Bases de dades avançades

curs 2017/18

Enric Biosca Trias ebiosca@ub.edu

Dept. Matemàtica Aplicada i Anàlisi.

Universitat de Barcelona

Temari

1. Sistemes analítics i Datawarehouse
2. Big Data & Storage
3. NoSQL

Big Data i Storage

Bases de dades avançades curs 17/18

Enric Biosca Trias ebiosca@ub.edu

Dept. Matemàtica Aplicada i Anàlisi.

Universitat de Barcelona

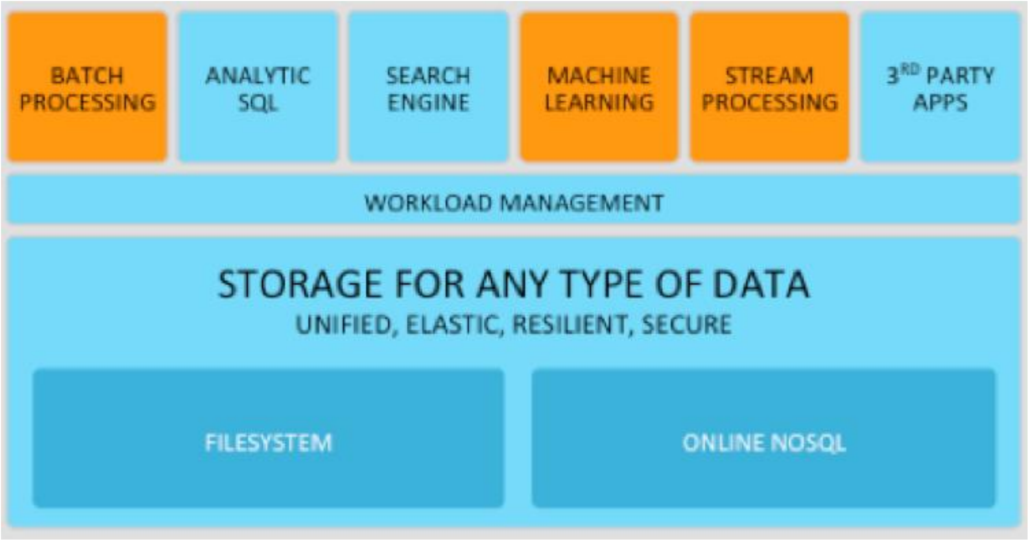
UNIVERSITAT DE BARCELONA

Introducció a Spark

Introduction to spark

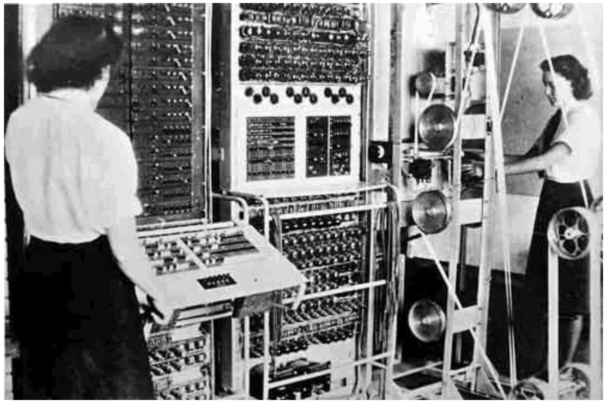
CDH

- 100% open source, enterpriseAready distribution of Hadoop and related projects
- The most complete, tested, and widelyA deployed distribution of Hadoop
- Integrates all key Spark and Hadoop ecosystem projects



Traditional Large Scale Computation

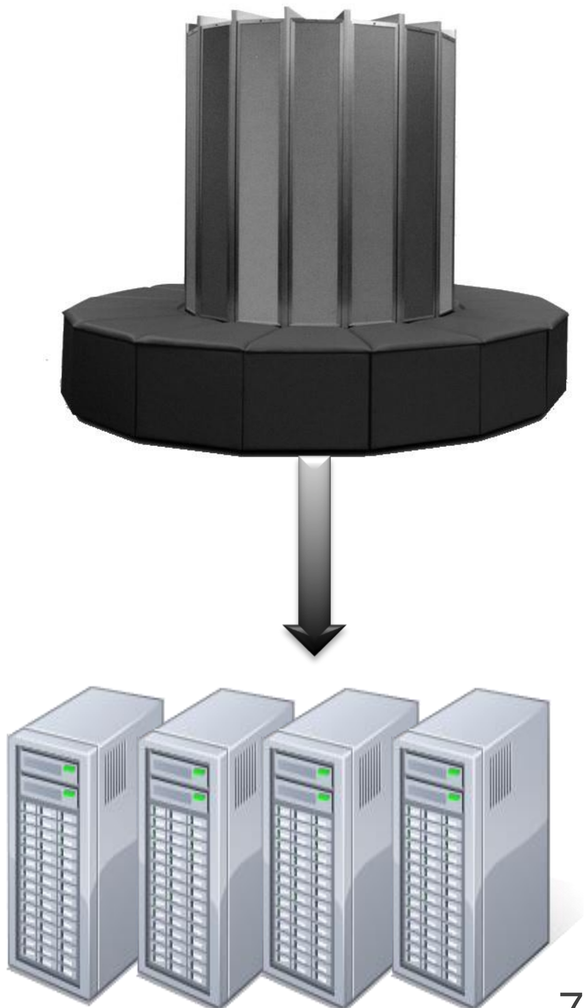
- **Traditionally, computation has been processor-bound**
  - Relatively small amounts of data
  - Lots of complex processing



- **The early solution: bigger computers**
  - Faster processor, more memory
  - But even this couldn't keep up

