Processing Big Data

• Some approaches to deal with big data
  – Old fashion: RDMS but non applicable
  – **MapReduce**: store and process data-sets at massive scale (Volume + Variety)
  – **Data stream processing**: process fast data *without storing* them

• In this lesson we focus on MapReduce
• In the next lesson, we will examine data stream processing
How much data?

• Every day in 2014 we created:
  – 2.3 Zettabytes (2.3x10^{21}) ...
  – 2,300 Exabytes ...
  – 2,300,000 Petabytes ...
  – 2,300,000,000 Terabytes ...
  – 2,300,000,000,000,000,000 bytes!
  – In 2012 “only” 2.5x10^{18} bytes every day
  – 90% of all the data in the world has been generated over the last two years (2013)

• Google processes more than 20 PB a day (2008)
• Facebook has 2.5 PB of user data + 15 TB/day (4/2009) (~25 PB today)
• eBay has more than 6.5 PB of user data + 50 TB/day (5/2009)
• CERN’s LHC generates 1 PB of data per second (2013)

Parallel programming: background

• Parallel programming
  – Break processing into parts that can be executed concurrently on multiple processors

• Challenge
  – Identify tasks that can run concurrently and/or groups of data that can be processed concurrently
  – Not all problems can be parallelized!
Parallel programming: background (2)

• Simplest environment for parallel programming
  – No dependency among data
    • Data can be split into equal-size chunks
  – Each process can work on a chunk
  – Master/worker approach
    • Master
      – Initializes array and splits it according to the number of workers
      – Sends each worker the sub-array
      – Receives the results from each worker
    • Worker:
      – Receives a sub-array from master
      – Performs processing
      – Sends results to master

• Single Program, Multiple Data (SMPD): technique to achieve parallelism
  – The most common style of parallel programming

Key idea behind MapReduce: Divide and conquer

• A feasible approach to tackling large-data problems
  – Partition a large problem into smaller sub-problems
  – Independent sub-problems executed in parallel
  – Combine intermediate results from each individual worker

• The workers can be:
  – Threads in a processor core
  – Cores in a multi-core processor
  – Multiple processors in a machine
  – Many machines in a cluster

• Implementation details of divide and conquer are complex
Divide and conquer: how?

- Decompose the original problem in smaller, parallel tasks
- Schedule tasks on workers distributed in a cluster, keeping into account:
  - Data locality
  - Resource availability
- Ensure workers get the data they need
- Coordinate synchronization among workers
- Share partial results
- Handle failures

Key idea behind MapReduce: 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
  - Datacenter efficiency is a difficult problem to solve, but recent improvements
- Processing data is quick, I/O is very slow
- 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)
MapReduce

• **Programming model** for processing huge amounts of data sets over thousands of servers
  – Originally proposed by Google in 2004
    “MapReduce: simplified data processing on large clusters”
  – Based on a shared nothing approach

• Also an associated **implementation** (framework) of the distributed system that runs the corresponding programs

• Some examples of applications:
  – Web indexing
  – Reverse Web-link graph
  – Distributed sort
  – Web access statistics

---

MapReduce: programmer view

• MapReduce hides system-level details
  – Key idea: separate the *what* from the *how*
  – MapReduce abstracts away the “distributed” part of the system
  – Such details are handled by the framework

• Programmers get simple API
  – Don’t have to worry about handling
    • Parallelization
    • Data distribution
    • Load balancing
    • Fault tolerance
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

Key idea: provide a functional abstraction of the two Map and Reduce operations

MapReduce: model

- Processing occurs in two phases: Map and Reduce
  - Functional programming roots (e.g., Lisp)
- Map and Reduce are defined by the programmer
- Input and output: sets of key-value pairs
- Programmers specify two functions: map and reduce
- `map(k_1, v_1) \rightarrow [(k_2, v_2)]`
- `reduce(k_2, [v_2]) \rightarrow [(k_3, v_3)]`
  - `(k, v)` denotes a (key, value) pair
  - `[…]` denotes a list
  - Keys do not have to be unique: different pairs can have the same key
  - Normally the keys of input elements are not relevant
Map

- Execute a function on a set of key-value pairs (input shard) to create a new list of values
  \[ \text{map (in\_key, in\_value) } \rightarrow \text{list(out\_key, intermediate\_value)} \]

  - *Example*: square \( x = x \times x \)
    
    ```
    map square [1,2,3,4,5]
    returns [1,4,9,16,25]
    ```

- Map calls are distributed across machines by automatically partitioning the input data into M “shards”
- MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the Reduce function

Reduce

- Combine values in sets to create a new value
  \[ \text{reduce (out\_key, list(intermediate\_value)) } \rightarrow \text{list(out_value)} \]

  - *Example*: sum = (each elem in arr, total +=)
    
    ```
    reduce [1,4,9,16,25]
    returns 55 (the sum of the square elements)
    ```
MapReduce program

• A MapReduce program, referred to as a job, consists of:
  – Code for Map and Reduce packaged together
  – Configuration parameters (where the input lies, where the output should be stored)
  – Input data set, stored on the underlying distributed file system
    • The input will not fit on a single computer’s disk

