BigData, i.e., Scalable Algorithm Design

The “Map Reduce” Programming Model

Marco Milanesio

UCA - MSc DATA SCIENCE 2020-21

- Jimmy Lin and Chris Dyer, "Data-Intensive Text Processing with MapReduce," Morgan & Claypool Publishers, 2010[1]
- Tom White, "Hadoop, The Definitive Guide," O'Reilly / Yahoo Press, 2012
- Anand Rajaraman, Jeffrey D. Ullman, Jure Leskovec, "Mining of Massive Datasets", Cambridge University Press, 2013
- Holden Karau, Andy Konwinski, Patrick Wendell and Matei Zaharia, "Learning Spark", O'Reilly

This lecture is built starting from material from Prof. Michiardi's "Cloud" course @Eurecom

[1] `http://lintool.github.io/MapReduceAlgorithms/`

What is Big Data?

- **Vast repositories of data**
  - The Web
  - Physics
  - Astronomy
  - Finance
- **Volume, Velocity, Variety**
- **It's not the algorithm, it's the data!**
  - More data leads to better accuracy
  - With more data, accuracy of different algorithms converges

What is the “Map Reduce” Programming Model?

- **A distributed programming model**:
  - Inspired by functional programming
  - Inspired by Bulk Synchronous Parallelism (BSP)
- **An instance of an execution framework**:
  - Designed for large-scale data processing
  - Designed to run on clusters of commodity hardware

Key Principles

Scale out, not up!

- **For data-intensive workloads, a large number of commodity servers is preferred over a small number of high-end servers**
  - Cost of super-computers is not linear
  - But datacenter efficiency is a difficult problem to solve
- **Some numbers ($\sim$ 2012):**
  - Data stored/processed by Google every day: $O(EB)$
  - Data stored/processed by Facebook every day: $O(PB)$

Implications of Scaling Out

- **Processing data is quick, I/O is very slow**
  - 1 Mechanical HDD $\sim$ 100 MB/sec
  - 1000 Mechanical HDDs $\sim$ 100 GB/sec
- **Sharing vs. Shared nothing**:
  - Sharing: manage a common/global state
  - Shared nothing: independent entities, no common state
- **Sharing is difficult**:
  - Synchronization, deadlocks
  - Finite bandwidth to access data from SAN
  - Temporal dependencies are complicated (restarts)

Failures are the norm, not the exception

- **Failures are part of everyday life**
  - Mostly due to the scale and shared environment
- **Sources of Failures**
  - Hardware / Software
  - Electrical, Cooling, ...
  - Unavailability of a resource due to overload
- **Failure Types**
  - Permanent
  - Transient

Move Processing to the Data

- **Drastic departure from high-performance computing model**
  - HPC: distinction between processing nodes and storage nodes
  - HPC: CPU intensive tasks
- **Data intensive workloads**
  - Generally not processor demanding
  - The network becomes the bottleneck
  - Framework generally assumes processing and storage nodes to be collocated

  $\rightarrow$ **Data Locality Principle**
- **Distributed filesystems are necessary**

Process Data Sequentially and Avoid Random Access

- **Data intensive workloads**
  - Relevant datasets are too large to fit in memory
  - Such data resides on disks
- **Disk performance is a bottleneck**
  - **Seek times** for random disk access are **the problem**
    - Example: 1 TB DB with 10^{10} 100-byte records. Updates on 1% requires 1 month, reading and rewriting the whole DB would take 1 day[2]
  - Organize computation for sequential reads

[2] From a post by Ted Dunning on the Hadoop mailing list

Implications of Data Access Patterns

- **Systems designed for:**
  - **Batch processing**
  - involving (mostly) **full scans** of the data

- **Typically, data is collected "elsewhere" and copied to the distributed filesystem**
  - E.g.: Apache Kafka, Hadoop Sqoop, $\cdots$

- **Data-intensive applications**
  - Read and process the whole Web (e.g. PageRank)
  - Read and process the whole Social Graph (e.g. LinkPrediction, a.k.a. "friend suggest")
  - Log analysis (e.g. Network traces, Smart-meter data, $\cdots$)

Hide System-level Details

- **Separate the *what* from the *how***
  - Framework abstracts away the “distributed” part of the system
  - Such details are handled by internal primitives

- **BUT: In-depth knowledge of the framework is key**
  - Custom data reader/writer
  - Custom data partitioning
  - Memory utilization

- **Auxiliary components**
  - Too many to list!