- **The better solution: more computers**
  - Distributed systems – use multiple machines for a single job

> “In pioneer days they used oxen for heavy pulling, and when one ox couldn’t budge a log, we didn’t try to grow a larger ox. We shouldn’t be trying for bigger computers, but for *more systems* of computers.”
>
> – Grace Hopper

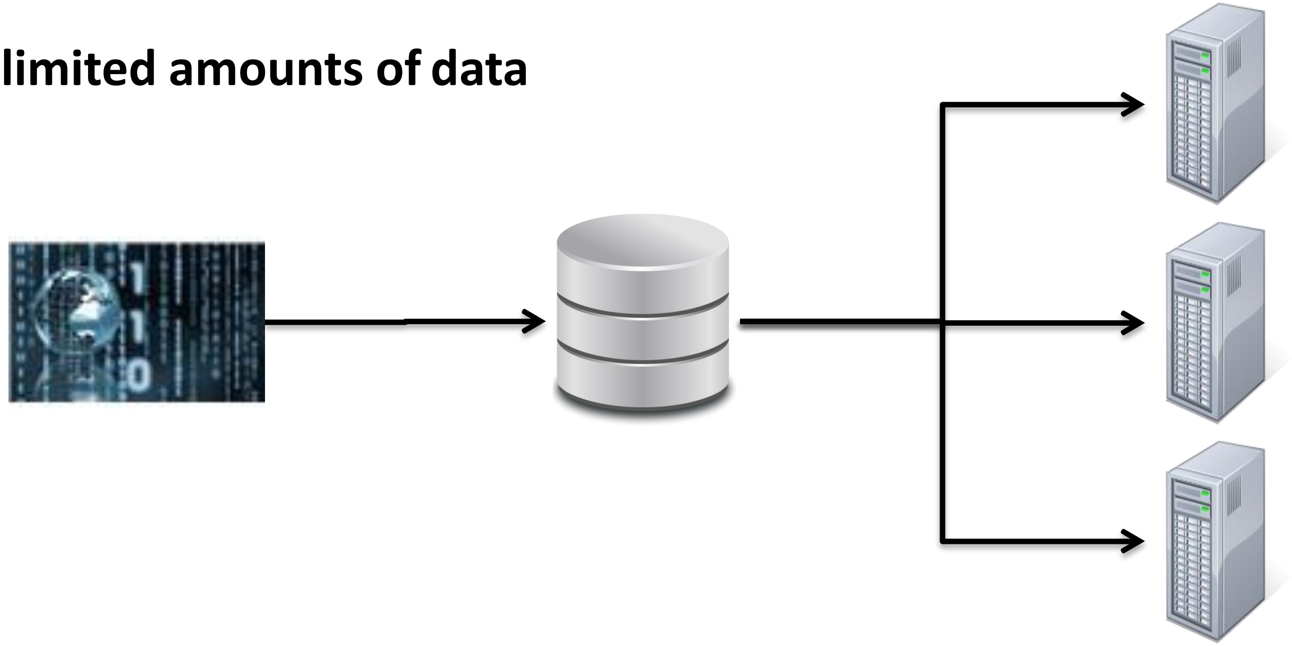


Distributed Systems: Challenges

- **Challenges with distributed systems**
  - Programming complexity
    - Keeping data and processes in sync
  - Finite bandwidth
  - Partial failures

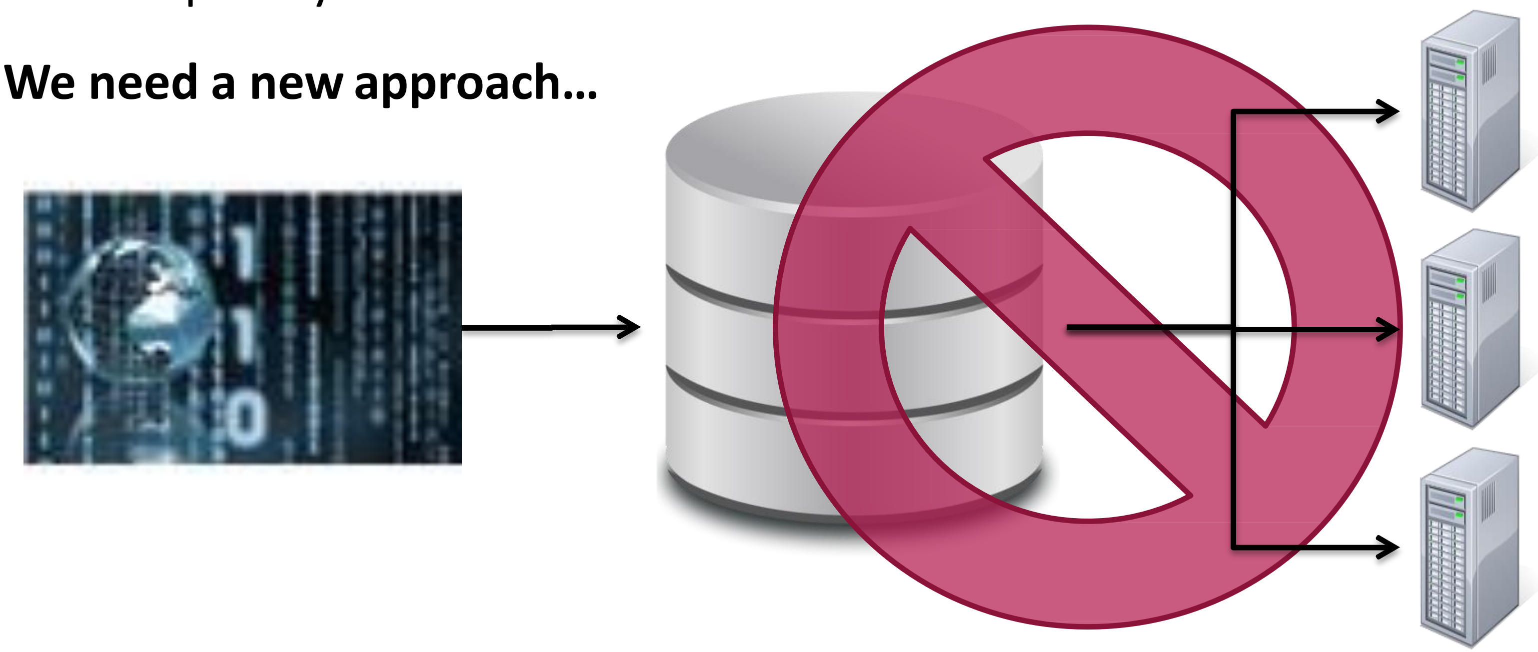
Distributed Systems: The Data Bottleneck (1)

