CentraleSupelec 2018-2019 MSC DSBA / DATA SCIENCES

Big Data Algorithms, Techniques and Platforms

Distributed Computing with MapReduce and Hadoop

Hugues Talbot & Céline Hudelot, professors.

References

- Mining of Massive Datasets - Jure Leskovec, Anand Rajaraman, Jeff Ullman.
  `http://www.mmds.org/`
- Data-Intensive Text Processing with MapReduce - Jimmy Lin and Chris Dyer
  `https://lintool.github.io/MapReduceAlgorithms/`
- MapReduce Design Patterns - Building Effective Algorithms and Analytics for Hadoop and Other Systems. Donald Miner and Adam Shook
- Hadoop : The Definitive Guide- Tom White

Processing Big Data

Solution : parallelism

- 1 server
  - 8 disks
  - Read the web : 230 days
- Cluster Hadoop Yahoo
  - 4000 servers with 8 disks each.
  - Read the web : 1h20

Processing Big Data

Some problems

- Synchronization.
- Programming models (shared memory, message passing (MPI)).
- Scalability and elasticity (arbitrary numbers of nodes).
- Fault Tolerance.

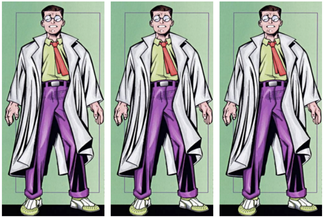
Tackling large data problems : big ideas

Scale out not up

A large number of **commodity low-end servers** (scaling out) as opposed to a small numbers of **high-end servers** (scaling up)

Scale up

Scale out



Tackling large data problems : big ideas

Assume failures as common

Assume

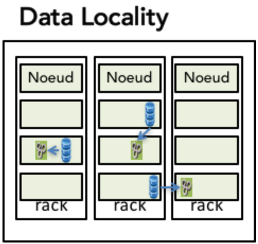
- A 10000-server cluster ;
- A mean time between failure of 1000 days.

10 failures for a day !

Tackling large data problems : big ideas

Move processing to the data

- Architectures where processors and storage are co-located (**data locality**)
- Distributed File Systems : GFS, HDFS...



Tackling large data problems : big ideas

Process data sequentially, avoid random access

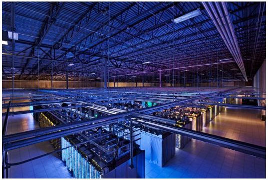
Relevant datasets are too large to fit in memory and must be held on the disk.

- Seeks are expensive, disk throughput is good.
- Organize computations so that the data is processed sequentially.

Tackling large data problems : big ideas

The data center is the computer

- Towards the right level of abstraction.
  Beyond the von Neumann architecture. What's the *instruction set* of the datacenter computer ?
- Hide system-levels details from the developers
  No need to explicitely worry about reliability, fault tolerance.
- Separating the *what* from the *how*.
  Developers specifies the computation that needs to be performed and an Execution framework handles actual execution.



Mapreduce : the first instanciation of this idea

Plan

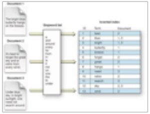
What is MapReduce ?

Two things

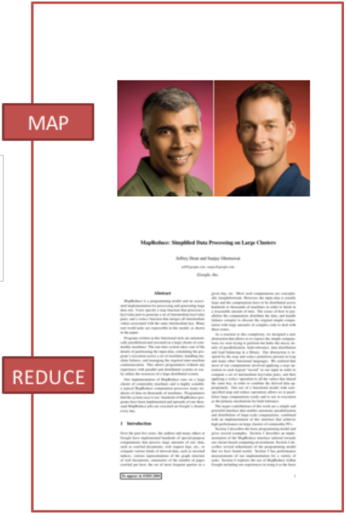
- A simple programming model for processing huge data sets in a distributed way.
- A framework that runs these programs on clusters of commodity servers, automatically handling the details of distributed computing :
  - Division of labor.
  - Distribution.
  - Synchronization.
  - Fault-tolerance.

Hadoop is one of the famous implementation of the MapReduce programming model and execution framework

MapReduce : origins



...



MapReduce : origins

MapReduce : Simplified Data Processing on Large Clusters [Dean and Ghemawat, OSDI 04]

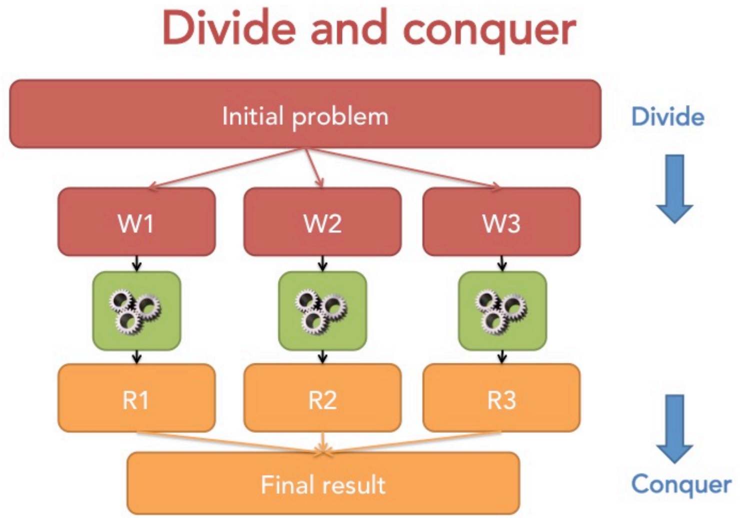
MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with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

MapReduce : origins

MapReduce : Simplified Data Processing on Large Clusters [Dean and Ghemawat, OSDI 04]

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed systems.

Divide and conquer paradigm



MapReduce : a distributed divide and conquer paradigm

Principle

- Divide a task into independent subtasks.
- Handle the sub-tasks in parallel.
- Aggregate the results of the subtasks to form the final output

Writing programs in MapReduce enables their automatic parallelization.

MapReduce : a distributed divide and conquer paradigm

A simple principle but :

- How do we break up the problem into smaller tasks that can be executed in parallel ?
- How do we assign tasks to workers ?
- How do we ensure that the workers get the data they need ?
- How do we coordinate synchronisation among the different workers ?
- How do we share partial results from one worker that is needed by another ?
- How do we accomplish all of the above in the face of software errors and hardware faults ?

Typical big-data problem

- Iterate over a large number of records.
- Extract something of interest from each record.
- Shuffle and sort intermediate results.
- Aggregate intermediate results.
- Generate final output.

The origins of MapReduce : this interesting observation made by Dean and Ghemawat. Many different problems can be processed in parallel in the same way.

Simple computations but complex issues from their parallelization.

Typical big-data problem

MAP

- Iterate over a large number of records.
- Extract something of interest from each record.

