MAP REDUCE

Instead of bringing data to compute, why not take compute to data

↳ Paper by google (Jeff Dean & Sanjay Ghemawat)

↳ Usecase at google: computation on huge amount of data
eg: creating index of all web content
: page ranking

↳ Allows running giant computations on 1000's of computers
↳ It is a framework / paradigm that allows engineers to write the guts of the application w/o worrying about the distributed systems

↗ of GFS & MR architecture

- Hadoop is the adoption of open-source implementation by Yahoo (now Apache project)
- MapReduce is the central processing component of the Hadoop framework. It provides a programming model for processing vast amounts of data in a distributed environment, offering several key advantages:
  - **Parallel Processing:** MapReduce enables the parallel processing of data across multiple servers in a distributed computational environment, which significantly boosts performance.
  - **Data Locality:** This is a fundamental principle of MapReduce. Instead of moving large amounts of data to the processing unit, the framework moves the processing logic to where the data is already stored. This reduces expensive network traffic and improves efficiency.
- Its an execution framework for large-scale data processing
  - Which distributes the computing across distributed commodity servers and then pulls the results together
  - Programmer writes code as if writing for a single machine but the framework executes the process in distributed manner
  - Distributed implementation that hides all the messy details
    - Fault tolerance (thus provides High Availibility)
    - I/O scheduling
    - parallelization

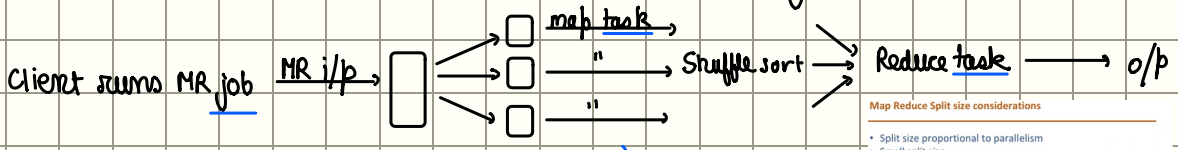
- Uses Divide and conquer – Partitions large problem into smaller subproblems
- Workers work on sub-problems in parallel (could be threads in a core or cores in multi-core processor, multiple processor in a machine, machines in a cluster) and produce intermediate results
- Intermediate results from workers are combined to form the final result

Terminology: MapReduce job - unit of work that client wants performed.

↓

divided into tasks - map tasks
- reduce tasks

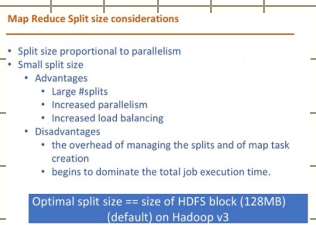
↳ scheduled using YARN ↗ automatically schedules failed tasks on another node



i/p is split

the optimal split size = chunk size in HDFS

map task is run on each record in each split



Map reduce wordcount example

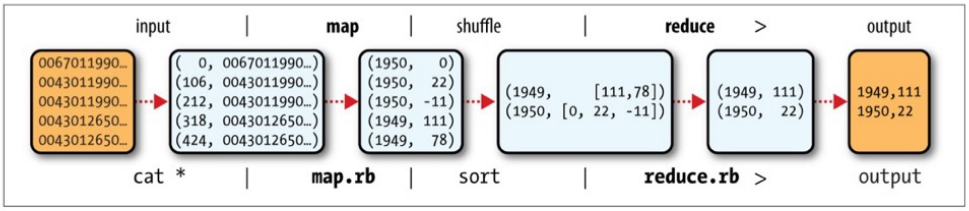


Figure 2-1. MapReduce logical data flow

file : hello world people
love hello
people hello

the i/p file is small, it will spawn just one mapper, but lets say it spawns one mapper per line

i/p to mapper : <key, value>
↳ line
↳ line num
} line num & line are just an example, it could be diff depending on our need

↓ mapper task enjoys data locality

: 0, hello world people → mapper 1
1, love hello → mapper 2
2, people hello hello → mapper 3

o/p of mappers : mapper 1 : <hello, 1>
<world, 1>
<people, 1>

↓ stored to local disk : it can be thrown away after Reducer task so storing it in HDFS w replication is overkill

