Big data: architectures and data analytics

Motivations of Hadoop and MapReduce

Data volumes

- The amount of data increases every day
- Some numbers (~ 2012):
  - Data processed by Google every day: 100+ PB
  - Data processed by Facebook every day: 10+ PB
- To analyze them, systems that scale with respect to the data volume are needed

3

Data volumes: Google Example

- Analyze 10 billion web pages
- Average size of a webpage: 20KB
- Size of the collection: 10 billion x 20KB = 200TB
- Hard disk read bandwidth: 100MB/sec
- Time needed to read all web pages (without analyzing them): 2 million seconds = more than **24 days**
- A single node architecture is not adequate

4

Failures

- Failures are part of everyday life, especially in data center
  - A single server stays up for 3 years (~1000 days)
    - 10 servers → 1 failure every 100 days (~3 months)
    - 100 servers → 1 failure every 10 days
    - 1000 servers → 1 failure/day
- Sources of failures
  - Hardware/Software
  - Electrical, Cooling, ...
  - Unavailability of a resource due to overload

5

Failures

- LALN data [DSN 2006]
  - Data for 5000 machines, for 9 years
  - Hardware failures: 60%, Software: 20%, Network 5%
- DRAM error analysis [Sigmetrics 2009]
  - Data for 2.5 years
  - 8% of DIMMs affected by errors
- Disk drive failure analysis [FAST 2007]
  - Utilization and temperature major causes of failures

6

Failures

- Failure types
  - Permanent
    - E.g., Broken motherboard
  - Transient
    - E.g., Unavailability of a resource due to overload

7

Network bandwidth

- Network becomes the bottleneck if big amounts of data need to be exchanged between nodes/servers
  - Network bandwidth: 1Gbps
  - Moving 10 TB from one server to another takes 1 day
    → Data should be moved across nodes only when it is indispensable
  - Usually, codes/programs are small (few MBs)
    → Move code/program and computation to data

8

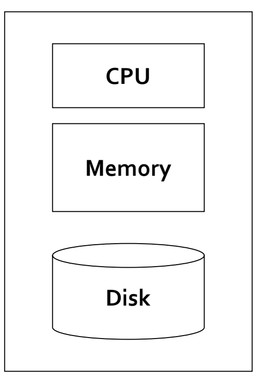
Network bandwidth

- Network becomes the bottleneck if big amounts of data need to be exchanged between nodes/servers
  - Network bandwidth: 1Gbps
  - Moving 10 TB from one server to another takes 1 day
    → Data should be moved across nodes only when it is indispensable
  - Usually, codes/programs are small (few MBs)
    → Move code/program and computation to data

Data locality

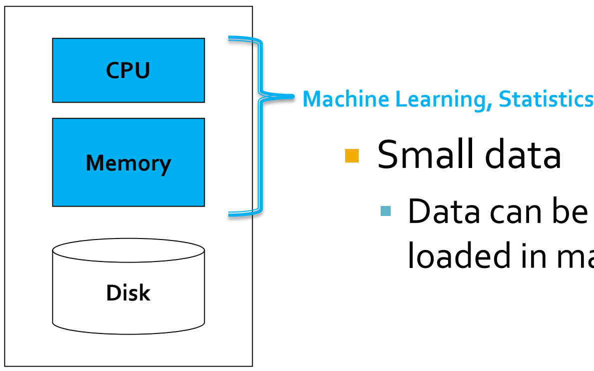
Single-node architecture

Server (Single node)



Single-node architecture

Server (Single node)

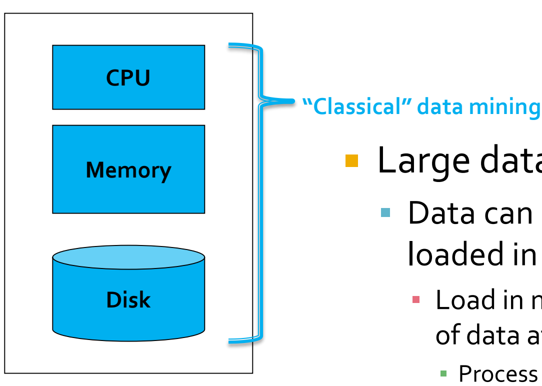


- Small data
  - Data can be completely loaded in main memory

11

Single-node architecture

Server (Single node)



- Large data
  - Data can not be completely loaded in main memory
    - Load in main memory one chunk of data at a time
      - Process it and store some statistics
    - Combine statistics to compute the final result

