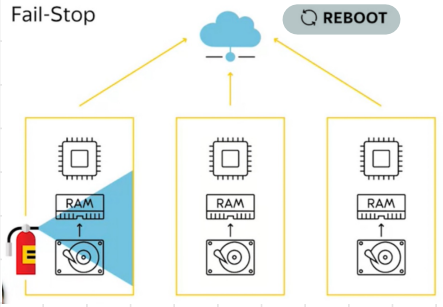
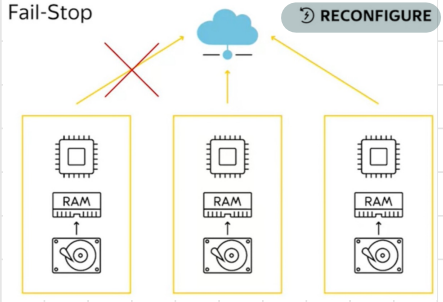
Hadoop MapReduce:

How to build reliable system from unreliable components.

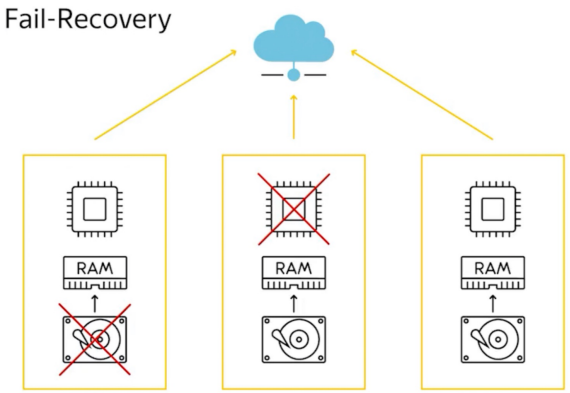
- Distributed systems are often built from unreliable components, cluster nodes can break any time because of power supply, disk damages, overheated CPUs, and so on.
- Types of Node Failure.

  1, Fail - Stop
  - means that if machines get out of service during a computation, then you have to have an external impact to bring system back to a working state.
    - Reboot: fix the node and reboot the whole system or part of.
    - Reconfigure: retire the broken machine and reconfigure the distributed system, not robust to node crashes.



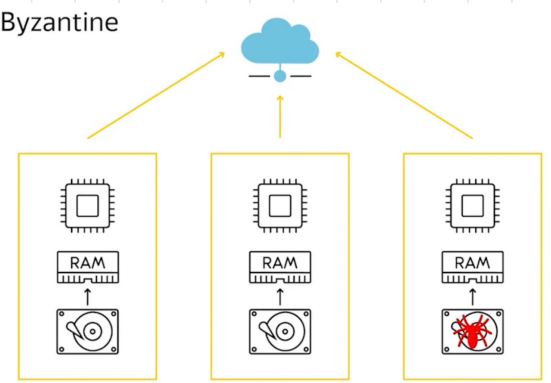


2, **Fail Recovery**.



- during computations, nodes can arbitrary crash and return back to servers.
- doesn't influence correctness and success of computation. no external impact necessary to reconfiguring the system.
- If hard drive was damaged, system administrator can physically change the hard drive. After reconnection, this node will be automatically picked up by a distributed system.

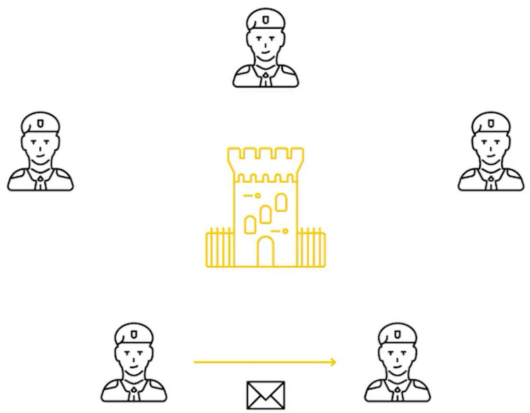
3, **Byzantine failure**.



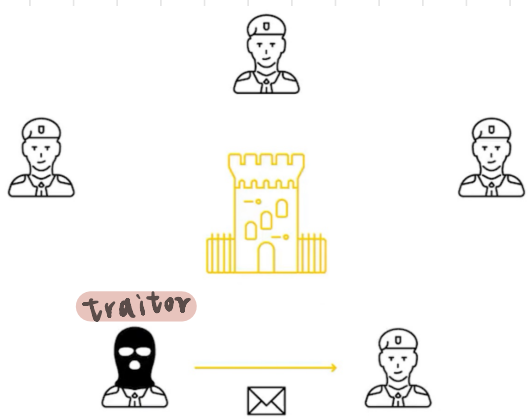
- A distributed system is robust Byzantine failure if it can work

despite some of nodes behaving out of protocal.

- **Byzantine Generals' Problem**
  - group of generals of the Byzantine army camped with their troops around an enemy city, communicating only by a messenger, the generals must agree upon a common battle plan.



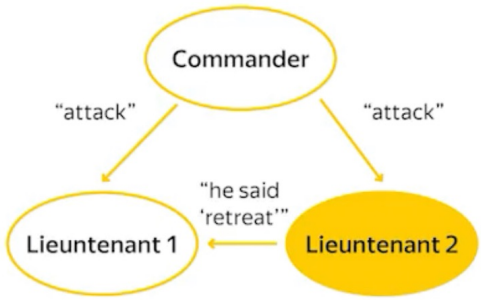
However, one or more of them may be a traitor who trying to confuse the others.



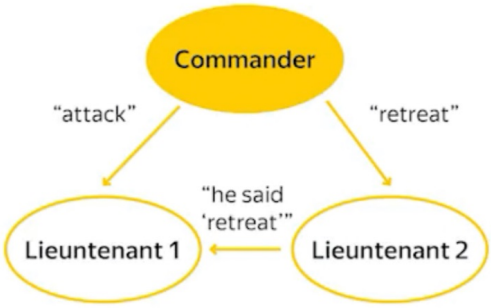
It is shown that using only unsigned messages, you have to have more than 2/3 of loyal generals.

In the case of one commanding general, two lieuntenant:

1. If second lieuntenant is traitor.

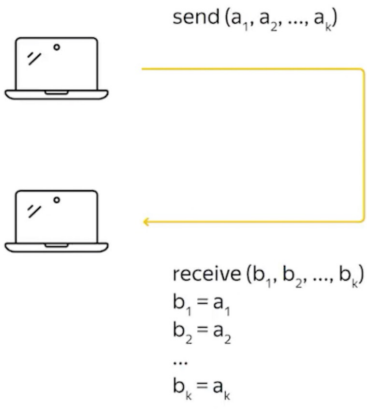


2. If commander is traitor.



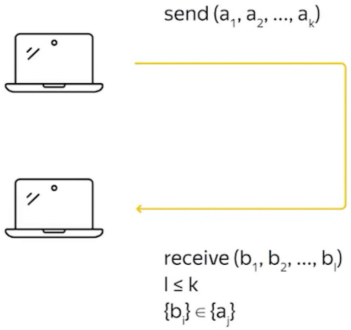
- Types of Link Failures

1. **Perfect Link**:

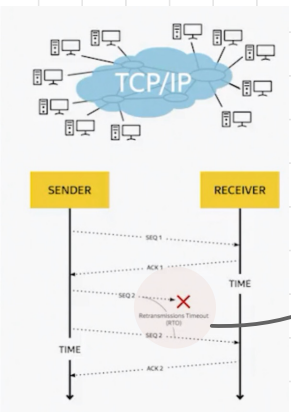


all the sent messages must be delivered and received without any modification in the same order.

2. Fair-Loss Link:

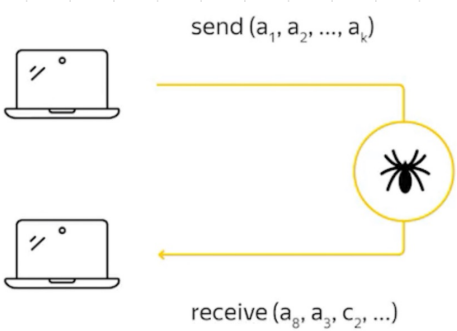


some part of the messages can be lost, but the probability of message loss doesn't depend on contents of a message.



Packet loss is a very common problem for network connection. For instance, the well-known TCP/IP protocol tries to solve this problem by re-transmitting messages if they were not received

3. Byzantine Link



Some messages can be filtered according to some rule, and some messages could be created out of nowhere. Nothing comes out of nowhere according to the law of conversation of energy.

- **The Two Generals Paradox.**
  - started in 1975 and then popularized by Jim Gray in 1978.
  - the first computer communication problem to be proved **unsolvable**
  - There are two generals on a campaign. They have an objective, if they simultaneously march the objective, they win. They can communicate only via messages. Unfortunately, the world is occupied by the city's defenders and there is a chance that any given method sent through the wall will be captured. The problem is to find a protocol that allows the generals to march together even. There is a simple proof that **no fixed plans protocol exists**.
    1. The first general send a message "attack at 5a.m." how does he know that the second general will receive?
    2. The second general should send an acknowledgement message comfirmed. But, how does he know that the first general will receive the acknowledgement message?

    ⇒ **continue infinitely!!**

1.

2.

- Types of clock synchronization problem.

  1, clock skew: time can be different on different machines.

  2, clock drift: different clock rate.

  - Any clock synchronization mechanism is subject to some precision,

    ⇒ logical clock were invented.

    - help to track happened before events, and therefore, order events to build reliable protocols.
    - logical clocks were named after his inventor, Leslie Lamport: Lamport logical clocks.

- Synchronous System.

