An Insufficient Introduction to Spark

Part 1: The MapReduce computing model

Riccardo Murri <riccardo.murri@gmail.com>

2019-02-11

What is Spark?

Apache Spark is a general-purpose distributed computation framework.

Parallel computing

"In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

- A problem is broken into discrete parts that can be solved concurrently
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed"

Reference: Introduction to Parallel Computing,
Blaise Barney, Lawrence Livermore National Laboratory,

https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

Parallel computing

"In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

- A problem is broken into discrete parts that can be solved concurrently
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed"

Reference: Introduction to Parallel Computing,
Blaise Barney, Lawrence Livermore National Laboratory,

https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

Distributed computing

"A distributed system is a model in which components located on networked computers communicate and coordinate their actions by passing messages.

[. . .]

Three significant characteristics of distributed systems are:

- concurrency of components,
- lack of a global clock, and
- independent failure of components."

Reference: https://en.wikipedia.org/wiki/Distributed_computing

Why distributed computing?

Scale out

Attack larger problems
by using multiple computers.

Speed

Solve independent parts
of a problem concurrently.

What's so hard about distributed computation?

- Synchronization: Tasks need to be coordinated between the different machines.
- Distributing and collecting data across multiple processors can be verbose and complicated.
- No longer have one machine but many; *hence*, hard to debug and prone to failures.

Liberation through limitation

Popular “framework” approach (Map/Reduce, BSP):

- *Limited to a specific model of parallel computation*
  - Users need/can only supply a few “functions” in a pre-determined scheme.
- *Framework takes cares of work+data distribution and fault-tolerance*

Usefulness of a framework depends on how broad a class of problems the parallel computing model can be applied to.

MapReduce

Let's start with some concrete examples.

Exercise 1.A: Write a function `Lengths(L)` that takes a list `L` of *strings* and returns a list of the their lengths.

Exercise 1.B: Write a function `LongerThan(L, m)` that takes a list `L` of strings and a single value `m`, then returns a list of those strings in `L` whose length is larger than `m`.

Exercise 1.C: Write a function `Sum(L)` that takes a list `L` of numbers and returns the sum of all of them.

Exercise 1.D: Write a function `RandList(N)` that takes generates and returns a list of `N` random floating-point numbers (each ranging from `0.0` to `1.0`).

map, reduce, filter (1)

Constructing a new list by looping over a given list and applying a function on all elements is *so common* that there are specialized functions for that:

map(fn, L)

Return a new list formed by applying function `fn(x)` to every element `x` of list `L`

reduce(fn2, L)

Apply *associative* function `fn2(x,y)` to the first two items `x` and `y` of list `L`, then apply `fn2` to the result and the third element of `L`, and so on until all elements have been processed — return the final result.

map, reduce, filter (2)

filter(fn, L)

Return a new list formed by elements `x` of list `L` for which `fn(x)` evaluates to a "True" value.

See also: http://www.python-course.eu/lambda.php and https://docs.python.org/3/howto/functional.html (more advanced)

This is how you could rewrite the examples using `map`, `reduce`, and `filter`.

```
# *** ex 1.A ***
def Lengths(S):
  return map(len, S)

# *** ex 1.C ***
def Sum(L):
  from operator import add
  return reduce(add, L)
```

```
# *** ex 1.B ***
def LargerThan(L, m):
  # note:  can define
  # func's in func's!
  def good(s):
    return (len(s) > m)
  return filter(good, L)
```

Exercise 1.E: A rough approximation to the constant π can be computed (using a Monte Carlo method) as follows:

1. Let $N > 0$ be a large integer,
2. pick N points in the square $\{(x, y) : 0 < x, y < 1\}$ uniformly at random;
3. count the number P of points that fall into the unit circle $\{(x, y) : x^2 + y^2 < 1\}$;
4. for large enough N, the ratio P/N approximates the area of a quarter of the unit circle, i.e. $\pi/4$.

Write Python code that computes an approximation to π using the above procedure.

What is the advantage of `map+reduce` over loops?

What is the advantage of `map`+`reduce` over loops?

Parallelism.

