Introduction to Hadoop and MapReduce Algorithms

Giuseppe Fiameni
Giovanni Simonini

School on Scientific Data Analytics and Visualization
20-24 June 2016

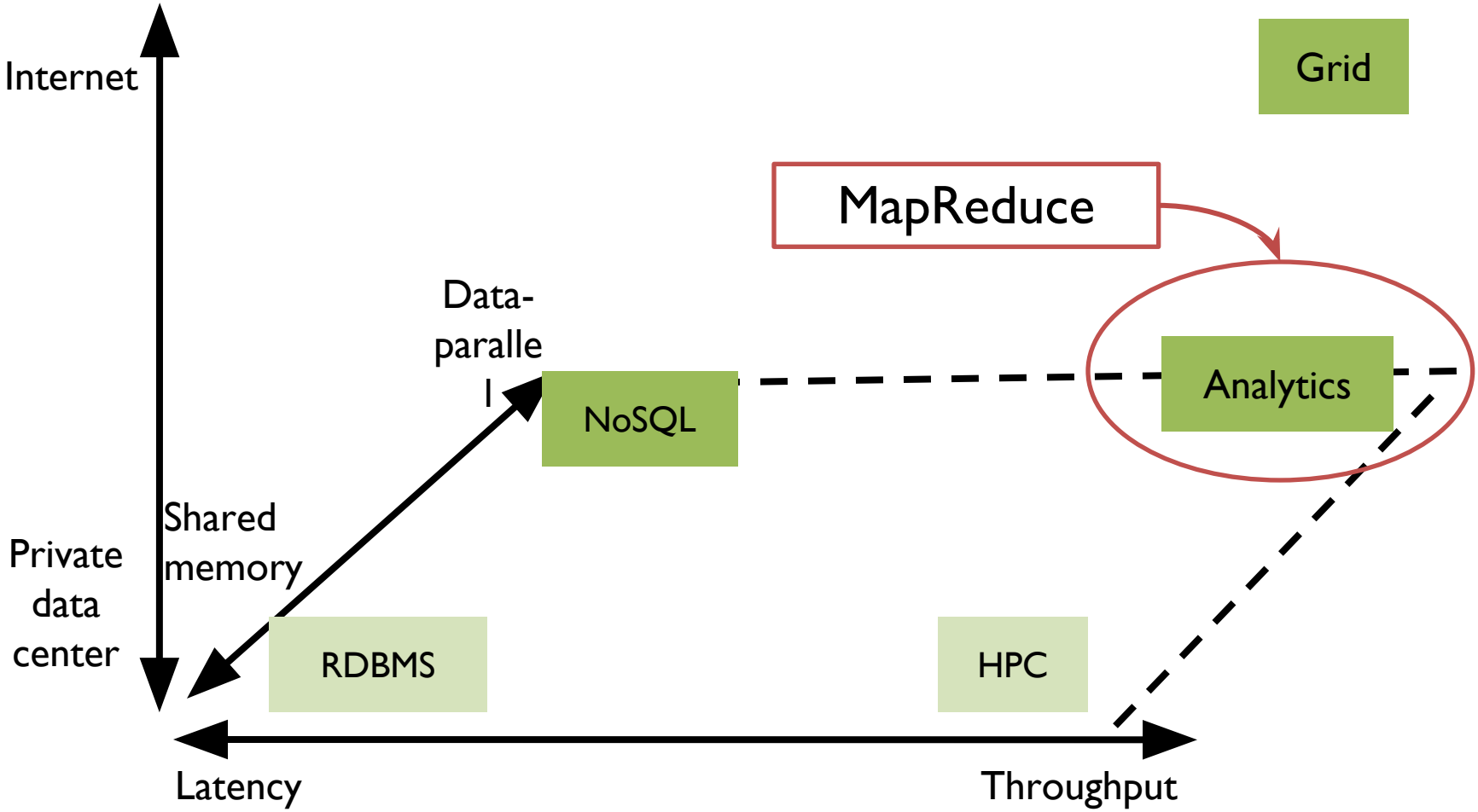
CINECA
Università di Modena e Reggio Emilia

Today

- **Introduction**:
  - Why a *functional programming approach* is needed in order to programming distributed parallel system
  - We need to know the architecture of the system to understand what we can do and what we can't

- **MapReduce/Hadoop**:
  - The Distributed File System
  - MapReduce
  - Some algorithms in MapReduce

Design Space



The Problem

- Programming distributed systems has always been very difficult task, needing specialized techniques and experts
- Moore's Law has held for over 40 years:
  - Processing power double every two years
  - Processing speed is no longer the problem
- Getting the data to the processor becomes the bottleneck
  - e.g.: Typical disk transfer rate: 75MB/sec
    Time taken to transfer 100GB of data to the processor: ~22minutes! (actual time is worse if servers have less than 100GB RAM)

"End of the Moore's low as we know it"

- Increasing performance **cannot** be achieved just through increasing hardware speed, new approach is needed
- Distributed computation **must be** exploited
  - Micro scale: **multicore processing**
  - Macro scale:**cloud computing / distributed data parallel systems**

Why so Hard?

