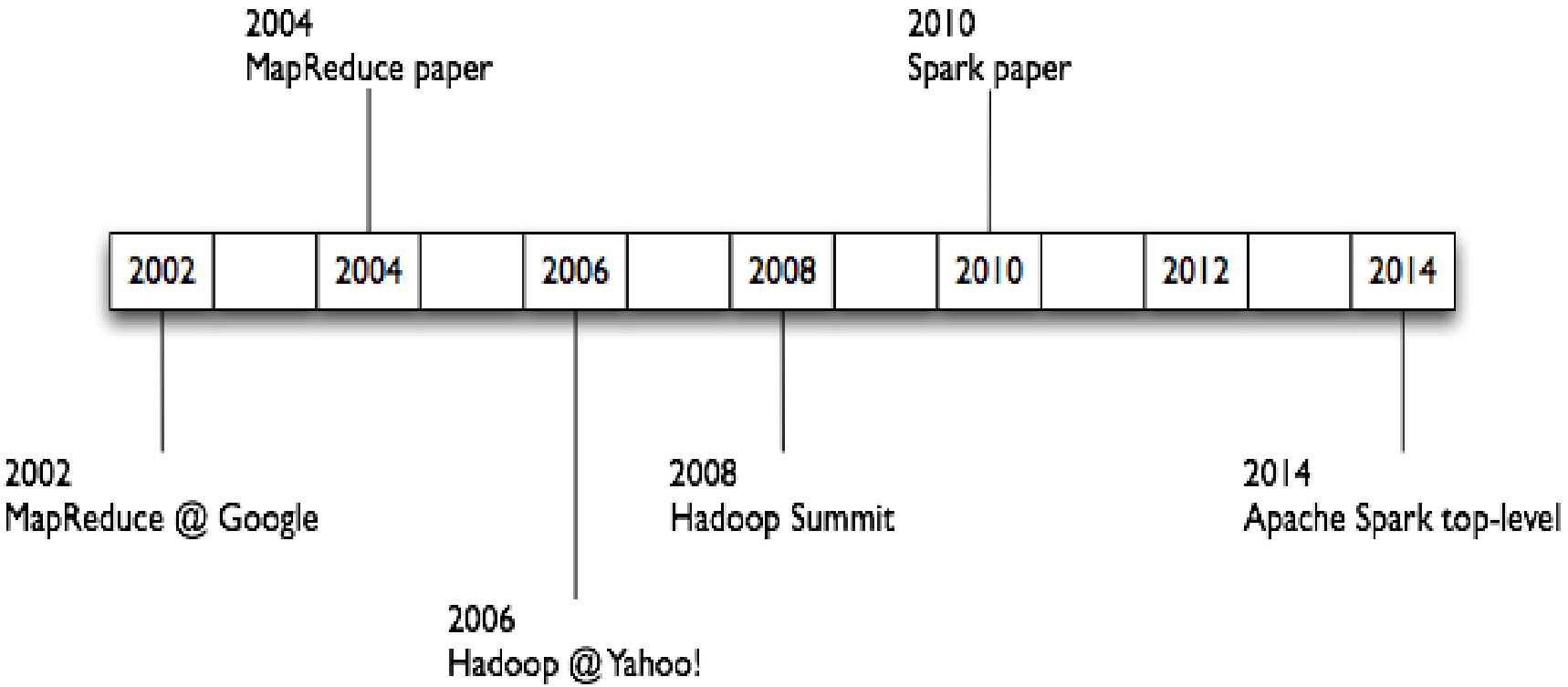
Introduction to Spark

Spark

- Apache Spark™ is a fast and general-purpose engine for large-scale data processing
- Spark aims at achieving the following goals in the Big data context
  - Generality: diverse workloads, operators, job sizes
  - Low latency: sub-second
  - Fault tolerance: faults are the norm, not the exception
  - Simplicity: often comes from generality

Spark History

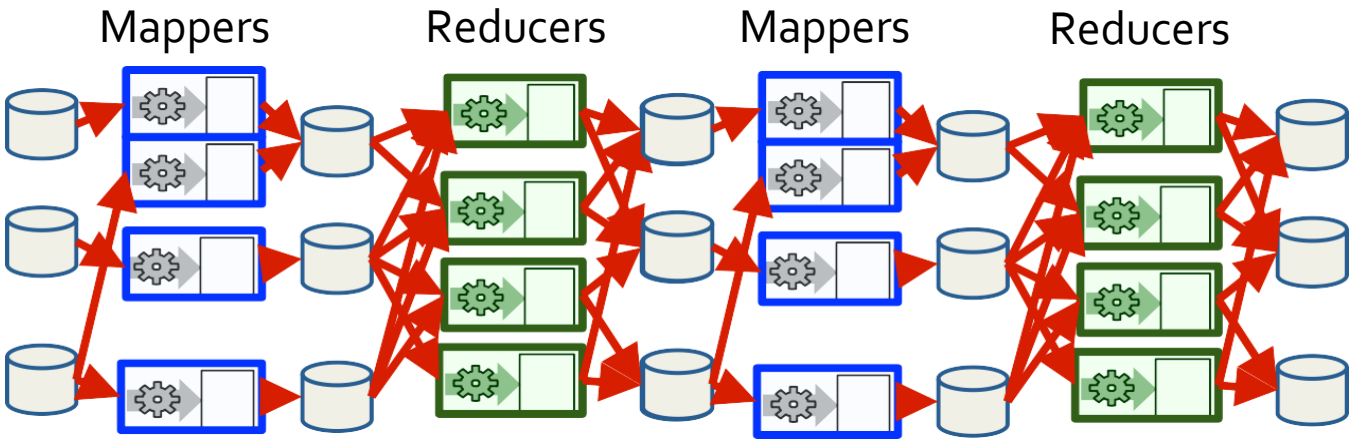
- Originally developed at the University of California - Berkeley's AMPLab