- Shuffle and sort intermediate results

REDUCE

- Aggregate intermediate results
- Generate final output.

MapReduce key idea :

- Every massive data processing can be expressed with these only two operations (outline stays the same, **map** and **reduce** change to fit the problem)
- Provide a functional abstraction of these two operations.

[Dean and Ghemawat, OSDI 2004]

MapReduce

Inspiration from functional programming

`map` and `reduce` (or `fold`) are existing list operators in functional programming (lisp, scheme, scala ...)

map

Apply a function f to each item of the list

```
map(f)[x0, ..., xn] = [f(x0), ...,f(xn)]
map(*4)[2, 3, 6] = [8, 12, 24]
```

reduce

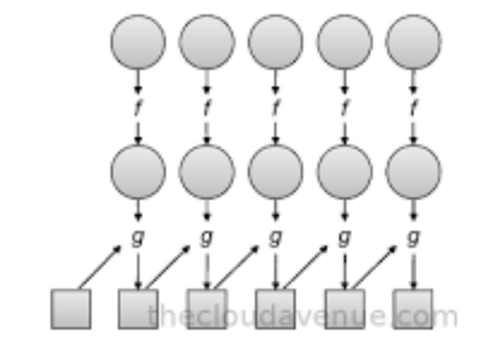
Apply a function recursively to the items of the list

```
reduce(f)[x0, ..., xn] = f(x0, f(x1, f(x2, ...)))
reduce(+)[2, 3, 6] = (2 + (3 + 6)) = 11
```

MapReduce

Inspiration from functional programming

Map and fold



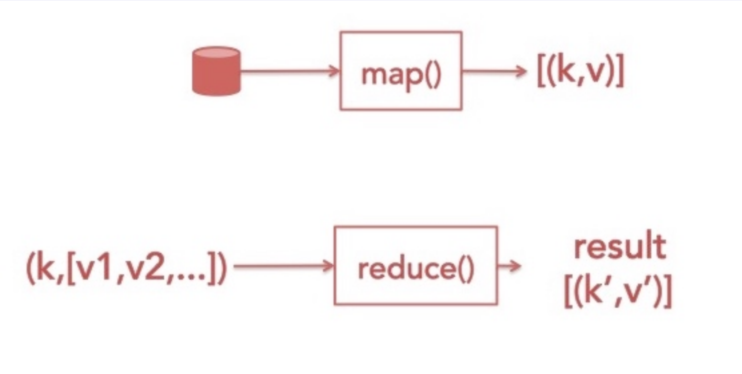
`map` : transformation of a dataset.

`reduce` : aggregation operation (some locality constraints).

MapReduce

Main principles

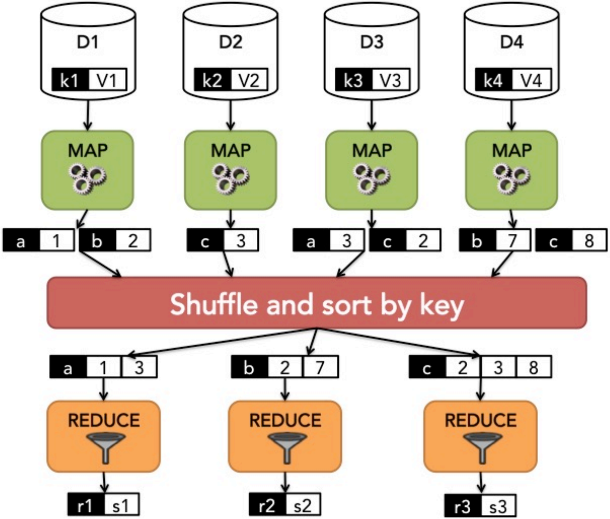
- All the data are structured in (key,value) pairs.
- Two functions :
  - MAP : Transform the input data in a list of (key,value) pairs.
  - REDUCE : Aggregate/Reduce all the values associated to the same key.



MapReduce overview

- Read a lot of data sequentially.
- MAP
  - Extract something you care about.
- SHUFFLE AND SORT
  - Group by key.
- REDUCE
  - Aggregate, summarize, filter or transform
- Write the result.

MapReduce scheme

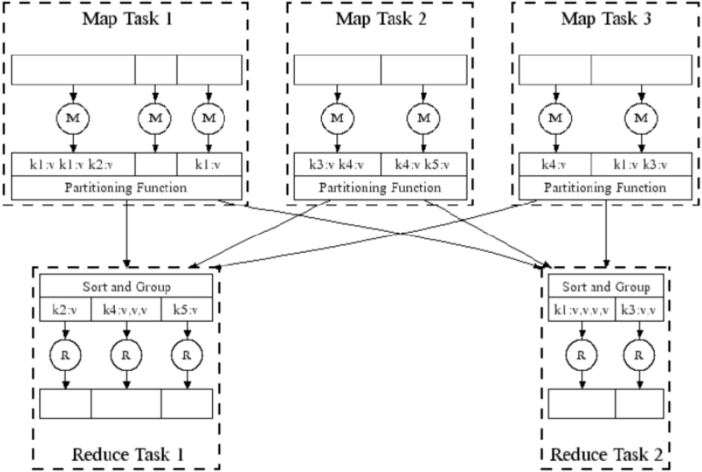


MapReduce programming model

Programmers must specify two functions :

- map $(k, v) \rightarrow [(k', v')...]$
  - Takes a key value pair and outputs a set of key value pairs (e.g. key = filename, value = the file content)
  - One Map call for every (k, v) pair.
- reduce $(k', [v'_1, ...v'_n]) \rightarrow [(k'', v'')...]$
  - All values v' with the same key k' are reduced together and processed in v' order.
  - One Reduce function call for each key k'
- The execution framework handles everything else

MapReduce : Execution in Parallel



Typical example

Task

- A huge text document.
- Count the number of times each distinct word appears in the file.

Sample applications

- Web server logs analysis to find popular URLs
- Information retrieval : document indexing
- ...

Just try (at home) !

- Write a program in python or java that achieves this task !
- Try it on files of different sizes !

Wordcount

Word Count !

Count the occurrences of each word in different documents.

Data : a set of textual documents

Le jour se lève sur notre grisaille, sur les trottoirs de nos ruelles et sur nos tours [...]	Le jour se lève sur notre envie de vous faire comprendre à tous que c'est à notre tour [...]

(**Grand Corps Malade,** Le Jour se lève. *Excerpt*)

Wordcount

Word Count !

SPLIT (with simplification)

jour lève notre grisaille

trottoir notre ruelle
notre tour

jour lève notre envie
vous

faire comprendre tous
notre tour

Wordcount

Word Count !