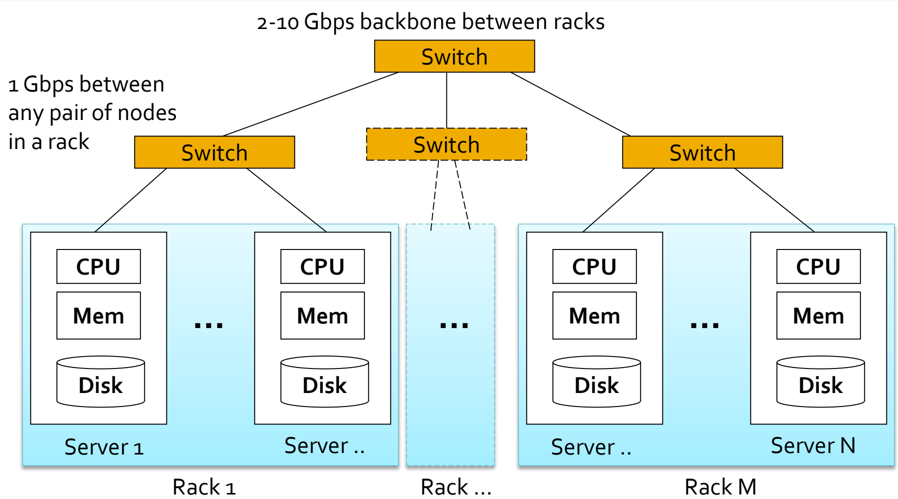
12

Cluster Architecture

- Cluster of servers (data center)
  - Computation is distributed across servers
  - Data are stored/distributed across servers
- Standard architecture in the Big data context
  - Cluster of commodity Linux nodes/servers
    - 32 GB of main memory per node
  - Gigabit Ethernet interconnection

13

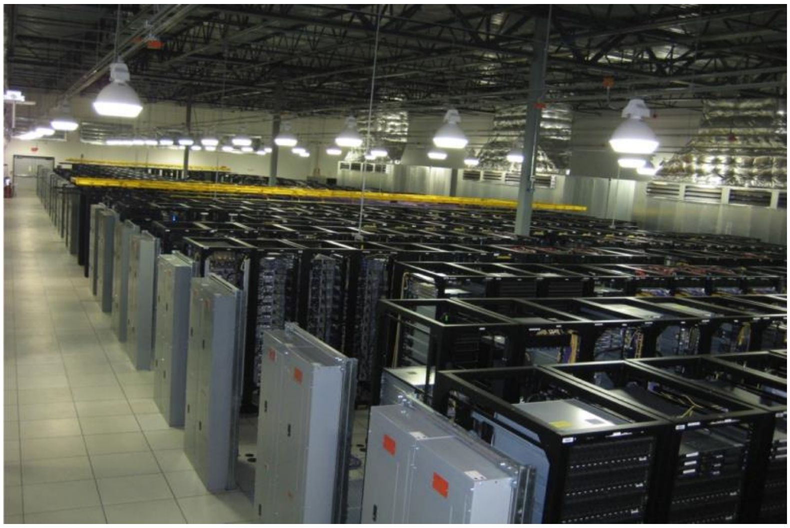
Commodity Cluster Architecture



Each rack contains 16-64 nodes

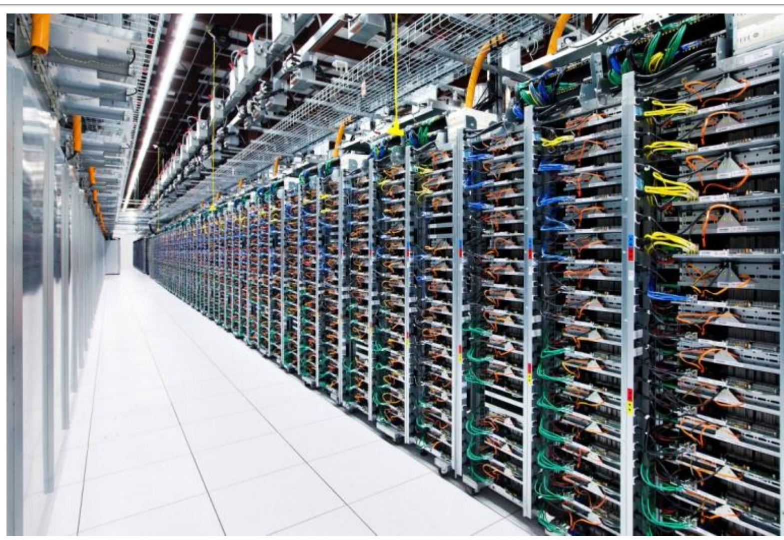
14

Data center



15

Data center



16

Scalability

- Current systems must scale to address
  - The increasing amount of data to analyze
  - The increasing number of users to serve
  - The increasing complexity of the problems
- Two approaches are usually used to address scalability issues
  - Vertical scalability (scale up)
  - Horizontal scalability (scale out)

Scale up vs. Scale out

- Vertical scalability (scale up)
  - Add more power/resources (main memory, CPUs) to a single node (high-performing server)
    - Cost of super-computers is not linear with respect to their resources
- Horizontal scalability (scale out)
  - Add more nodes (commodity servers) to a system
    - The cost scales approximately linearly with respect to the number of added nodes
    - But data center efficiency is a difficult problem to solve

Scale up vs. Scale out

- For data-intensive workloads, a large number of commodity servers is preferred over a small number of high-performing servers
  - At the same cost, we can deploy a system that processes data more efficiently and is more fault-tolerant
- Horizontal scalability (scale out) is preferred for big data applications
  - But distributed computing is hard

    →New systems hiding the complexity of the distributed part of the problem to developers are needed

19

Cluster computing challenges

- Distributed programming is hard
  - Problem decomposition and parallelization
  - Task synchronization
- Task scheduling of distributed applications is critical
  - Assign tasks to nodes by trying to
    - Speed up the execution of the application
    - Exploit (almost) all the available resources
    - Reduce the impact of node failures

20

Cluster computing challenges

- Distributed data storage
  - How do we store data persistently on disk and keep it available if nodes can fail?
    - Redundancy is the solution, but it increases the complexity of the system
- Network bottleneck
  - Reduce the amount of data send through the network
    - Move computation (and code) to data