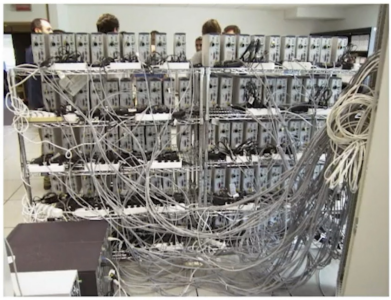
  - every messages between nodes is delivered within limited time.
  - clock drift is limited
  - each instruction exection is also limited.

- Different Distributed Systems.

  1, Fail-stop + Perfect Link + Synchronous

  - usually refevred as a parallel computation model and widely adopted by supercomputers where many process connected by a local high speed computer bus.

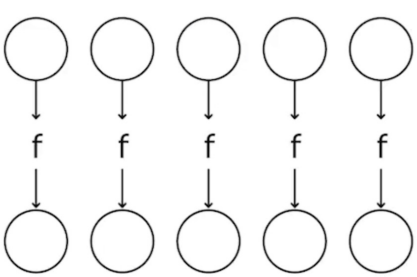
2. Fail Recovery + Fair-Loss Link + Asynchronous.



3. Byzantine-Failure + Byzantine Link + Asynchronous

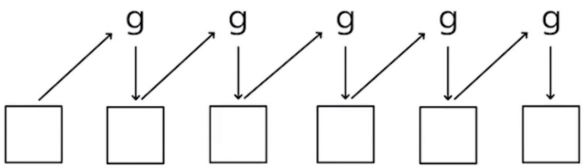
- It is a model adopted for the systems where computational components spread across the globe of unreliable and untrusted network connections.
- The common representative of this model is grid computing.

- MapReduce
  - Invented by Jeffery Dean and Sanjay Ghenawat, presented on Symposium and Operating Systems Design and Implementation in 2004.
  - Map: apply the same function to each element of your collection.

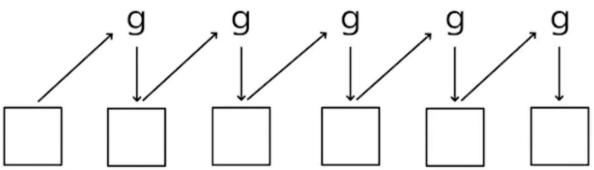



```
>>> map(lambda x: x*x, [1,2,3,4])
[1, 4, 9, 16]
```

- **Reduce** (Fold / Aggregate)




```
>>> reduce(operator.sum, [1, 4, 9, 16])
>>> reduce(operator.sum, [5, 9, 16])
>>> reduce(operator.sum, [14, 16])
30
```

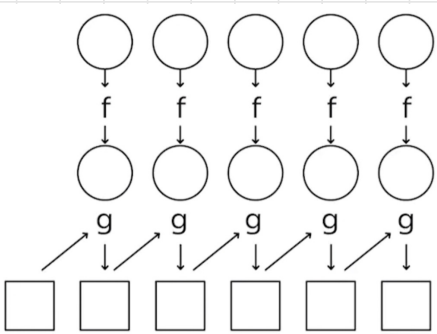



```
>>> average = lambda x, y: (x + y) / 2.
>>> reduce(average, [1, 2, 3])
2.25
>>> reduce(average, [3, 2, 1])
>>> reduce(average, [2.5, 1])
1.75
```

different result.

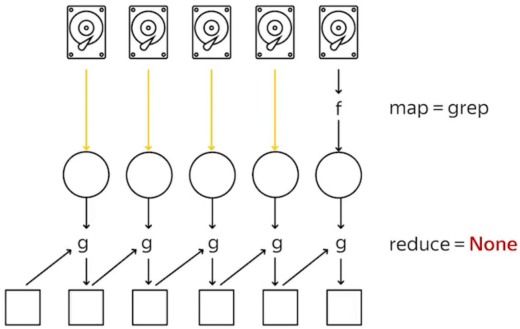
- Reduce operator cause a sequence of elements by applying the following procedure iteratively.
- reducing function sum is associative.
- mean function is not associative: changing the order of the atom effects the result.

- **MapReduce**: combine Map and Reduce together.



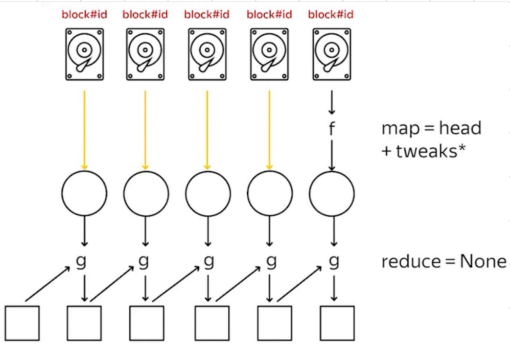

```
>>> reduce(operator.add, map(lambda x: x*x,
[1, 2, 3, 4]))
30
```

- Distributed Shell: grep

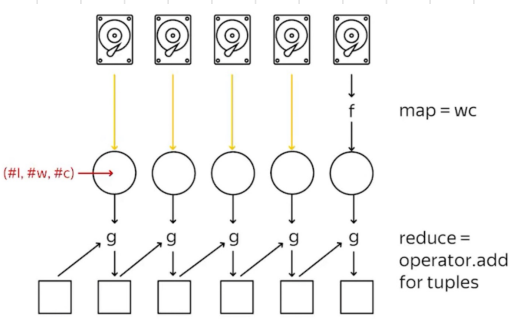


(in MapReduce application, you don't always need map or reduce function)

- Distributed Shell: head



- Distributed Shell: wc



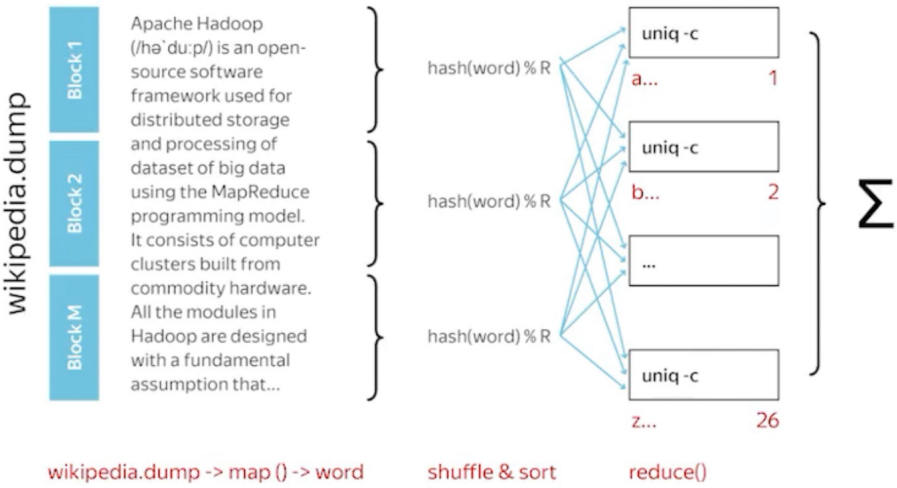
- Distributed Shell: Word Count

```
distributed: cat * | tr ' ' '\n'  | sort | uniq -c
map=sort
reduce=sort (doesn't feat in Memory / Disk)
```

⇒ introduces shuffle and sort.

Map → Shuffle & Sort → Reduce

- text is split into words.
- words are distributed to a reduce phase in the way that reduce function can be executed independently on different machines.



external sorting

- MapReduce Formal Model.
  - All input and output of map and reduce function should be a key value pair.

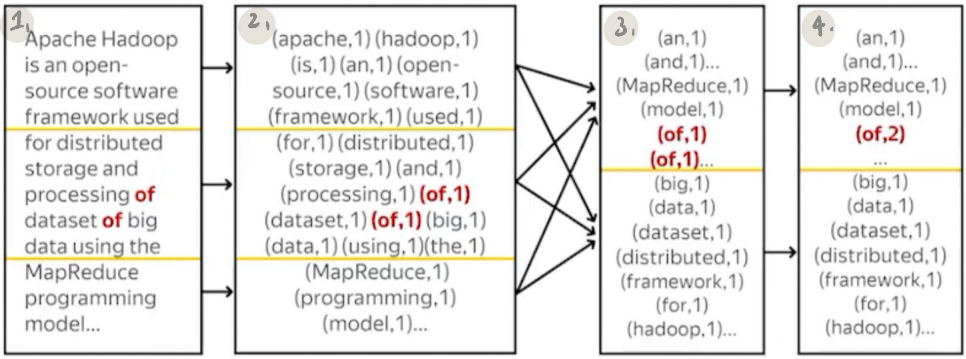
    map: (key, value) → (key, value)
    reduce: (key, value) → (key, value)

  - WordCount example

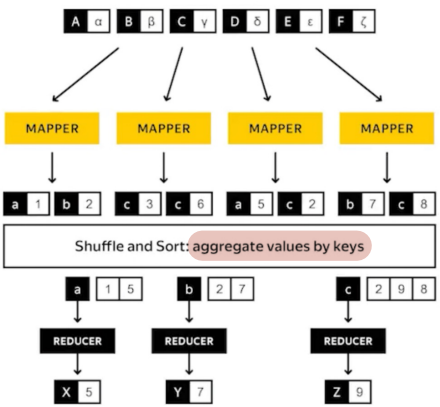
```
$ cat -n wikipedia.dump | tr ' ' '\n'|
sort | uniq -c
```

› cat **-n** wikipedia.dump: [(line_no, line), ...] 1,
› tr ' ' '\n': (-, line) —> [(word, 1), ...] 2,
› sort: Shuffle & Sort 3,
› uniq -c: (word, [1, ...]) —> (word, count) 4.

1, read data and get pairs with line number, line content.
2, map phase: ignore line number and split lines into words.
3, shuffle and sort phase: spread the words by the hashes.
4. reduce phase: sum up to get the answer.



Text (Big Data) Map Shuffle & Sort Reduce



› read: [(k_in, v_in), ...]