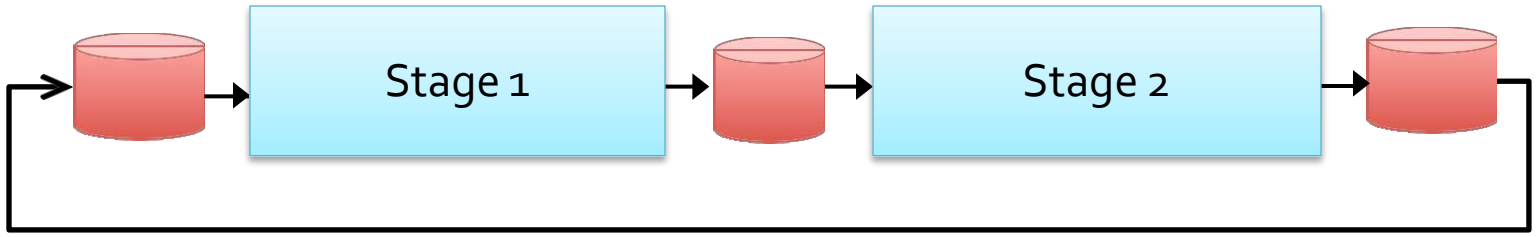


Spark: Motivations

MapReduce and Iterative Jobs

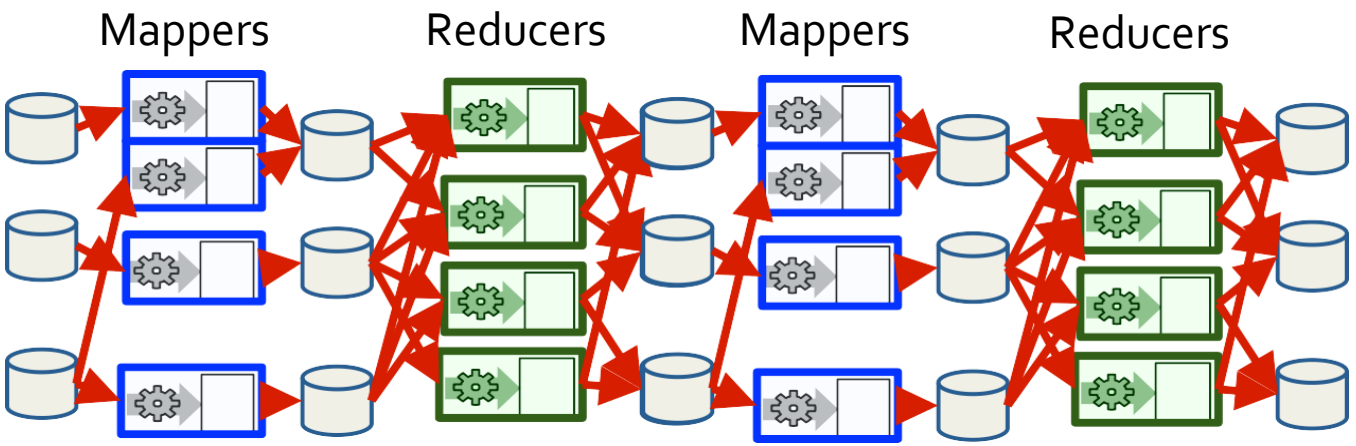
- Iterative jobs, with MapReduce, involve a lot of disk I/O for each iteration and stage

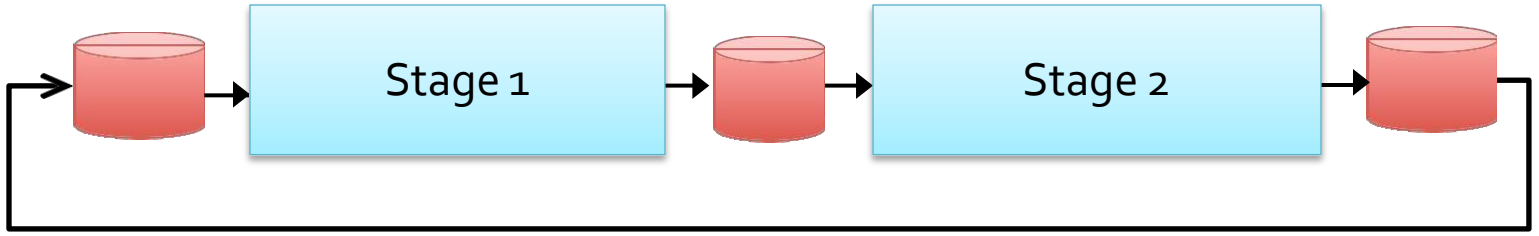




MapReduce and Iterative Jobs

- Disk I/O is very slow (even if it is local I/O)



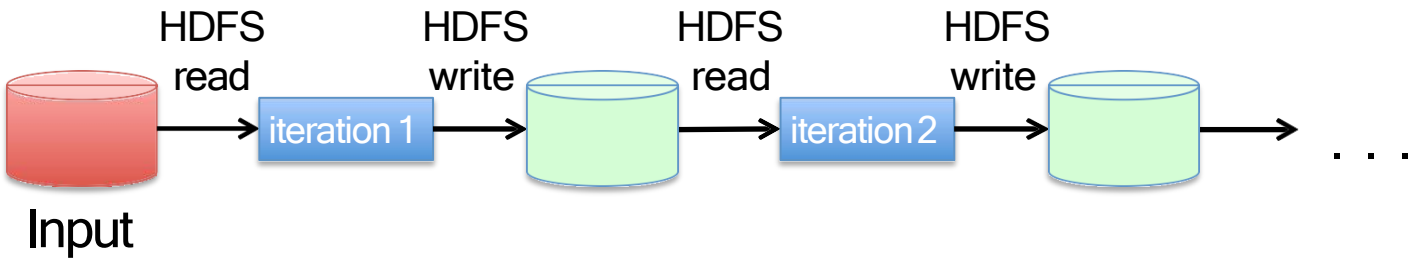


Apache Spark: Motivation and Opportunity

- **Motivation**
  - Using MapReduce for complex **iterative jobs** or **multiple jobs on the same data** involves lots of disk I/O
- **Opportunity**
  - The **cost** of **main memory decreased**
    - Hence, large main memories are available in each server
- **Solution**
  - Keep **more data in main memory**
    - Basic idea of Spark