MAP

For each word, generate the pair (word, 1)

```
def  map (key,value):     #key : nom du doc, value : contenu du doc
for word w in value:
    emitIntermediate (w,1)
```

Wordcount


Word Count !
MAP
jour lève notre grisaille
MAP
[(jour,1), (lève,1), (notre,1), (grisaille,1)]
trottoir notre ruelle notre tour
MAP
[(trottoir,1), (notre,1), (ruelle,1), (notre,1), (tour,1)]
jour lève notre envie vous
MAP
[(jour,1),(lève,1), (notre,1), (envie,1), (vous,1)]
faire comprendre tous notre tour
MAP
[(faire,1), (comprendre,1), (tous,1), (notre,1), (tour,1)]

Wordcount

Word Count !

SHUFFLE : group and sort all the pairs by key

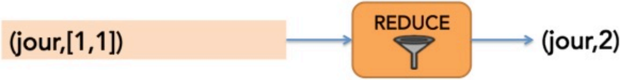
(comprendre, [1])	(notre, [1,1,1,1,1])
(envie,[1])	(ruelle,[1])
(faire,[1])	(tour,[1,1])
(grisaille,[1])	(tous,[1])
(jour,[1,1])	(trottoir,[1])
(lève, [1,1])	(vous, [1])

Wordcount

Word Count !

REDUCE : add all the values associated to a same key

```
def  reduce (key,values):      #key : un mot, values : liste de valeurs
result =0
for count c in values:
    result = result +1
emit(key,result)
```

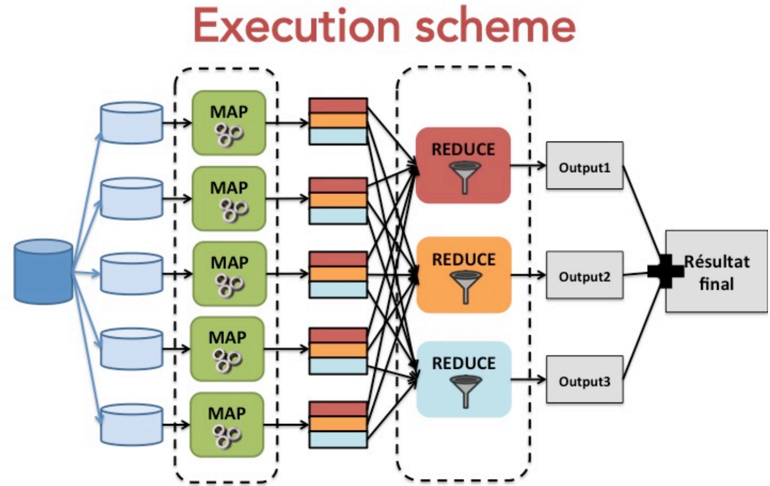


Basic WordCount algorithm in MapReduce

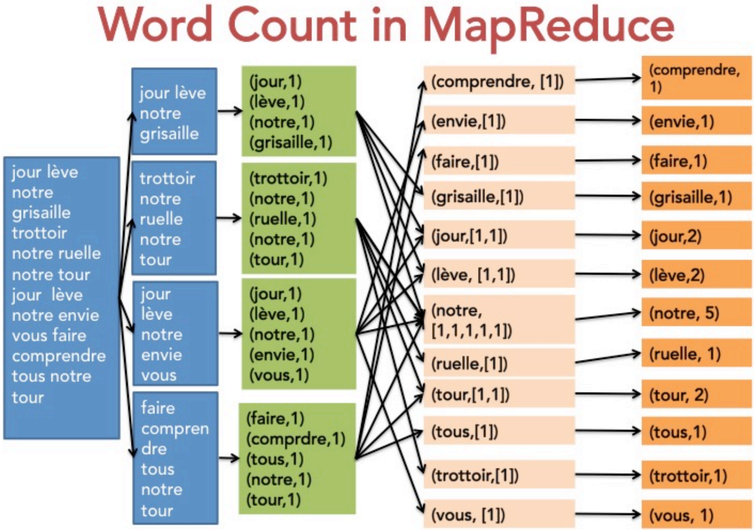
```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)
```

```
class Reducer
    method Reduce(term t, counts [c1, c2, ...])
        sum ← 0
        for all count c ∈ counts [c1, c2, ...] do
            sum ← sum + c
        Emit(term t, count sum)
```

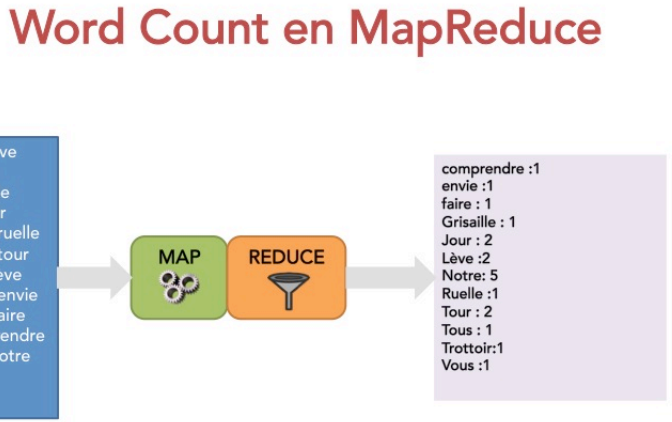
Wordcount


Execution scheme
MAP
MAP
MAP
MAP
MAP
REDUCE
REDUCE
REDUCE
Output1
Output2
Output3
Résultat final

