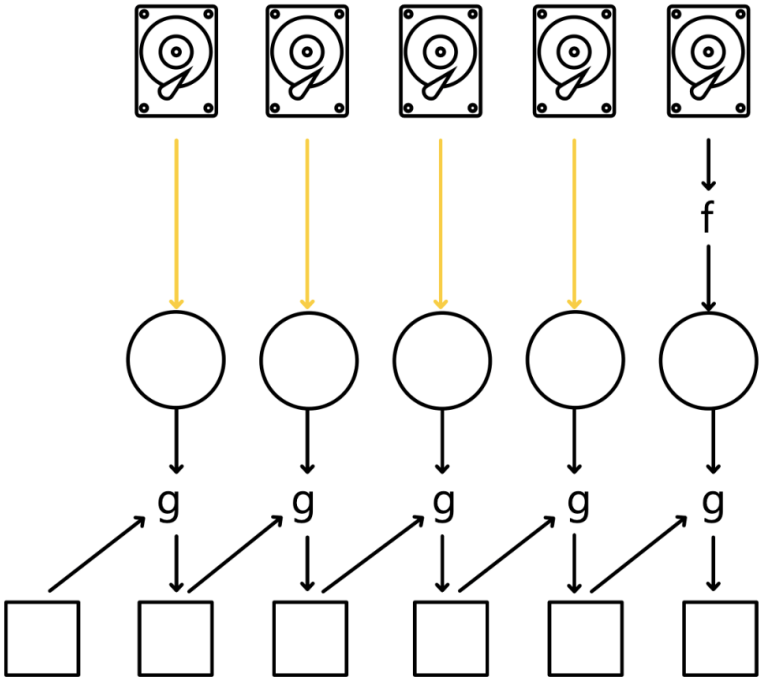
- **grep** 항상 map과 reduce 모두 필요한 것은 아님

```
$ grep <pattern> <file> //used to find matches in files
ex) $ grep "hadoop" A.txt
$ grep -i "hadoop" A.txt
$ man grep
```



- **head** 항상 map/reduce로 해결할 것이 아니라 단순 command line으로 해결 가능

```
$ head <file>
```



~~map = head (?)~~

파일이 분산 저장되어 있기 때문에
head를 수행하기 위해
map 작업은 비효율적이고 복잡하다.

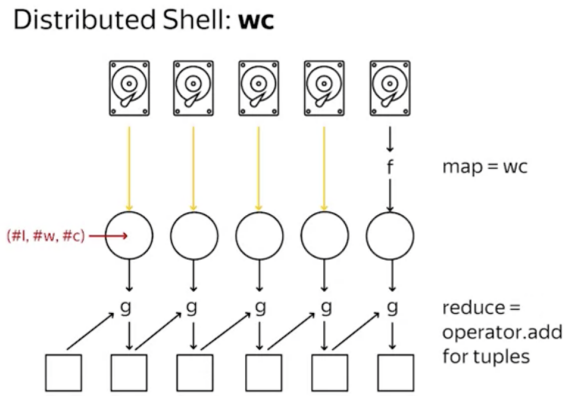
hdfs client 명령어로 수행할 수 있다.

~~reduce = None (?)~~

```
hdfs dfs -text distributed_A.txt | head
```

- wc

```
$ wc <file> //lines, words, bytes
```

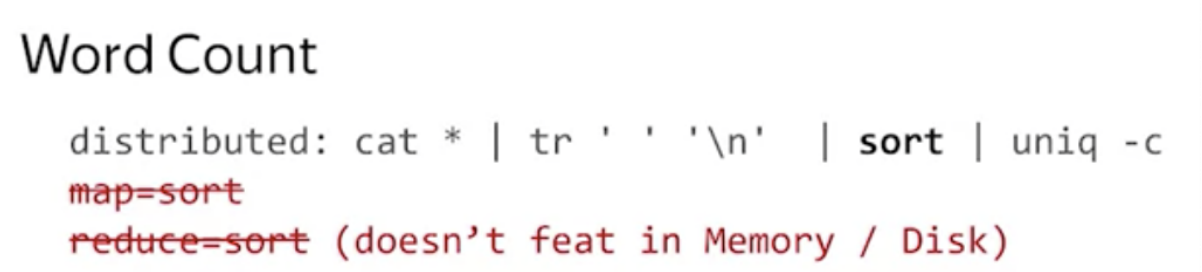


- wordcount on wikipedia

uniq? sort?