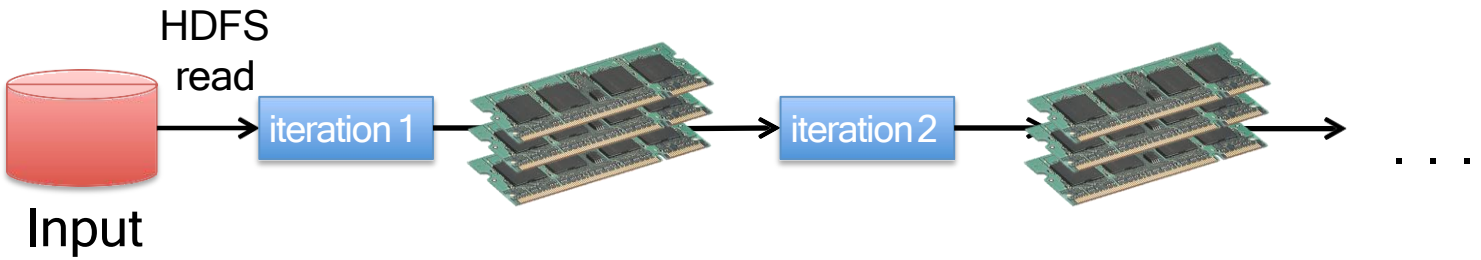
From MapReduce to Spark

- MapReduce: Iterative job



From MapReduce to Spark

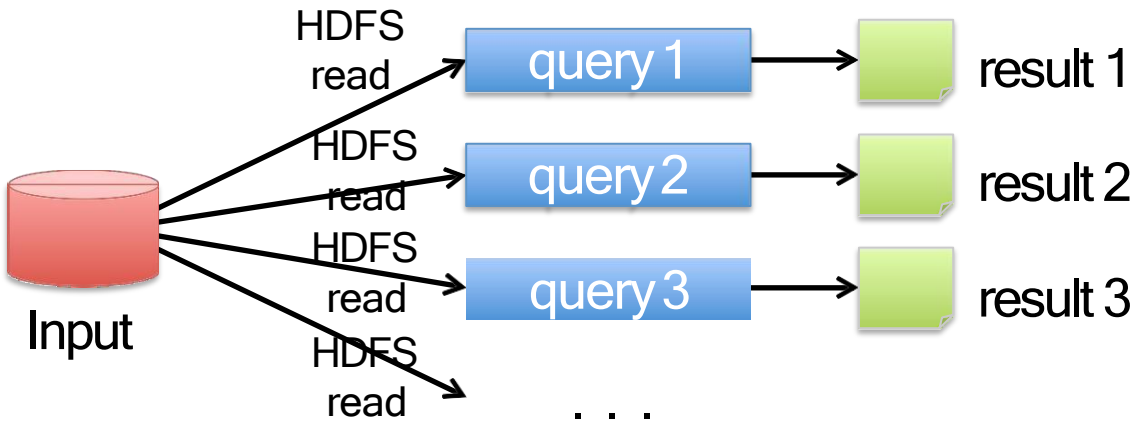
- Spark: Iterative job



- Data are shared between the iterations by using the main memory
  - Or at least part of them
- 10 to 100 times faster than disk

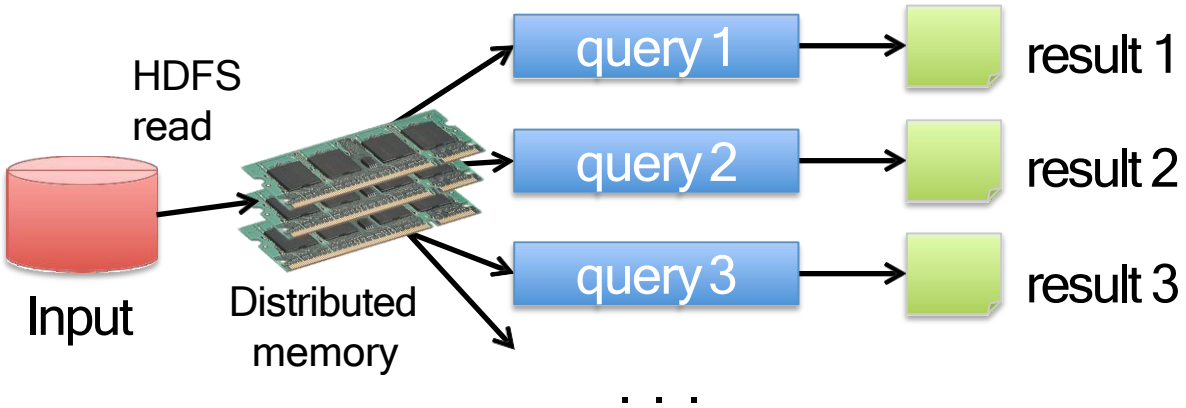
From MapReduce to Spark

- MapReduce: Multiple analyses of the same data



From MapReduce to Spark

- Spark: Multiple analyses of the same data



- Data are read only **once** from HDFS and stored in main memory
  - Split of the data across the main memory of each server

MapReduce vs Spark

	Hadoop MapReduce	Spark
Storage	Disk only	In-memory or on disk
Operations	Map and Reduce	Map, Reduce, Join, Sample, etc…
Execution model	Batch	Batch, interactive, streaming
Programming environments	Java	Scala, Java, Python, and R

Industry Adoption at Scale



JPMorgan Chase

Utilizes real-time, in-memory processing for sophisticated credit card fraud detection and identifying high-value investment patterns.



Healthcare Systems

Analyzes vast datasets of patient medical histories to proactively identify potential health issues and optimize diagnostic accuracy.

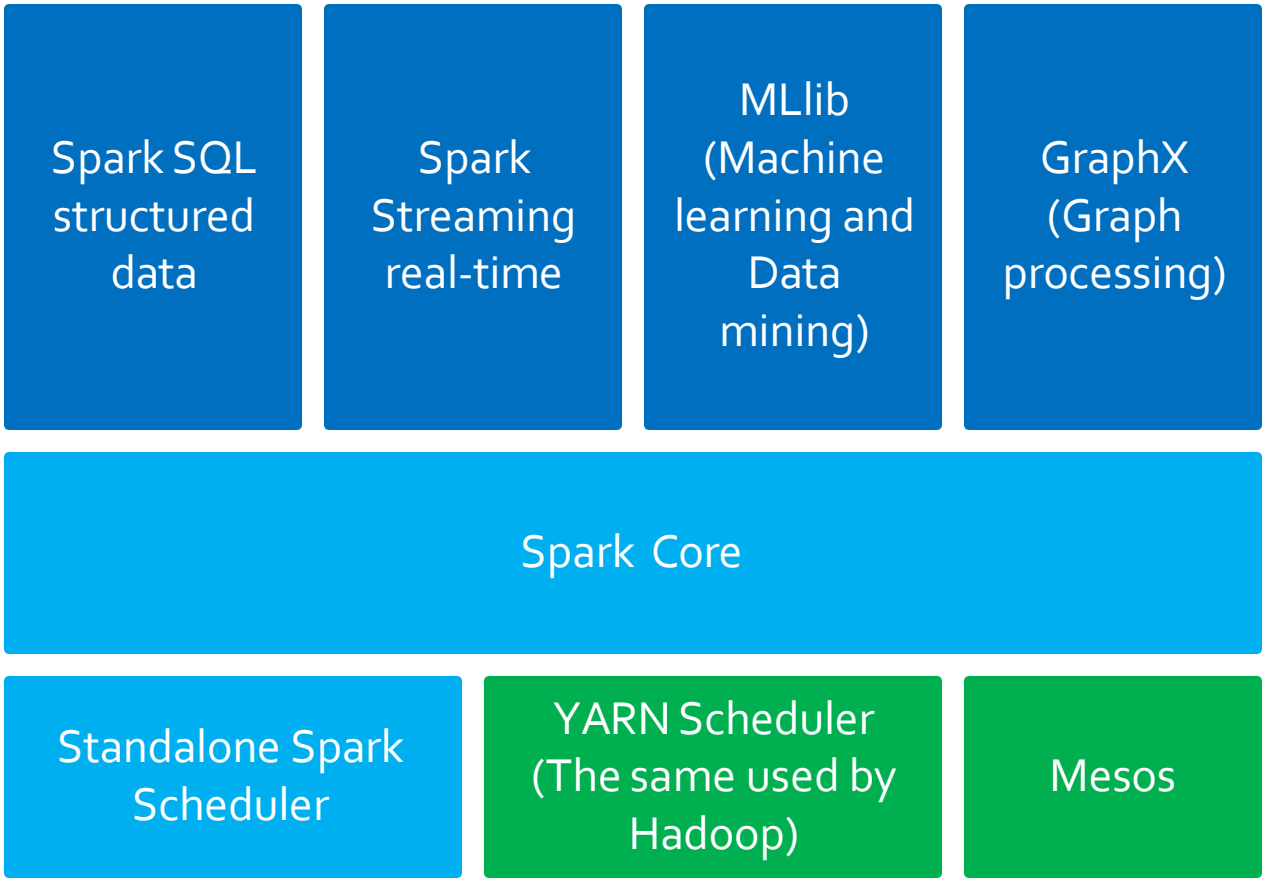


The Bottom Line

Apache Spark is the standard for organizations requiring highly resilient, low-latency analysis over massive datasets.