- **Traditionally, data is stored in a central location**
- **Data is copied to processors at runtime**
- **Fine for limited amounts of data**

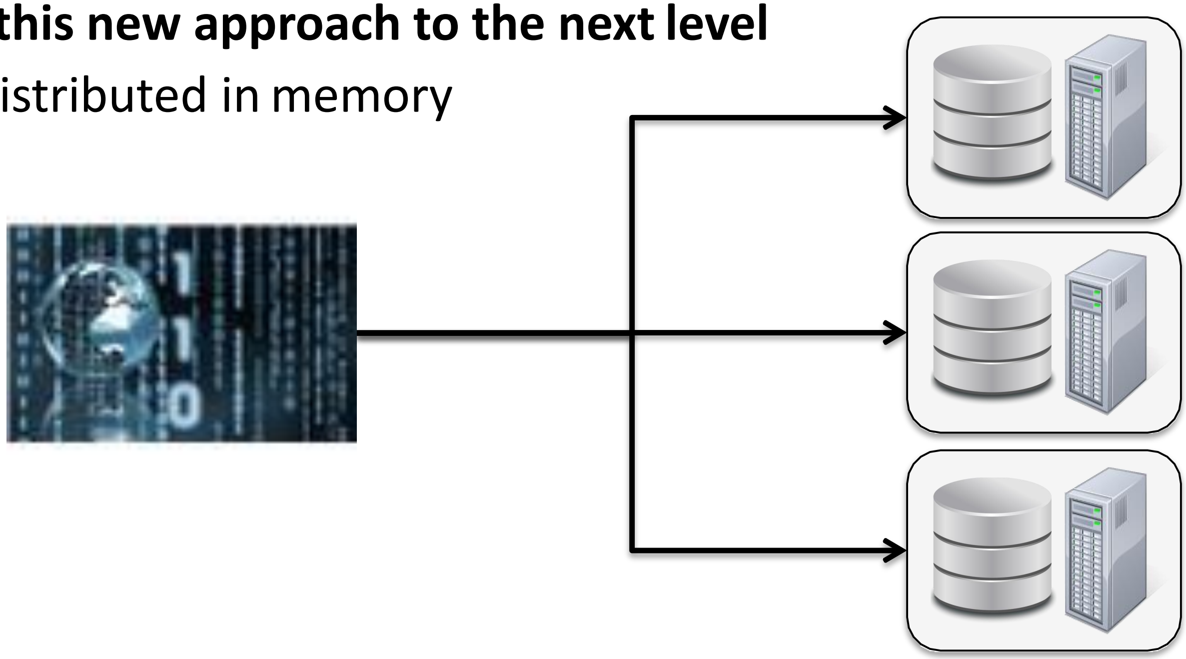


Distributed Systems: The Data Bottleneck (2)

- **Modern systems have much more data**
  - terabytes+ a day
  - petabytes+ total
- **We need a new approach...**