2 : <love, 1>
<hello, 1>

3 : <people, 1>
<hello, 1>

all these outputs go to shuffle sort

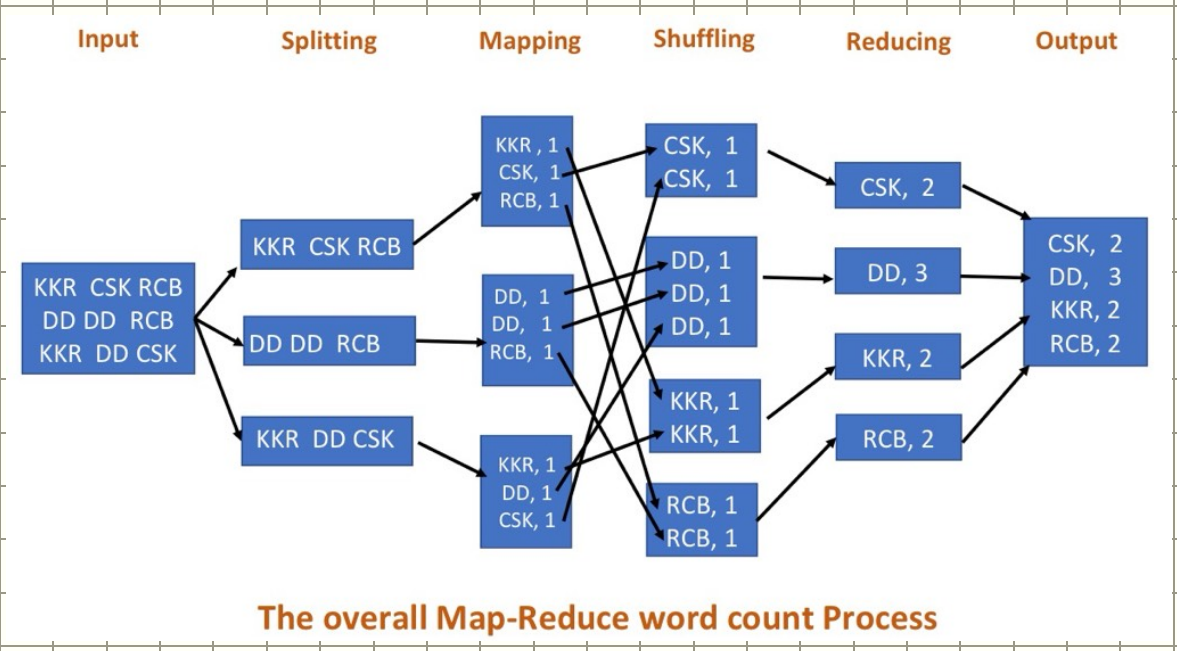
i/p of shuffle sort :
<hello, 1>
<world, 1>
<people, 1>
<love, 1>
<hello, 1>
<people, 1>
<hello, 1>

o/p of shuffle sort : hello → [1,1,1,1]
world → [1]
people → [1,1]
love → [1]

o/p of SS is i/p to reducer : reducer doesn't have advantage of data locality. sorted mapper outputs have to be transferred across network to the node where reduce task is running

o/p of reducer : hello 4
world 1
people 2
love 1

(Stored in HDFS : one replica on local node, other on off rack nodes.