› map: (k_in, v_in) —> [(k_interm, v_interm), ...] ← intermediate key-value pairs.

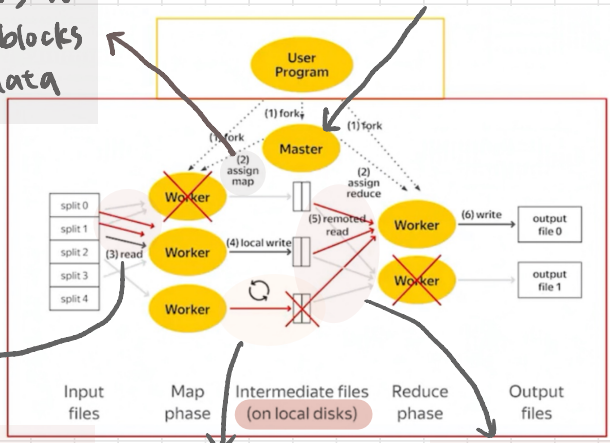
› Shuffle & Sort: sort and group by k_interm ← aggregated by intermediate keys.

› reduce: (k_interm, [(v_interm, ...)]) —> [(k_out, v_out), ...] ← transformed by reducers, can be stored on local disks of a distributed system.

- MapReduce framework: **Fault Tolerance Model**

launch mappers to process input blocks or splits of data

control the execution

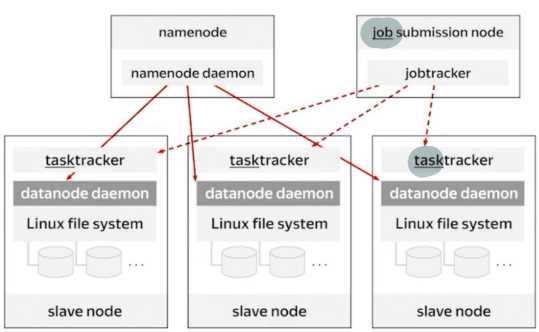


assign another worker to execute mapper against these data.

re-execute mapper on another alive worker if this data is lost

Shuffle and sort data for this particular reducer to another worker.

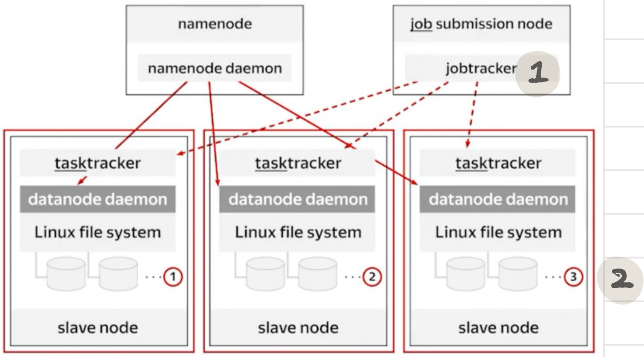
  - **All this complexity is hidden in MapReduce framwork** to make your life way more easier.
  - You only need to provide deterministic map and reduce functions.
- MapReduce concepts



jobs: One MapReduce application is a job, is a big chunck of work.

task: A map or reduce function applied to some chuck of data.

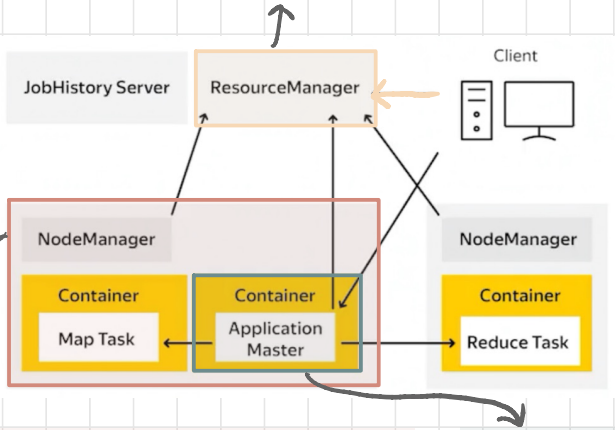
- Hadoop MapReduce v1.



1. There was one global JobTracker to direct execution of MapReduce jobs. It is usually located on one high-cost and high-performance node with HDFS namenode.
2. TaskTrackers are located once per every node where you store data or where datanode daemon is working. TaskTracker spawns workers from mapper or reducer.
3. In this scenario, JobTracker is a single point of failure. That is why to reduce the load on JobTracker, some functionality of future version was delegated to other cluster nodes.

- **YARN** (Yet Another Resource Negotiation)

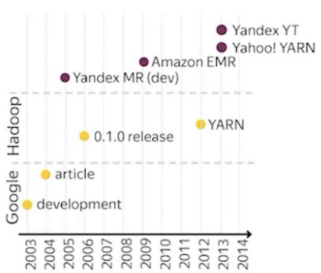
ResourceManager oversees NodeManagers and client request resources for execution. MapReduce applications can work on top of this resource layer.



TaskTrackers are substituted by NodeMangers who can provide a layer of CPU and RAM containers.

No global JobTracker because Application Master can start on any node.

- MapReduce Frameworks: History Timeline

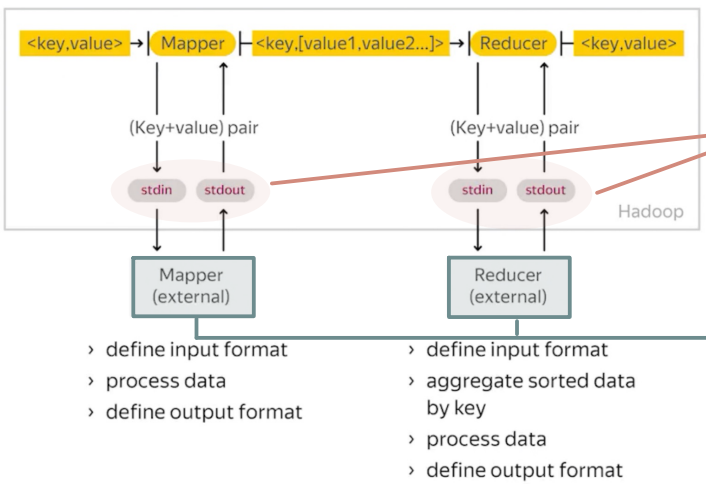


- [2003] Google MapReduce (development)
- [2004] Google MapReduce (article)
- [2005] Yandex MapReduce (development)
- [2006] Hadoop 0.1.0 release
- [2009] Amazon EMR (Hadoop inside)
- [2012] MapReduce —> YARN
- [2013] Yahoo! YARN deployed in production
- [2013] Yandex YT
- ...
- MapReduce in MongoDB,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

Hadoop MapReduce Streaming Applications in Python.

- Streaming

Hadoop provides a possibility to write mappers, and reducers in any language, which is called streaming.



communicate via standard input and output channels.

You have to implement your own mappers and reducers.

- example: Line Count.

```
/opt/cloudera/parcels/CDH-5.9.0-1.cdh5.9.0.p0.23/lib/hadoop-mapreduce/hadoop-streaming.jar

HADOOP_STREAMING_JAR="/path/to/hadoop-streaming.jar"
yarn jar $HADOOP_STREAMING_JAR \
        -mapper 'wc -l' \
        -numReduceTasks 0 \
        -input /data/wiki/en_articles \
        -output wc_mr
```

define the path to the streaming jar.

a bash command mapper.

need to be specified if no reducer.

HDFS folder or file going to process.

HDFS folder for output.

execute yarn application.

output:

```
$ hdfs dfs -ls wc_mr
Found 3 items
-rw-r--r--   3 adral adral      0 2017-03-21 14:48 wc_mr/_SUCCESS
-rw-r--r--   3 adral adral      6 2017-03-21 14:48 wc_mr/part-00000 (1)
-rw-r--r--   3 adral adral      6 2017-03-21 14:48 wc_mr/part-00001 (2)
```

only two mappers were executed.

```
$ hdfs dfs -text wc_mr/*
1986
2114
```

1986 + 2114 = 4100 Wikipedia articles.

provide reducer which aggregates the number of articles from all the mappers:

```
HADOOP_STREAMING_JAR="/path/to/hadoop-streaming.jar"
yarn jar $HADOOP_STREAMING_JAR \
          -mapper 'wc -l' \
          -reducer "awk '{line_count += \$1} END { print line_count
}'" \
          -numReduceTasks 1 \
          -input /data/wiki/en_articles \
          -output wc_mr
```

escape special characters in the Shell map and reduce commands

```
$ hdfs dfs -ls wc_mr_with_reducer
Found 2 items
-rw-r--r--   3 adral adral       wc_mr_with_reducer/_SUCCESS
-rw-r--r--   3 adral adral       wc_mr_with_reducer/part-00000

$ hdfs dfs -text wc_mr_with_reducer/*
4100
```

provide reducer from shell script:

reducer.sh

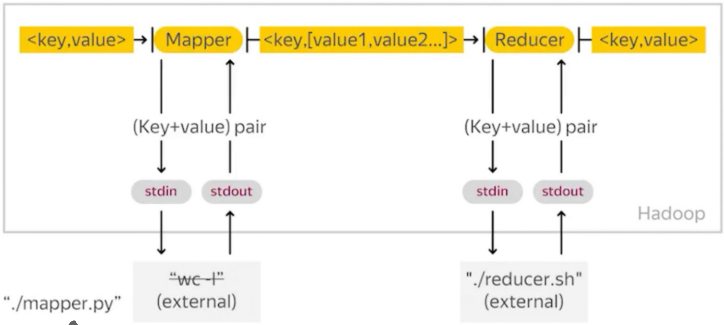
```
#!/usr/bin/env bash
awk '{line_count += $1} END { print line_count }'
```

don't need escaping.