One method to avoid ND output is to eliminate ND execution, for example by means of coordination method (e.g. locks)

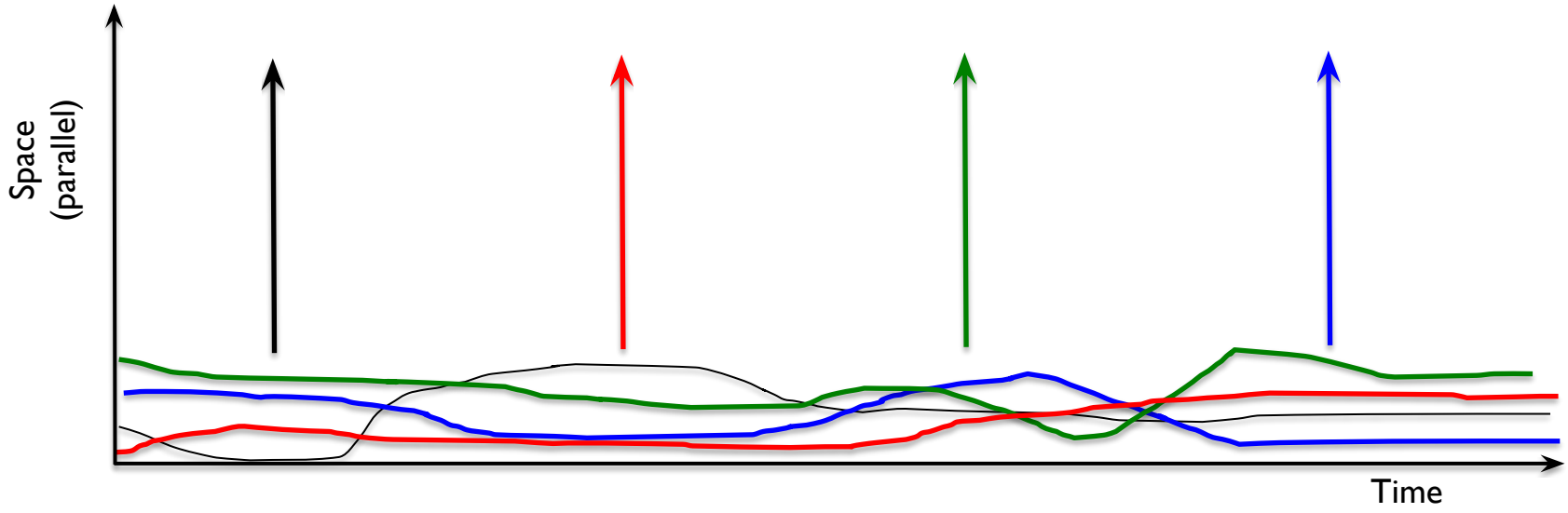
Non-deterministic Output = ~~**Non-deterministic Execution**~~ **+** **Mutable State**

```
var x = 0
async { x = x + 1}
async { x = x * 2}
```

Odersky M. : "*Working hard to keep it simple*". Keynote at OSCON '11.

Order is Hard, Go Parallel!

- To overcome the Von Neumann bottleneck, a different programming style must then be embraced: instead of specifying how the computation flow should proceed sequentially in time, programmers must be pushed to think more in space:
- Computation intended as a set of order-agnostic transformations applied in parallel to a collection of input data elements.
- The output is then a new set of elements which can be used as new building block for the successive computations.



Odersky M. : "*Working hard to keep it simple*". Keynote at OSCON '11.