21

Cluster computing challenges

- Distributed computing is not a new topic
  - HPC (High-performance computing) ~1960
  - Grid computing ~1990
  - Distributed databases ~1990
- Hence, many solutions to the mentioned challenges are already available
- But we are now facing big data driven-problems

  → The former solutions are not adequate to address big data volumes

22

Typical Big Data Problem

- Typical Big Data Problem
  - Iterate over a large number of records/objects
  - Extract something of interest from each
  - Aggregate intermediate results
  - Generate final output
- The challenges:
  - Parallelization
  - Distributed storage of large data sets (Terabytes, Petabytes)
  - Node Failure management
  - Network bottleneck
  - Diverse input format (data diversity & heterogeneity)

23

Apache Hadoop

Apache Hadoop

- Scalable fault-tolerant distributed system for Big Data
  - Distributed Data Storage
  - Distributed Data Processing
  - Borrowed concepts/ideas from the systems designed at Google (Google File System for Google's MapReduce)
  - Open source project under the Apache license
    - But there are also many commercial implementations (e.g., Cloudera, Hortonworks, MapR)

25

Hadoop History

- Dec 2004 – Google published a paper about GFS
- July 2005 – Nutch uses MapReduce
- Feb 2006 – Hadoop becomes a Lucene subproject
- Apr 2007 – Yahoo! runs it on a 1000-node cluster
- Jan 2008 – Hadoop becomes an Apache Top Level Project
- Jul 2008 – Hadoop is tested on a 4000 node cluster

26

Hadoop History

- Feb 2009 – The Yahoo! Search Webmap is a Hadoop application that runs on more than 10,000 core Linux cluster
- June 2009 – Yahoo! made available the source code of its production version of Hadoop
- In 2010 Facebook claimed that they have the largest Hadoop cluster in the world with 21 PB of storage
  - On July 27, 2011 they announced the data has grown to 30 PB.

27

Who uses Hadoop?

- Amazon
- Facebook
- Google
- IBM
- Joost
- Last.fm
- New York Times
- PowerSet
- Veoh
- Yahoo!
- .....

28

Hadoop vs. HPC

- Hadoop
  - Designed for Data intensive workloads
  - Usually, no CPU demanding/intensive tasks
- HPC (High-performance computing)
  - A supercomputer with a high-level computational capacity
    - Performance of a supercomputer is measured in floating-point operations per second (FLOPS)
  - Designed for CPU intensive tasks
  - Usually it is used to process "small" data sets

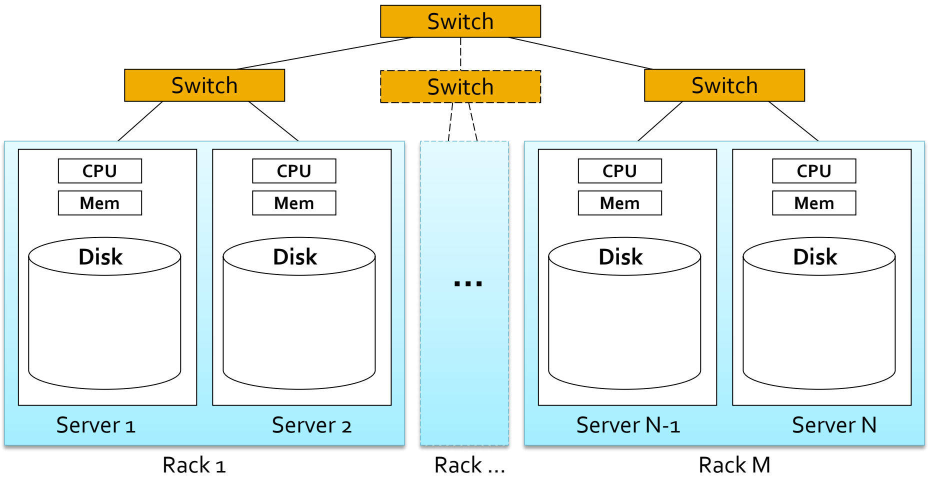
29

Hadoop: main components

- Core components of Hadoop:
  - Distributed Big Data Processing Infrastructure based on the MapReduce programming paradigm
    - Provides a high-level abstraction view
      - Programmers do not need to care about task scheduling and synchronization
    - Fault-tolerant
      - Node and task failures are automatically managed by the Hadoop system
  - HDFS (Hadoop Distributed File System)
    - Fault-tolerant
    - High availability distributed storage