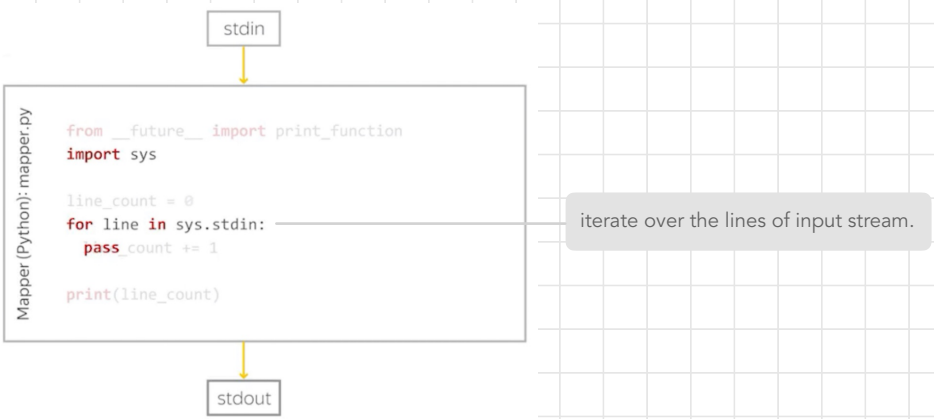
```
HADOOP_STREAMING_JAR="/path/to/hadoop-streaming.jar"
yarn jar $HADOOP_STREAMING_JAR \
          -mapper 'wc -l' \
          -reducer './reducer.sh' \
          -file reducer.sh \
          -numReduceTasks 1 \
          -input /data/wiki/en_articles \
          -output wc_mr_with_reducer
```

copy reducer.sh from the local storage to be available for MapReduce workers.

- Streaming in Python.



replace shell command to Python script that reads data from standard input, and prints it out to the standard output.




```
HADOOP_STREAMING_JAR="/path/to/hadoop-streaming.jar"
yarn jar $HADOOP_STREAMING_JAR \
    -files mapper.py,reducer.sh \
    -mapper 'python mapper.py' \
    -reducer './reducer.sh' \
    -numReduceTasks 1 \
    -input /data/wiki/en_articles \
    -output wc_mr_with_reducer
```

specify multiple files to distribute over the workers as a comma separated list.

use python mapper.

```
$ hdfs dfs -ls wc_mr_with_reducer
Found 2 items
-rw-r--r-- 3 adral adral 0 <date> wc_mr_with_reducer/_SUCCESS
-rw-r--r-- 3 adral adral 0 <date> wc_mr_with_reducer/part-00000
```

# of replicas filesize

```
$ hdfs dfs -text wc_mr_with_reducer/*
--
```

empty output by running this streaming MapReduce job.

stdin

Mapper (Python): mapper.py

```python
from __future__ import print_function
import sys

line_count = 0
for line in sys.stdin:
  line_count += 1

print(line_count)
```

provide standard output from Python.

stdout

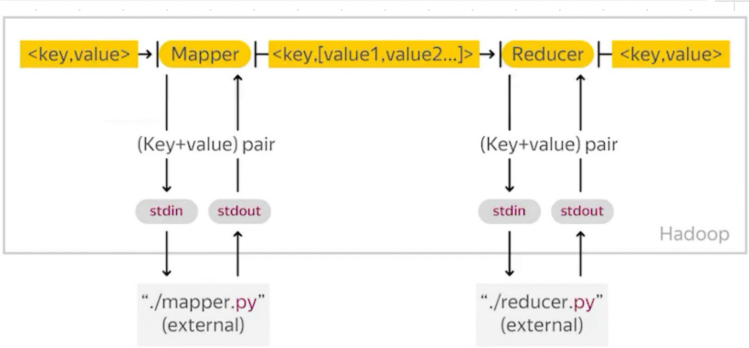
stdin

Reducer (Python): reducer.py

```python
from __future__ import print_function
import sys

line_count = sum(
   int(value) for value in sys.stdin
)

print(line_count)
```

stdout

```
HADOOP_STREAMING_JAR="/path/to/hadoop-streaming.jar"
yarn jar $HADOOP_STREAMING_JAR \
        -files mapper.py,reducer.py \
        -mapper 'python mapper.py' \
        -reducer 'python reducer.py' \
        -numReduceTasks 1 \
        -input /data/wiki/en_articles \
        -output wc_mr_with_reducer
```



- Word Count in Python.
  - key-value pairs for the input and output streams.

<article id> <tab> <article content>

key value

key: before tab , value: after tab

if no tab character in the line:

<article content>

key value: None

mapper.py

```
from __future__ import print_function
import sys

for line in sys.stdin:
 article_id, content = line.split("\t", 1)
    words = content.split()
    for word in words:
        print(word, 1, sep="\t")
```

```
yarn jar $HADOOP_STREAMING_JAR \
          -files mapper.py \
          -mapper 'python mapper.py' \
          -numReduceTasks 0 \
          -input /data/wiki/en_articles \
          -output word_count
```

```
$ hdfs dfs -text /data/wiki/en_articles/* | head -c 80
12 <tab> Anarchism          Anarchism is often defined as a
political philosophy which …
```

```
$ hdfs dfs -ls -h word_count
Found 3 items
-rw-r--r--   3 adral adral   0 2017-03-22 11:40 word_count/_SUCCESS
-rw-r--r--   3 adral adral  47.8 M 2017-03-22 11:40 word_count/part-00000
-rw-r--r--   3 adral adral  47.9 M 2017-03-22 11:40 word_count/part-00001
```

result:

part-00000	part-00001
Basel 1	Anarchism 1
Basel 1	Anarchism 1
(1	is 1
) 1	often 1
or 1	defined 1
…	…

```
yarn jar $HADOOP_STREAMING_JAR \
          -files mapper.py \
          -mapper 'python mapper.py' \
          -numReduceTasks 1 \
          -input /data/wiki/en_articles \
          -output word_count
```

one reducer with default implementation which does nothing, the shuffle and sort phase will be executed.

```
$ hdfs dfs -text word_count/part-00000 | head
! 1
! 1
! 1
! 1
! 1
…
```

sorted output:

modify mapper.py

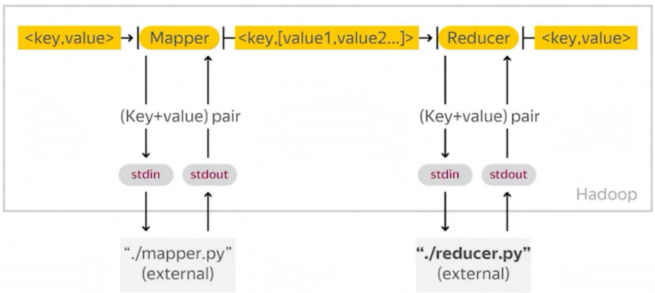
```
from __future__ import print_function
import re
import sys

for line in sys.stdin:
 article_id, content = line.split("\t", 1)
    words = re.split("\W+", content)
    for word in words:
        if word:
            print(word, 1, sep="\t")
```

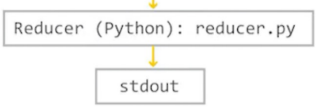
split all non-word characters.

```
yarn jar $HADOOP_STREAMING_JAR \
           -files mapper.py \
           -mapper 'python mapper.py' \
           -numReduceTasks 1 \
           -input /data/wiki/en_articles \
           -output word_count
```

```
$ hdfs dfs -text word_count/part-00000 | head -4
0  1
0  1
0  1
0  1
```

```
 $ hdfs dfs -tail word_count/part-00000 | tail -4
 zyu1 1
 zyu1 1
 zz  1
 zz  1
```





- define input format
- **aggregate sorted data by key**
- process data
- define output format

reducer has responsibility to aggregate value by keys.

reducer.py

```
from __future__ import print_function
import sys

current_word = None
word_count = 0

for line in sys.stdin:
    word, counts = line.split("\t", 1)
    counts = int(counts)
    if word == current_word:
        word_count += counts
    else:
        if current_word:
            print(current_word, word_count, sep="\t")
        current_word = word
        word_count = counts

if current_word:
    print(current_word, word_count, sep="\t")
```

initialize current_word and word_count.

parse input key-value pair.

if see the same word, increase the counter.

otherwise,
1. output aggregate stats for previous word.
2. update the counter for a new key.

output accumulated stat for the last key.

```
yarn jar $HADOOP_STREAMING_JAR \
            -files mapper.py,reducer.py \
            -mapper 'python mapper.py' \
            -reducer 'python reducer.py' \
            -numReduceTasks 1 \
            -input /data/wiki/en_articles \
            -output word_count
```

one reducer

```
$ hdfs dfs -ls -h word_count
Found 2 items
-rw-r--r--   3 adral adral          0 2017-03-22 13:05 word_count/_SUCCESS
-rw-r--r--   3 adral adral      3.2 M 2017-03-22 13:05 word_count/part-00000
```

one file in output

```
$ hdfs dfs -text word_count/part-00000
0 14905
00 844
000 8186
…
zymodemes 1
zymogen 2
zymosan 1
zyu1 4
zz 2
```

```
…
zymosan 1
zyu1 1
zyu1 1
zyu1 1
zyu1 1
zz 1
```

```
yarn jar $HADOOP_STREAMING_JAR \
            -files mapper.py,reducer.py \
            -mapper 'python mapper.py' \
            -reducer 'python reducer.py' \
            -input /data/wiki/en_articles \
            -output word_count
```

remove -numReduceTasks argument, MapReduce job will have an arbitrary number of reducer.

```
$ hdfs dfs -ls -h word_count
Found 11 items
-rw-r--r--   3 adral adral          0 2017-03-22 13:19 word_count/_SUCCESS
-rw-r--r--   3 adral adral    331.0 K 2017-03-22 13:18 word_count/part-00000
-rw-r--r--   3 adral adral    332.1 K 2017-03-22 13:18 word_count/part-00001
-rw-r--r--   3 adral adral    331.7 K 2017-03-22 13:18 word_count/part-00002
-rw-r--r--   3 adral adral    329.8 K 2017-03-22 13:18 word_count/part-00003
-rw-r--r--   3 adral adral    326.1 K 2017-03-22 13:18 word_count/part-00004
-rw-r--r--   3 adral adral    332.2 K 2017-03-22 13:18 word_count/part-00005
-rw-r--r--   3 adral adral    332.3 K 2017-03-22 13:18 word_count/part-00006
-rw-r--r--   3 adral adral    331.4 K 2017-03-22 13:18 word_count/part-00007
-rw-r--r--   3 adral adral    330.5 K 2017-03-22 13:19 word_count/part-00008
-rw-r--r--   3 adral adral    330.7 K 2017-03-22 13:19 word_count/part-00009
```

will see several files in the output.

```
$ hdfs dfs -tail word_count/part-… | tail -5
```

part-00000	part-00005
…	…
	zsu 1
zuang 1	
zucchini 5	
	zuchetto 1
zuerst 1	
	zure 1
…	…

In each file, data is sorted by keys, but not globally sorted as I shuffled between reducers.