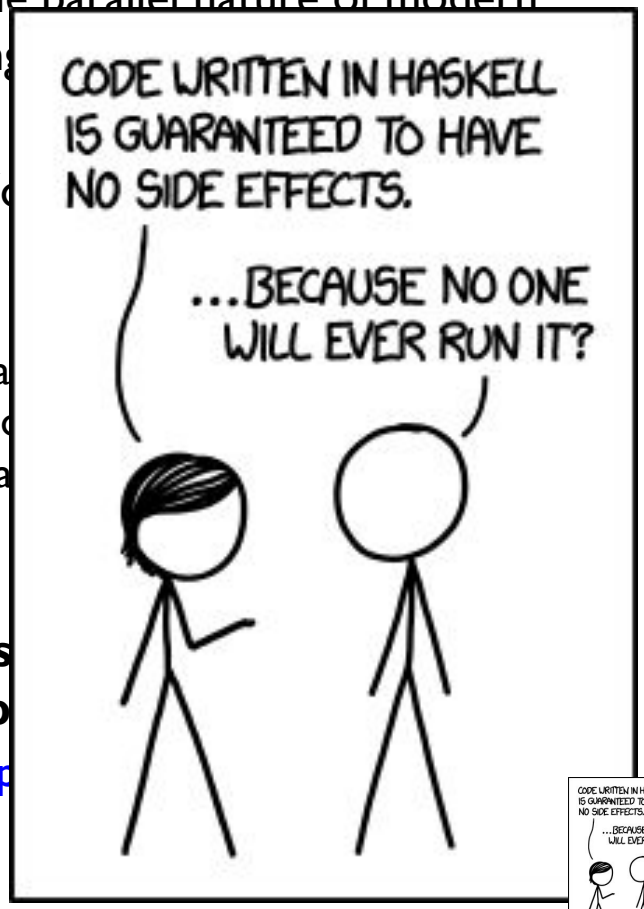
Functional Programming Approach

Non-deterministic Output = Non-deterministic Execution + ~~Mutable State~~

- State is immutable by default. As a consequence, the parallel nature of modern architectures can be fully exploited while maintainin

- Thanks to the immutability of states, not only the V avoided and parallel programming becomes natural, concerns can be easily addressed:
  - **if only deterministic operations** are considered, a **immutable states is logged**, every time a state is l can be recomputed starting from the previous state, a operations.

- **Due to the above features, we are not surpris parallel frameworks embracing a functional p**
  - **Exploited by modern data-parallel system "Map**
    - **mainly driven by industrial needs**

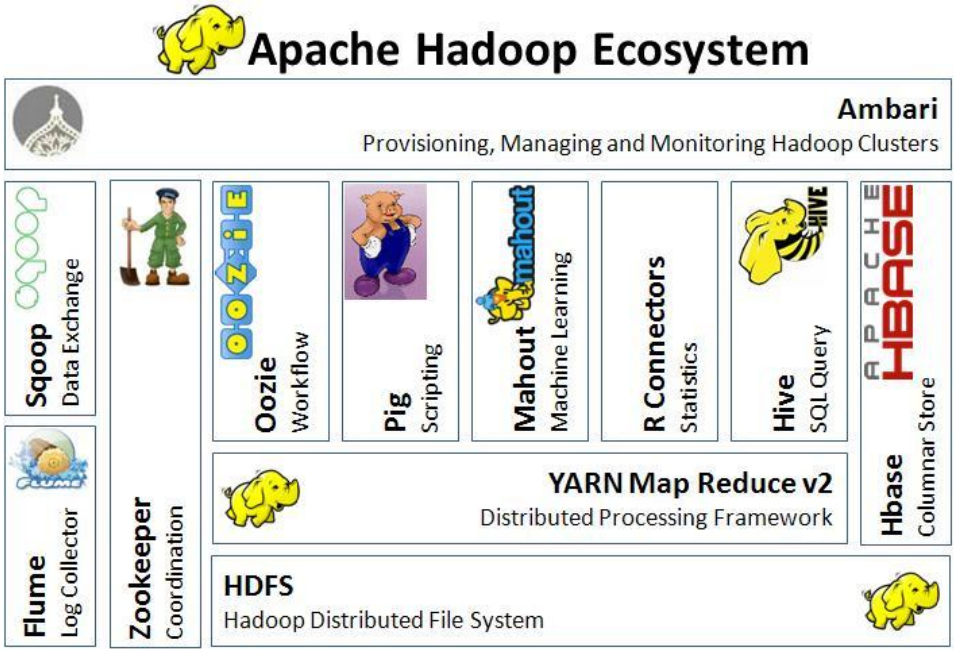


MapReduce

What is needed to implement a functional programming approach on a distributed system?