The overall Map-Reduce word count Process

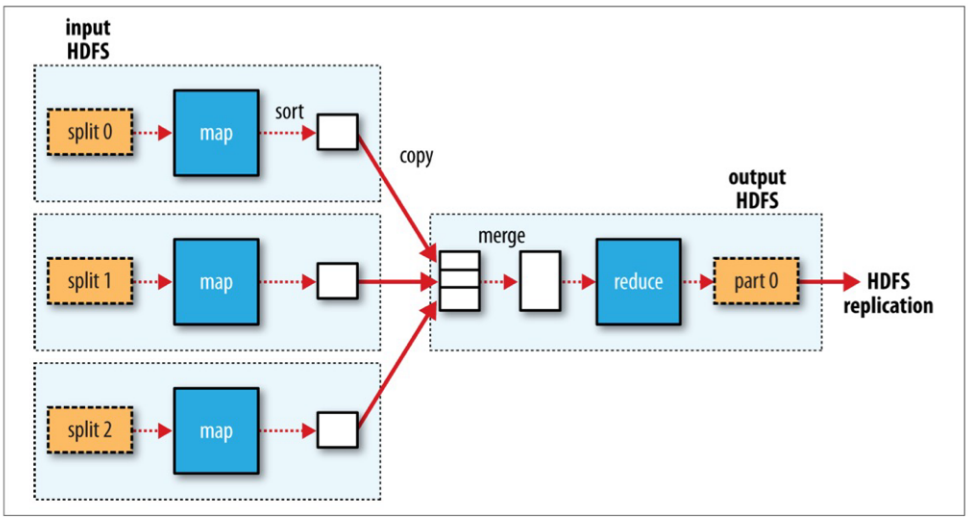


Figure 2-3. MapReduce data flow with a single reduce task

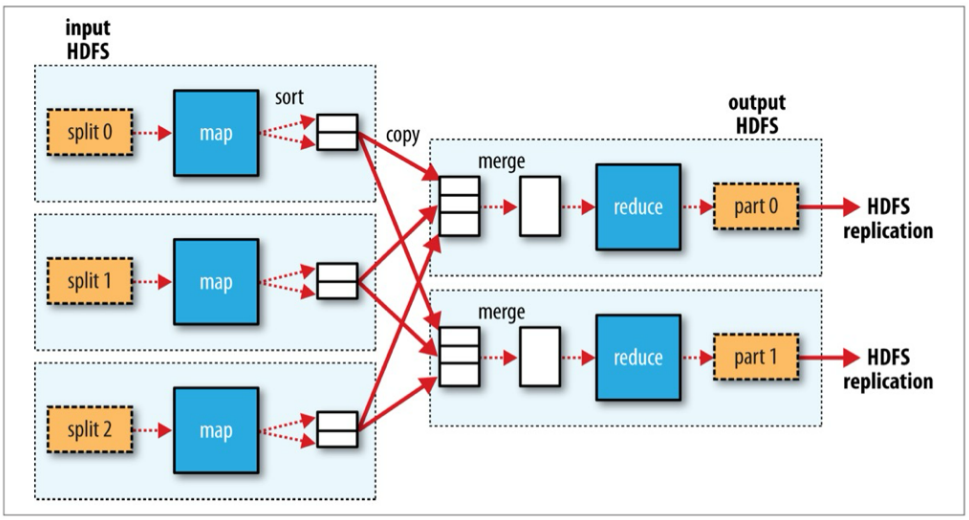


Figure 2-4. MapReduce data flow with multiple reduce tasks

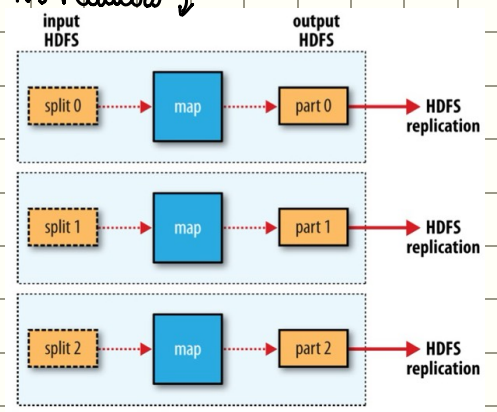
# of reduce tasks is not governed by size of i/p, it is specified independently

When there are multiple reducers, the map tasks *partition* their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.

Finally, it's also possible to have zero reduce tasks. This can be appropriate when you don't need the shuffle because the processing can be carried out entirely in parallel

whenever the problem does not require grouping, aggregating, or combining across multiple records, a reducer is unnecessary.

No reducers



Eg : Log file filtering :

- **Task Goal**: Extract only the error messages (lines containing "ERROR") from large log files.
- **Why reducer is not needed**: Each log line can be independently checked by the mappers. We don't need to aggregate or combine results — just output matching lines.

Data Flow

1. **Input**: A set of log files distributed across HDFS. Example:

Copy

```
2025-09-14 INFO  Server started
2025-09-14 ERROR Connection failed
2025-09-14 INFO  Request received
2025-09-14 ERROR Disk full
```

2. **Mapper Function**:
   - Reads each line.
   - Checks if the line contains "ERROR".
   - If yes, outputs the line.
   - If not, outputs nothing.

Example pseudocode:

python Copy

```
def map(key, value):
    # key: byte offset of line
    # value: log line
    if "ERROR" in value:
        emit(None, value)
```

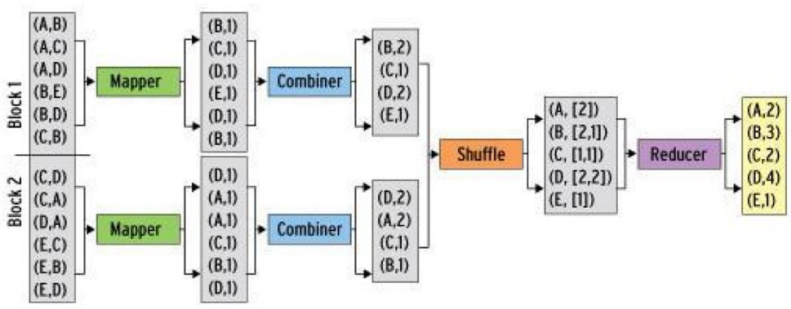
3. **Reducer**:
   - **Not needed**.
   - The framework can directly write mapper output to HDFS.
4. **Output**:

Copy

```
2025-09-14 ERROR Connection failed
2025-09-14 ERROR Disk full
```

Combiner function :

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a *combiner function* to be run on the map output, and the combiner function's output forms the input to the reduce function. Because the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all. In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer.