TotalOrderPartitioner can sort keys between the reducers.

- Distributed Cache

if I only want to count some specific word in word count example ?

mapper.py

```
from __future__ import print_function
import re
import sys

def read_vocabulary(file_path):
    return set(word.strip() for word in open(file_path))

vocabulary = read_vocabulary("vocabulary.txt")

for line in sys.stdin:
    article_id, content = line.split("\t", 1)
    words = re.split("\W+", content)
    for word in words:
        if word in vocabulary:
            print(word, 1, sep="\t")
```

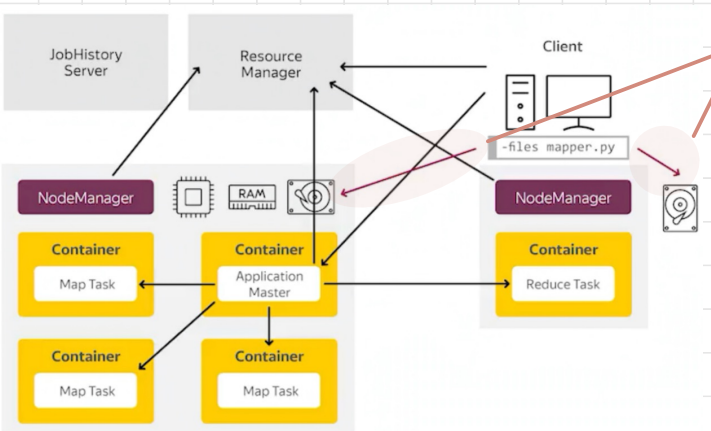
read vocabulary in memory.

filter words by these vocabularies.

```
yarn jar $HADOOP_STREAMING_JAR \
          -files mapper.py,reducer.py,vocabulary.txt \
          -mapper 'python mapper.py' \
          -reducer 'python reducer.py' \
          -input /data/wiki/en_articles \
          -output word_count
```

add this vocabulary.txt into distribute files during execution.

- Distributed Cache.



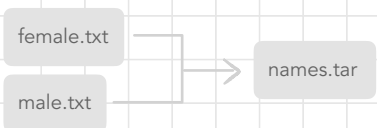
If you provide -files, each of this file will be copied once by each node before any task execution.

- ways to distribute files.

1. -files
2. -archives : better utilize network profile transmission.
   all archives will be unpacked on worker nodes.
3. -libjars : JAR stands for Java Archive.

- example for -archives : count the most popular male and female name.

(1) create tar file



(2) execute MapReduce application with -archives

```
yarn jar $HADOOP_STREAMING_JAR \
          -files mapper.py,reducer.py,vocabulary.txt \
          -archives names.tar \
          -mapper 'python mapper.py' \
          -reducer 'python reducer.py' \
          -input /data/wiki/en_articles \
          -output word_count
```

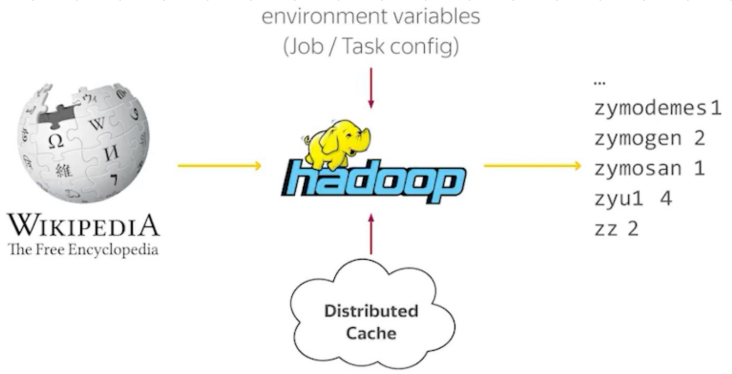
(3) mapper.py

```
from __future__ import print_function
import re
import sys

def read_vocabulary(file_path):
    return set(word.strip() for word in open(file_path))

male_names = read_vocabulary("names.tar/male.txt")
female_names = read_vocabulary("names.tar/female.txt")

for line in sys.stdin:
    article_id, content = line.split("\t", 1)
    words = re.split("\W+", content)
    for word in words:
        if word in male_names or word in female_names:
            print(word, 1, sep="\t")
```

tar file will be unpacked to the folder with the same name

- Enviroment, Counters.

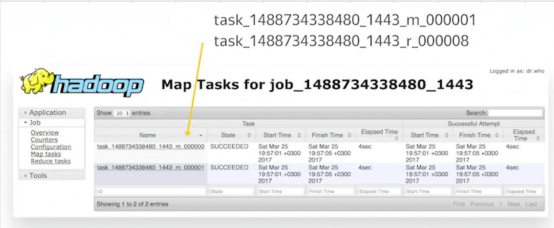
Distributed cache is not a only way to pass information to script, you can also get job configuration options through enviroment variables.

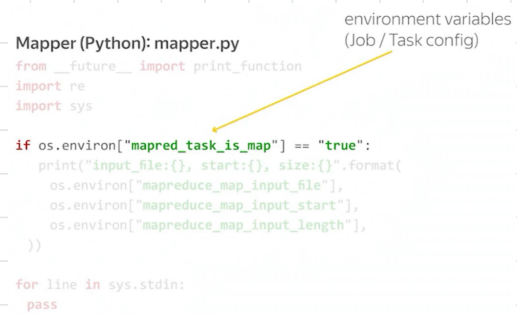


When you launch MapReduce application, framework will assign of data to available workers, you can access these data from your script.

os.environ["mapreduce_task_id"]

Get an absolute task id, is usually available on job tracker UI.





os.environ["mapreduce_task_prtition"]

Get the relative order of the task within map or reduce phase.
task_1488734338480_1443_m_000001 → 1
task_1488734338480_1443_r_000008 → 8

```
yarn jar $HADOOP_STREAMING_JAR -D word_pattern="\w+\d+" \
          -files mapper.py \
          -mapper 'python mapper.py' \
          -reducer 'python reducer.py' \
          -input /data/wiki/en_articles \
          -output word_count
```

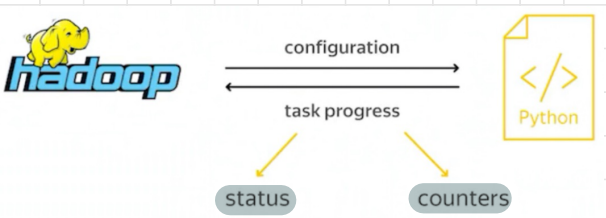
use -D to provide arbitrary environment variables.

Mapper (Python): wc_mapper.py

```
from __future__ import print_function
import os
import re
import sys

pattern = re.compile(os.environ["word_pattern"])

for line in sys.stdin:
  article_id, content = line.split("\t", 1)
    words = re.findall(pattern, content)
    for word in words:
        print(word, 1, sep="\t")
```

use os.environ() to get the environment variables which specified in executing yarn app.

There is also a backward communication channel between framework and your script. For instance, you can provide information about task progress: status and counters.



- status

Mapper (Python): reporter_mapper.py

```
from __future__ import print_function
import re
import sys

for line in sys.stdin:
    article_id, content = line.split("\t", 1)
    words = re.findall("\w+", content)
    for index, word in enumerate(words):
        print(word, 1, sep="\t")
        print("reporter:status:processed {} words"
            .format(index + 1), file=sys.stderr)
```

you can provide an arbitrary message in status for each task execution.