Spark: Main components

Spark Components



Spark Components

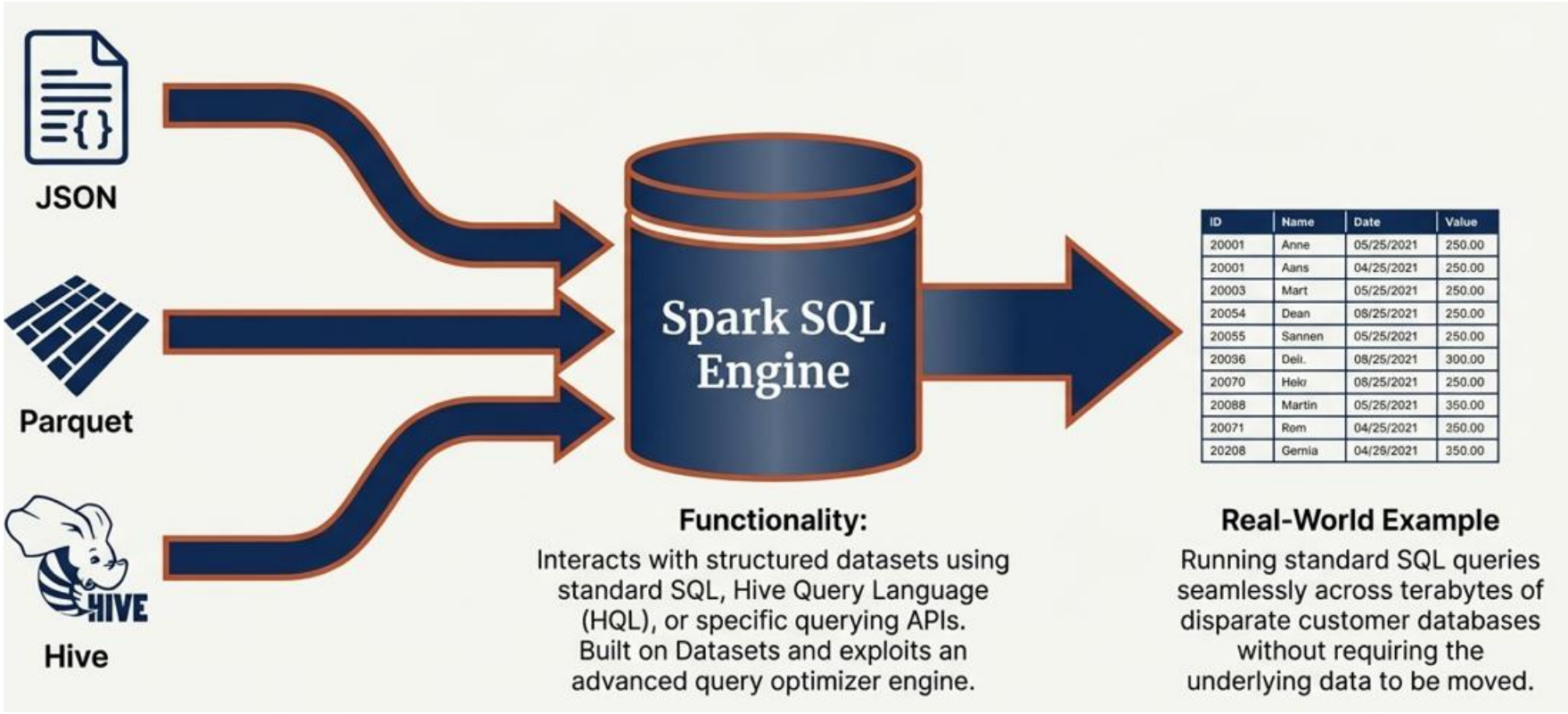
- **Spark Core**
  - Contains the basic functionalities of Spark exploited by all components
    - Task scheduling
    - Memory management
    - Fault recovery
    - …
  - Provides the APIs that are used to create RDDs and applies transformations and actions on them

Spark Components

- **Spark SQL structured data**
  - This component is used to **interact with structured datasets by means of the SQL** language or specific querying APIs
    - Based on Datasets
  - It supports also
    - Hive Query Language (HQL)
  - It interacts with many data sources
    - Hive Tables, Parquet, Json, ..
  - It exploits a query optimizer engine

Spark Components

- **Spark SQL structured data**

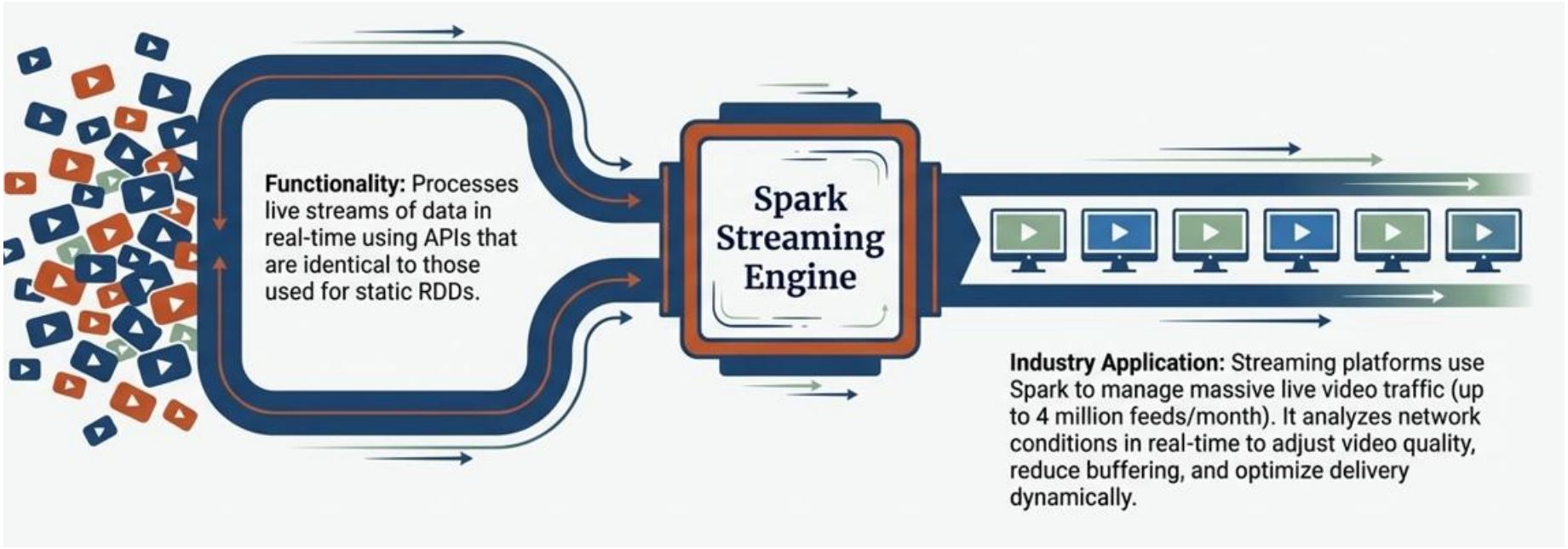


Spark Components

- **Spark Streaming real-time**
  - It is used to process live streams of data in real-time
  - The APIs of the Streaming real-time components operated on RDDs and are similar to the ones used to process standard RDDs associated with "static" data sources