30

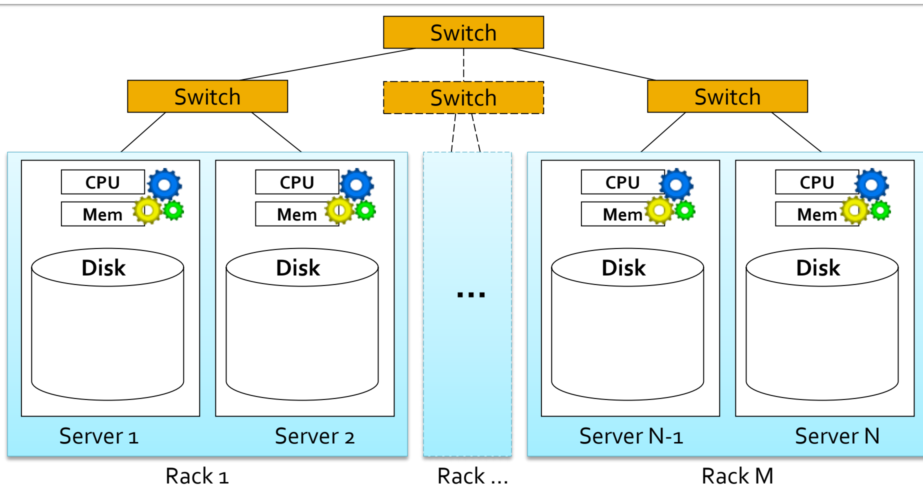
Hadoop: main components



Example with number of replicas per chunk = 2

31

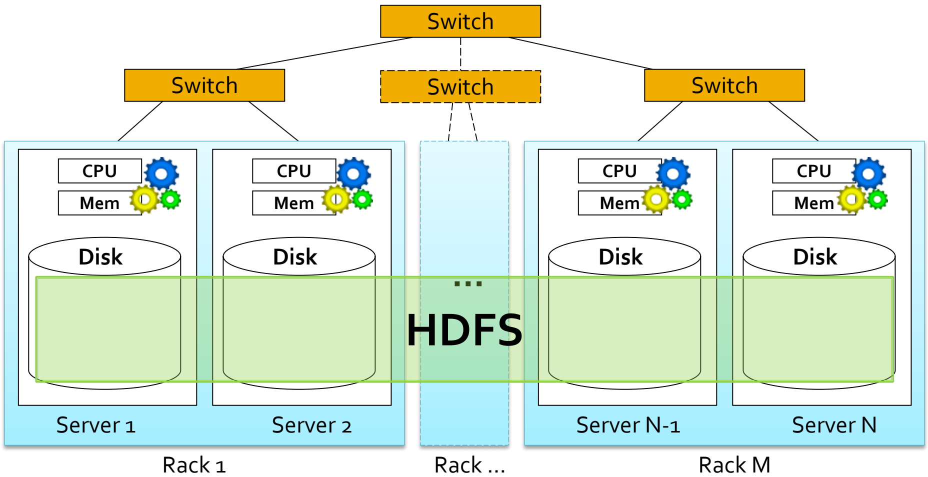
Hadoop: main components



Example with number of replicas per chunk = 2

32

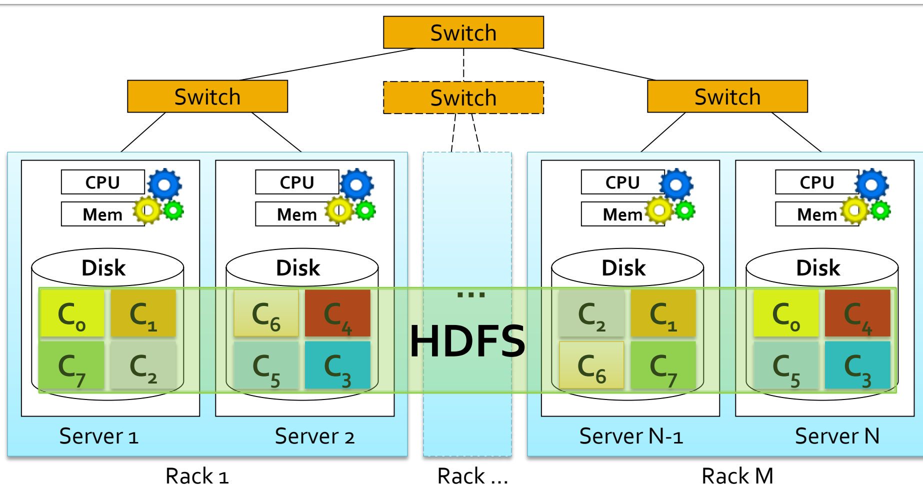
Hadoop: main components



Example with number of replicas per chunk = 2

33

Hadoop: main components



Example with number of replicas per chunk = 2

34

Distributed Big Data Processing Infrastructure

- Separates the what from the how
  - Hadoop programs are based on the MapReduce programming paradigm
  - MapReduce abstracts away the "distributed" part of the problem (scheduling, synchronization, etc)
    - Programmers focus on what
  - The distributed part (scheduling, synchronization, etc) of the problem is handled by the framework
    - The Hadoop infrastructure focuses on how

35

Distributed Big Data Processing Infrastructure

- But an in-depth knowledge of the Hadoop framework is important to develop efficient applications
  - The design of the application must exploit data locality and limit network usage/data sharing

36

HDFS

- HDFS
  - Standard Apache Hadoop distributed file system
  - Provides global file namespace
  - Stores data redundantly on multiple nodes to provide persistence and availability
    - Fault-tolerant file system
- Typical usage pattern
  - Huge files (GB to TB)
  - Data is rarely updated
  - Reads and appends are common
    - Usually, random read/write operations are not performed