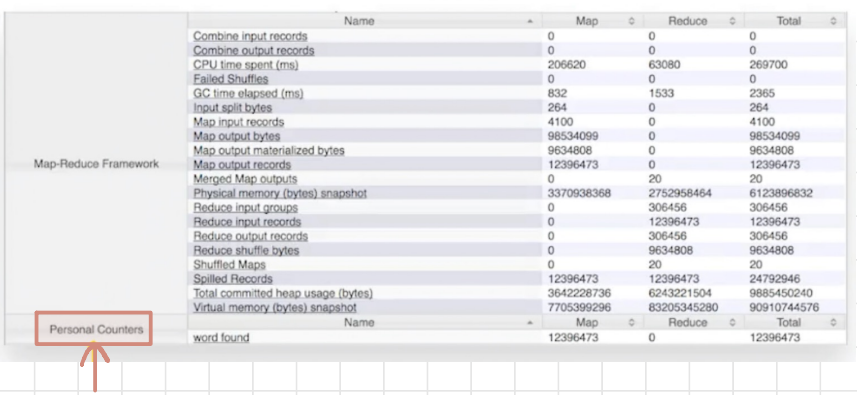
- counter

Mapper (Python): reporter_mapper.py

```
from __future__ import print_function
import re
import sys

for line in sys.stdin:
    article_id, content = line.split("\t", 1)
    words = re.findall("\w+", content)
    for index, word in enumerate(words):
            print(word, 1, sep="\t")
            print("reporter:status:processed {} words"
                .format(index + 1), file=sys.stderr)
            print(" reporter:counter:Personal Counters,word found,1",
                file=sys.stderr)
                  file=sys.stderr)
)
```

reporter:counter:<group>,<counter>,<amount>

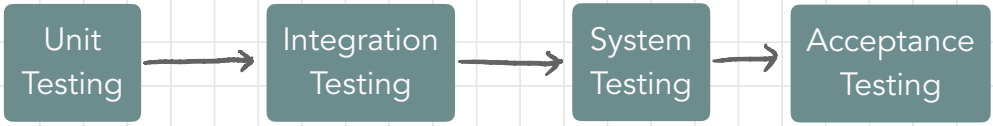
- counter family name
- counter namr
- value to add to the counter



	Name	Map	Reduce	Total
Map-Reduce Framework	Combine input records	0	0	0
	Combine output records	0	0	0
	CPU time spent (ms)	206620	63080	269700
	Failed Shuffles	0	0	0
	GC time elapsed (ms)	832	1533	2365
	Input split bytes	264	0	264
	Map input records	4100	0	4100
	Map output bytes	98534099	0	98534099
	Map output materialized bytes	9634808	0	9634808
	Map output records	12396473	0	12396473
	Merged Map outputs	0	20	20
	Physical memory (bytes) snapshot	3370938368	2752958464	6123896832
	Reduce input groups	0	306456	306456
	Reduce input records	0	12396473	12396473
	Reduce output records	0	306456	306456
	Reduce shuffle bytes	0	9634808	9634808
	Shuffled Maps	0	20	20
	Spilled Records	12396473	12396473	24792946
	Total committed heap usage (bytes)	3642228736	6243221504	9885450240
	Virtual memory (bytes) snapshot	7705399296	83205345280	90910744576
Personal Counters	Name	Map	Reduce	Total
	word found	12396473	0	12396473

this information is available on JobTracker UI,

- Testing.



- Unit Testing
  - test function edge cases.
  - python library: pytest

```python
def get_words(input_line):
    return input_line.split()

def test_get_words_parse_simple_string():
    assert get_words("a b cd efg ") == ["a", "b", "cd", "efg"]

def test_get_words_parse_empty_string():
    assert get_words("") == []

def test_get_words_raise_exception_if_no_input():
    with pytest.raises(AttributeError):
        get_words(None)
```

prefix test function with test

```
(venv) → coursera pytest -v test_function.py
============================= test session starts ==============================
platform darwin -- Python 2.7.12, pytest-3.0.7, py-1.4.33, pluggy-0.4.0 -- /Users/aadral/workspace/personal/mipt/coursera/venv/bin/python2.7
cachedir: .cache
rootdir: /Users/aadral/workspace/personal/mipt/coursera, inifile:
collected 3 items

test_function.py::test_get_words_parse_simple_string PASSED
test_function.py::test_get_words_parse_empty_string PASSED
test_function.py::test_get_words_raise_exception_if_no_input PASSED

=========================== 3 passed in 0.01 seconds ===========================
```

Pytest will scan and execute all the functions with this prefix.

- Integration Testing.
  - Validate how mappers and reducers scripts are integrated with Hadoop MapReduce streaming API.
  - will be working out of the box if your scripts rely on MapReduce job configuration options.
    ⇒ Hadoop MapReduce framework provides an empty config for System Testing.

- **System Testing**.
  - execute whole pipeline end to end.

```
hdfs --config $HADOOP_EMPTY_CONFIG dfs -rm -r word_count
yarn --config $HADOOP_EMPTY_CONFIG jar $HADOOP_STREAMING_JAR
```

If you provide a Hadoop empty config, you execute the MapReduce application in a standalone mode.

```
$ locate "hadoop/conf.empty"
```

find a path to an empty config.

  - **standalone mode**.

    HDFS client point out to a local file system, and a node manager is **working on the same node**. Streaming scripts will be able to communicate with MapReduce framework via environment variables. You will be able to read configuration of variables and validate correctness.

- **Acceptance Testing**.
  - **final stage** before shipping code into production system.
  - validation against a **sample dataset** without wasting your time and CPU cycles.
  - validation against big datasets and **measuring performance or efficiency**.

Hadoop MapReduce Application Tuning: Job Configuration, Comparator, Combiner, Partitioner.

Efficient MapReduce is based on three:
- Combiner
- Partitioner
- Comparator

- **Combiner**

in word count example,

input: word word a word b c d word d e...

Mapper (Python): reporter_mapper.py

```
from __future__ import print_function
import sys

for line in sys.stdin:
    article_id, content = line.split("\t", 1)
    words = content.split()
    for word in words:
            print(word, 1, sep="\t")
```

output: (word, 1), (word, 1), (a, 1), ... → a lot of repeated items.

would better squash the repeated items, for example,

(word,1), (word,1), (a,1) → (word,2), (a,1)

This approach can help you to dramatically change the usage of these IO operations and network bandwidth.

implement combiner in mapper:

input: word word a word b c d word d e...

Mapper (Python): reporter_mapper.py

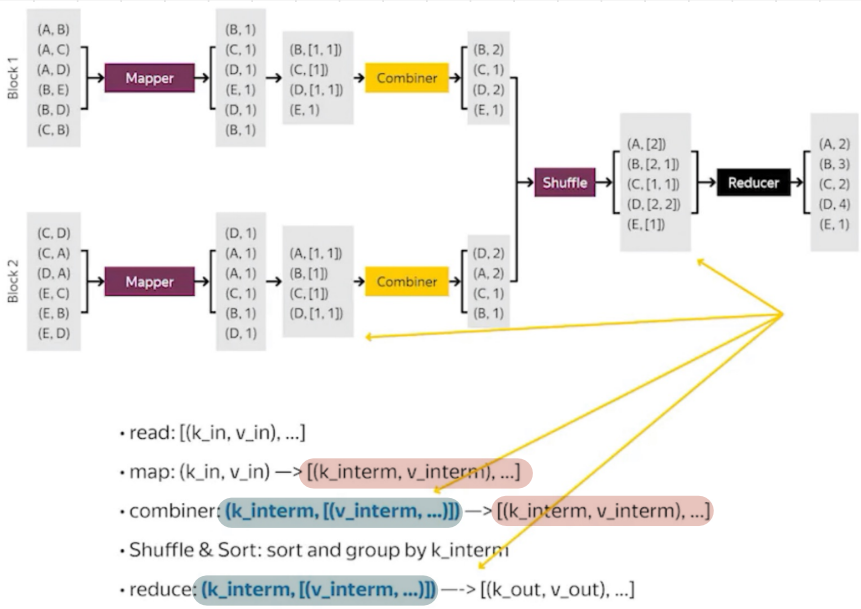
```
from __future__ import print_function
from collections import Counter
import sys

for line in sys.stdin:
    article_id, content = line.split("\t", 1)
    words = content.split()
    counts = Counter(words)
    for word, word_count in counts.items():
        print(word, word_count, sep="\t")
```

output: (a, 1), (b, 1), (c, 1), (d, 2), (e, 1), (word, 4), ...

improvement in calculations:

	without Combiner	with Combiner
Wall time (sec)	935	528
CPU time (sec)	9790	6584
Local FS Read (MB)	3006	1324
Local FS Write (MB)	4527	1963
Peek Map phys. memory (MB)	526	606
Peek Map virt. memory (MB)	2131	2144
Peek Reduce phys. memory (MB)	2744	631
Peek Reduce virt. memory (MB)	3196	3194

Combiner interface:



1. input in the form of reducer input.
2. output in the form of mapper output.

⇒ so combiner can be applied arbitrarily number of times between map and reduce phase.

```
yarn jar $HADOOP_STREAMING_JAR \
            -files mapper.py,reducer.py \
            -mapper 'python mapper.py' \
            -reducer 'python reducer.py' \
            -combiner 'python reducer.py' \
            -numReduceTasks 2 \
            -input griboedov.txt \
            -output word_count
```

Call combiner with argument -combiner.
In word count example, there is no difference between the combiner and the reducer.

```
Map-Reduce Framework
    Map input records=2681
    Map output records=182
    Map output bytes=1218
    Map output materialized bytes=955
    Input split bytes=126
    Combine input records=182
    Combine output records=99
    Reduce input groups=99
    Reduce shuffle bytes=955
    Reduce input records=99
    Reduce output records=99
```

How many records were processed by the combiner.

Sometimes you need to write your own combiner with different signature.

Example

Task: Count how many times on average you see a word in an article?

In Mapper:

input: word word a word b c d word d e...

Mapper (Python): mapper.py

```python
from __future__ import print_function
from collections import Counter
import sys

for line in sys.stdin:
    article_id, content = line.split("\t", 1)
    words = content.split()
    counts = Counter(words)
    for word, word_count in counts.items():
        print(word, word_count, sep="\t")
        print(word, 1, word_count, sep="\t")
```

~~**output**: (a, 1), (d, 2), (word, 4), ...~~
output: (a, (1, 1)), (d, (1, 2)), (word, (1, 4)), ...

a pair containing the number of articles processed and the cumulative amount of words.