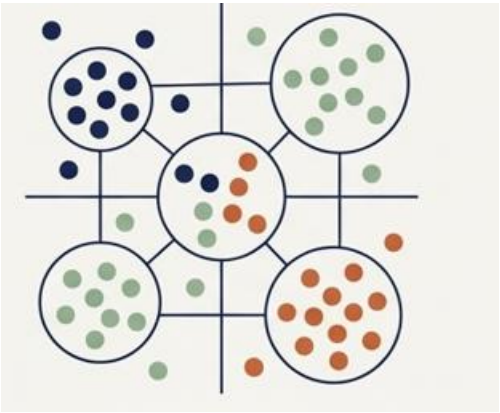
Spark Components

- **Spark Streaming real-time**



Spark Components

- **MLlib**
  - It is a machine learning/data mining library
  - It can be used to apply the parallel versions of some machine learning/data mining algorithms
    - Data preprocessing and dimensional reduction
    - Classification algorithms
    - Clustering algorithms
    - ….



Spark Components

- **GraphX**
  - A graph processing library
  - Provides algorithms for manipulating graphs
    - Subgraph searching
    - PageRank
    - ….



- GraphFrames
  - A graph library based on DataFrames and Python

Spark Schedulers

- Spark can exploit **many schedulers** to execute its applications
  - HadoopYARN
    - Standard scheduler of Hadoop
  - Mesos cluster
    - Another popular scheduler
  - Standalone Spark Scheduler
    - A simple cluster scheduler included in Spark

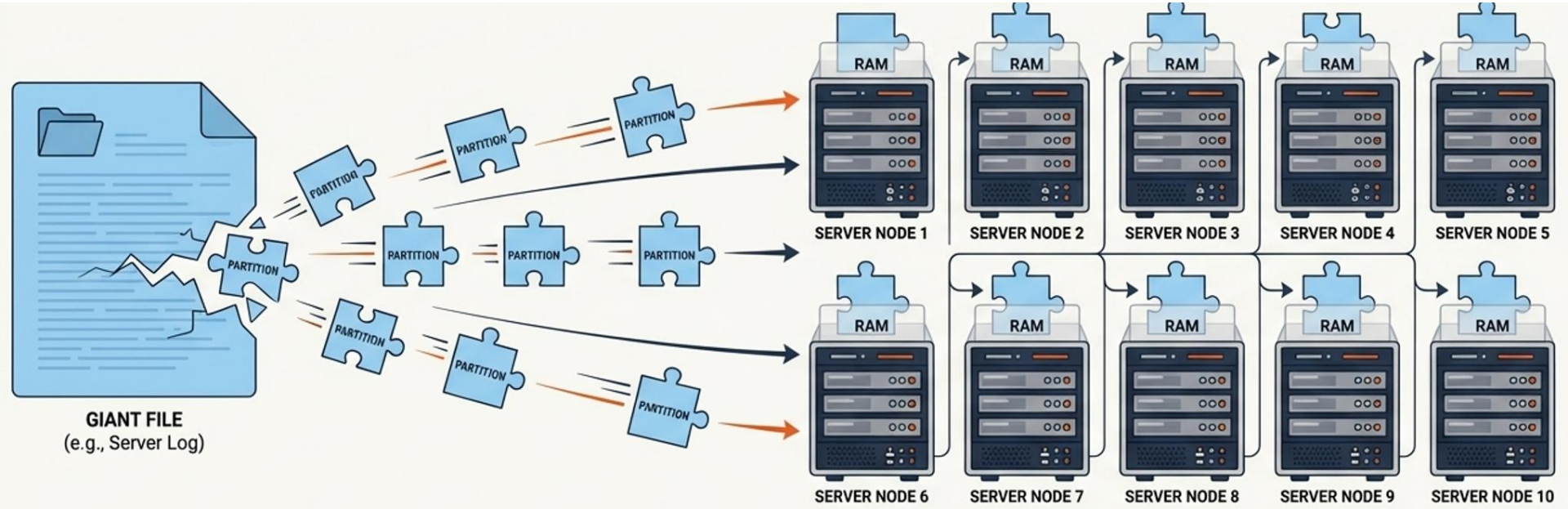
Spark Basic Concepts

Resilient Distributed Data sets (RDDs)

- RDDs are the primary abstraction in Spark
- RDDs are **distributed collections of objects spread across the nodes of a clusters**

- Resilient: Automatically recovers from computer failures.
- Distributed: Data is partitioned and spread across many machines.
- Dataset: A collection of your data, like transactions or logs.

Resilient Distributed Data sets (RDDs)