Seamless Scalability

- **We can define scalability along two dimensions**
  - In terms of data: given twice the amount of data, the same algorithm should take no more than twice as long to run
  - In terms of resources: given a cluster twice the size, the same algorithm should take no more than half as long to run
- **Embarrassingly parallel problems**
  - Simple definition: independent (shared nothing) computations on fragments of the dataset
  - How to to decide if a problem is embarrassingly parallel or not?

The Programming Model

Functional Programming Roots

- **Key feature: higher order functions**
  - Functions that accept other functions as arguments
  - **Map** and **Fold**

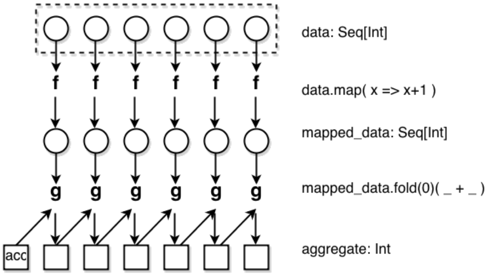


Figure: Illustration of *map* and *fold*.

Functional Programming Roots

- **map phase:**
  - Given a list, *map* takes as an argument a function f (that takes a single argument) and applies it to all element in a list
- **fold phase:**
  - Given a list, fold takes as arguments a function g (that takes two arguments) and an initial value (an accumulator)
  - g is first applied to the initial value and the first item in the list
  - The result is stored in an intermediate variable, which is used as an input together with the next item to a second application of g
  - The process is repeated until all items in the list have been consumed

Functional Programming Roots

- **We can view map as a transformation over a dataset**
  - This transformation is specified by the function *f*
  - Each functional application happens in **isolation**
  - The application of *f* to each element of a dataset can be parallelized in a straightforward manner
- **We can view fold as an aggregation operation**
  - The aggregation is defined by the function *g*
  - Data locality: elements in the list must be "brought together"
  - If we can **group** elements of the list, also the fold phase can proceed in parallel
- **Associative and commutative operations**
  - Allow performance gains through local aggregation and reordering

Functional Programming and “Map Reduce”

- **Equivalence of “Map Reduce” and Functional Programming:**
  - The **map** of Hadoop MapReduce corresponds to the map operation
  - The **reduce** of Hadoop MapReduce corresponds to the fold operation
- **The framework coordinates the map and reduce phases:**
  - Grouping intermediate results happens in parallel
- **In practice:**
  - User-specified computation is applied (in parallel) to all input records of a dataset
  - Intermediate results are aggregated by another user-specified computation

What can we do with this Programming Model??

- **Introducing the Data Flow abstraction**
  - The “old” Hadoop MapReduce programming model appears quite limited and strict
  - Apache Spark programming model is much more flexible, and operates on a directed acyclic graph representative of the computations
- **Generally, everything can be computed with the “Map Reduce” model**
  - We will focus on illustrative cases
  - “design patterns”

Data Structures

- **Key-value pairs are the basic data structure in "Map Reduce"**
  - Keys and values can be: integers, float, strings, raw bytes
  - They can also be **arbitrary data structures**
- **The design of "Map Reduce" algorithms involves**:
  - Imposing the key-value structure on arbitrary datasets[3]
    - E.g.: for a collection of Web pages, input keys may be URLs and values may be the HTML content
  - In some algorithms, input keys are not used, in others they uniquely identify a record
  - Keys can be combined in complex ways to design various algorithms

[3] There's more about it: here we only look at the input to the map function.

A Generic "Map Reduce" Algorithm

- **The programmer defines a mapper and a reducer as follows**[4][5]:
  - map: $(k_1, v_1) \rightarrow [(k_2, v_2)]$
  - reduce: $(k_2, [v_2]) \rightarrow [(k_3, v_3)]$
- **In words**:
  - A dataset stored on an underlying **distributed** filesystem, which is split in a number of **blocks** across machines
  - The mapper is applied to every input key-value pair to generate intermediate key-value pairs
  - The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs

[4] We use the convention $[\cdots]$ to denote a list.
[5] Pedices indicate different data types.