In Reducer:

Mapper (Python): reducer.py

```python
from __future__ import print_function
import sys

current_word = None
word_count, article_count = 0, 0

for line in sys.stdin:
    word, articles, counts = line.split("\t", 2)
    articles, counts = int(articles), int(counts)
    if word == current_word:
        word_count += counts
        article_count += articles
    else:
        if current_word:
            print(current_word, word_count / article_count, sep="\t")
            current_word = word
        word_count = counts
        article_count = articles

if current_word:
    print(current_word, article_count  / word_count, sep="\t")
```

In Combiner:

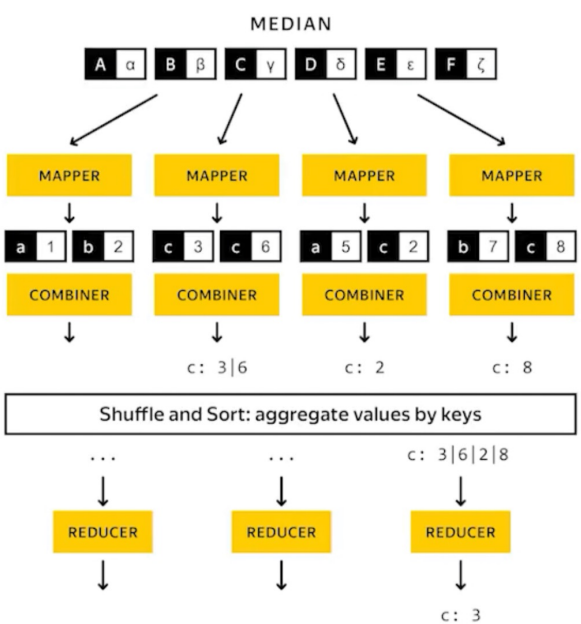
Mapper (Python): combiner.py

```
from __future__ import print_function
import sys

current_word = None
word_count, article_count = 0, 0

for line in sys.stdin:
    word, articles, counts = line.split("\t", 2)
    articles, counts = int(articles), int(counts)
    if word == current_word:
        word_count += counts
        article_count += articles
    else:
        if current_word:
            assert len(current_word.rstrip()) > 0
            print(current_word, word_count, article_count, sep="\t")
        current_word = word
        word_count = counts
        article_count = articles

if current_word:
    print(current_word, word_count, article_count, sep="\t")
```

does some spellers for each code in the pair, it could help us to speed up calculations, use less IO resources.

Example

Task: Count how many times on median you see a word in an article?



- Partitioner

Example: Word Count - Bigram

input

```
$ head -c 100 wikipedia_sample.txt
12 Anarchism Anarchism is often defined as a political
philosophy which holds the state to ...
```

Mapper (Python): bigram_mapper.py

```
from __future__ import print_function
import re
import sys
for line in sys.stdin:
 article_id, content = line.split("\t", 1)
 words = re.split("\W+", content)
 for index in range(len(words) - 1):
  print(words[index], words[index + 1], 1, sep="\t")
```

output

```
$ head word_count/part-00000
Anarchism Anarchism 1
Anarchism is 1
is often 1
```

Hadoop MapReduce framework will distribute and sort data by the first word

The data on the reducer will not be sorted by the second word. Of course, you can update reducer.py to count all bigrams for the first corresponding word in memory:

Mapper (Python): inmemroy_bigram_reducer.py

```
from __future__ import print_function
from collections import Counter
import sys

current_word = None
bigram_count = Counter()

for line in sys.stdin:
 first_word, second_word, counts = line.split("\t", 2)
  counts = Counter({second_word: int(counts)})
 if first_word == current_word:
   bigram_count += counts
 else:
  if current_word:
    for second_word, bigram_count in bigram_count.items():
      print(current_word, second_word,
```

But it will be memory consuming.

Output:

```
yarn --config $HADOOP_EMPTY_CONFIG jar $HADOOP_STREAMING_JAR \
        -files bigram_mapper.py,inmemory_bigram_reducer.py \
        -mapper 'python bigram_mapper.py' \
        -reducer 'python inmemory_bigram_reducer.py' \
        -numReduceTasks 5 \
        -input wikipedia_sample.txt \
        -output word_count \
```

```
$ grep $'^new\t' word_count/* | sort -nrk3,3 | head -4
word_count/part-00001:new    york   112
word_count/part-00001:new    world  15
word_count/part-00001:new    jewrsey  12
word_count/part-00001:new    constitution   12
$ grep $'\tyork\t' word_count/* | sort -nrk3,3 | head
word_count/part-00001:new    york   112
word_count/part-00004:sergeant york   1
```

In addition to the unnecessary memory consumption, there would be an uneven load on the reducers.
Some words which occur far more frequently than others, for instance, "the".
In a default scenario you will have the far more load on the reducer that will be busy processing this article "the".
But you have no need to send all of the bigrams starting with "the" to one reducer.
⇒ Partitioner comes into play.
You can specify the way to split as three mappers or reducer output to a key-value pair through CLI.

```
yarn --config $HADOOP_EMPTY_CONFIG jar $HADOOP_STREAMING_JAR \
        -D stream.num.map.output.key.fields=2 \
        -files bigram_mapper.py,bigram_reducer.py \
        -mapper 'python bigram_mapper.py' \
        -reducer 'python bigram_reducer.py' \
        -numReduceTasks 5 \
        -input wikipedia_sample.txt \
        -output word_count \
```

split the line into key-value pairs by the second tab character.

⇒ will complete this MapReduce job faster.

```
$ grep $'^new\t' word_count/* | sort -nrk3,3 | head -4
word_count/part-00001:new york 112
word_count/part-00001:new world 15
word_count/part-00002:new constitution 12
word_count/part-00000:new jersey 12
```

bigrams starting with arbitrary allocated in different files

Other useful flags:

input

```
$ cat subnet.txt
1.2.3.4
2.3.4.5
3.4.5.6
4.5.6.7
```

```
yarn --config $HADOOP_EMPTY_CONFIG jar $HADOOP_STREAMING_JAR \
        -D stream.map.output.field.separator=. \
        -D stream.num.map.output.key.fields=1 \
        -D stream.reduce.output.field.separator=. \
        -D stream.num.reduce.output.key.fields=2 \
        -files identity_mr.py \
        -mapper 'python identity_mr.py' \
        -reducer 'python identity_mr.py' \
        -numReduceTasks 2 \
        -input subnet.txt \
        -output subnet_out \
```

specify what a delimiter is

specify number of fields related to a key

output

```
cat subnet_out/*
2    3.4    5
4    5.6    7
1    2.3    4
3    4.5    6
```

input

```
$ cat subnet.txt
1a.2.3.4
2a.3.4.5
3b.4.5.6
4b.5.6.7
```

```
yarn --config $HADOOP_EMPTY_CONFIG jar $HADOOP_STREAMING_JAR \
        -D map.output.key.field.separator=. \
        -D mapreduce.partition.keypartitioner.options=-k1.2,1.2 \
        -files identity_mr.py \
        -mapper 'python identity_mr.py' \
        -reducer 'python identity_mr.py' \
        -numReduceTasks 2 \
        -input subnet.txt \
        -output subnet_out \
        -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
```

specify the field index and the starting character index in the start position

specify the field index and the character index in the end position

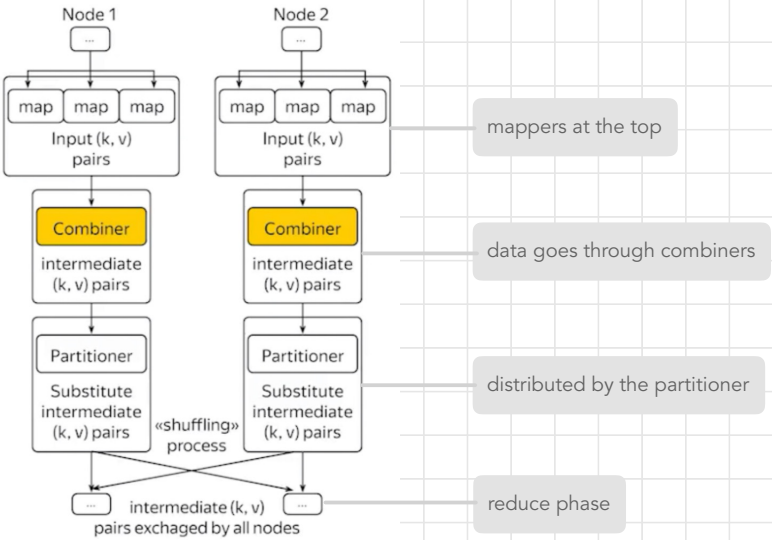
set a special partitioner called KeyFieldBasedPartiioner.
It is a Java class located in `$HADOOP_STREAMING_JAR`

output

```
cat subnet_out/*
3b.4.5.6
4b.5.6.7
1a.2.3.4
2a.3.4.5
```

```
ls -lth subnet_out
total 8.0K
0 Apr 1 14:59 _SUCCESS
20 Apr 1 14:59 part-00001
20 Apr 1 14:59 part-00000
```

Whole Pipeline of MapReduce Application Execution:



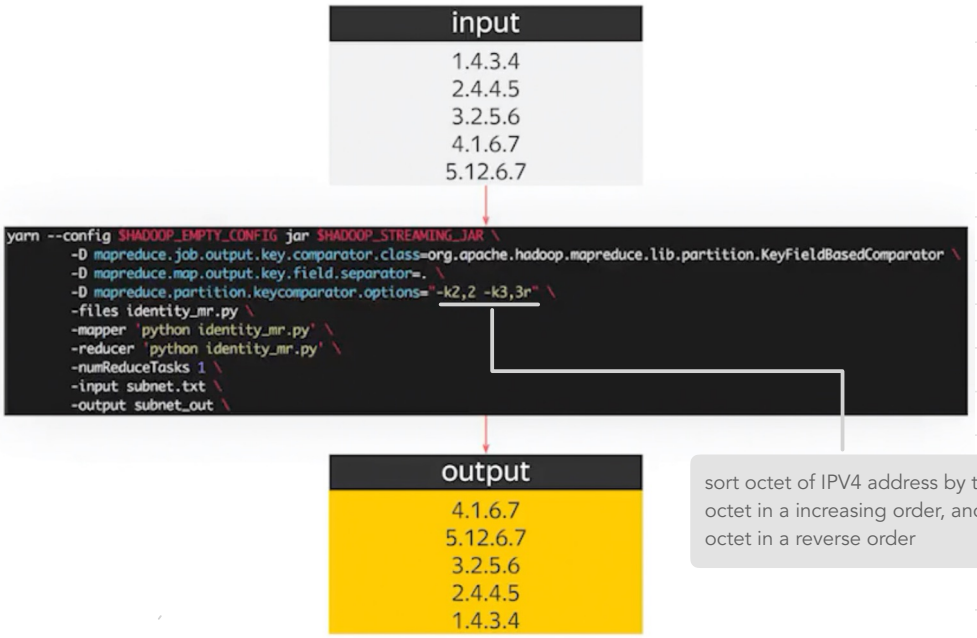
- **Comparator**

All the keys in MapReduce implement writable comparable interface.