Definition

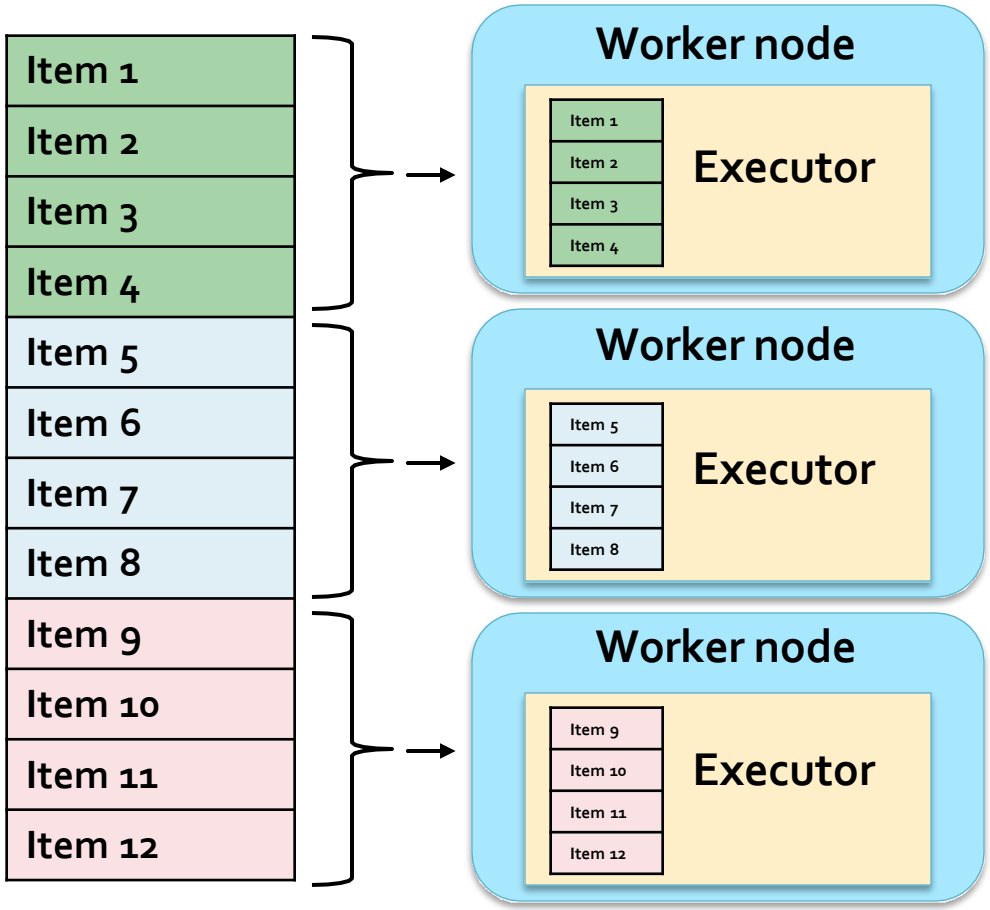
Partitioned and distributed collections of objects spread across the nodes of a cluster. Each node running an application contains at least one partition of the defined RDD.

Example Context

Think of a 1,000-page server log file. Instead of one computer reading it line by line, Spark splits it into 10 chunks (partitions) and distributes them across 10 different servers to be processed simultaneously.

Resilient Distributed Data sets (RDDs)

- Example of an RDD split in 3 partitions

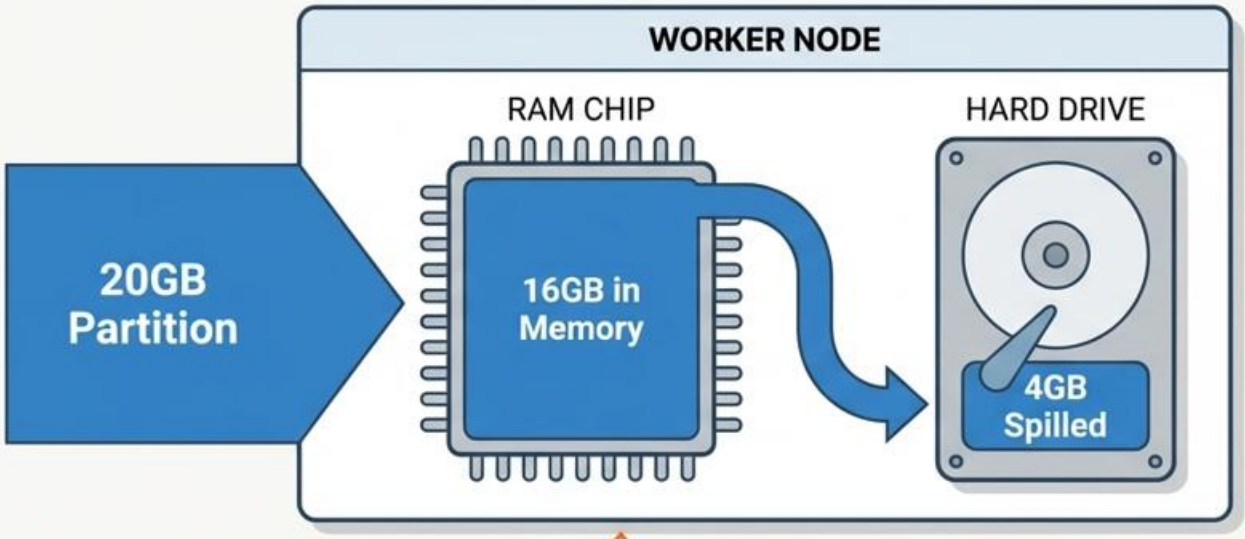


Resilient Distributed Data sets (RDDs)

- RDDs
  - Are **stored in the main memory** *of the executors running in the nodes of the cluster* (when it is possible) or in the local disk of the nodes if there is not enough main memory
    - Each executor of a worker node runs the specified code on its partition of the RDD

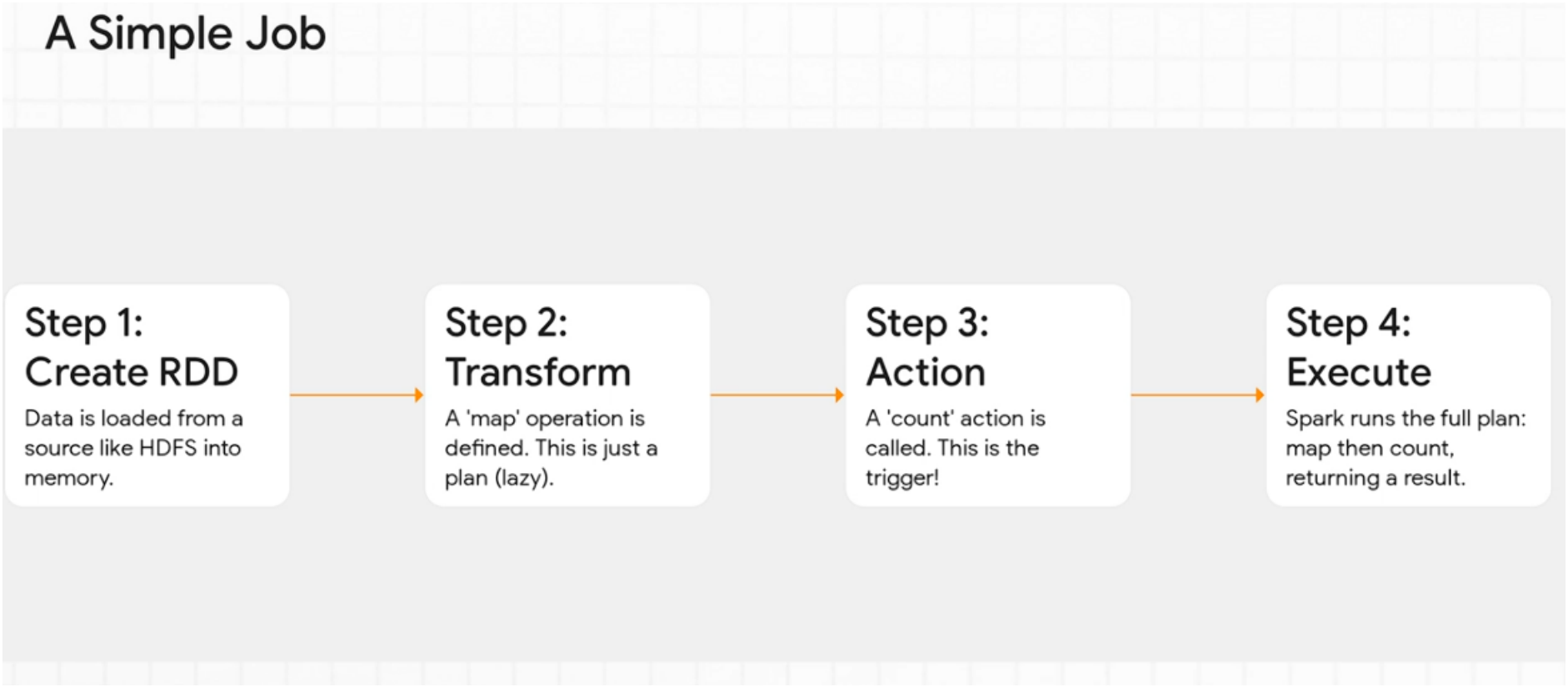
Resilient Distributed Data sets (RDDs)

