Università degli Studi di Roma “Tor Vergata”
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Introduction to Data Intensive Computing

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2017/18

Valeria Cardellini
How much data?

- By 2020 40 Zettabytes of data will be created:
  - 40 Zettabytes \((40 \times 10^{21} \approx 40 \times 2^{70})\) ...
  - 40,000 Exabytes \((40,000 \times 10^{18})\) ...
  - 40,000,000 Petabytes \((40,000,000 \times 10^{15})\) ...
  - 40,000,000,000 Terabytes \((40,000,000,000 \times 10^{12})\) ...
  - 40,000,000,000,000,000,000,000 bytes!

- 90% of all the data in the world has been generated over the last two years (in 2013)

Source: [The Four V's of Big Data](http://www.slideshare.net/valericardellini/the-four-v-s-of-big-data)
How Big? Growth rate

• Big Data is growing fast

![Graph showing growth of Big Data categories such as sensors & devices, social media, VoIP, and enterprise data from 2010 to 2015. The graph illustrates the exponential increase in volume of data in exabytes and the percentage of uncertain data.](image-url)
How Big? IoT impact

- Internet of Things (IoT) will largely contribute to increase Big Data challenges
What is Big Data?

• “Big Data” is similar to “Small Data”, but bigger
• …but having data bigger it requires different approaches (scale changes everything!)
  – New methodologies, tools, architectures
• …with an aim to solve new problems
• …or old problems in a better way

![Comic strip with characters discussing big data]
3V model for Big Data

- **Volume**: challenging to store and process (how to index, retrieve)
- **Variety**: different data types (text, audio, video, record) and degree of structure (structured, semi-structured, unstructured data)
- **Velocity**: speed of generation, rate of analysis
- Defined in 2001 by D. Laney

Valeria Cardellini - SDCC 2017/18
The extended (3+n)V model

1. **Volume** (lots of data)
2. **Variety** (complexity, curse of dimensionality)
3. **Velocity** (rate of data and information flow)
4. **Value** (Big data can generate huge competitive advantages)
   - “Big data technologies describe a new generation of technologies and architectures, designed to economically extract value from very large volumes of a wide variety of data, by enabling high-velocity capture, discovery, and/or analysis.” [Gan11]
5. **Variability** (data flows can be highly inconsistent with periodic peaks)
6. **Veracity** (untrusted, uncleaned data)
7. **Visualization**
Processing Big Data

• Some approaches to deal with big data
  – Old fashion: RDMS but non applicable
  – **Batch processing**: store and process data sets at massive scale (especially Volume+Variety), also known as batch processing
    • **MapReduce** is the most notable approach
  – **Data stream processing**: process fast data (in real-time) as data is being generated, without storing
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
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

\[
\text{map (in\_key, in\_value) \rightarrow list(out\_key, intermediate\_value)}
\]

• Example: sum of squares from 1 to n
  \[
  x = x \times x
  \]
  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 of squares from 1 to n
  - \( \text{sum} = (\text{each elem in arr, total +=}) \)
    \[
    \text{reduce } [1,4,9,16,25]
    \]
    returns 55 (the sum of the square elements)
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
“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:
    \((w_1, 1), (w_2, 1), \ldots, (w_n, 1)\)

• Grouping: groups by key and generates pairs of the form
  \((w_1, [1, 1, \ldots, 1]), \ldots, (w_n, [1, 1, \ldots, 1])\)

• Reduce: adds up all the values and emits \((w_1, k), \ldots, (w_n, l)\)

• 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”

```
Map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, “1”);
```
WordCount: Reduce

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

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))

• This is pseudo-code; for the complete code of the example see the MapReduce paper http://bit.ly/2sfzbO8
MapReduce: execution overview
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 Facebook, IBM, Linkedin, Twitter, 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

• 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)
Hadoop in the Cloud

• Pros:
  – Gain Cloud scalability and elasticity
  – No need to manage and provision the infrastructure and the platform