Comparable means that you can specify the rule according to one key is bigger than another.

By default, you have the keys sorted by increasing order.

Example:



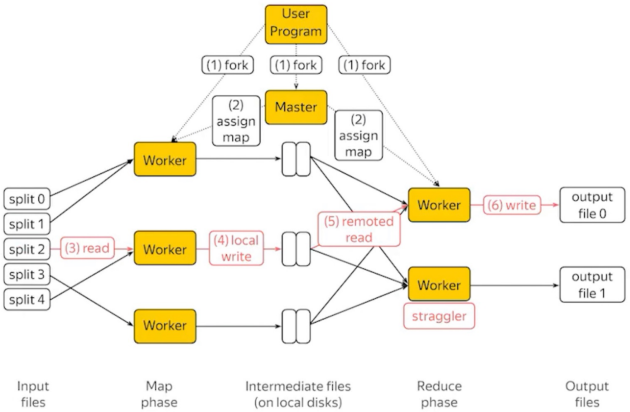
- **Speculative Execution / Backup Tasks**

One of the most common problems that causes a MapReduce application to wait longer for a job completion is a straggler.

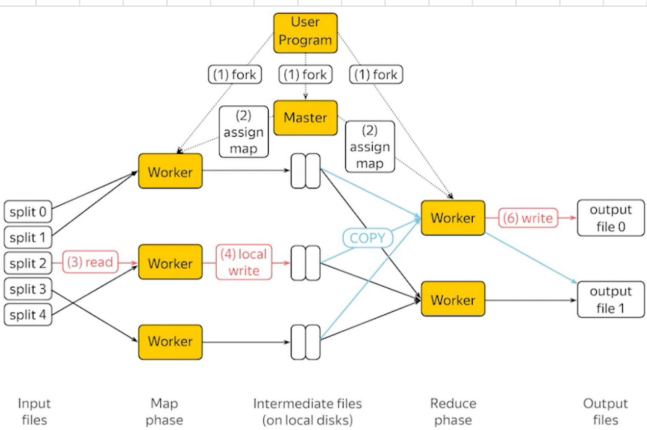
straggler problem: machine that takes an unusually long time to complete one of the last few tasks in the computation.

(落伍者)

It can be result of a number of different issues, e.g. hard drive, OS configs, swap space uses network connections or CPU overutilization.



The solution for this problem was provided by the authors of MapReduce in the original article, called Backup Tasks. Due to the deterministic behavior of the Mapper and Reducer, you can easily re-execute straggler body of work on other node.



In this case, the worker which process data the first outputs data to a distributed file system, all the other concurrent executions will be killed.

MapReduce framework is not going to have a copy for each

running task, it is only used when a MapReducer application is close to completion.

MapReduce sort application was faster by 30~40%

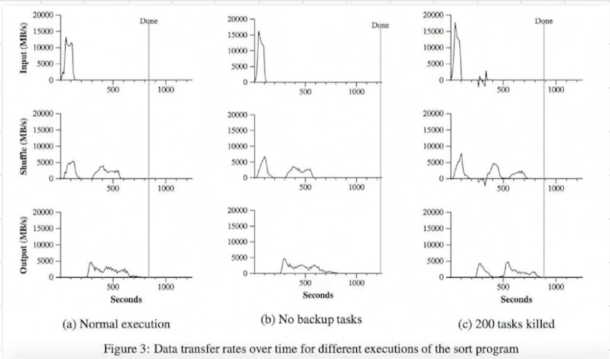


Figure 3: Data transfer rates over time for different executions of the sort program

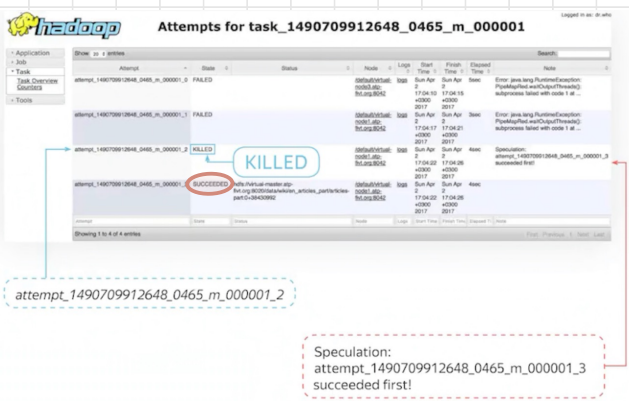
"MapReduce: Simplified Data Processing on Large Clusters"
by Jeffrey Dean and Sanjay Ghemawat

```
yarn jar $MR_STREAMING_JAR \
        -D mapreduce.map.speculative="true" \
        -D mapreduce.job.speculative.speculative-cap-running-tasks=0.99 \
        -D mapreduce.job.speculative.speculative-cap-total-tasks=0.99 \
        -D mapreduce.job.speculative.retry-after-no-speculate=10 \
        -files flaky_mapper.py \
        -mapper 'python flaky_mapper.py' \
        -output flaky_mr \
        -numReduceTasks 0 \
        -input /data/wiki/en_articles_part
```

set speculative execution (default true)

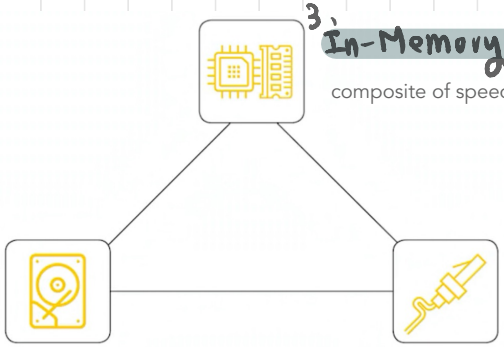
specify the allowed percentage of running backup tasks at each point in the stream of the time and overall

limit the time of your waiting till the next round of speculation



attemp_1490709912648_0465_m_000001_3 first complete the task,
so the concurrent tasks attemp_1490709912648_0465_m_000001_2 is killed

- **Compression**



3. **In-Memory Calculation Capacity**
   composite of speed and usage of CPU and RAM.

1. **Data Compression**
   a trade-off between the disk I/O required to read and write data.

2. **Network Bandwidth**
   send data across the network.

The correct balance of these factors depends on your

(1) cluster

(2) data

(3) applications

(4) usage patterns

Data located in HDFS can be compressed, what's more, there is a shuffle and sort phase between map and reduce where you can compress the intermediate data.

Compression Options:

Compression Format	Splittable	Comments
.deflate .gz (gzip)	NO	Uses DEFLATE algorithm
.bz2 (bzip)	YES	more effective than gzip, but slower compression
.lzo	YES*	decompres way faster than gzip, compres less efficient
.snappy	NO	faster than LZO for decompression

more aggressive for space requirements, but slower during compression.

suitable for the scenario where you read data far more frequently than write.

flags: -1(optimised for speed)...-9(optimised for space)

built-in Hadoop Compression Codec:

Comptession format	Hadoop CompressionCodec
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
LZ4	org.apache.hadoop.io.compress.Lz4Codec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

You can tune the compression codec through CLI arguments

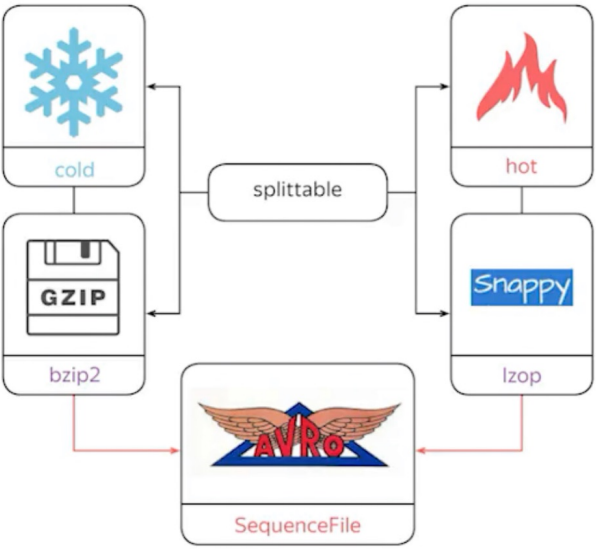
mapper
```
-D mapreduce.compress.map.output=true
-D mapreduce.map.output.compression.codec=...
```

reducer
```
-D mapreduce.output.compress=true
-D mapreduce.output.compression.codec=true
```

* Running test is essential to see what options are the most suitable for your data processing patterns.

Rule of thumb:

gzip or bzip are a good choice for cold data which is accessed infrequently.



snappy or lzo are a better choice for hot data, which is accessed frequently.

snappy and gzip are not splittable at file level compression.
But you can use block level compression and splittable container formats such as Avro or SequenceFile