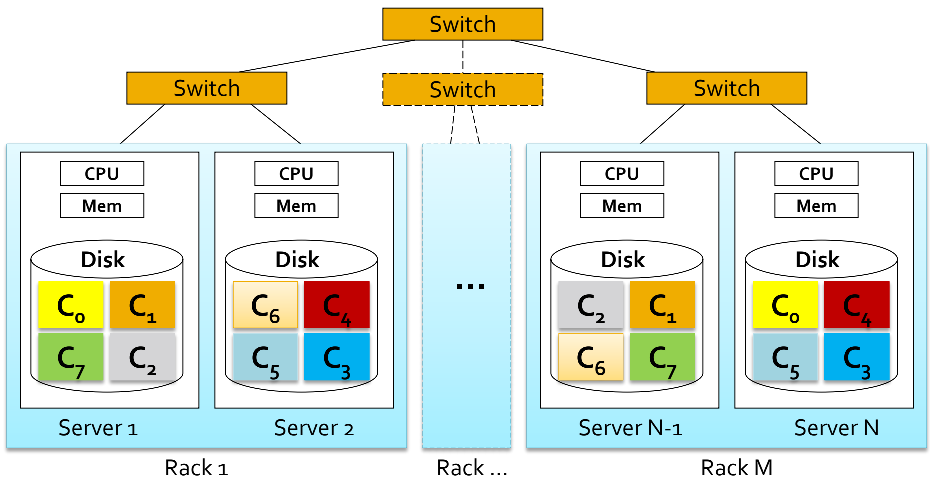
37

HDFS

- Each file is split in "chunks" that are spread across the servers
  - Each chuck is replicated on different servers (usually there are 3 replicas per chuck)
    - Ensures persistence and availability
    - To increase persistence and availability, replicas are stored in different racks, if it is possible
  - Typically each chunk is 64-128MB

38

HDFS



Example with number of replicas per chunk = 2

39

HDFS

- The Master node, a.k.a. Name Nodes in HDFS, is a special node/server that
  - Stores HDFS metadata
    - E.g., the mapping between the name of a file and the location of its chunks
  - Might be replicated
- Client applications: file access through HDFS APIs
  - Talk to the master node to find data/chuck servers associated with the file of interest
  - Connect to the selected chunk servers to access data

40

Hadoop ecosystem

- Many Hadoop-related projects/systems are available
  - Pig
    - A data flow language and execution environment, based on MapReduce, for exploring very large datasets
  - Hive
    - A distributed data warehouse, based on MapReduce, for querying data stored in HDFS by means of a query language based on SQL
  - HBase
    - A distributed column-oriented database. HBase uses HDFS for storing data

41

Hadoop ecosystem

  - Sqoop
    - A tool for efficiently moving data between relational databases and HDFS
  - ZooKeeper
    - A distributed coordination service. It provides primitives such as distributed locks
  - ....
- Each project/system addresses one specific class of problems

42

Principles of Hadoop and MapReduce

Warm up: Word Count

- Input
  - A large textual file of words
- Problem
  - Count the number of times each distinct word appears in the file
- Output
  - A list of pairs <word, number of occurrences in the input file>

Word Count

- Case 1: Entire file fits in main memory

Word Count

- Case 1: Entire file fits in main memory
  - A traditional single node approach is probably the most efficient solution in this case
    - The complexity and overheads of a distributed system impact negatively on the performance when files of few GBs are analyzed

Word Count

- Case 1: Entire file fits in main memory
  - A traditional single node approach is probably the most efficient solution in this case
    - The complexity and overheads of a distributed system impact negatively on the performance when files of few GBs are analyzed
- Case 2: File too large to fit in main memory

Word Count

- Case 1: Entire file fits in main memory
  - A traditional single node approach is probably the most efficient solution in this case
    - The complexity and overheads of a distributed system impact negatively on the performance when files of few GBs are analyzed
- Case 2: File too large to fit in main memory

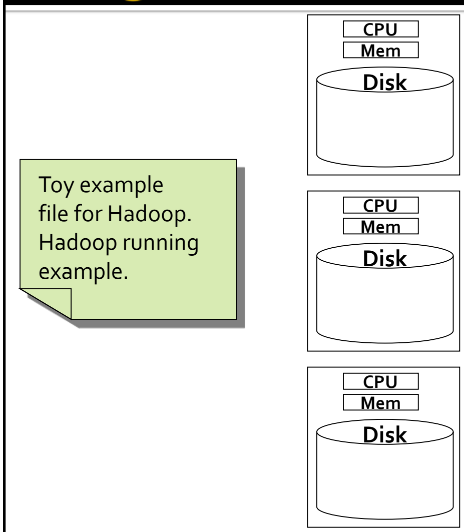
  - How can we split this problem in a set of (almost) independent sub-tasks and execute it on a cluster of servers?

Word Count: Toy example (Case 2: large file)

- Suppose that
  - The cluster has 3 servers
  - The content of the input file is
    - "Toy example file for Hadoop. Hadoop running example."
  - The input file is split in two chunks (number of replicas=1)

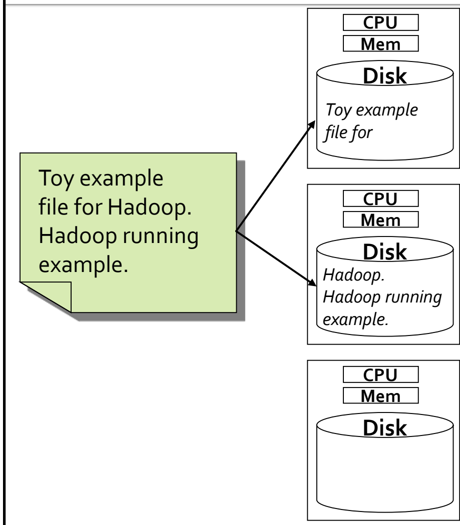
49

Word Count: Toy example (Case 2: large file)