MapReduce: advantages of the model

> *"Programs written in this style*
> *are automatically parallelized*
> *and executed on a large cluster of machines"*

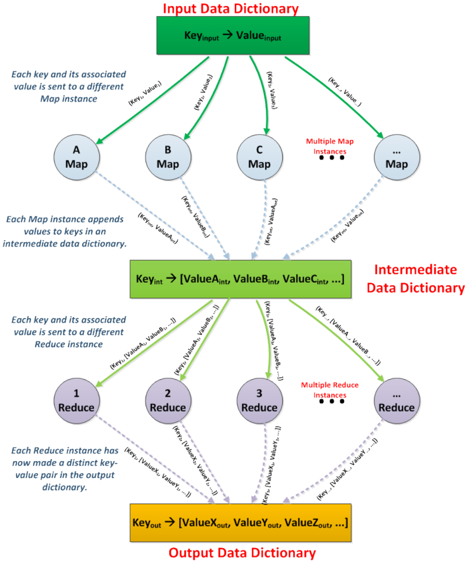
Reference: Dean and Ghemawat,

MapReduce: Simplified Data Processing on Large Clusters

MapReduce

The Map function processes a key/value pair to produce intermediate key/value pairs.

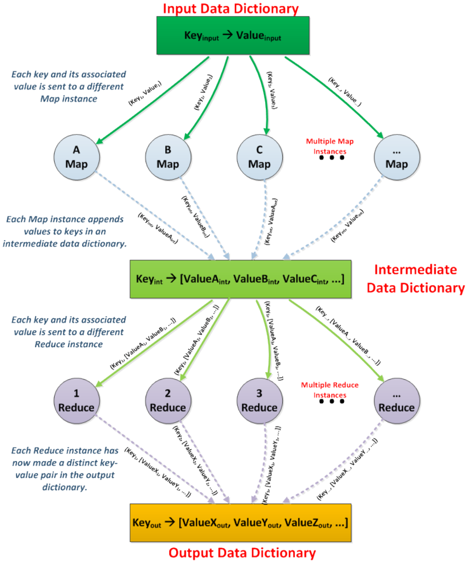
The Reduce function merges all intermediate values associated with a given key.



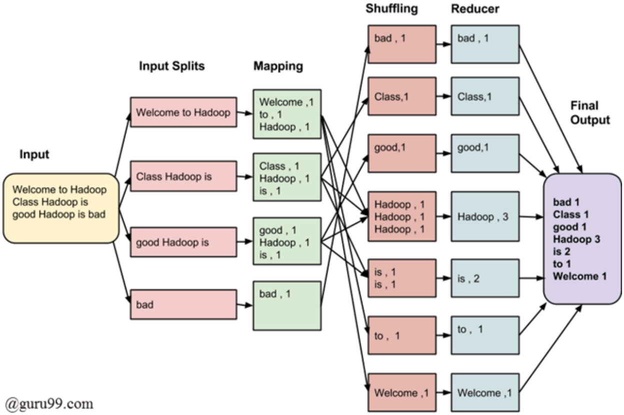
MapReduce

The Map function processes a key/value pair to produce intermediate key/value pairs.

The Reduce function merges all intermediate values associated with a given key.



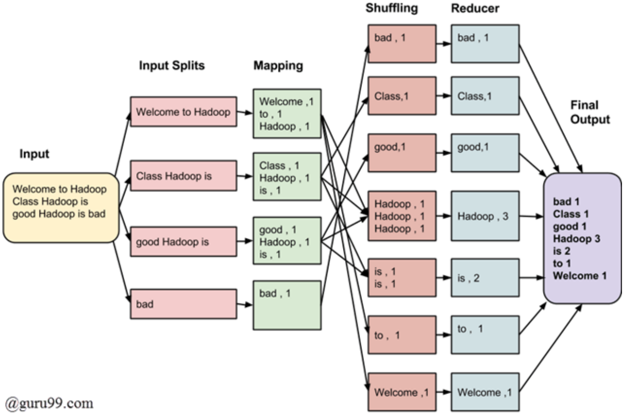
Example: word count



Input is a text file, to be *split* at line boundaries.

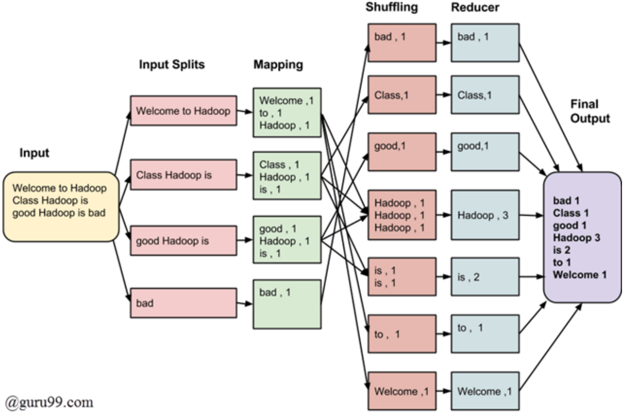
Image source: http://www.guru99.com/introduction-to-mapreduce.html