- Google File System (paper published in 2003)
- Google MapReduce (paper published in 2003 – implemented at Google in 2002)

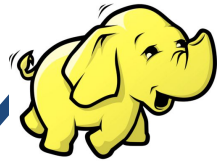
- Hadoop (2006-2008)
  - HDFS
  - MapReduce
  - A whole ecosystem



http://thebigdatablog.weebly.com/blog/the-hadoop-ecosystem-overview

- Google File System
- Design Assumption
- Architecture
- HDFS

HADOOP FILE SYSTEM



GFS Assumptions

- Goals (as previous distributed file systems):
  - performance, scalability, reliability, and availability

However, its design has been driven by key observations of particular application work-load and technological environment

- Design assumptions
  - Hardware failures are common (commodity machines)
    - If medium-time-between-failure is 1 year – Then 10000 servers have one failure / hour
  - Files are huge (GB) and their number is limited (millions, not billions)
  - Sequential writes: typically most files are mutated by appending new data rather than overwriting existing data
    - Random writes within a file are typically non-existent (possible, but not efficient)
  - Sequential reads: once written, the files are only read, and often only sequentially
    - Random modification in files possible, though not efficient
  - High sustained bandwidth rather than low latency
    - Batch processing

GFS: Files and Chunks

- Files are divided into fixed-size chunks
  - Size: typically 64/128 MB (modifiable parameter)
  - Files are replicated (by default 3 times, remember: fault-tolerance)
  - Advantages of (large) fixed-size chunks:
    - Disk seek time small compared to transfer time
    - A single file can be larger than a node's disk space
    - Fixed size makes allocation computations easy
  - Why not increase the chunk size further?
    Maps task operate on one chunk at a time .... the increasing of the chunk size decreases the parallelism (see MapReduce)

GFS: Master and ChunkServers

- Single Master maintains all file system metadata:
  - the namespace, access control information, the mapping from files to chunks, and the current locations of chunks
  - All metadata is kept in master's **memory** (fast random access)

- Multiple Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range.
  - chunkserver has the final word over what chunks it has

- Heartbeat messages between master and chunkservers
  - Is the chunkserver still alive? What chunks are stored at the chunkserver?

- Single Master can became the bottleneck
  - **HDFS Federation** in 2.X versions: several NameNodes share control (partition of filesystem namespace)

HDFS for programmer

- The File System (FS) shell includes various shell-like commands that directly interact with the Hadoop Distributed File System (HDFS)

- The FS shell (Unix-like) is invoked by:
  bin/hadoop fs <args>

- appendToFile
- cat
- chgrp
- chmod
- chown
- copyFromLocal
- copyToLocal
- count
- cp
- du
- dus
- expunge
- get
- getfacl
- getfattr
- getmerge
- ls
- lsr
- mkdir
- moveFromLocal
- moveToLocal
- mv
- put
- rm
- rmr
- setfacl
- setfattr
- setrep
- stat
- tail
- test
- text
- touchz

GFS	HDFS
Master	NameNode
Chunkserver	DataNode
Chunk	Block

http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html

- Data Model
- Architecture
- First Algorithms
- Advanced optimization

MAP-REDUCE

Map-Reduce

- Developed by Google and first presented in:
  - Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design \& Implementation - Volume 6* (OSDI'04), Vol. 6. USENIX Association, Berkeley, CA, USA, 10-10.

- Who use Map-Reduce? (Actually Hadoop, Map-Reduce open source implementation)
  - Amazon CloudSearch, Accela Communication, Adobe, AOL, adyard, Able Grape, Adknowledge, Aguja, Alibaba, AOL, ARA.COM.TR, Archive.is, Atbrox, BabaCar, Basenfasten, Benipal Technologies, Beebles, Bixo Labs, BrainPad, Brilig, Brockmann Consult GmbH, Caree.rs, CDU now!, Charleston, Cloudspace, Contestweb, Cooliris, Cornell University Web Lab, CRS4, crowdmedia, Datagraph, Dataium, Deepdyve, Detektei Berlin, Detikcom, devdaily.com, DropFire, eBay, eCircle, Enet, Enormo, Eyealike, Explore.To Yellow Pages, Facebook…
  - More at http://wiki.apache.org/hadoop/PoweredBy