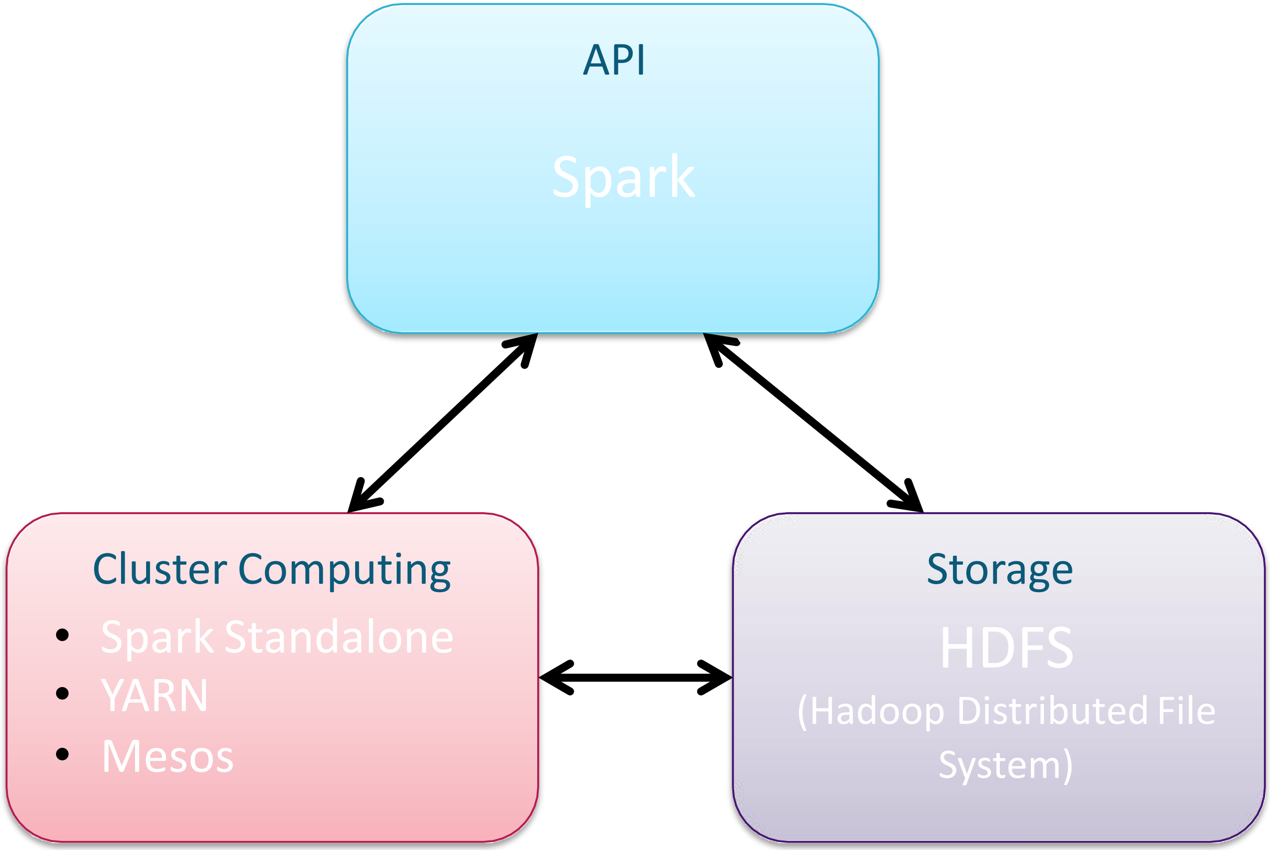


- **Hadoop introduced a radical new approach based on two key concepts**
  - Distribute the data when it is stored
  - Run computation where the data is
- **Spark takes this new approach to the next level**
  - Data is distributed in memory



Introducing Apache Spark

- **Apache Spark** is a fast, general engine for large-scale data processing on a cluster
- **Originally developed at AMPLab at UC Berkeley**
  - Started as a research project in 2009
- **Open source Apache project**
  - Committers from Cloudera, Yahoo, Databricks, UC Berkeley, Intel, Groupon, …
  - One of the most active and fastest-growing Apache projects
  - Cloudera provides enterprise5level support for Spark

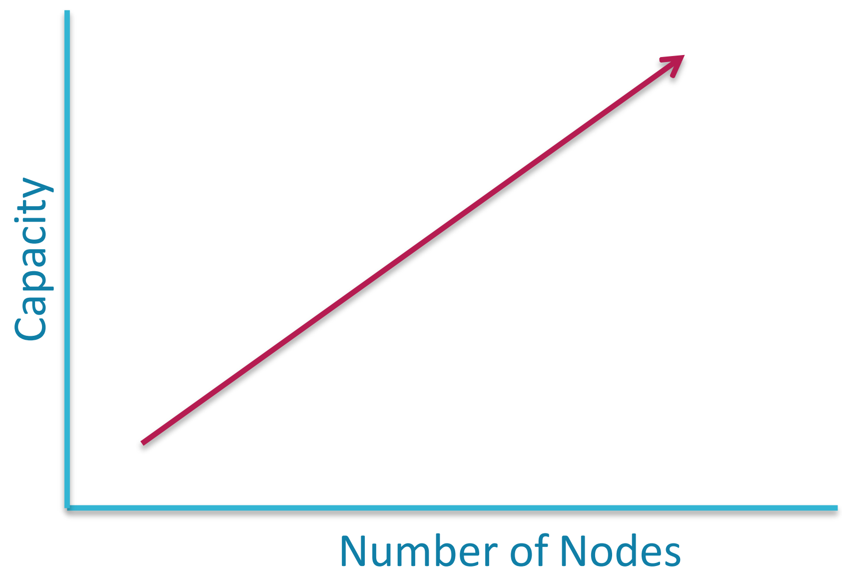

API
Spark
Cluster Computing
Spark Standalone
YARN
Mesos
Storage
HDFS
(Hadoop Distributed File System)

Advantages of Spark

- **High-level programming framework**
  - Programmers can focus on logic, not plumbing
- **Cluster computing**
  - Application processes are distributed across a cluster of worker nodes
  - Managed by a single “master”
  - Scalable and fault tolerant
- **Distributed storage**
  - Data is distributed when it is stored
  - Replicated for efficiency and fault tolerance
  - “Bring the computation to the data”
- **Data in memory**
  - Configurable caching for efficient iteration

Scalability

- **Increasing load results in a graceful decline in performance**
  - Not failure of the system
- **Adding nodes adds capacity proportionally**



Fault Tolerance

- **Node failure is inevitable**
- **What happens?**
  - System continues to function
  - Master re5assigns tasks to a different node
  - Data replication = no loss of data
  - Nodes which recover rejoin the cluster automatically

Who Uses Spark?