uniq: 같은 문장끼리 인접해있어야 count할 수 있음 tr A B: A를 B로 변경

```
one computer
$ cat * | tr ' ' '\n' | sort | uniq -c
```



sort 명령어는 map 명령 수행 불가능 (모든 단어가 한 node에 있어야 가능), reduce 명령 불가능 (메모리 제한)

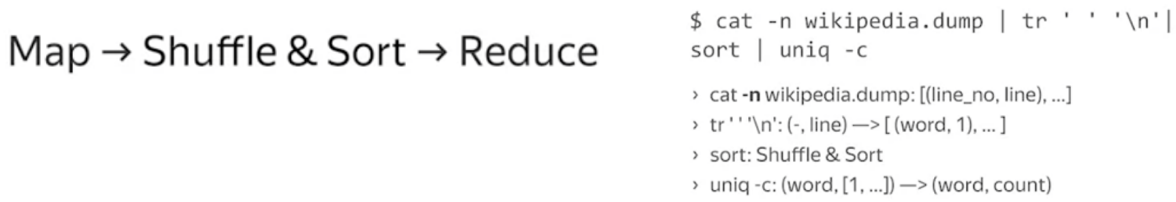
⇒ shuffle & sort

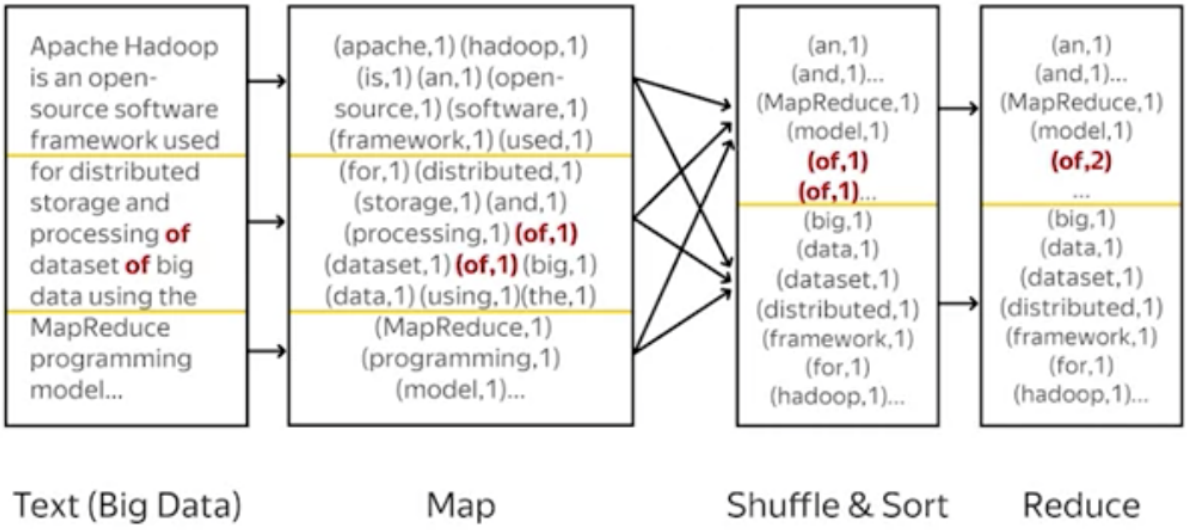
- shuffle & sort

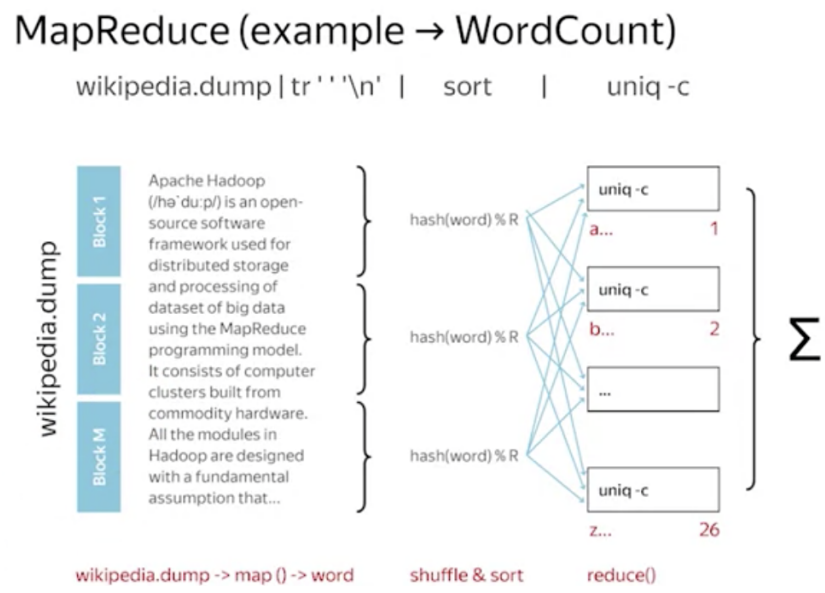
shuffle: map 결과의 key들은 mapreduce framework에 의해 정렬된다. 정렬된 key 값을 기준으로 데이터가 보내질 reducer 위치가 결정되고 전송하는 과정

sort (by key): reducer가 여러 mapper에서 온 data를 다시 key-value pair로 정렬

e.g.) (a, 1), (b,2) (a, 2) (c,3) (b, 4)=> (a, [1,2]) (b, [2,4]) (c,3)





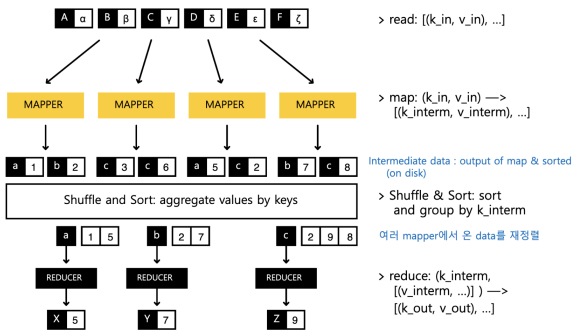


일반적으로 3종류의 key-value 쌍이 있음 input, intermediate, output k-v pair

WordCount example

```
$ cat -n wikipedia.dump | tr ' ' '\n'|
sort | uniq -c
```

› cat **-n** wikipedia.dump: [(line_no, line), ...]
› **read: [(k_in, v_in), ...]**