• Main challenges:
  – Move data to the Cloud
    • Latency is not zero!
    • Minor issue: network bandwidth
  – Data security and privacy
Amazon Elastic MapReduce (EMR)

- Distributed the computational work across a cluster of virtual servers running on EC2 instances
- Cluster managed with Hadoop
- Input and output: Amazon S3, DynamoDB
Hadoop ecosystem: a partial big picture

See hadoopecosystemtable.github.io for a longer and updated list
Apache Spark

• Separate, fast and general-purpose engine for large-scale data processing
  – Not a modified version of Hadoop
  – The leading candidate for “successor to MapReduce”
• In-memory data storage for very fast iterative queries
  – At least 10x faster than Hadoop
• Suitable for general execution graphs and powerful optimizations
• Compatible with Hadoop’s storage APIs
  – Can read/write to any Hadoop-supported system, including HDFS and HBase
Data sharing in MapReduce

• Slow due to replication, serialization and disk I/O
Data sharing in Spark

• Distributed in-memory: 10x-100x faster than disk and network
Spark stack

Spark SQL
structured data

Spark
Streaming
real-time

MLlib
machine
learning

GraphX
graph
processing

Spark Core

Standalone Scheduler

YARN

Mesos
Why data stream processing?

• Applications such as:
  – Sentiment analysis on multiple tweet streams @Twitter
  – User profiling @Yahoo!
  – Tracking of query trend evolution @Google
  – Fraud detection
  – Bus routing management @city of Dublin

• Require:
  – Continuous **processing** of unbounded **data streams**
    generated by multiple, distributed sources
  – In (near) **real-time** fashion
Batch processing vs. data stream processing

• Batch processing (MapReduce & Hadoop)
  – Goal: address volume and variety in the Big Data architecture
  – Challenge: latency of computation

• Data stream processing:
  – Goal: decrease the overall latency to obtain results
  – No data persistence on stable storage
    See “Latency numbers every programmer should know”
  – Compute one data element or a small window of recent data at one time
Data stream

• “A data stream is a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety.

Queries over streams run continuously over a period of time and incrementally return new results as new data arrive.”

DSP application model

• A DSP application is made of a network of operators (processing elements) connected by streams, at least one data source and at least one data sink

• Represented by a directed graph
  – Graph vertices: operators
  – Graph edges: streams

• Graph can be cyclic
  – Some systems only support directed acyclic graph (DAG)

• Graph topology rarely changes
DSP programming model

- Data flow programming
- **Flow composition**: techniques for creating the topology associated with the flow graph for an application
- **Flow manipulation**: the use of processing elements (i.e., operators) to perform transformations on data
Data flow manipulation

• How the streaming data is manipulated by the different operators in the flow graph?

• Operator properties:
  – Operator type
  – Operator state
  – Windowing
DSP operator

- A self-contained processing element that:
  - transforms one or more input streams into another stream
  - can execute a generic user-defined code
    - Algebraic operation (filter, aggregate, join, ..)
    - User-defined (more complex) operation (POS-tagging, ..)
  - can execute in parallel with other operators
- Can be stateless or stateful
  - **Stateless**: know nothing about the state (e.g., filter, map)
  - **Stateful**: keep some sort of state
    - E.g., some aggregation or summary of processed elements, or state-machine for detecting patterns for fraudulent financial transaction
    - State might be shared between operators
Sliding windows

- Window: a buffer associated with an input port to retain previously received tuples
- Eviction data policy: how many data items should we keep in the buffer and process each time?
  - Count-based window, e.g., last $n$ items held in the window
  - Time-based window, e.g. from $[t-T]$ to $[t]$
• How often should we evaluate the window?
  – **Eager approach**: output new result items as soon as available (but can be difficult to implement efficiently)
  – **Lazy approach**: slide window by $s$ seconds (or $m$ items)
“Hello World”: a variant of WordCount

- Goal: emit the top-k words in terms of occurrence when there is a rank update

- Where are the bottlenecks?
- How to scale the DSP application in order to sustain the traffic load?
“Hello World”: a variant of WordCount