- **Yahoo!**
  - Personalization and ad analytics
- **Conviva**
  - Real-time video stream optimization
- **Technicolor**
  - Real-time analytics for telco clients
- **Ooyala**
  - Cross-device personalized video experience
- **Plus…**
  - Intel, Groupon, TrendMicro, Autodesk, Nokia, Shopify, ClearStory, Technicolor, and many more…

Common Spark Use Cases

- Extract/Transform/Load (ETL)
- Text mining
- Index building
- Graph creation and analysis
- Pattern recognition
- Collaborative filtering
- Prediction models
- Sentiment analysis
- Risk assessment

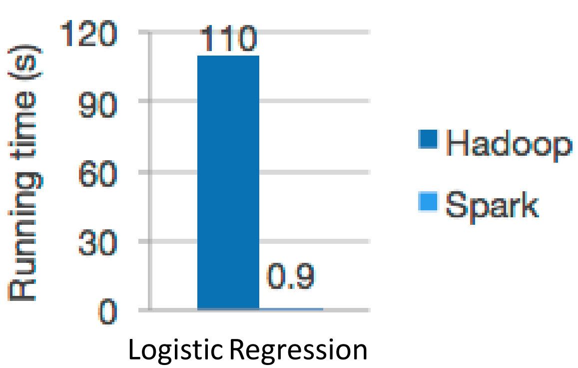
- What do these workloads have in common? Nature of the data…
  - Volume
  - Velocity
  - Variety

Benefits of Spark

- **Previously impossible or impractical analysis**
- **Lower cost**
- **Less time**
- **Greater flexibility**
- **Near-linear scalability**

Spark v. Hadoop MapReduce

- **Spark takes the concepts of MapReduce to the next level**
  - Higher level API = faster, easier development
  - Low latency = near real5time processing
  - In5memory data storage = up to 100x performance improvement




```
sc.textFile(file) \
   .flatMap(lambda s: s.split()) \
   .map(lambda w: (w,1)) \
   .reduceByKey(lambda v1,v2: v1+v2) \
   .saveAsTextFile(output)
```

```
public class WordCount {
  public static void main(String[] args) thr ow
   Job job = new Job();
    job.setJarByClass(WordCount.class);
    job.setJobName("Word Count");
    FileInputFormat.setInputPaths(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.setMapperClass(WordMapper.class);
    job.setReducerClass(SumReducer.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    boolean success = job.waitForCompletion(true);
    System.exit(success ? 0 : 1);
  }
}

public class WordMapper extends Mapper<LongWritable, Text, Text,
IntWritable> {
public void map(LongWritable key, Text value,
Context context) throws IOException, InterruptedException {
    String line = value.toString();
    for (String word : line.split("\\W+")) {
      if (word.length() > 0)
        context.write(new Text(word), new IntWritable(1));
      }
    }
  }
}

public class SumReducer extends Reducer<Text, IntWritable, Text,
IntWritable> {
  public void reduce(Text key, Iterable<IntWritable>
  values, Context context) throws IOException, InterruptedException {
    int wordCount = 0;
    for (IntWritable value : values) {
      wordCount += value.get();
    }
    context.write(key, new IntWritable(wordCount));
  }
}
```

Key Points

- **Traditional large-scale computing involved complex processing on small amounts of data**
- **Exponential growth in data drove development of distributed computing**
- **Distributed computing is difficult!**
- **Spark addresses big data distributed computing challenges**
  - Bring the computation to the data
  - Fault tolerance
  - Scalability
  - Hides the 'plumbing' so developers can focus on the data
  - Caches data in memory

What is Apache Spark?

- **Apache Spark is a fast and general engine for large-scale data processing**
- **Written in Scala**
  - Functional programming language that runs in a JVM
- **Spark Shell**
  - Interactive – for learning or data exploration
  - Python or Scala
- **Spark Applications**
  - For large scale data processing
  - Python, Scala, or Java

Spark Shell

- The Spark Shell provides interactive data exploration (REPL)
- Writing standalone Spark applications will be covered later

Python Shell: **`pyspark`**

```
$ pyspark

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /   / .__/\_,_/_/ /_/\_\   version 1.0.0
      /_/

Using Python version 2.6.6 (r266:84292, Jan
22 2014 09:42:36)
SparkContext available as sc.

>>>
```

Scala Shell: **`spark-shell`**

```
$ spark-shell

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.0.0
      /_/

Using Scala version 2.10.3 (Java HotSpot(TM)
64-Bit Server VM, Java 1.7.0_51)
Created spark context..
Spark context available as sc.

scala>
```

Spark Context

- **Every Spark application requires a *Spark Context***
  - The main entry point to the Spark API
- **Spark Shell provides a preconfigured Spark Context called `sc`**

```
Using Python version 2.6.6 (r266:84292, Jan 22 2014 09:42:36)
Spark context available as sc.

>>> sc.appName
u'PySparkShell'
```

```
Using Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM,
Java 1.7.0_51)
Created spark context..
Spark context available as sc.

scala> sc.appName
res0: String = Spark shell
```

UNIVERSITAT DE BARCELONA

RDD (Resilient Distributed Dataset)

- **RDD (Resilient Distributed Dataset)**
  - Resilient – if data in memory is lost, it can be recreated
  - Distributed – stored in memory across the cluster
  - Dataset – initial data can come from a file or be created programmatically
- **RDDs are the fundamental unit of data in Spark**
- **Most Spark programming consists of performing operations on RDDs**

Creating an RDD

- **Three ways to create an RDD**
  - From a file or set of files
  - From data in memory
  - From another RDD