RDDs are prioritized for main memory but adapt to hardware limitations. They are stored in the main memory of the executors running in the nodes of the cluster. If there is not enough main memory, they are stored on the local disk of the nodes. This is entirely separate from Local Variables, which can only store "small" objects limited by the main memory of the driver process.



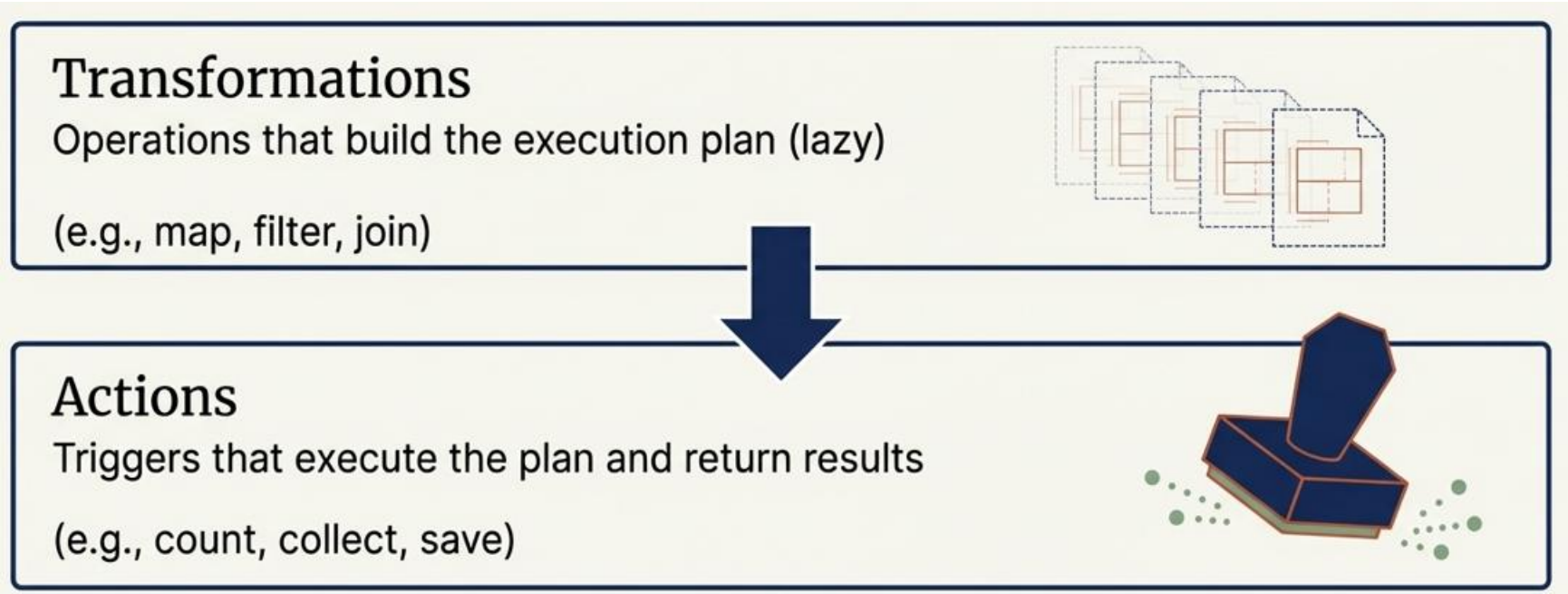
Example Context: A single server node has 16GB of RAM but is handed a 20GB partition of data to process. Spark gracefully holds 16GB in memory and spills the remaining 4GB to the local disk without crashing.

Resilient Distributed Data sets (RDDs)



Resilient Distributed Data sets (RDDs)

- Spark programs are written in terms of operations on resilient distributed data sets:



Resilient Distributed Data sets (RDDs)

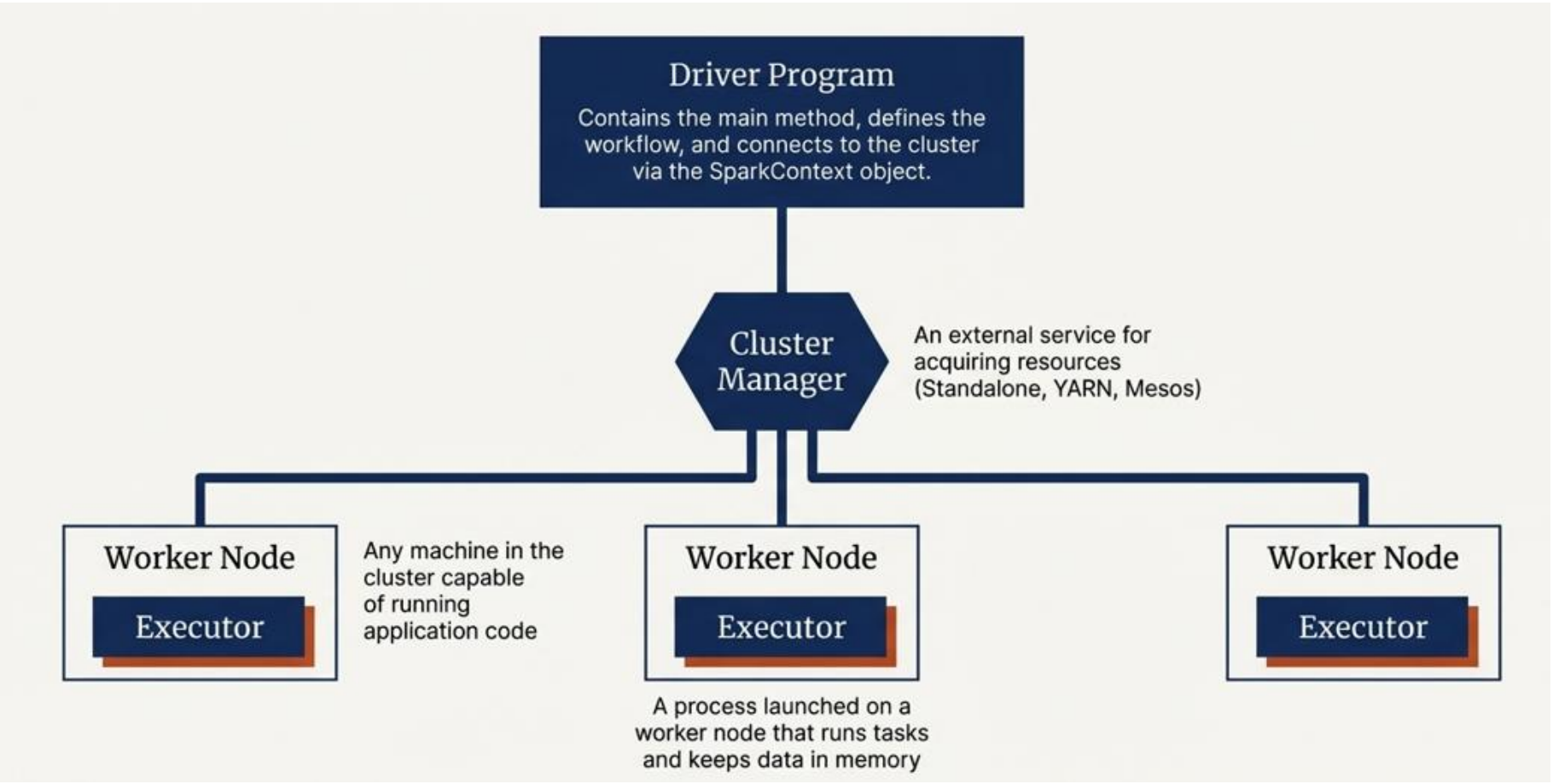
- RDDs
  - Are immutable once constructed
    - i.e., the content of an RDD cannot be modified
- Spark tracks lineage information to efficiently recompute lost data (due to failures of some executors)
  - i.e., **for each RDD, Spark knows how it has been constructed and can rebuilt it if a failure occurs**
  - This information is represented by means of a DAG (Direct Acyclic Graph) connecting input data and RDDs