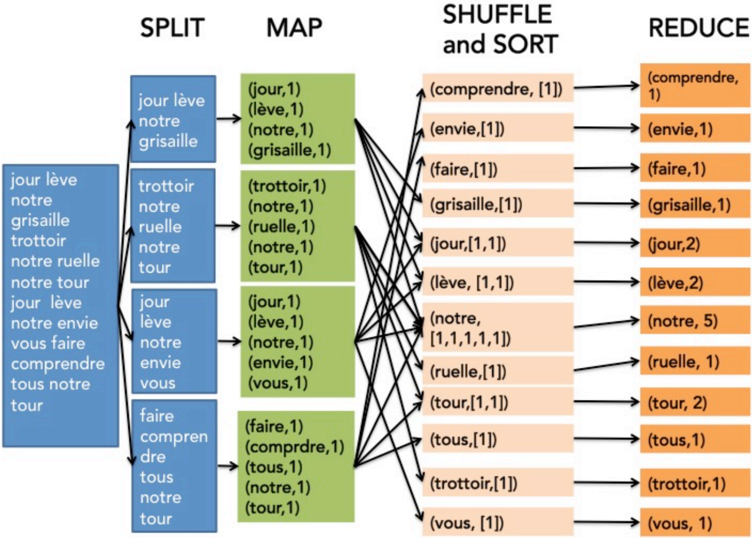
Wordcount


Word Count in MapReduce
jour lève notre grisaille trottoir notre ruelle notre tour jour lève notre envie vous faire comprendre tous notre tour
jour lève notre grisaille
trottoir notre ruelle notre tour
jour lève notre envie vous
faire compren dre tous notre tour
(jour,1) (lève,1) (notre,1) (grisaille,1)
(trottoir,1) (notre,1) (ruelle,1) (notre,1) (tour,1)
(jour,1) (lève,1) (notre,1) (envie,1) (vous,1)
(faire,1) (comprdre,1) (tous,1) (notre,1) (tour,1)
(comprendre, [1])
(envie,[1])
(faire,[1])
(grisaille,[1])
(jour,[1,1])
(lève, [1,1])
(notre, [1,1,1,1,1])
(ruelle,[1])
(tour,[1,1])
(tous,[1])
(trottoir,[1])
(vous, [1])
(comprendre, 1)
(envie,1)
(faire,1)
(grisaille,1)
(jour,2)
(lève,2)
(notre, 5)
(ruelle, 1)
(tour, 2)
(tous,1)
(trottoir,1)
(vous, 1)

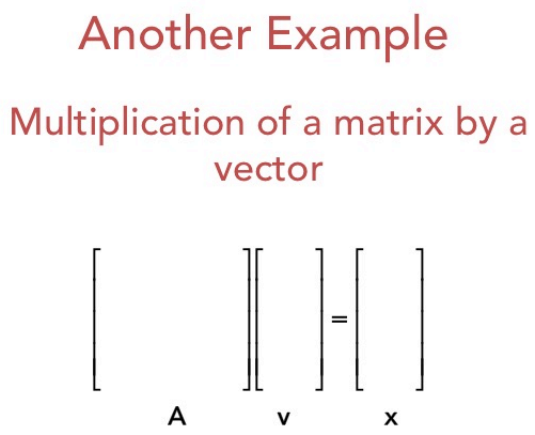
Wordcount


Word Count en MapReduce
jour lève notre grisaille trottoir notre ruelle notre tour jour lève notre envie vous faire comprendre tous notre tour
MAP
REDUCE
comprendre :1
envie :1
faire : 1
Grisaille : 1
Jour : 2
Lève :2
Notre: 5
Ruelle :1
Tour : 2
Tous : 1
Trottoir:1
Vous :1

Wordcount


SPLIT
MAP
SHUFFLE and SORT
REDUCE
jour lève notre grisaille trottoir notre ruelle notre tour jour lève notre envie vous faire comprendre tous notre tour
jour lève notre grisaille
trottoir notre ruelle notre tour
jour lève notre envie vous
faire compren dre tous notre tour
(jour,1) (lève,1) (notre,1) (grisaille,1)
(trottoir,1) (notre,1) (ruelle,1) (notre,1) (tour,1)
(jour,1) (lève,1) (notre,1) (envie,1) (vous,1)
(faire,1) (comprdre,1) (tous,1) (notre,1) (tour,1)
(comprendre, [1])
(envie,[1])
(faire,[1])
(grisaille,[1])
(jour,[1,1])
(lève, [1,1])
(notre, [1,1,1,1,1])
(ruelle,[1])
(tour,[1,1])
(tous,[1])
(trottoir,[1])
(vous, [1])
(comprendre, 1)
(envie,1)
(faire,1)
(grisaille,1)
(jour,2)
(lève,2)
(notre, 5)
(ruelle, 1)
(tour, 2)
(tous,1)
(trottoir,1)
(vous, 1)

Another example



Another example

Multiplication of a matrix by a vector

	$\bar{x}$	$\bar{x}P^1$	$\bar{x}P^2$	$\bar{x}P^3$	$\bar{x}P^4$	$\bar{x}P^5$	$\bar{x}P^6$	$\bar{x}P^7$	$\bar{x}P^8$	$\bar{x}P^9$	$\bar{x}P^{10}$	$\bar{x}P^{11}$	$\bar{x}P^{12}$	$\bar{x}P^{13}$
d_0	0.14	0.06	0.09	0.07	0.07	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05	0.05
d_1	0.14	0.08	0.06	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04
d_2	0.14	0.25	0.18	0.17	0.15	0.14	0.13	0.12	0.12	0.12	0.12	0.11	0.11	0.11
d_3	0.14	0.16	0.23	0.24	0.24	0.24	0.24	0.25	0.25	0.25	0.25	0.25	0.25	0.25
d_4	0.14	0.12	0.16	0.19	0.19	0.20	0.21	0.21	0.21	0.21	0.21	0.21	0.21	0.21
d_5	0.14	0.08	0.06	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04
d_6	0.14	0.25	0.23	0.25	0.27	0.28	0.29	0.29	0.30	0.30	0.30	0.30	0.31	0.31

Use for the page rank !

Source : Wikipedia

Another example

Multiplication of a matrix by a vector

✧ How to represent the matrix?

$$A = \begin{bmatrix} 3 & 2 & 0 \\ 0 & 4 & 1 \\ 2 & 0 & 1 \end{bmatrix} \Rightarrow \begin{array}{c} (0,0,3) \\ (0,1,2) \\ (1,1,4) \\ (1,2,1) \\ (2,0,2) \\ (2,2,1) \end{array}$$

List of triples (i,j, A[i][j])

Another example

Multiplication of a matrix by a vector

✧ How to represent the vector v?

$$\begin{bmatrix} 4 \\ 3 \\ 1 \end{bmatrix} \Rightarrow \begin{array}{c} (0,4) \\ (1,3) \\ (2,1) \end{array}$$

v — List of pairs(j, v[j])

Another example

Multiplication of a matrix by a vector

Simple case : the vector v fits in memory of the MAP node

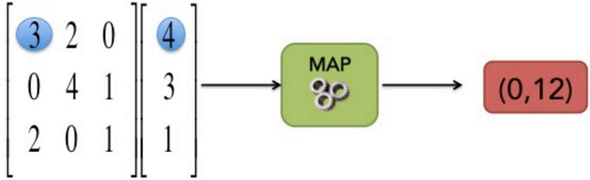
Another example

Multiplication of a matrix by a vector

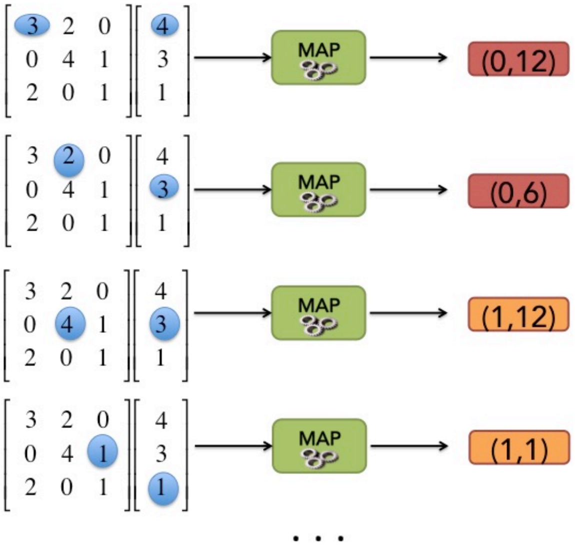
✧ MAP

Each entry (i,j,A[i][j]) of the matrix is transformed in (i, A[i][j] * v[j])

```
def map (key,value):     #key = None, value : (i,j,A[i][j])
    emitIntermediate (i,A[i][j]*v[j])
```