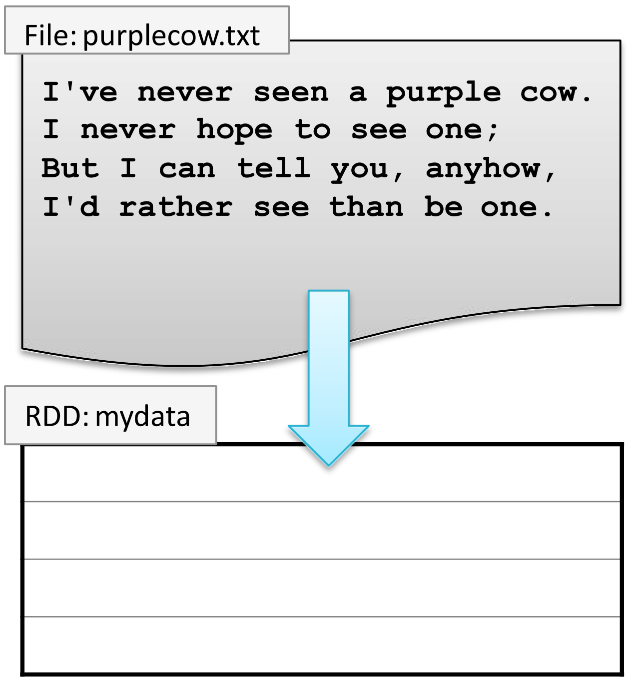
UNIVERSITAT DE BARCELONA

File-Based RDDs

- **For file-based RDDS, use `SparkContext.textFile`**
  - Accepts a single file, a wildcard list of files, or a comma-separated list of files
  - Examples
    - `sc.textFile("myfile.txt")`
    - `sc.textFile("mydata/*.log")`
    - `sc.textFile("myfile1.txt,myfile2.txt")`
  - Each line in the file(s) is a separate record in the RDD
- **Files are referenced by absolute or relative URI**
  - Absolute URI: `file:/home/training/myfile.txt`
  - Relative URI (uses default file system): `myfile.txt`

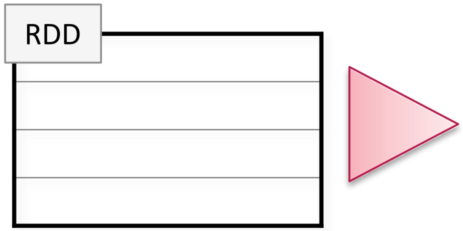
Example: A File-based RDD

```
> mydata = sc.textFile("purplecow.txt")
…
14/01/29 06:20:37 INFO storage.MemoryStore:
   Block broadcast_0 stored as values to
   memory (estimated size 151.4 KB, free 296.8
   MB)

> mydata.count()


…
14/01/29 06:27:37 INFO spark.SparkContext: Job
   finished: take at <stdin>:1, took
   0.160482078 s
4
```

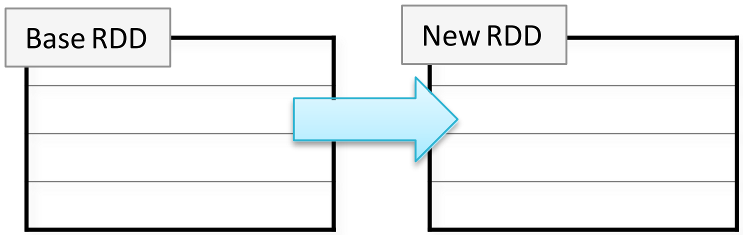


RDD Operations

- **Two types of RDD operations**
  - Actions – return values



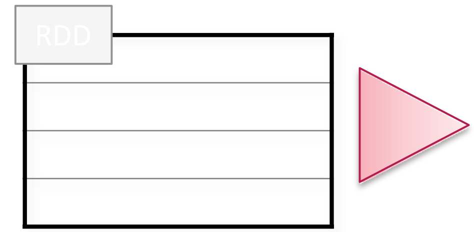
  - Transformations – define a new RDD based on the current one(s)



- **Quiz:**
  - Which type of operation is `count()`?

RDD Operations: Actions

- **Some common actions**
  - **`count()`** – return the number of elements
  - **`take(`*`n`*`)`** – return an array of the first *n* elements
  - **`collect()`** – return an array of all elements
  - **`saveAsTextFile(`*`filename`*`)`** – save to text file(s)

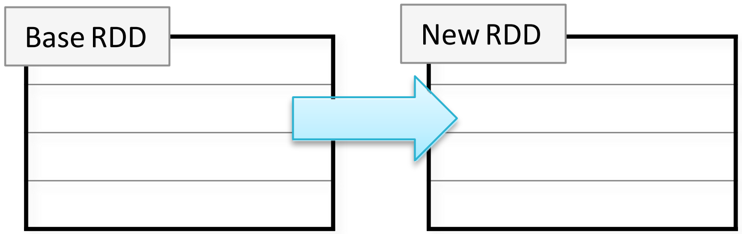



```
> mydata =
   sc.textFile("purplecow.txt")

> mydata.count()
4

> for line in mydata.take(2):
     print line
I've never seen a purple cow.
I never hope to see one;
```

```
> val mydata =
   sc.textFile("purplecow.txt")

> mydata.count()
4

> for (line <- mydata.take(2))
     println(line)
I've never seen a purple cow.
I never hope to see one;
```

RDD Operations: Transformations

- **Transformations create a new RDD from an existing one**



- **RDDs are immutable**
  - Data in an RDD is never changed
  - Transform in sequence to modify the data as needed
- **Some common transformations**
  - **`map`**`(`*`function`*`)` – creates a new RDD by performing a function on each record in the base RDD
  - **`filter`**`(`*`function`*`)` – creates a new RDD by including or excluding each record in the base RDD according to a boolean function

Example: `map` and `filter` Transformations

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

Python: `map(lambda line: line.upper())`

Scala: `map(line => line.toUpperCase)`

```
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.
```