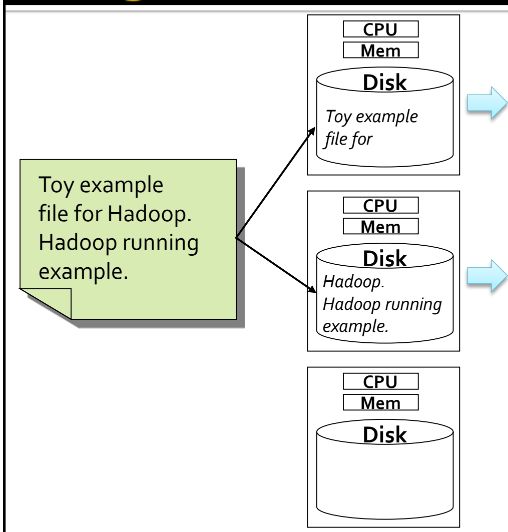
50

Word Count: Toy example (Case 2: large file)



51

Word Count: Toy example (Case 2: large file)



<toy, 1>
<example, 1>
<file, 1>
<for, 1>

<hadoop, 2>
<running, 1>
<example, 1>

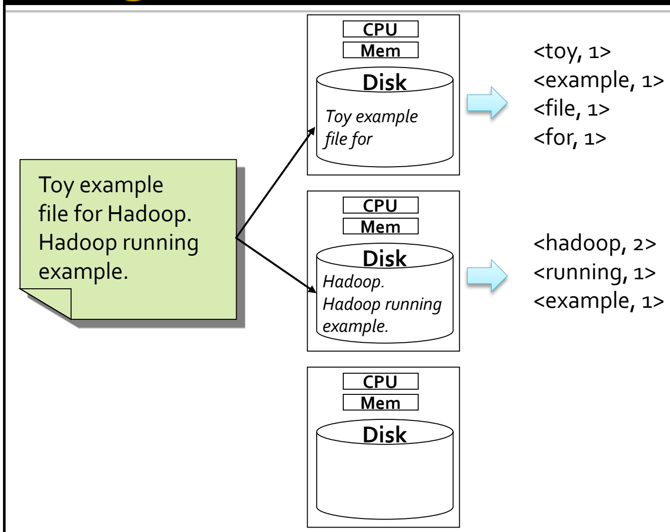
52

Word Count: Toy example (Case 2: large file)

- The problem can be easily parallelized
  1. Each server processes its chunk of data and counts the number of times each word appears in its chunk
     - Each server can perform it independently →Synchronization is not needed in this phase

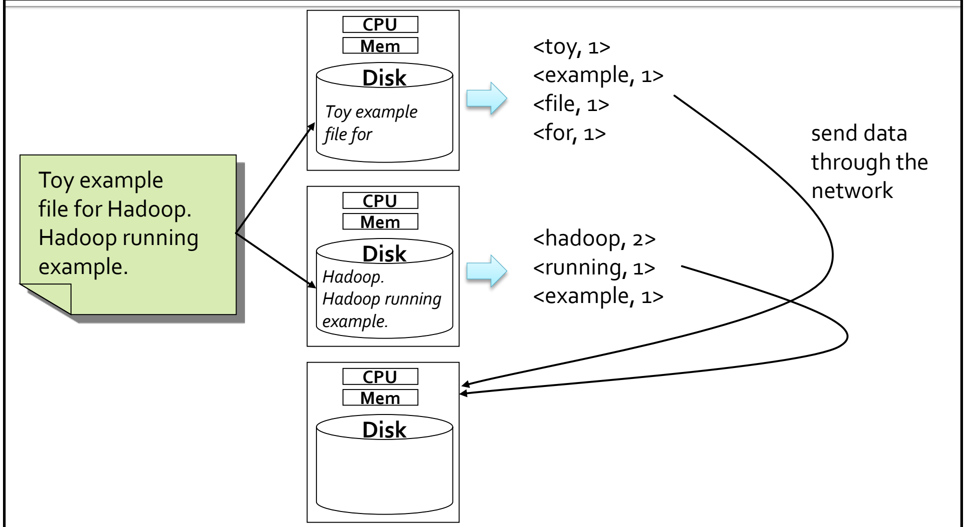
53

Word Count: Toy example (Case 2: large file)