Another example

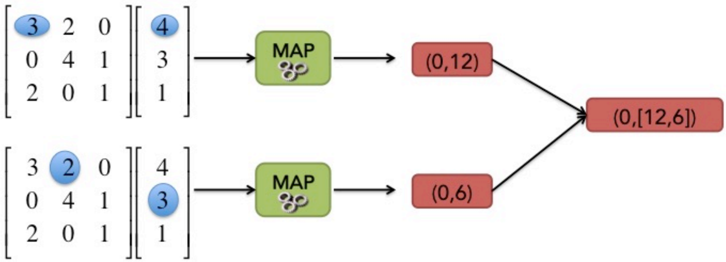


Another example

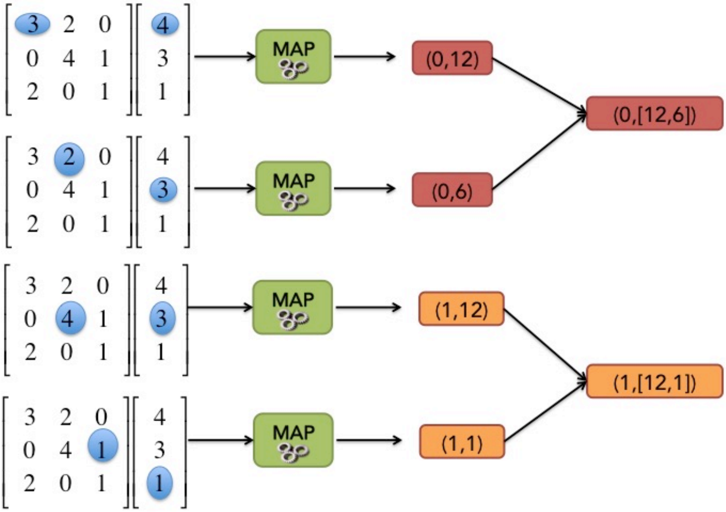
Multiplication of a matrix by a vector

✧ SHUFFLE

The pairs of the key i are grouped



Another example



. . .

Another example

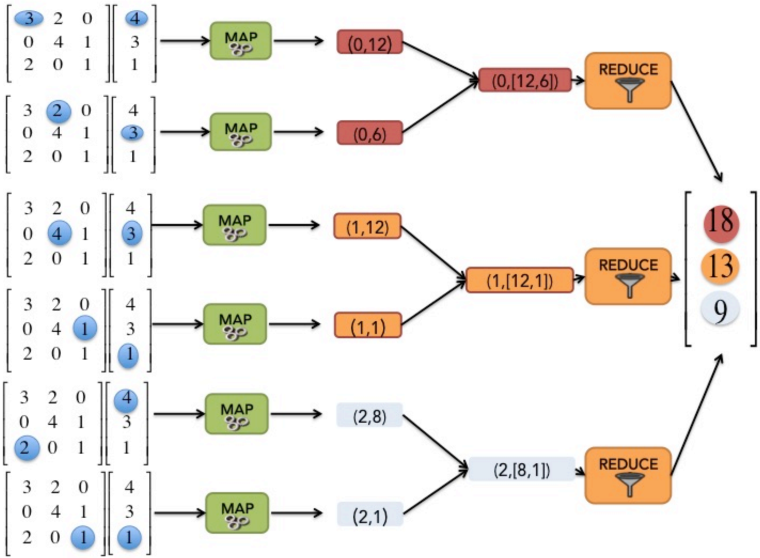
Multiplication of a matrix by a vector

✧ REDUCE

We sum the values associated to the same key

```
def reduce (key,values):     #key = row, values = liste de valeurs
      result =0
for v in values:
      result = result +v
emit(key,result)
```

Another example



MapReduce refinement : Partitioners and Combiners

Two additional elements complete the programming model : partitioners and reducers

Partitioner

Inputs to map tasks are simply created by contiguous splits of input file, but reduce tasks need to ensure that records with the same intermediate key end up at the same worker.
partition : $(k', \#partitions) \rightarrow$ partition for k'

Combiner

Combiners are an optimization in MapReduce that allow for local aggregation before the shuffle and sort phase.
combine : $(k', v') \rightarrow [(k', v'), ...]$

MapReduce : Partitioners

Partitioners are responsible for :

- Dividing up the intermediate key space.
- Assigning intermediate key-value pairs to reducers.

Hash-based partitioner

- System uses a default partition function :

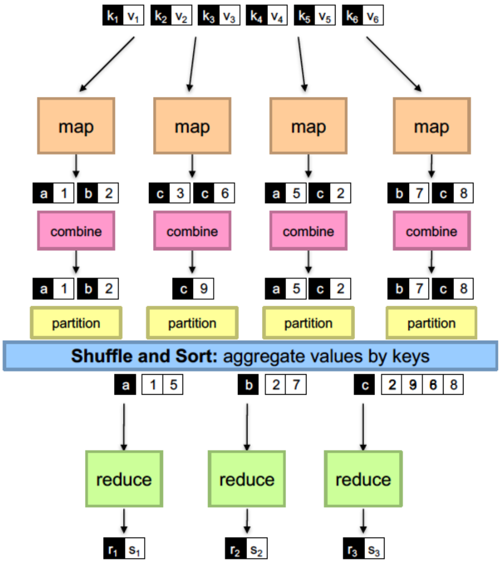
$$hash(key) mod\ R$$

  with R the number of reducers.
- When dealing with complex keys, it may be useful to override the default partition function.

MapReduce : Combiners

- A map task can produce many pairs with the same key (e.g. popular words in wordcount) : $(k, v_1), (k, v_2), \ldots$
- They need to be sent over the network to the reducer : costly.
- Combiner : local aggregation of these pairs into a single key-value pair at the mapper
- Decrease the size of intermediate data and save network time.
- Combiners = mini-reducers on the output of the mappers.