Python: `filter(lambda line: line.startswith('I'))`

Scala: `filter(line => line.startsWith('I'))`

```
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.
```

Lazy Execution (1)

- **RDDs are not always immediately materialized**
  - Spark logs the *lineage* of transformations used to build datasets

File: purplecow.txt

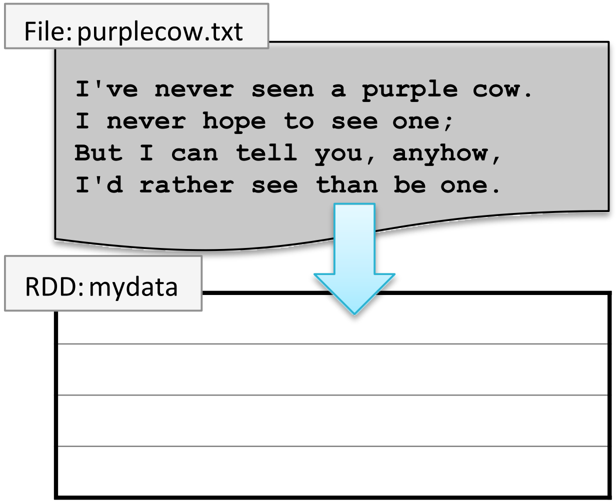
```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

```
>
```

Lazy Execution (2)

- **Data in RDDs is not processed until an *action* is performed**
  - RDD is materialized in memory upon the first action that uses it

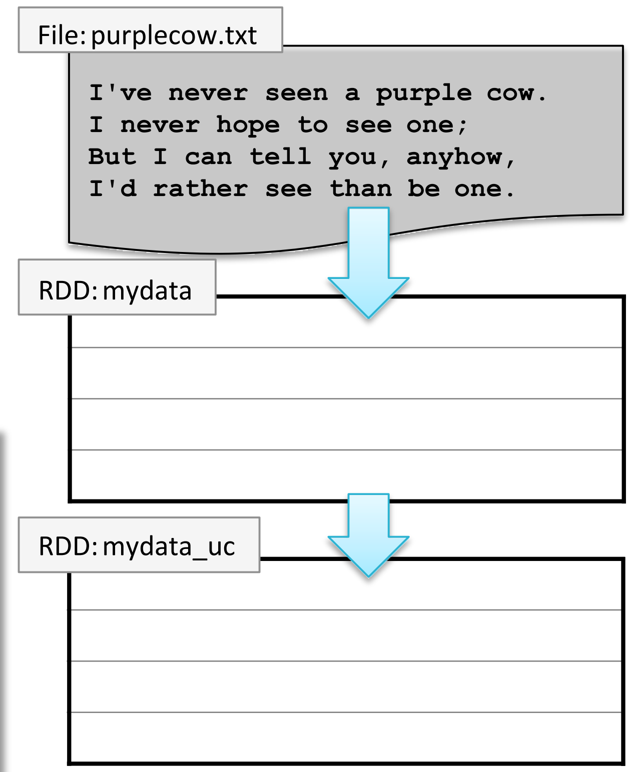
```
> mydata = sc.textFile("purplecow.txt")
```



Lazy Execution (3)

- **Data in RDDs is not processed until an *action* is performed**
  - RDD is materialized in memory upon the first action that uses it

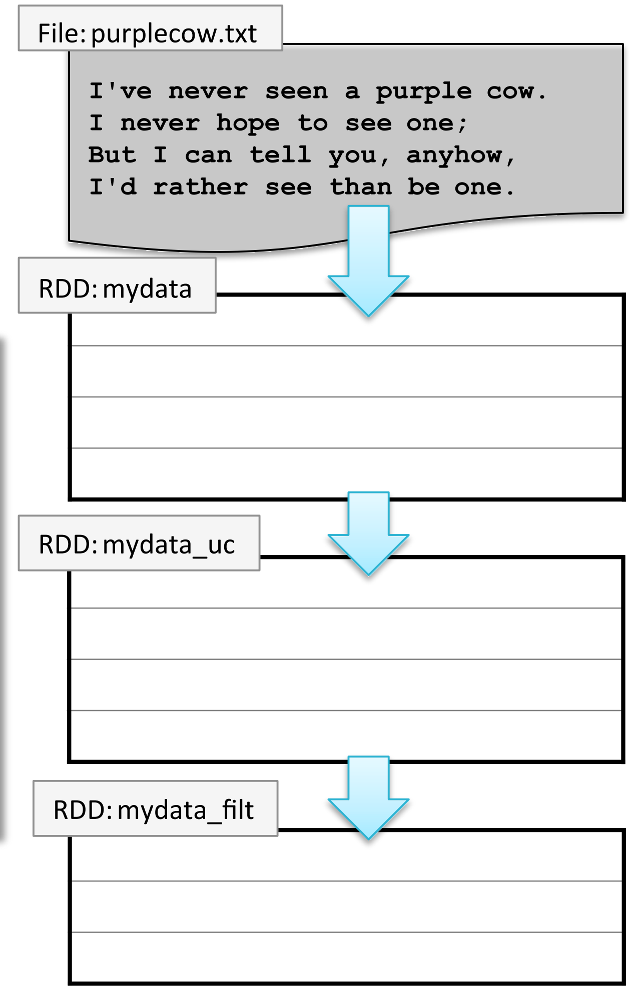
```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda line:
   line.upper())
```



Lazy Execution (4)

- **Data in RDDs is not processed until an *action* is performed**
  - RDD is materialized in memory upon the first action that uses it

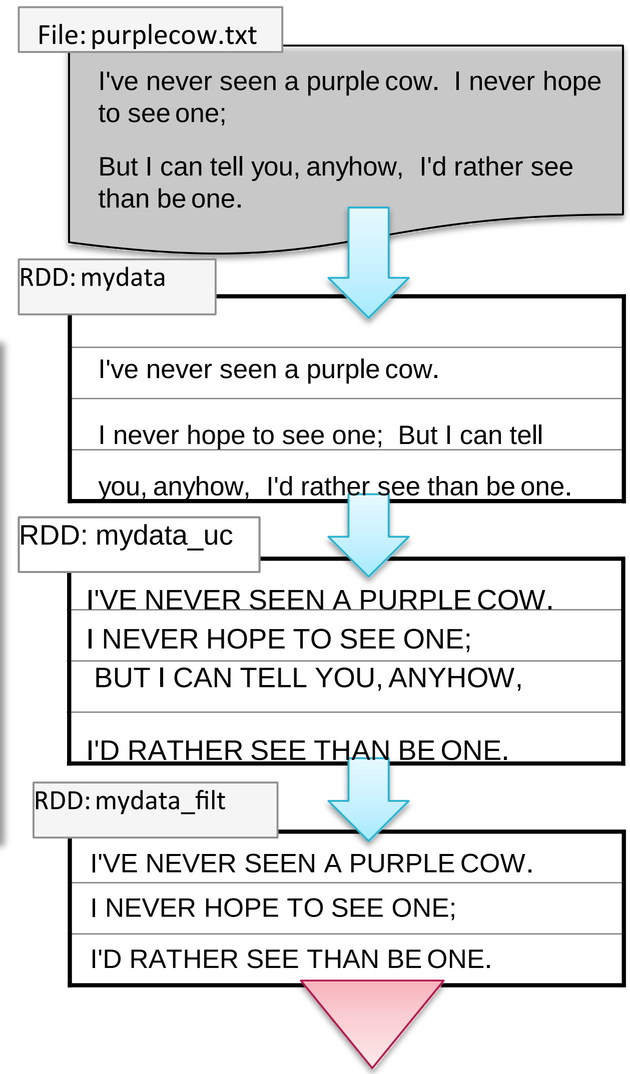
```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda line:
   line.upper())
> mydata_filt = \
     mydata_uc.filter(lambda line: \
     line.startswith('I'))
```



Lazy Execution (5)

- **Data in RDDs is not processed until an *action* is performed**
  - RDD is materialized in memory upon the first action that uses it