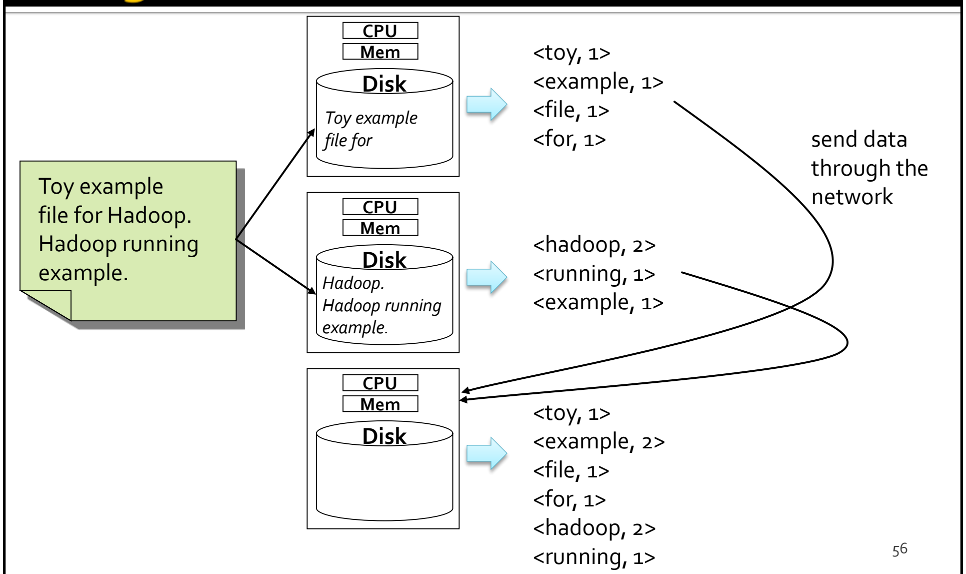
54

Word Count: Toy example (Case 2: large file)



55

Word Count: Toy example (Case 2: large file)



56

Word Count: Toy example (Case 2: large file)

2. Each server sends its local (partial) list of pairs <word, number of occurrences in its chunk> to a server that is in charge of aggregating local results and computing the global list/global result
   - The server in charge of computing the global result needs to receive all the local (partial) results to compute and emit the final list

     → A simple synchronization operation is needed in this phase

Word Count: a more realistic case

- Case 2: File too large to fit in main memory
- Suppose that
  - The file size is 100 GB and the number of distinct words occurring in it is at most 1,000
  - The cluster has 101 servers
  - The file is spread across 100 servers and each of these servers contains one (different) chunk of the input file
    - i.e., the file is optimally spread across 100 servers (each server contains 1/100 of the file in its local hard drives)

Word Count: complexity

- Each server reads 1 GB of data from its local hard drive (it reads one chunk from HDFS)
  - Few seconds
- Each local list is composed of at most 1,000 pairs (because the number of distinct words is 1,000)
  - Few MBs
- The maximum amount of data sent on the network is 100 x size of local list (number of servers x local list size)
  - Some MBs

Word Count: scalability

- We can define scalability along two dimensions
  - In terms of data:
    - Given twice the amount of data, the word count algorithm takes approximately no more than twice as long to run
      - Each server processes 2 x data => 2 x execution time to compute local list
  - In terms of resources
    - Given twice the number of servers, the word count algorithm takes approximately no more than half as long to run
      - Each server processes ½ x data => ½ x execution time to compute local list

60

Word Count: scalability

- The time needed to send local results to the node in charge of computing the final result and the computation of the final result are considered negligible in this running example
- Frequently, this assumption is not true
  - It depends
    - on the complexity of the problem
    - on the ability of the developer to limit the amount of data sent on the network

61

Key Ideas of Hadoop and MapReduce

- Scale "out", not "up"
  - Increase the number of servers and not the resources of the already available ones
- Move processing to data
  - The network has a limited bandwidth
- Process data sequentially, avoid random access
  - Seek operations are expensive
  - Big data applications usually read and analyze all records/objects
    - Random access is useless

62

Data locality

- Traditional distributed systems (e.g., HPC) move data to computing nodes (servers)
  - This approach cannot be used to process TBs of data
    - The network bandwidth is limited
- Hadoop moves code to data
  - Code (few KB) is copied and executed on the servers that contain the chucks of the data of interest
  - This approach is based on "**data locality**"

63

Hadoop and MapReduce

- Hadoop/MapReduce is designed for
  - Batch processing involving (mostly) full scans of the data
  - Data-intensive applications
    - Read and process the whole Web (e.g., PageRank computation)
    - Read and process the whole Social Graph (e.g., LinkPrediction, a.k.a. "friend suggest")
    - Log analysis (e.g., Network traces, Smart-meter data, ..)

64

Hadoop and MapReduce

- Hadoop/MapReduce is not the panacea for all big data problems
- Hadoop/MapReduce does not feet well
  - Iterative problems
  - Recursive problems
  - Stream data processing

65

The MapReduce Programming Paradigm

66

MapReduce and Functional programming

- The MapReduce programming paradigm is based on the basic concepts of Functional programming
- MapReduce "implements" a subset of functional programming
  - The programming model appears quite limited and strict
    - Everything is based on two "functions", complaint with specific interfaces, defined by the developer

67

What can we do with MapReduce?

- Solving complex problems is difficult
- However, there are several important problems that can be adapted to MapReduce
  - Log analysis
  - PageRank computation
  - Social graph analysis
  - Sensor data analysis
  - Smart-city data analysis
  - Network capture analysis

68

Building blocks: Map and Reduce

- MapReduce is based on two main "building blocks"
  - **Map** and **Reduce** functions
- Map function
  - It is **applied over each element** of an input data set and **emits** a set of **(key, value) pairs**
- Reduce function
  - It is **applied over each set of (key, value) pairs** (emitted by the map function) **with the same key** and **emits** a set of **(key, value) pairs** → Final result

69

Word count running example

- Input
  - A textual file (i.e., a list of words)
- Problem
  - Count the number of times each distinct word appears in the file
- Output
  - A list of pairs <word, number of occurrences in the input file>

70

Word count running example

- The input textual file is considered as a list of words L

Word count running example

L = [toy, example, toy, example , hadoop]

[…] denotes a list. (k, v) denotes a key-value pair.