MapReduce scheme with combiners and partitioners



Basic WordCount algorithm in MapReduce

class Mapper
  method Map(docid a, doc d)
    for all term $t \in$ doc d **do**
      Emit(term t, count 1)

class Reducer
  method Reduce(term t, counts $[c_1, c_2, \ldots]$)
    $sum \leftarrow 0$
    for all count $c \in$ counts $[c_1, c_2, \ldots]$ **do**
      $sum \leftarrow sum + c$
    Emit(term t, count sum)

WordCount algorithm in MapReduce with combiners

```
class Mapper
    method Map(docid a, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do
            H{t} ← H{t} + 1                    ▷ Tally counts for entire document
        for all term t ∈ H do
            Emit(term t, count H{t})
```

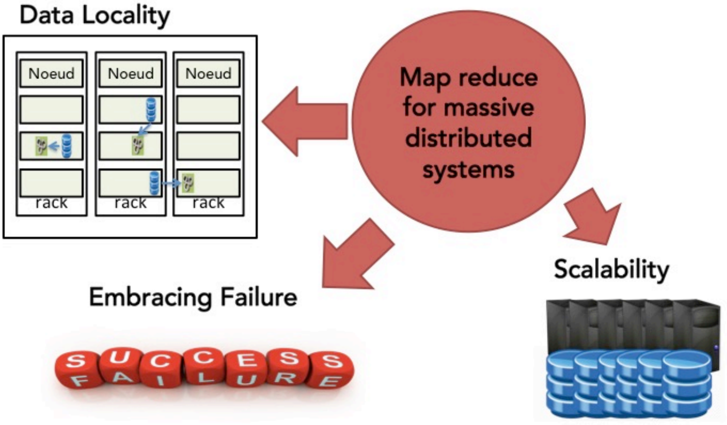
A key-value pair is emitted for each unique *term* in the document.

Combiners : precautions

- The use of combiners is at the discretion of the execution framework.
- In Hadoop, they also may be invoked in the reduce phase.
- They have to be carefully written.

Plan

MapReduce in real life



MapReduce : the execution framework

MapReduce program : a job

- Code for mappers and reducers.
- Code for combiners and partitioners.
- Configuration parameters.

A MapReduce job is submitted to the cluster

- The framework takes care of everything else.

MapReduce : runtime execution

MapReduce runtime execution environment handles :

- sheduling : assigns workers to map and reduce tasks.
- data distribution : partitions the input data and moves processes to data.
- synchronisation : manages required inter-machine communication to gather, sort and shuffle intermediate data.
- errors and faults : detects worker failures and restarts.

MapReduce : Scheduling

- **Each job is broken into tasks (smaller units)**.
  - Map tasks work on fractions of the input dataset.
  - Reduce tasks work on intermediate inputs and write back to the distributed file sytem.
- **The number of tasks may exceeed the number of available machines in a cluster**
  - The scheduler takes care of maintening something similar to a queue of pending tasks to be assigned to machines with avaialble ressources.
- **Jobs to be executed in a cluster requires scheduling as well**
  - Different users may submit jobs.
  - Jobs may be of various complexity.
  - Fairness is generally a requirement.

MapReduce : data flow

- The input and final output are stored on a distributed file system
  - Scheduler tries to schedule map tasks close to physical storage location of input data.
- The intermediate results are stored on the local file system of map and reduce workers.
- The output is often the input of another MapReduce task.

MapReduce : Synchronization

- Synchronization is achieved by the *Shuffle and Sort* step.
  - Intermediate key-value pairs are group by key.
  - This requires a distributed sort involving all mappers, and taking into account all reducers.
  - If you have m mappers and r reducers, it involves $m \times r$ copying operations.
- Important : the reduce operation cannot start until all mappers have finished (to guarantee that all values associated with the same key have been gathered)

MapReduce : coordination

Master keeps an eye on each task status (Master-slave architecture)

- idle tasks get scheduled as workers become available
- When a map task completes, it sends Master the location and sizes of its R intermediate files, one for each reducer and then Master pushes this info to reducers.
- Master pings workers periodically to detect and manage failures.

MapReduce : implementations

- Google has a proprietary implementation in C++.
- Hadoop is an open-source implementation in JAVA.
  - Development led by Yahoo, used in production.
  - Apache project.
  - A big software ecosystem.
- Lot of custom research implementations.

Plan

1. MapReduce : the programming model

2. MapReduce : the execution framework

3. MapReduce Algorithm Design patterns
   - Summarization Patterns
     - Local Aggregation
   - Filtering Design Patterns

Algorithm Design with MapReduce

Developing algorithms involve :

- Preparing the input data.
- Writing the mapper and the reducer.
- And, optionally, writing the combiner and the partitioner.

How to recast existing algorithms in MapReduce :

- Not always obvious.
- Importance of data structures : complex data structures as keys and values.
- Difficult to optimize.

Learn by practice and examples

- MapReduce design patterns.
- Synchronization of intermediate results is difficult : the tricky task.

Algorithm Design with MapReduce

Aspects that are not under the control of the designer

- *Where* a mapper or a reducer will run.
- *When* a mapper or a reducer begins or finishes.
- *Which* input key-value pairs are processed by a specific mapper.
- *Which* intermediate key-value pairs are processed by a specific reducer.

Algorithm Design with MapReduce

Aspects that can be controlled

- Construct data structures as keys and values.
- Execution of user-specific initialisation or termination code at each Mapper or Reducer.
- State preservation in Mapper - Reducer across multiple inputs or intermediates keys.
- User-controlled partitionning of the key space.

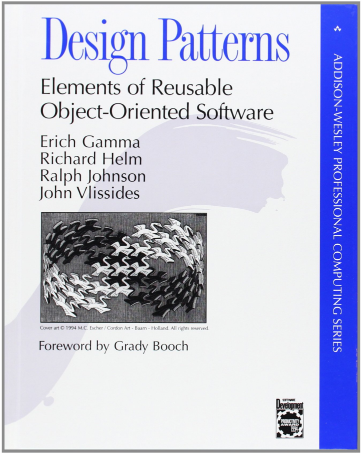
Algorithm Design with MapReduce

Designing MapReduce algorithms can be complex

- Many algorithms cannot be easily expressed as a single MapReduce job.
- Decompose complex algorithms into a sequence of MapReduce jobs :
  - Requires orchestrating data so that the output of one job becomes the input of the next
- Often require an external driver.

What are design patterns ?

- Reusable solutions to problems.
- Domain independent.
- Makes the intent of code easier to understand.
- Provides a common language for solutions.

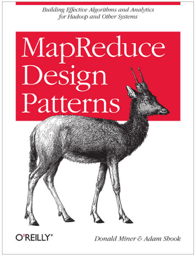


Why Map Reduce design patterns ?

- Recurring patterns in data-related problem solving.
- Map Reduce is a new way of thinking.
- Foundations for higher-level tools (Pig, Hive).
- To make easier the use of Map Reduce.