› tr ' ' '\n': (-, line) —> [(word, 1), ...]
› **map: (k_in, v_in) —> [(k_interm, v_interm), ...]**
› **Shuffle & Sort: sort and group by k_interm**
› uniq -c: (word, [1, ...]) —> (word, count)
› **reduce: (k_interm, [(v_interm, ...)]) —> [(k_out, v_out), ...]**



reduce function: **max**

MapReduce

› You know the phases of MapReduce: Map, Shuffle & Sort, Reduce;

› You know how to solve simple tasks such as distributed "grep", "head", "wc" and "Word Count" with MapReduce.

If you would like to group key-values pairs by key. How do you solve this task with Hadoop MapReduce? (please provide the map and reduce functions' interfaces)

map: identity, reduce: identity

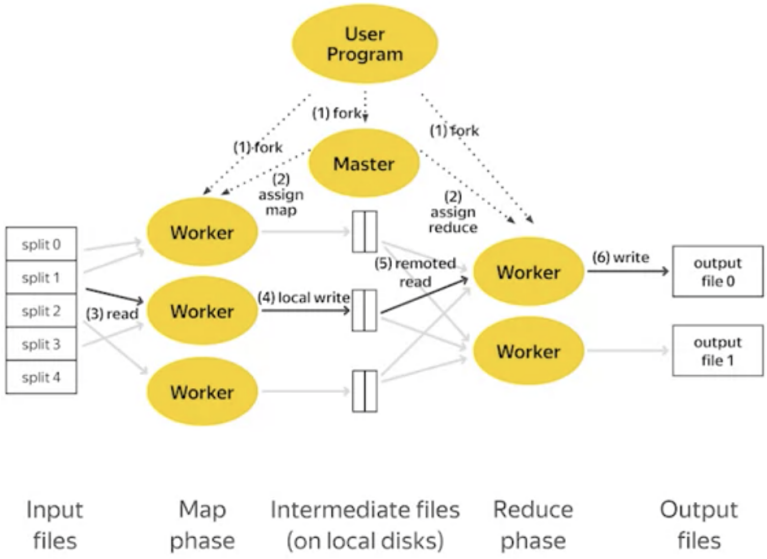
you are asked to count how many times each word occurs in a dataset. Please, provide the map and reduce functions' interfaces to complete this job.