• Each MapReduce job is divided by the system into smaller units called tasks
  – Map tasks
  – Reduce tasks

• The output of MapReduce job is also stored on the underlying distributed file system

MapReduce computation

1. Some number of Map tasks each are given one or more chunks of data from a distributed file system.
2. These Map tasks turn the chunk into a sequence of key-value pairs.
   – The way key-value pairs are produced from the input data is determined by the code written by the user for the Map function.
3. The key-value pairs from each Map task are collected by a master controller and sorted by key.
4. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task.
5. The Reduce tasks work on one key at a time, and combine all the values associated with that key in some way.
   – The manner of combination of values is determined by the code written by the user for the Reduce function.
6. Output key-value pairs from each reducer are written persistently back onto the distributed file system
7. The output ends up in r files, where r is the number of reducers.
   – Such output may be the input to a subsequent MapReduce phase
Where the magic happens

• Implicit between the map and reduce phases is a distributed “group by” operation on intermediate keys
  – Intermediate data arrive at each reducer in order, sorted by the key
  – No ordering is guaranteed across reducers

• Intermediate keys are transient:
  – They are not stored on the distributed file system
  – They are “spilled” to the local disk of each machine in the cluster

Valeria Cardellini - SDCC 2014/15

MapReduce computation: the complete picture
A simplified view of MapReduce: example

- Mappers are applied to all input key-value pairs, to generate an arbitrary number of intermediate pairs
- Reducers are applied to all intermediate values associated with the same intermediate key
- Between the map and reduce phase lies a barrier that involves a large distributed sort and group by

“Hello World” in MapReduce: WordCount

- Problem: counts the number of occurrences for each word in a large collection of documents
- Input: a repository of documents, each document is an element
- Map: reads a document and emits a sequence of key-value pairs where:
  - Keys are words of the documents and values are equal to 1: (w1, 1), (w2, 1), ..., (wn, 1)
- Grouping: groups by key and generates pairs of the form (w1, [1, 1, ..., 1]), ..., (wn, [1, 1, ..., 1])
- Reduce: adds up all the values and emits (w1, k), ..., (wn, l)
- Output: (w,m) pairs where:
  - w is a word that appears at least once among all the input documents and m is the total number of occurrences of w among all those documents
WordCount: Map

- **Map** emits each word in the document with an associated value equal to “1”

\[
\text{Map}(\text{String } \text{key}, \text{ String } \text{value}): \\
// \text{key}: \text{document name} \\
// \text{value}: \text{document contents} \\
\text{for each word } w \text{ in } \text{value}: \\
\quad \text{EmitIntermediate}(w, "1");
\]

WordCount: Reduce

- **Reduce** adds up all the “1” emitted for a given word

\[
\text{Reduce}(\text{String } \text{key}, \text{ Iterator } \text{values}): \\
// \text{key}: \text{a word} \\
// \text{values}: \text{a list of counts} \\
\text{int result}=0 \\
\text{for each } v \text{ in } \text{values}: \\
\quad \text{result} += \text{parseInt}(v) \\
\text{Emit}(<\text{AsString}(\text{result}))
\]

- This is pseudo-code; for the complete code of the example see the [MapReduce paper](#)
What is Apache Hadoop?

- Open-source software framework for reliable, scalable, distributed data-intensive computing
  - Originally developed by Yahoo!
- Goal: storage and processing of data-sets at massive scale
- Infrastructure: clusters of commodity hardware
- Core components:
  - HDFS: Hadoop Distributed File System
  - Hadoop MapReduce
- Includes a number of related projects
  - Among which Apache Pig, Apache Hive, Apache HBase
- Used in production by Google, Facebook, Yahoo! and many others
- Provided by Amazon (ElasticMapReduce, EMR) as a service running on EC2
Hadoop core

• HDFS
  – A distributed file system characterized by a master/worker architecture
  – Data is replicated with redundancy across the cluster
  – Servers can fail and not abort the computation process
  – Quite similar to Google File System (see next lessons)

• Hadoop MapReduce
  – Allows to easily write applications which process vast amounts of data (multi-terabyte data-sets) in parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner
  – The powerhouse behind most of today’s big data processing (e.g., Facebook)

HDFS

• An HDFS cluster has two types of nodes:
  – Multiple DataNodes
  – One NameNode
HDFS concepts

- **The datanodes** (workers) just store and retrieve the blocks (also shards or chunks) when they are told to (by clients or the namenode)
- The **namenode** (master):
  - Manages the filesystem tree and the metadata for all the files and directories
  - Knows the datanodes on which all the blocks for a given file are located
- Without the namenode HDFS cannot be used
  - It is important to make the namenode resilient to failure

HDFS: file read

- NameNode is only used to get block location

Source: “Hadoop: The definitive guide”
HDFS: file write

Source: “Hadoop: The definitive guide”

- Clients ask NameNode for a list of suitable DataNodes
- This list forms a pipeline: first DataNode stores a copy of a block, then forwards it to the second, and so on

WordCount on Hadoop

- Let’s analyze the WordCount code on Hadoop
  http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v1.0