- Combine multiple map outputs before doing a reduce
- Can write a combiner function in program
  - Combiner will be run before reduce
- Mini-reducer

Example : for wordcount example, there was just one occurance of each word in a line

If the i/p was : file : hello world people people people
love hello love love
people hello hello

w/o combiner : all this would have to be transferred to reducer

o/p of mappers : mapper 1 : <hello, 1>
<world, 1>
<people, 1>
<people, 1>
<people, 1>
2 : <love, 1>
<hello, 1>
<love, 1>
<love, 1>
3 : <people, 1>
<hello, 1>
<hello, 1>

w combiner

o/p of mappers : mapper 1 : <hello, 1>
<world, 1>
<people, 3>
2 : <love, 3>
<hello, 1>
3 : <people, 1>
<hello, 2>

} transferring this uses less bandwidth

Example :

This is best illustrated with an example. Suppose that for the maximum temperature example, readings for the year 1950 were processed by two maps (because they were in different splits). Imagine the first map produced the output:

```
(1950, 0)
(1950, 20)
(1950, 10)
```

and the second produced:

```
(1950, 25)
(1950, 15)
```

The reduce function would be called with a list of all the values:

```
(1950, [0, 20, 10, 25, 15])
```

with output:

```
(1950, 25)
```

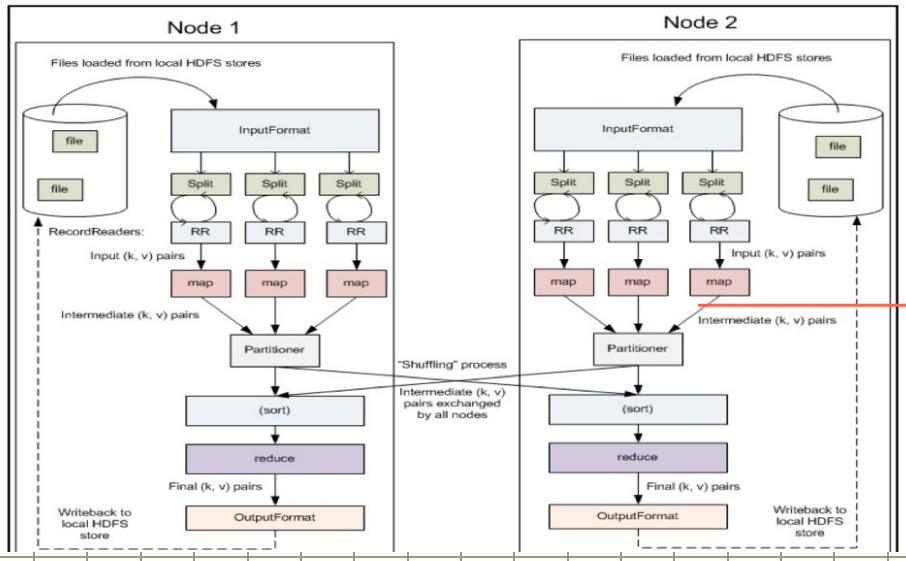
since 25 is the maximum value in the list. We could use a combiner function that, just like the reduce function, finds the maximum temperature for each map output. The reduce function would then be called with:

```
(1950, [20, 25])
```

Combiner func is usually the same as reduce function

the combiner function is defined using the Reducer class, and for this application, it is the same implementation as the reduce function in MaxTemperatureReducer. The only change we need to make is to set the combiner class on the Job

High level view



combiner comes after map & before partitioner

Hadoop Streaming

Hadoop provides an API to MapReduce that allows you to write your map and reduce functions in languages other than Java. *Hadoop Streaming* uses Unix standard streams as the interface between Hadoop and your program, so you can use any language that can read standard input and write to standard output to write your MapReduce program.[3]

Streaming is naturally suited for text processing. Map input data is passed over standard input to your map function, which processes it line by line and writes lines to standard output. A map output key-value pair is written as a single tab-delimited line. Input to the reduce function is in the same format—a tab-separated key-value pair—passed over standard input. The reduce function reads lines from standard input, which the framework guarantees are sorted by key, and writes its results to standard output.