- map: (docid, line) → [(word, 1), ...]
- reduce: (word, [1, 1, ...]) → (word, count)

Fault Tolerance

HDFS: unreliable and cheap nodes.. but any computational model should be robust to failures.

MapReduce framework, fault tolerance model.. to overcome node failtures, duplication exists.



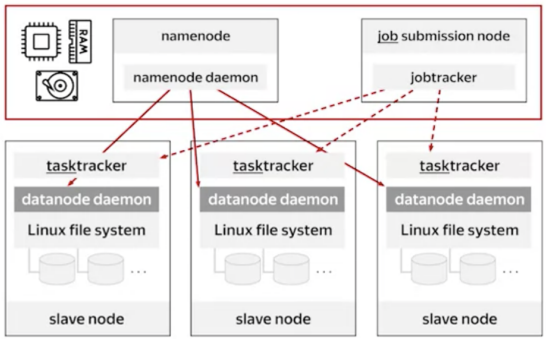
모든 것을 관리하는 application master 존재, map 단계에서, reduce 단계에서 각각 worker가 죽었을 때 다른 worker로 위임, → 기능적 요인때문에 intermediate files가 local에 저장되는 것. 이게 죽었을 때는 앞에서부터 다시 하면 됨

⇒ **deterministic** functional programming.. map, reduce 함수를 제공하는 것에 초점...

⇒ but **Application Master** is a single point of failure in this case. If it dies, then the execution of the whole job is cancelled.

- **jobs**(one mapreduce application), **tasks**(can be either mapper or reducer)

Hadoop MRv1



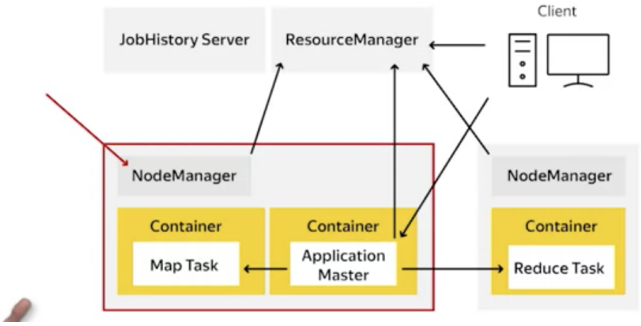
버전1

하나의 global **jobtracker**가 master 역할 수행

tasktracker: mapper or reducer 관리

cf) namenode: dfs 메타데이터 담당

MRv1 vs YARN

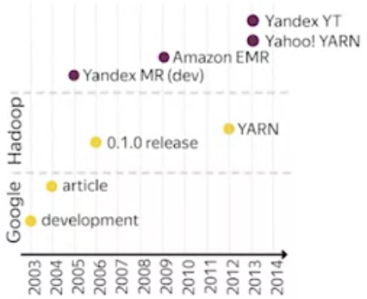


global jobtracker X. **ResourceManager**: cluster resources and processes requests from NodeManagers.

ApplicationManager: A service to run and monitor containers for application-specific processes on cluster nodes. jobtracker 대신 application master는 어느 node에서든 실행 가능

tasktracker X. **NodeManager**: CPU와 RAM을 가진 컨테이너를 생성 및 관리

MapReduce Frameworks: History Timeline



- [2003] Google MapReduce (development)
- [2004] Google MapReduce (article)
- [2005] Yandex MapReduce (development)
- [2006] Hadoop 0.1.0 release
- [2009] Amazon EMR (Hadoop inside)
- [2012] MapReduce —> YARN
- [2013] Yahoo! YARN deployed in production
- [2013] Yandex YT
- ...
- MapReduce in MongoDB, Riak, ...

MapReduce Framework

- You can explain what will happen if Mapper or Reducer dies;
- You know what JobTracker and TaskTracer in MRv1, ResourceManager and NodeManager in YARN are.

Hadoop MapReduce Intro (Quiz)