- The usual answer: replication!
- Use data parallelism
Example of DSP application: DEBS’14 GC

http://debs.org/?p=75

• Real-time analytics over high volume sensor data: analysis of energy consumption measurements for smart homes
  – Smart plugs deployed in households and equipped with sensors that measure values related to power consumption

• Input data stream:
  2967740693, 1379879533, 82.042, 0, 1, 0, 12

• Query 1: make load forecasts based on current load measurements and historical data
  – Output data stream:
    ts, house_id, predicted_load

• Query 2: find the outliers concerning energy consumption
  – Output data stream:
    ts_start, ts_stop, household_id, percentage
Example of DSP application: DEBS’15 GC

http://debs.org/?p=56

- Real-time analytics over high volume spatio-temporal data streams: analysis of taxi trips based on data streams originating from New York City taxis
- Input data streams: include starting point, drop-off point, corresponding timestamps, and information related to the payment

```
07290D3599E7A0D62097A346EFCC1FB5,E7750A37CAB07D0DFF0AF7E3573AC141,2013-01-01 00:00:00,2013-01-01 00:02:00,120,0.44,-73.956528,40.716976,-73.962440,40.715008,CSH,3.50,0.50,0.50,0.00,0.00,4.50
```
Example of DSP application: DEBS’15 GC

- Query 1: **top10 frequent routes** NYC taxis in the last 30 minutes
  - Use Redis for data ingestion

![Diagram of the DSP application process]

Valeria Cardellini - SDCC 2017/18
Streaming system

- Distributed system that executes stream graphs
  - continuously calculates results for long-standing queries
  - over potentially infinite data streams
  - using operators
    - that can be stateless or stateful
- System nodes may be heterogeneous
- Must be highly optimized and with minimal overhead so to deliver real-time response for high-volume DSP applications
- Some open-source frameworks for data stream processing: Apache Storm, Apache Flink, Heron, Apache Spark Streaming
- Also DSP as Cloud service: AWS Kinesis, AWS EMR (Flink), Google Dataflow
DSP: processing model

- Two stream processing models:
  - *One-at-a-time*: each tuple is individually sent
  - *Micro-batched*: some tuples are grouped before being sent

<table>
<thead>
<tr>
<th></th>
<th>One-at-a-time (e.g., Apache Storm)</th>
<th>Micro-batched (e.g., Apache Spark)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower latency</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Higher throughput</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>At-least-once semantics</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Exactly-once semantics</td>
<td>In some cases</td>
<td>✓</td>
</tr>
<tr>
<td>Simpler programming model</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>


The two approaches are complementary with distinct trade-offs and are suitable to different types of applications.
Apache Storm

• **Apache Storm**
  – Open-source, real-time, scalable streaming system
  – Provides an abstraction layer to execute DSP applications
  – Initially developed by Twitter

• **Topology**
  – DAG of **spouts** (sources of streams) and **bolts** (operators and data sinks)
Stream grouping in Storm

- Data parallelism in Storm: how are streams partitioned among multiple tasks (threads of execution)?
- Shuffle grouping
  - Randomly partitions the tuples

- Field grouping
  - Hashes on a subset of the tuple attributes
Stream grouping in Storm

• All grouping (i.e., broadcast)
  – Replicates the entire stream to all the consumer tasks

• Global grouping
  – Sends the entire stream to a single bolt

• Direct grouping
  – Sends tuples to the consumer bolts in the same executor
Storm architecture

- Master-worker architecture
Storm components: Nimbus and Zookeeper

• Nimbus
  – The master node
  – Clients submit topologies to it
  – Responsible for distributing and coordinating the topology execution

• Zookeeper
  – Nimbus uses a combination of local disk(s) and Zookeeper to store state about the topology
Storm components: worker

- **Task**: operator instance
  - The actual work for a bolt or a spout is done in the task
- **Executor**: smallest schedulable entity
  - Execute one or more tasks related to same operator
- **Worker process**: Java process running one or more executors
- **Worker node**: computing resource, a container for one or more worker processes
WordCount in Storm