Where the magic happens

- **Implicit between the map and reduce phases is a parallel "group by" operation on intermediate keys**
  - Intermediate data arrive at each reducer in order, sorted by the key
  - No ordering is guaranteed across reducers
- **Output keys from reducers are written back to the distributed filesystem**[6]
  - The output may consist of *r* distinct files, where *r* is the number of reducers
  - Such output may be the input to a subsequent phase[7]

[6] Only Hadoop MapReduce. Apache Spark keeps in memory intermediate data.

[7] Think of **iterative algorithms**.

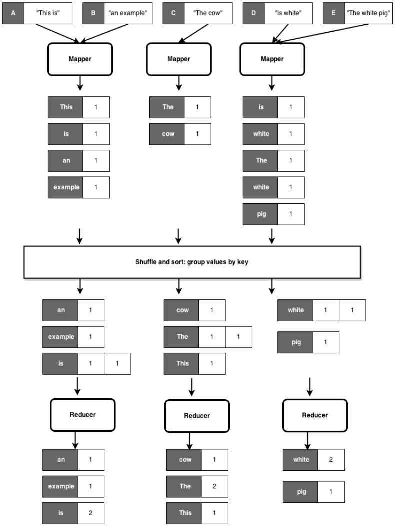
Where the magic happens

- **Intermediate keys are transient**:
  - They are not stored on the distributed filesystem
  - They are "spilled" to the local disk of each machine in the cluster

"Hello World" in "Map Reduce"

```
class Mapper
    method Map(offset a, line l)
        for all term t ∈ line l 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)
```


A "This is"
B "an example"
C "The cow"
D "is white"
E "The white pig"
Mapper
Mapper
Mapper
This 1
is 1
an 1
example 1
The 1
cow 1
is 1
white 1
The 1
white 1
pig 1
Shuffle and sort: group values by key
an 1
example 1
is 1 1
cow 1
The 1 1
This 1
white 1 1
pig 1
Reducer
Reducer
Reducer
an 1
example 1
is 2
cow 1
The 2
This 1
white 2
pig 1

“Hello World” in “Map Reduce”

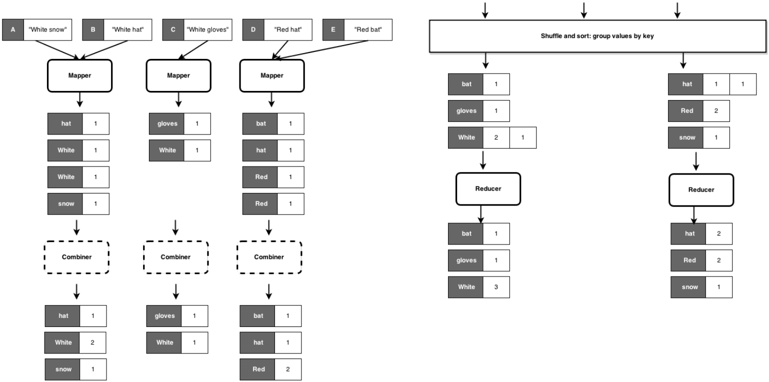
- **Input:**
  - Key-value pairs: (offset, line) of a file stored on the distributed filesystem
  - a: unique identifier of a line offset
  - l: is the text of the line itself
- **Mapper:**
  - Takes an input key-value pair, tokenize the line
  - Emits intermediate key-value pairs: the word is the key and the integer is the value
- **The framework:**
  - Guarantees all values associated with the same key (the word) are brought to the same reducer
- **The reducer:**
  - Receives all values associated to some keys
  - Sums the values and writes output key-value pairs: the key is the word and the value is the number of occurrences

Combiners

- **Combiners are a general mechanism to reduce the amount of intermediate data**
  - They could be thought of as “mini-reducers”
- **Back to our running example: word count**
  - Combiners aggregate term counts across documents processed by each map task
  - If combiners take advantage of all opportunities for local aggregation we have at most $m \times V$ intermediate key-value pairs
    - m: number of mappers
    - V: number of unique terms in the collection
  - Note: due to Zipfian nature of term distributions, not all mappers will see all terms