Word count running example

L = [toy, example, toy, example , hadoop]

Apply a function on each element

L_m =[(toy, +1), (example, +1), (toy, +1), (example, +1), (hadoop, +1)]

[...] denotes a set. (k, v) denotes a key-value pair.

73

Word count running example

L = [toy, example, toy, example , hadoop]

L_m =[(toy, +1), (example, +1), (toy, +1), (example, +1), (hadoop, +1)]

Group by key

(toy, [+1, +1]) (example, [+1, +1]) (hadoop, [+1])

[...] denotes a list. (k, v) denotes a key-value pair.

74

Word count running example

L = [toy, example, toy, example , hadoop]

L_m =[(toy, +1), (example, +1), (toy, +1), (example, +1), (hadoop, +1)]

(toy, [+1, +1]) (example, [+1, +1]) (hadoop, [+1])

Apply a function on each group

[(toy, 2) , (example, 2), (hadoop, 1)]

[...] denotes a list. (k, v) denotes a key-value pair.

75

Word count running example

L = [toy, example, toy, example , hadoop]

Map phase

L_m =[(toy, +1), (example, +1), (toy, +1), (example, +1), (hadoop, +1)]

Shuffle and Sort phase

(toy, [+1, +1]) (example, [+1, +1]) (hadoop, [+1])

Reduce phase

[(toy, 2) , (example, 2), (hadoop, 1)]

[...] denotes a list. (k, v) denotes a key-value pair.

76

Word count running example

- The input textual file is considered as a list of words L

Word count running example

- The input textual file is considered as a list of words L
- A key-value pair (w, 1) is emitted for each word w in L
  - i.e., the **map** function is m(w) = (w, 1)
  - A new list of (key, value) pairs L_m is generated

Word count running example

- The key-value pairs in L_m are aggregated by key (i.e., by word in our example)
  - One group G_w is generated for each word
  - Each group G_w is a key-list pair (w, [list of values]) where [list of values] contains all the values of the pairs associated with the word w
    - i.e., [list of values] is a list of ones in our example
    - Given a group G_w, the number of ones is equal to the occurrences of word w in the input file

79

Word count running example

- A key-value pair (w, sum G_w.[list of values]) is emitted for each group G_w
  - i.e., the **reduce** function is $r(G_w)$ = (w, sum(G_w.[list of values]))
- The list of emitted pairs is the result of the word count problem
  - One pair (word, num. of occurrences) for each words in our running example

80

MapReduce: Map

- The Map phase can be viewed as a transformation over each element of a data set
  - This transformation is a function **m** defined by the designer
  - Each application of **m** happens in **isolation**
    - The application of **m** to each element of a data set can be parallelized in a straightforward manner

81

MapReduce: Reduce

- The Reduce phase can be viewed as an aggregate operation
  - The aggregate function is a function **r** defined by the designer
  - Also the reduce phase can be performed in parallel
    - Each group of key-value pairs with the same key can be processed in isolation

82

MapReduce: Shuffle and Sort

- The shuffle and sort phase is always the same
  - i.e., group the output of the map phase by key
  - It does not need to be defined by the designer

Data Structures

- Key-value pair is the basic data structure in MapReduce
  - Keys and values can be: integers, float, strings, …
  - They can also be (almost) arbitrary data structures defined by the designer
- Both input and output of a MapReduce program are lists of key-value pairs
  - Note that also the input is a list of key-value pairs

Data Structures

- The design of MapReduce involves
  - Imposing the key-value structure on the input and output data sets
    - E.g.: for a collection of Web pages, input keys may be URLs and values may be the HTML content

Formal definition of Map and Reduce functions

- The map and reduce functions are formally defined as follows:
  - map: $(k1, v1) \rightarrow [(k2, v2)]$
  - reduce: $(k2, [v2]) \rightarrow [(k3, v3)]$
- Since the input data set is a list of key-value pairs, the argument of the map function is a key-value pair

[…] denotes a list. (k, v) denotes a key-value pair

Formal definition of Map and Reduce functions

- Map function
  - map: $(k1, v1) \rightarrow [(k2, v2)]$
- The argument of the map function is a key-value pair
- Note that the map function emits a list of key-value pairs

[…] denotes a list. (k, v) denotes a key-value pair

87

Formal definition of Map and Reduce functions

- Reduce function
  - reduce: $(k2, [v2]) \rightarrow [(k3, v3)]$
- Note that the reduce function emits a list of key-value pairs

[…] denotes a list. (k, v) denotes a key-value pair

88

MapReduce Algorithms

- In many applications, the keys of the input data set are ignored
  - i.e., the map function does not consider the key of its key-value pair argument
    - E.g., word count problem
- Some specific applications exploit also the keys of the input data
  - E.g., keys can be used to uniquely identify records/objects

Word Count using MapReduce: Pseudocode

Input file: a textual document
The map function is invoked over each word of the input file

```
map(key, value):
    // key: offset of the word in the file; value: a word of the input
    // document
    emit(value, 1)
```

```
reduce(key, values):
    // key: a word; value: a list of integers
    occurrences = 0
    for each c in values:
        occurrences = occurrences + c

    emit(key, occurrences)
```