```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda line:
   line.upper())
> mydata_filt = \
     mydata_uc.filter(lambda line: \
     line.startswith('I'))
> mydata_filt.count()
3
```



Chaining Transformations

- **Transformations may be chained together**

```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda line: line.upper())
> mydata_filt = mydata_uc.filter(lambda line: line.startswith('I'))
> mydata_filt.count()
3
```

is exactly equivalent to

```
> sc.textFile("purplecow.txt").map(lambda line: line.upper()) \
  .filter(lambda line: line.startswith('I')).count()
3
```

Functional Programming in Spark

- **Spark depends heavily on the concepts of *functional programming***
  - Functions are the fundamental unit of programming
  - Functions have input and output only
    - No state or side effects
- **Key concepts**
  - Passing functions as input to other functions
  - Anonymous functions

Passing Functions as Parameters

- **Many RDD operations take functions as parameters**
- **Pseudocode for the RDD map operation**
  - Applies function **`fn`** to each record in the RDD

```
RDD {
    map(fn(x)) {
        foreach record in rdd
        emit fn(record)
    }
}
```

Example: Passing Named Functions

- **Python**

```
> def toUpper(s):
      return s.upper()
> mydata = sc.textFile("purplecow.txt")
> mydata.map(toUpper).take(2)
```

- **Scala**

```
> def toUpper(s: String): String =
     { s.toUpperCase }
> val mydata = sc.textFile("purplecow.txt")
> mydata.map(toUpper).take(2)
```

Anonymous Functions

- **Functions defined in-line without an identifier**
  - Best for short, one-off functions
- **Supported in many programming languages**
  - Python: `lambda`
  - Scala: `x => ...`
  - Java 8: `x -> ...`

Example: Passing Anonymous Functions

- **Python:**

```
> mydata.map(lambda line: line.upper()).take(2)
```

- **Scala:**

```
> mydata.map(line => line.toUpperCase()).take(2)
```

```
> mydata.map(_.toUpperCase()).take(2)
```

Scala allows anonymous parameters using underscore (_)

Example: Java

```
...
  JavaRDD<String> lines = sc.textFile("file");
  JavaRDD<String> lines_uc = lines.map(
    new MapFunction<String, String>() {
      public String call(String line) {
        return line.toUpperCase();
      }
    }
...
```

```
...
 JavaRDD<String> lines = sc.textFile("file");
 JavaRDD<String> lines_uc = lines.map(
    line -> line.toUpperCase());
...
```

Key Points

- **Spark can be used interactively via the Spark Shell**
  - Python or Scala
  - Writing non7interactive Spark applications will be covered later
- **RDDs (Resilient Distributed Datasets) are a key concept in Spark**
- **RDD Operations**
  - Transformations create a new RDD based on an existing one
  - Actions return a value from an RDD
- **Lazy Execution**
  - Transformations are not executed until required by an action
- **Spark uses functional programming**
  - Passing functions as parameters
  - Anonymous functions in supported languages (Python and Scala)

Bases de dades avançades

curs 2017/18

Enric Biosca Trias ebiosca@ub.edu

Dept. Matemàtica Aplicada i Anàlisi.

Universitat de Barcelona