MapReduce Design Patterns

- Building effective algorithms and analytics for Map Reduce
- 23 patterns grouped into 6 catagories :
  - Summarization
  - Filtering
  - Data Organization
  - Joins
  - Input and Output
  - Metapatterns



Summarization Patterns

Summarization patterns

Top-down summaries to get a top-level view of your data, i.e. grouping similar data together and then performing an operation such as :

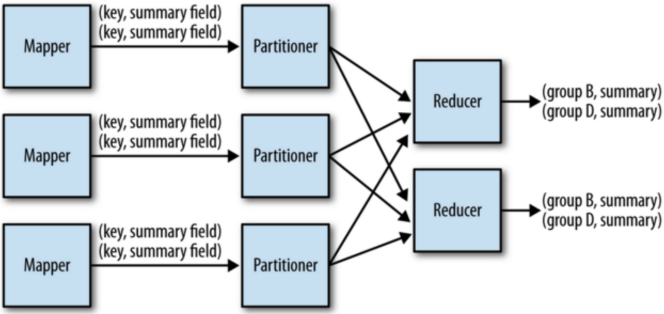
- Calculating a statistic (min, max, average,...)
- Building an index
- Counting with counters

Summarization Patterns

Numerical summarization pattern

A general pattern for calculating aggregate statistical values over your data.

- Mapper : outputs keys that consist of each field to group by and values consisting of any pertinent numerical items.
- Reducer : receives a set of numerical values $(v_1, v_2, ..., v_n)$ associated with a group-by-key and performs the aggregate function $\theta(v_1, v_2, ..., v_n)$.

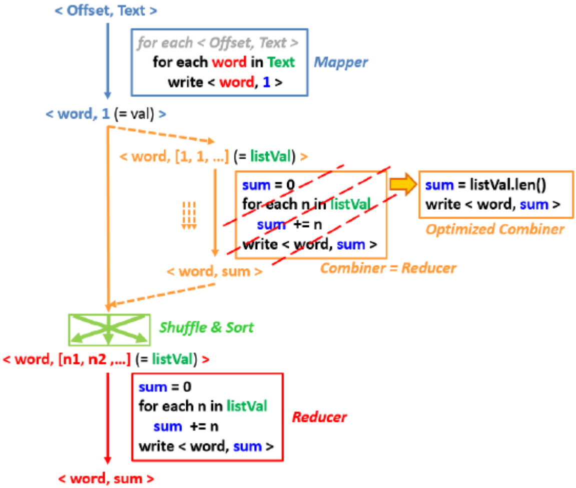


Summarization Patterns

Numerical summarization pattern : performance analysis

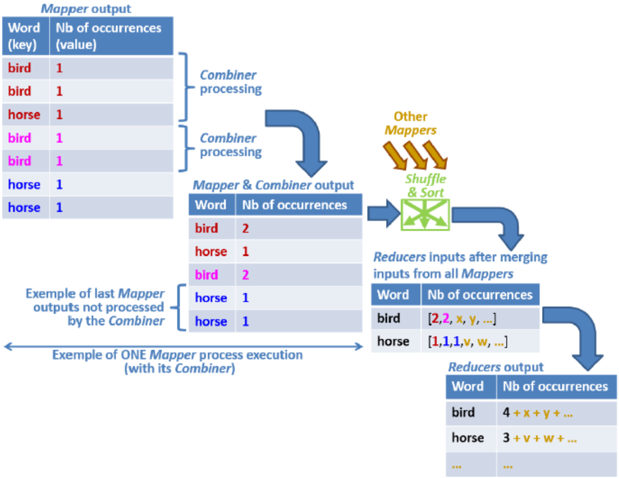
- Importance of a combiner properly used : Local aggregation.
- Data skew of reduce groups is problematic.

Summarization Patterns



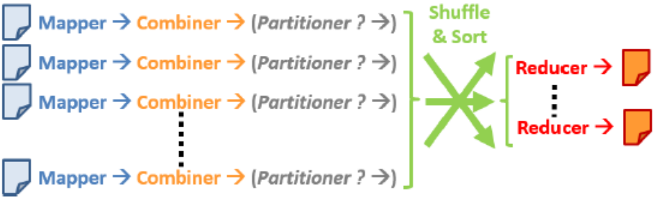
Source : S. Vialle course

Summarization Patterns



Source : S. Vialle course

Summarization Patterns



Source : S. Vialle course

Local Aggregation

- In data-intensive distributed processing, the most important aspect of synchronization is the exchange of intermediate results
  - It involves the copy of intermediate results from the processes that produced them to those that consumed them.
  - It involves data transfers over the network.
  - In some MapReduce implementation (as Hadoop), disk IO is involved : intermediate results are written to disk.

Network and disk latencies are expensive.

Idea : reduce the number and size of key-values pair to be shuffle.

In-Mapper Combiners

Idea

- Use an associative array to cumulate intermediate results.
  - Wordcount : The array is used to tally up term counts within a single document.
  - The `Emit` method is called only after all `Inputrecords` have been processed.

In-Mapper Combiners

WordCount example

class Mapper
    method Map(docid a, doc d)
        for all term $t \in$ doc d **do**
            Emit(term t, count 1)

class Reducer
    method Reduce(term t, counts $[c_1, c_2, \ldots]$)
        $sum \leftarrow 0$
        for all count $c \in$ counts $[c_1, c_2, \ldots]$ **do**
            $sum \leftarrow sum + c$
        Emit(term t, count sum)

Basic WordCount algorithm : emit a key-value pair for each term in the document

In-Mapper Combiners

WordCount example

```
class Mapper
    method Map(docid a, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do
            H{t} ← H{t} + 1                    ▷ Tally counts for entire document
        for all term t ∈ H do
            Emit(term t, count H{t})
```

WordCount algorithm with a combiner : emit a key-value pair for each unique term in the document

In-Mapper Combiners

Combiner Across Multiple Documents

- Prior to processing any input key-value pairs, an associative array for holding term counts is initialized (`initialize` method in MapReduce implementation API, `setup` in Hadoop)
- We can continue to accumulate partial term counts in the associative array across multiple documents and emit the key-values pairs only when the mapper has processed all or part of the documents.
- It requires an API that provides an opportinuty to execute user-specified code after the Map method has been applied to all input key-value pairs of the input data split of the map task (`cleanup` in Hadoop)

In-Mapper Combiners

WordCount example

```
class Mapper
    method Initialize
        H ← new AssociativeArray
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            H{t} ← H{t} + 1                ▷ Tally counts across documents
    method Close
        for all term t ∈ H do
            Emit(term t, count H{t})
```

WordCount algorithm with in-mapper combiner.