- Count the occurrences of each word (see WordCount for MapReduce)

Which kind of application?
Find topic trends on Twitter
WordCount in Storm

Classe Java standard: main

<table>
<thead>
<tr>
<th>Step</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TopologyBuilder tb = new TopologyBuilder();</td>
</tr>
<tr>
<td>2</td>
<td>tb.setSpout(&quot;spout&quot;, new RandomSentenceSpout(), 5);</td>
</tr>
<tr>
<td>3</td>
<td>tb.setBolt(&quot;split&quot;, new SplitSentence(), 8).shuffleGrouping(&quot;spout&quot;);</td>
</tr>
<tr>
<td></td>
<td>StormSubmitter.submitTopology(&quot;word-count&quot;, new Config(), tb.createTopology());</td>
</tr>
</tbody>
</table>

Partizionamento stream  Parallelismo componenti  API Storm

Valeria Cardellini - SDCC 2017/18
WordCount in Storm

RandomSentenceSpout (extends BaseRichSpout)

```java
public void nextTuple() {
    Utils.sleep(100);
    collector.emit(new Values(getRandomSentence()));
}

public void declareOutputFields(OutputFieldsDeclarer d) {
    d.declare(new Fields("sentence"));
}
```

WordCount (extends BaseBasicBolt)

```java
public void execute(Tuple tuple, BasicOutputCollector collector) {
    String word = tuple.getStringByField("word");
    Integer count = updateWordCountHashMap(word);
    collector.emit(new Values(word, count));
}

public void declareOutputFields(OutputFieldsDeclarer d) {
    d.declare(new Fields("word", "count"));
}
```
Other frameworks

- Apache Flink
- Twitter Heron
- Apache Spark Streaming
  - Reduce the size of each stream and process streams of data (*micro-batch processing*)
- Cloud-based frameworks
  - Google Cloud Dataflow
  - Amazon Kinesis
Apache Flink

- Distributed data flow processing system
- One common runtime for DSP applications and batch processing applications
  - Batch processing applications run efficiently as special cases of DSP applications
- Integrated with many other projects in the open-source data processing ecosystem
- Derives from Stratosphere project by TU Berlin, Humboldt University and Hasso Plattner Institute
- Support a Storm-compatible API
**Flink: software stack**

- On top: libraries with high-level APIs for different use cases, still in beta

<table>
<thead>
<tr>
<th>APIs &amp; Libraries</th>
<th>Core</th>
<th>Deploy</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEP Event Processing</td>
<td><strong>Runtime</strong> Distributed Streaming Dataflow</td>
<td>Local Single JVM</td>
</tr>
<tr>
<td>Table Relational</td>
<td></td>
<td>Cluster Standalone, YARN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cloud GCE, EC2</td>
</tr>
<tr>
<td><strong>Flink</strong> Machine Learning</td>
<td><strong>DataStream API</strong> Stream Processing</td>
<td><strong>DataSet API</strong> Batch Processing</td>
</tr>
<tr>
<td><strong>Gelly</strong> Graph Processing</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Flink: some features

• Supports stream processing and windowing with Event Time semantics
  – Event time makes it easy to compute over streams where events arrive out of order, and where events may arrive delayed

• Exactly-once semantics for stateful computations

• Highly flexible streaming windows
Flink: some features

- Continuous streaming model with backpressure
- Flink's streaming runtime has natural flow control: slow data sinks backpressure faster sources
Flink: APIs and libraries

• Streaming data applications: DataStream API
  – Supports functional transformations on data streams, with user-defined state, and flexible windows
  – Example: compute a sliding histogram of word occurrences of a data stream of texts

```scala
case class Word(word: String, freq: Long)
val texts: DataStream[String] = ...
val counts = text
  .flatMap { line => line.split("\W+") }
  .map { token => Word(token, 1) }
  .keyBy("word")
  .timeWindow(Time.seconds(5), Time.seconds(1))
  .sum("freq")
```