Example: word count



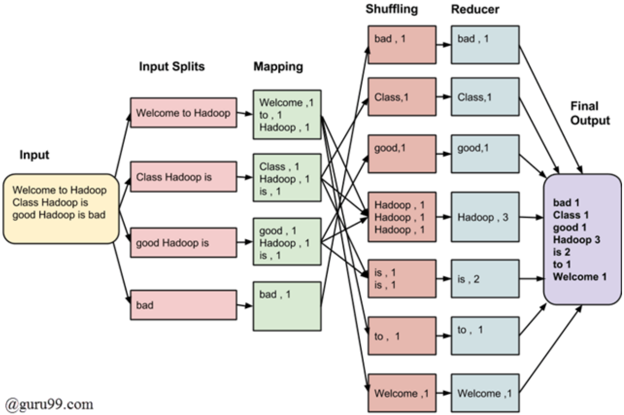
The *Map* function scans an input line and outputs a pair *(word, 1)* for each word in the text line.

Example: word count



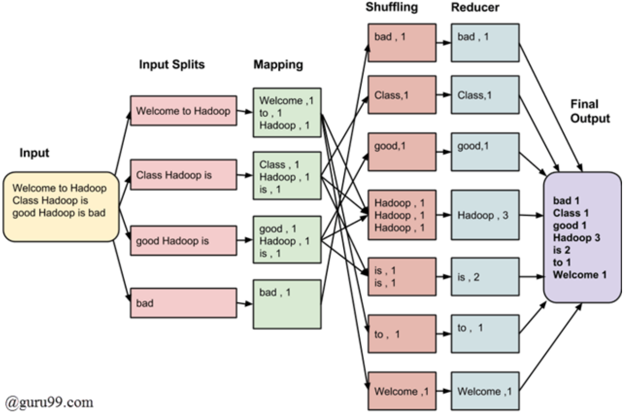
The pairs are *shuffled* and sorted so that each reducer gets all pairs *(word, 1)* with the same *word* part.

Example: word count



The *Reduce* function gets all pairs *(word, 1)* with the same *word* part, and outputs a single pair *(word, count)* where *count* is the number of input items received.

Example: word count



The global output is a list of pairs *(word, count)* where *count* is the number of occurences of *word* in the input text.

MapReduce: features of the implementation

The run-time system takes care of the details:

- *partitioning the input data,*
- *scheduling the program execution,*
- *handling machine failures,*
- *managing the required inter-machine communication.*

These are all *highly nontrivial* tasks to handle!

Reference: Dean and Ghemawat:

MapReduce: Simplified Data Processing on Large Clusters

MapReduce: features of the implementation

The run-time system takes care of the details:

- *partitioning the input data,*
- *scheduling the program execution,*
- *handling machine failures,*
- *managing the required inter-machine communication.*

These are all *highly nontrivial* tasks to handle!

Reference: Dean and Ghemawat:

MapReduce: Simplified Data Processing on Large Clusters

MapReduce: features of the implementation

The run-time system takes care of the details:

- *partitioning the input data,*
- *scheduling the program execution,*
- *handling machine failures,*
- *managing the required inter-machine communication.*

These are all *highly nontrivial* tasks to handle!

Reference: Dean and Ghemawat:

MapReduce: Simplified Data Processing on Large Clusters

MapReduce: features of the implementation

The run-time system takes care of the details:

- *partitioning the input data,*
- *scheduling the program execution,*
- *handling machine failures,*
- *managing the required inter-machine communication.*

These are all *highly nontrivial* tasks to handle!

Reference: Dean and Ghemawat:

MapReduce: Simplified Data Processing on Large Clusters

MapReduce: features of the implementation

The run-time system takes care of the details:

- *partitioning the input data,*
- *scheduling the program execution,*
- *handling machine failures,*
- *managing the required inter-machine communication.*

These are all *highly nontrivial* tasks to handle!

Reference: Dean and Ghemawat:

MapReduce: Simplified Data Processing on Large Clusters

What MapReduce is *not* good for

Low-latency computation
(e.g., interactive tasks).

Iterative computation
(no provision to re-use
already-computed results)

Problems which cannot easily
be partitioned or recombined
(i.e., do not fit the paradigm)

Appendix

References

Dean, J., and Ghemawat, S.: "MapReduce: Simplified Data Processing on Large Clusters", OSDI'04

Greiner, J. and Wong, S.: "Distributed Parallel Processing with MapReduce"