In-Mapper Combiners

Advantages

- Provide control over when local aggregation occurs and how it exactly takes place

Precautions

- In-mapper combining breaks the functional programming paradigm due to state preservation.
- State preservation implies that the algorithm behavior might depend on the execution order.

Scalability bottleneck

- In-mapper combining depends on having sufficient memory to store intermediate results.

In-Mapper Combiners : Another example

Computing the mean (in TD)

We suppose that we have a large dataset where input keys are strings and input values are integers : we wish to compute the mean of all integers associated with the same key (e.g. logs from a popular website, keys are user ids and values some measure of activity).

1. Write MapReduce pseudo-code to solve the problem.
2. Modify the solution to use combiners.

$$mean(1,2,3,4,5) \neq mean(mean(1,2), mean(3,4,5))$$

3. Modify the solution to use in-mapper combining.

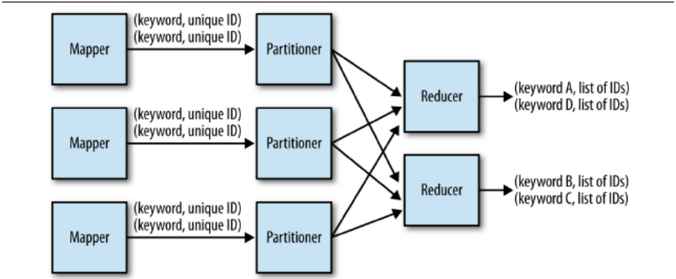
Summarization Patterns

Summarization pattern : inverted index

- Intent : Generate an index from a data set to allow for faster searches or data enrichment capabilities

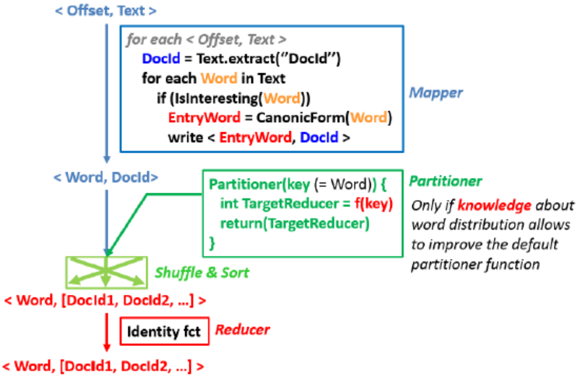
family	d_1, d_3, d_5, d_6
football	d_4
jaguar	$d_1, d_2, d_3, d_4, d_5, d_6$
new	d_1, d_2, d_5
rule	d_6
us	d_4, d_5
world	d_1

Summarization Patterns : inverted index



Source : Miner and Shook book

Summarization Patterns : inverted index



Source : S. Vialle course

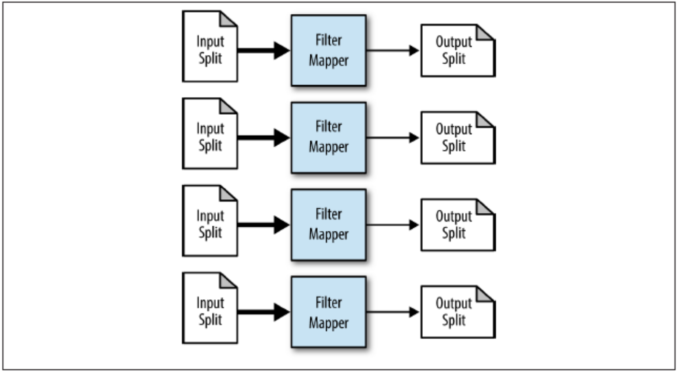
Filtering Patterns

- To understand a smaller piece of data.
  - Sampling
  - Top-k listing
  - Distinct
  - ...
- Filtering does not change the actual records.

Filtering Design Pattern

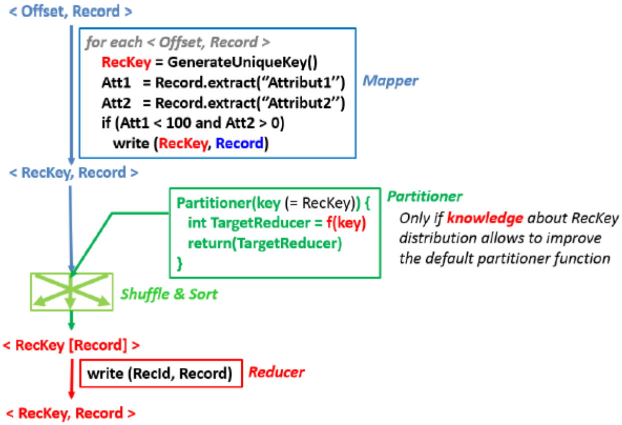
No reducer is needed

```
map(key, record):
   if we want to keep record then
      emit key,value
```



Source : Miner and Shook book

Filtering Design Pattern



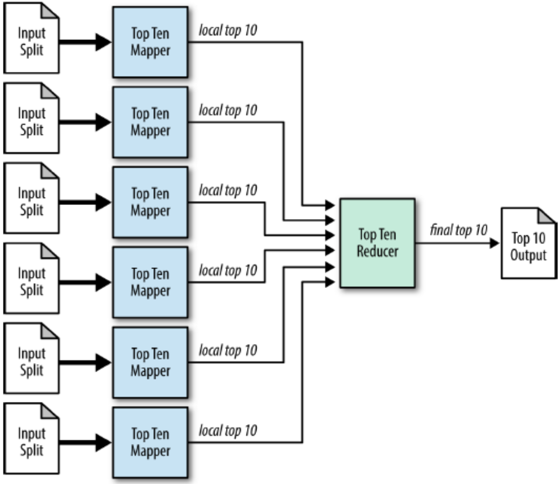
Source : S. Vialle course

Filtering Patterns

Top-k filter

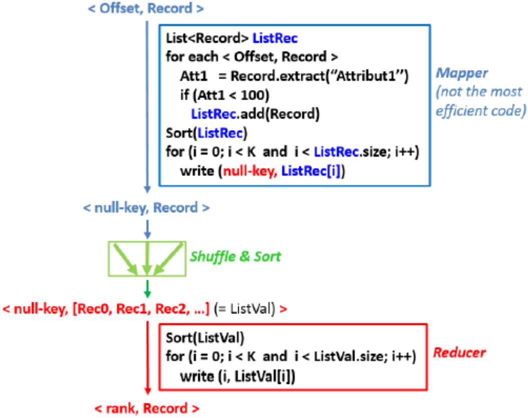
- Intent : Retrieve a small number of Top-k records according to some ranking, criteria.
- Applications
  - Outliers analysis
  - Selecting interesting data

Filtering Patterns : Top-k filter



Source : Miner and Shook book

Filtering Patterns : Top-k filter



Source : S. Vialle course

And now

Your turn