MapReduce Data Model

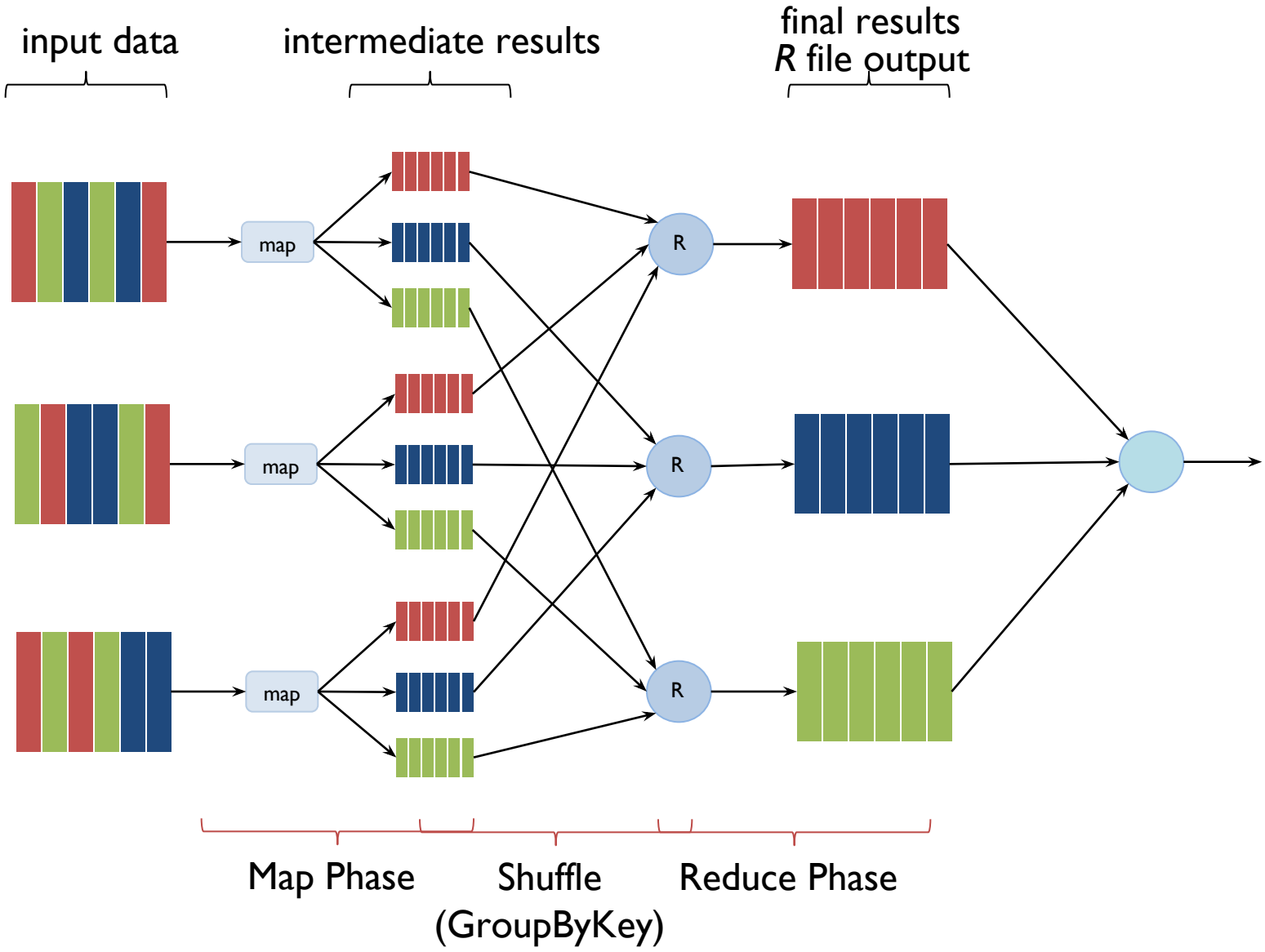
- MapReduce is an high-level programming model and implementation for large-scale parallel data processing.
- A MapReduce program consists of two functions (inspired by primitives of functional programming language):
  - MAP function:
    - Input: **(input key, value)**
    - Output: bag of **(intermediate key, value)**
  - REDUCE function:
    - Input: **(intermediate key, bag of values)**
    - Output: bag of output **(values)**

System executes the program in two steps:

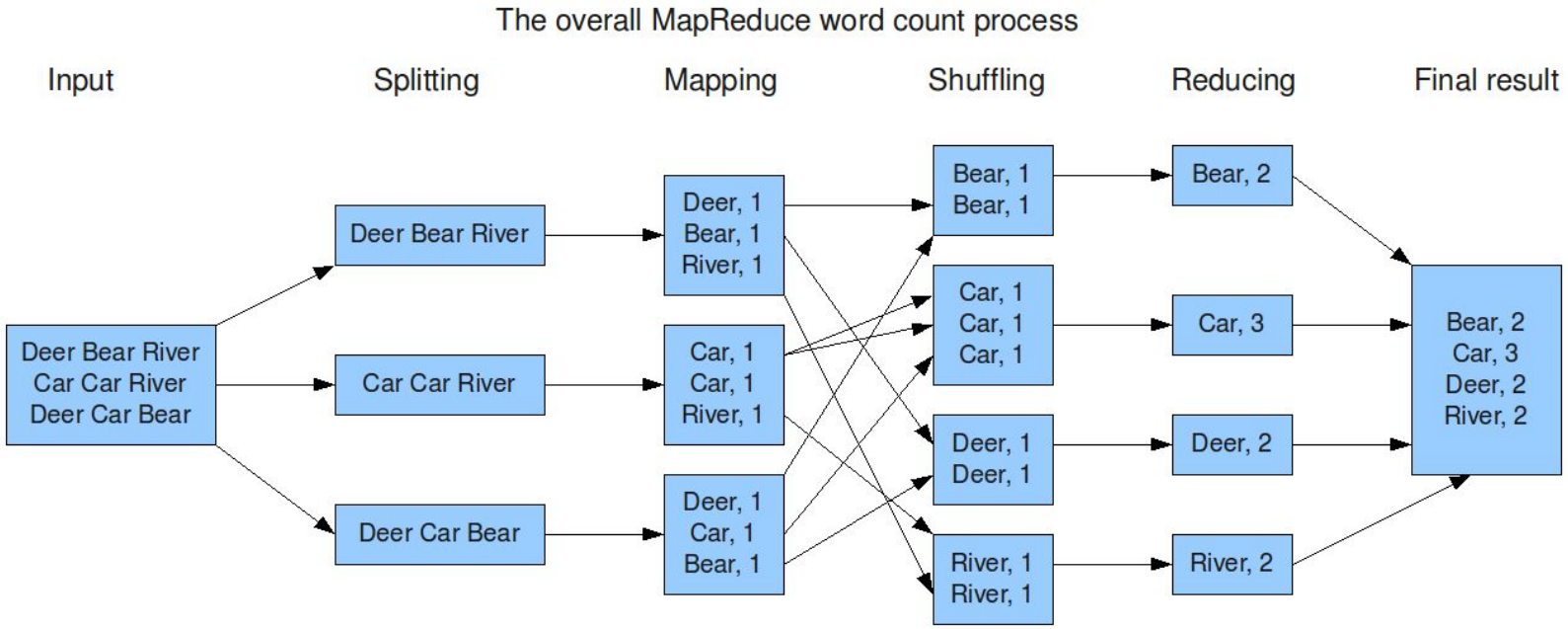
step 1) the map function is applied in parallel to all **(input key, value**) pairs in the input file

step 2) the system will group all pairs with the same intermediate key ("shuffle"), and passes the bag of values to the REDUCE function

