Divide:

Sharding: split documents into different servers.

Conquer: Each server process a "soln"

Combine: merge all "solns"

Issues

* fit in main memory
* How to split
* Server crash (re-do all work)

MapReduce

map function: like "counting words"

reduce function: combining local into a global

MapReduce invented by Google.

↳ Simplified programming framework for big data analytics, users must only specify the computing logic. (map(), reduce(), framework handles the rest)

Defn: MapReduce: is a data-parallel programming framework (or model) for clusters of commodity machines.

Goals: Scalability of large volumes, cost-efficiency.

Map() → Transform input data into some intermediate (local) results. (word Count)

Reduce → Aggregate or combine the local results into global results.

Parameters must be flexible and powerful

MapReduce: uses lists & key-value pairs as its main data primitives.

- $Map(k_1, v_1) \rightarrow List(k_2, v_2)$
- $Reduce(k_2, list(v_2)) \rightarrow (k_3, v_3)$

Input has to be a list of key-value pair.
↳ list of documents. (can be multiple pairs, for many docs)

< doc 1, "Hello World..." >

Input list of key-value pairs is partitioned into different servers and each key value, $\langle k_1, v_1 \rangle$ is processed by calling the map function → convert to list $\langle k_2, v_2 \rangle$

i.e → <Hello,1>, <word,1>, ...

All pairs that share the same key will be grouped together

<Hello,1>, <Hello,1> → <Hello, list(1,1)>
singles → <world, list(1)>
} <k2, list(v2)>

Reduce function converts each <k2, list(v2)> → <k3, v3>

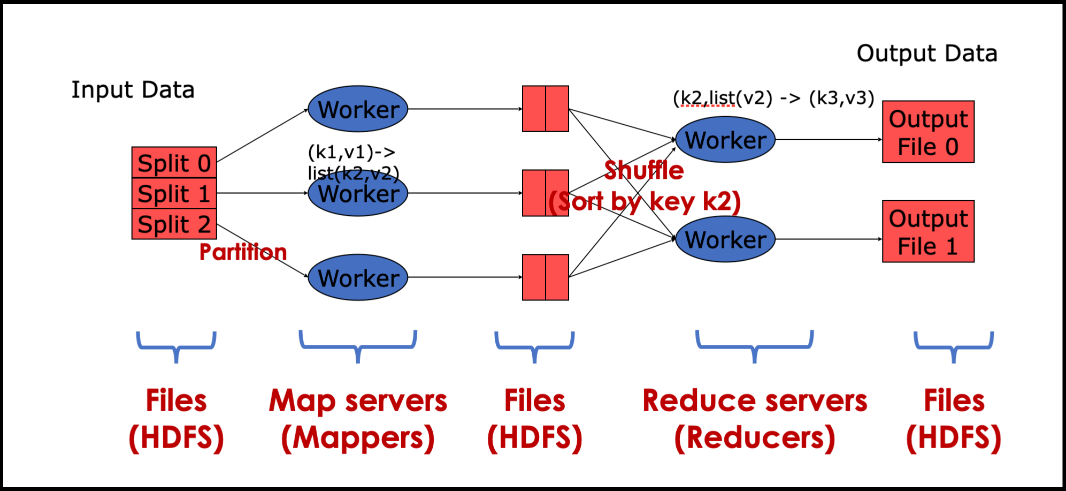
<Hello, list(1,1)> → <Hello, 2>
⋮ <world, 1>
} <k3, v3>

In general:

__Map:__ Extract something you care about from each record.

__Reduce:__ aggregate, summarize, filter or transform.

__Architecture__



Details

* Data represented in key value pairs
* Data is chunked (64 MB) based on an input split
* Multiple servers to run Map and Reduce
* Mappers read a chunk of data
* Mappers emit (write out) a set of data
* Intermediate data: (output of mappers) is sorted by key and split to a number of reducers.
* Reducers receive each key of data, along with All values associated with it. (each key must be sent to same reducer)
* Reducers emit a set of data. (reduced from its input, written to disk)

Worker Failure:

Detect failure via periodic heartbeats from master node.
Re-execute in-progress map/reduce tasks

Master failure

- single-point of failure; Resume from Execution log.

If a node fails:
- query is restarted
- Bad for long jobs (10hrs ...)

Recovery in the face of partial failure is an important contribution of MapReduce.

slow-workers lengthen completion time.
(other jobs, bad disks ...)

Soln: spawn backup copies of tasks, whichever one finishes first "wins"

MR vs DB

Failure → restart required

High scalability

No schemas

No query language

flexible UDFs

Optimizations, but limited options

can lose machines and keep going

MR yields new data