A word of caution

- **The use of combiners must be thought carefully**
  - In Hadoop, they are optional: the correctness of the algorithm cannot depend on computation (or even execution) of the combiners
  - In Apache Spark, they're mostly automatic
- **Combiners I/O types**
  - Input: $(k_2, [v_2])$ [Same input as for Reducers]
  - Output: $[(k_2, v_2)]$ [Same output as for Mappers]
- **Commutative and Associative computations**
  - Reducer and Combiner code may be interchangeable (e.g. Word Count)
  - This is not true in the general case


A
"White snow"
B
"White hat"
C
"White gloves"
D
"Red hat"
E
"Red bat"
Mapper
Mapper
Mapper
hat 1
White 1
White 1
snow 1
gloves 1
White 1
bat 1
hat 1
Red 1
Red 1
Combiner
Combiner
Combiner
hat 1
White 2
snow 1
gloves 1
White 1
bat 1
hat 1
Red 2
Shuffle and sort: group values by key
bat 1
gloves 1
White 2 1
hat 1 1
Red 2
snow 1
Reducer
Reducer
bat 1
gloves 1
White 3
hat 2
Red 2
snow 1

Algorithmic Correctness: an Example

- **Problem statement**
  - 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
    - In practice: the dataset can be a log from a website, where the keys are user IDs and values are some measure of activity
- **Next, a baseline approach**
  - We use an **identity mapper**, which groups and sorts appropriately input key-value pairs
  - Reducers keep track of running sum and the number of integers encountered
  - The mean is emitted as the output of the reducer, with the input string as the key

Example: Computing the mean

class MAPPER
  method MAP(string t, integer r)
    EMIT(string t, integer r)

class REDUCER
  method REDUCE(string t, integers $[r_1, r_2, \ldots]$)
    $sum \leftarrow 0$
    $cnt \leftarrow 0$
    for all integer $r \in$ integers $[r_1, r_2, \ldots]$ **do**
      $sum \leftarrow sum + r$
      $cnt \leftarrow cnt + 1$
    $r_{avg} \leftarrow sum/cnt$
    EMIT(string t, integer r_{avg})

Algorithmic Correctness

- **Note: operations are not distributive**
  - `Mean`(1,2,3,4,5) $\neq$ `Mean`(`Mean`(1,2), `Mean`(3,4,5))
  - Hence: a combiner cannot output partial means and hope that the reducer will compute the correct final mean
- **Rule of thumb:**
  - Combiners are optimizations, the algorithm should work even when “removing” them

Example: Computing the mean with combiners

class MAPPER
    method MAP(string t, integer r)
        EMIT(string t, pair $(r, 1)$)

class COMBINER
    method COMBINE(string t, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ **do**
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        EMIT(string t, pair (sum, cnt))

class REDUCER
    method REDUCE(string t, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ **do**
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        $r_{avg} \leftarrow sum/cnt$
        EMIT(string t, integer r_{avg})

Algorithm design

Algorithm Design

- **Developing algorithms involve:**
  - Preparing the input data
  - Implement the mapper and the reducer
  - Optionally, design the combiner and the partitioner
- **How to recast existing algorithms in “Map Reduce”?**
  - It is not always obvious how to express algorithms
  - Data structures play an important role
  - Optimization is hard
- **Learn by examples**
  - “Design patterns”
  - “Shuffle” is perhaps the most tricky aspect

Algorithm Design

- **Aspects that are not under the control of the designer**
  - *Where* a mapper or reducer will run
  - *When* a mapper or 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

- **Aspects that can be controlled**
  - Construct data structures as keys and values
  - Execute user-specified initialization and termination code for mappers and reducers
  - Preserve state across multiple input and intermediate keys in mappers and reducers
  - Control the sort order of intermediate keys, and therefore the order in which a reducer will encounter particular keys
  - Control the partitioning of the key space, and therefore the set of keys that will be encountered by a particular reducer

Conclusions...

- **“Map Reduce” algorithms can be complex**
  - Hadoop MapReduce requires algorithm decomposition in several jobs
  - **Apache Spark is much simpler**
  - In general, iterative algorithms require a driver
  - Design patterns: `http://www.dcs.bbk.ac.uk/~dell/teaching/cc/book/ditp/ditp_ch3.pdf`