Hadoop MapReduce

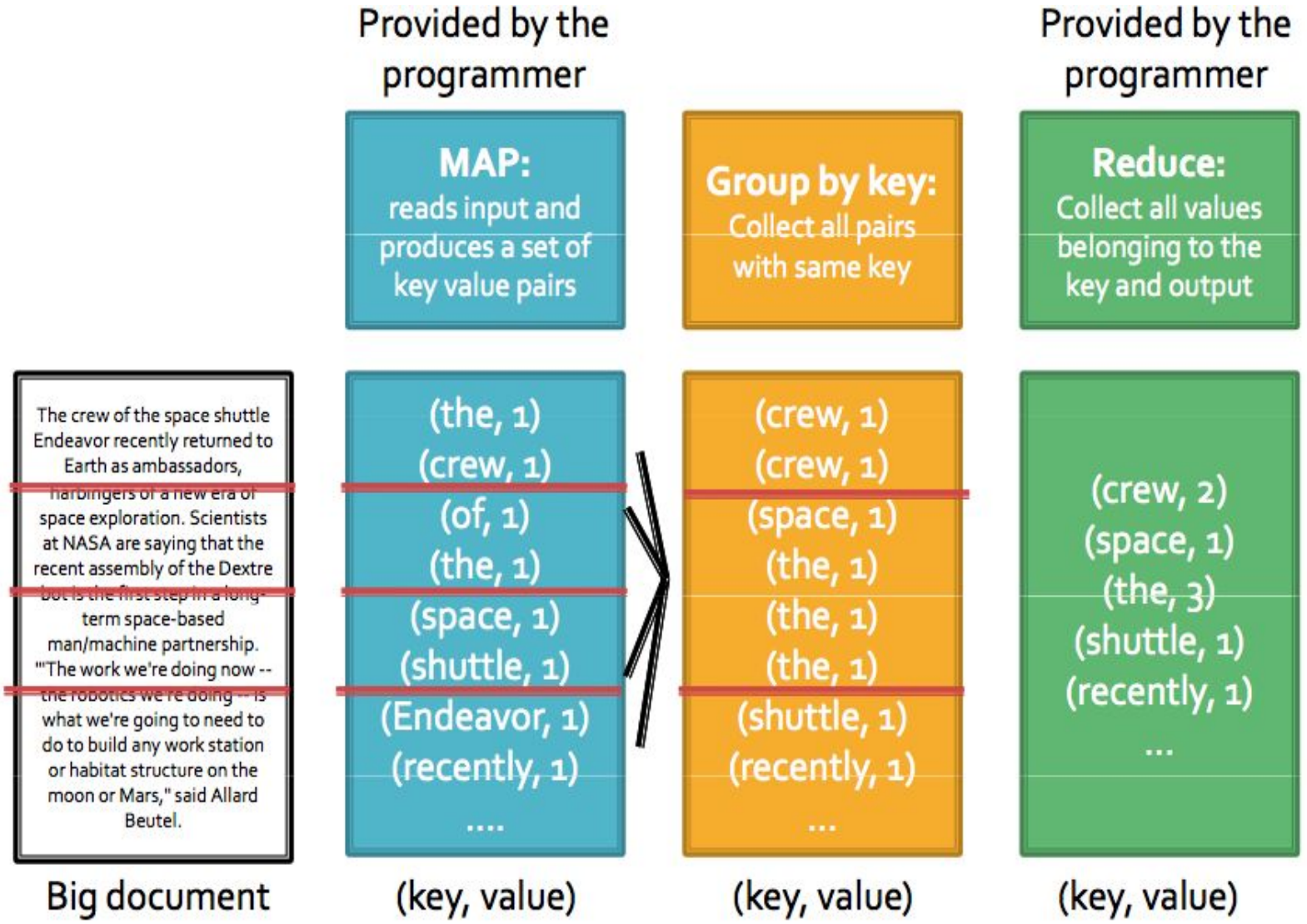


Word Count Example



http://blog.trifork.com//wp-content/uploads/2009/08/MapReduceWordCountOverview1.png

Word Count Example – Programmer point of view



Word Count Example – (Pseudo)Code

- Consider the problem of counting the number of occurrences of each word in a large collection of documents:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value: EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values: result += ParseInt(v);
  Emit(AsString(result));
```

- The map function emits each word plus an associated count of occurrences (just '1' in this simple example).
- The reduce function sums together all counts emitted for a particular word.

Hadoop Environment

Hadoop environment takes care of:

- **Partitioning** the input data
- **Scheduling** the program's execution across a set of machines
- **Performing the group by key** step
- **Handling node failures**
- **Managing** required inter-machine **communication**

Hadoop Daemons*:

Each daemon runs in its own Java Virtual Machine (JVM)

1. JobTracker
   - Manages MapReduce jobs, distribute individual tasks (map/reduce) to machines running the…

2. TaskTracker
   - Instantiates and monitors individual Map and Reduce tasks
   - When a TaskTracker receives a request to run a task, it instantiates a separate JVM for that task
     - Can run multiple tasks at the same time depending on the hardware resources

* For what concerns Map-Reduce "alone", in total they are five:
NameNode (HDFS), Secondary NameNode (HDFS - performs housekeeping to alleviate NameNode computations), DataNode (HDFS), JobTracker, and TaskTracker

JobTracker

- JobTracker takes care of:
  - task status: (idle, in-progress, completed)
  - scheduling idle tasks as resources (managed by taskTrackers) become available
  - gathering location and size of each intermediate file produced by the Map tasks
  - sending this info to the reducer tasks

- JobTracker pings taskTrackers periodically to detect failures:
  - if a Map failure occurs:
    - Map tasks completed or in-progress are reset to idle
    - Reduce tasks are notified when the map task is rescheduled on another taskTracker
  - if Reduce failure occurs:
    - Only in-progress tasks are reset to idle
  - JobTracker failure
    - MapReduce task is aborted and client is notified

Inside Map-Reduce

TaskTracker

- How to chose the number of Mappers and Reducers?
  - *M* map tasks, *R* reduce tasks
  - Rule of thumb:
    - Make *M* and *R* much larger than the number of nodes in cluster
      - One block (chunk) per map is common
      - Improves dynamic load balancing and speeds recovery from worker failure
  - Usually *R* is smaller than *M*, because output is spread across *R* files

MapReduce Data Locality

- Whenever possible, Hadoop will attempt to assign a Map task to a node working on a block of data stored locally (the chunk of file in HDFS)
- If this is not possible, the Map task will have to transfer the data across the network as it process that data
- Once the Map tasks have finished, data is then transferred across the network to the Reducers
  - Intermediate outputs of the Map tasks are written only on the local filesystem (on the node where it is running, not on HDFS); if the node fails, all computed data is lost, and the JobTracker reassign the computation to another worker.
  - Although the Reducers may run on the same physical machines as the Map tasks, there is no concept of data locality for the Reducers
    - All Mappers will, in general, have to communicate with all Reducers
- It appears that the shuffle and sort phase is a bottleneck:
  - The reduce method cannot start until all Mapper have finished
  - In practice, Hadoop will start to transfer data from Mappers to Reducers as the Mappers finish work

Combiners

- Often a map task will produce many pairs of the form (k,v1), (k,v2), … for the same key k (e.g. Word Count)
- Can save network time by pre-aggregating at mapper
  - combine(k1, list(v1)) → v2
  - Usually same as reduce function
  -
- Works only if reduce function is commutative and associative:
  - **Sum**
  - **Average**
    - if mapper emit (k, (partial_sum, num_of_instances_summed))
    - reduce: compute sum([partial_sum])/sum([num_of_instances_summed])
  - **Median**
    - not possible

- Create a monoid out of the intermediate value emitted by the mapper:
  - ***A monoid is an algebraic structure with a single associative binary operation and an identity element. As a simple example, the natural numbers form a monoid under addition with the identity element 0***

Partition Function

- Inputs to map tasks are created by contiguous splits of input file
- For reduce, we need to ensure that records with the same intermediate key end up at the same worker
- Hadoop uses a default partition function e.g., hash(key) mod R
- Sometimes useful to override
  - E.g., hash(hostname(URL)) mod R ensures URLs from a host end up in the same output file

- Custom Partitioners are also useful:
  - to avoid potential performance issues, redistributing the workload across Reducers
  - to perform Secondary Sort (allow to customize shuffle and sort)

Secondary Sort

E.g. Find the most frequent word starting with "a"

- How to perform secondary sort?
  - "natural key" vs "actual key":

    e.g. (key ='a#23', value='apple')
    (key ='a#9', value='airplane')
    (key ='a#22', value='air')

    - *custom partitioner*

      "group by" performed on a sub-set of the key

      e.g. all key starting with 'a' are sent to the same reducer

    - *custom comparator*

      record ordered according to a custom function

      e.g. sort by the second half of the key

Example: Inverted Index

Given a collection of textual documents, how to create an inverted index?

input:

tweet_01 "apple computers are …"

tweet_02 "I an apple today …"

tweet_03 "todays computers are … "

desired output:

"apple", (tweet_01, tweet_02)

"computers", (tweet_01, tweet_03)

"todays", (tweet_02, tweet_03)

…

```
Map(k,val):
    for word in val:
        emit(w,k)
```

```
Reduce(k,values):
    emit(k, set(values))
```

Join

Order			
type	orderid	account	date
ord	001	john	14-12
ord	002	sim	13-12
ord	003	mary	09-12

LineItem			
type	orderid	itemid	qty
line	001	i1	3
line	001	i2	2
line	002	i1	5
line	002	i3	2
line	003	i2	3

desired output:

001, john, 14-12, i1, 3
001, john, 14-12, i2, 2
002, sim, 13-12, i1, 5
002, sim, 13-12, i3, 2
003, mary, 09-12, i2, 3

Reduce-side Join

```
Map(k,val):
    orderid = val[1]
    emit(orderid, val)
```

```
Reduce(k,values):
    lines = []
    for val in values:
        type = val[0]
        if type == 'ord':
            order = val
        # if val[0] == 'line'
        else:
            lines.append(val)
    for line in lines:
        emit(order + line)
```

References

- Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." ACM SIGOPS Operating Systems Review. Vol. 37. No. 5. ACM, 2003.
- Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.
- Lam, Chuck. Hadoop in action. Manning Publications Co., 2010.
- Rajaraman, Anand, and Jeffrey David Ullman. Mining of massive datasets. Cambridge University Press, 2011.
- http://hadoop.apache.org/
- http://www.cloudera.com/content/cloudera/en/about/hadoop-and-big-data.html
- http://www.st.ewi.tudelft.nl/~hauff/BDP-Lectures/5_filesystem.pdf
- https://www.coursera.org/course/datasci
- https://www.coursera.org/course/mmds
- https://www.coursera.org/course/bigdata