Spark Programs

Supported languages

- Spark supports **many programming languages**
  - Scala
    - The same language that is used to develop the Spark framework and all its components (Spark Core, Spark SQL, Spark Streaming, MLlib, GraphX)
  - Java
  - Python
  - R

Structure of Spark programs



Structure of Spark programs

- The Driver program
  - Contains the main method
  - "Defines" the workflow of the application
  - Accesses Spark through the **SparkContext** object
    - The SparkContext object represents a connection to the cluster
  - Defines Resilient Distributed Datasets (RDDs) that are "allocated" in the nodes of the cluster

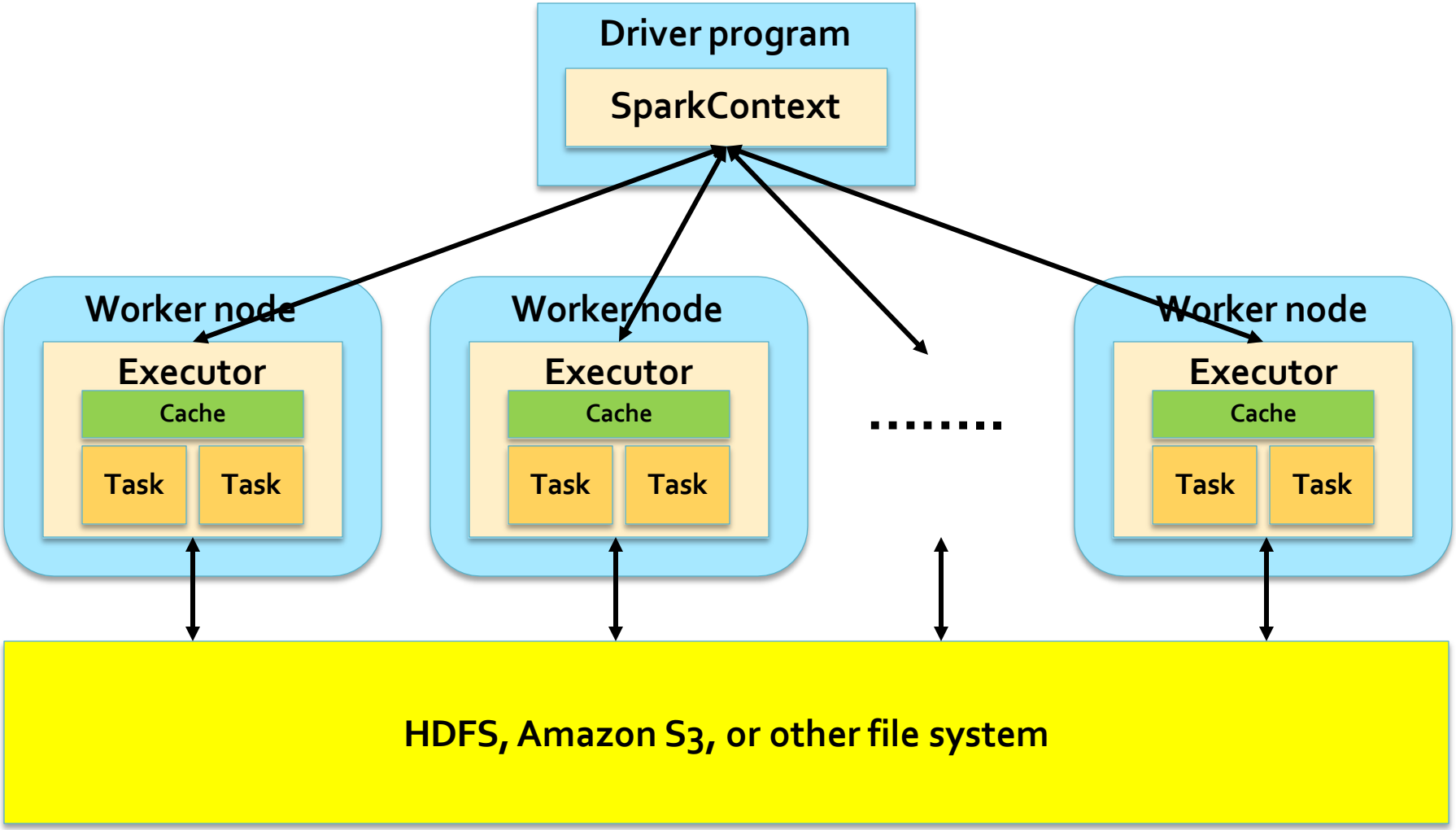
Structure of Spark programs

- The Driver program defines
  - **Local variables**
    - The standard variables of the Python programs
  - **RDDs**
    - Distributed "variables" stored in the nodes of the cluster
  - The SparkContext object allows
    - ***Creating RDDs***
    - "**Submitting**" executors (processes) that execute in parallel specific operations on RDDs
      - Transformations and Actions

Structure of Spark programs

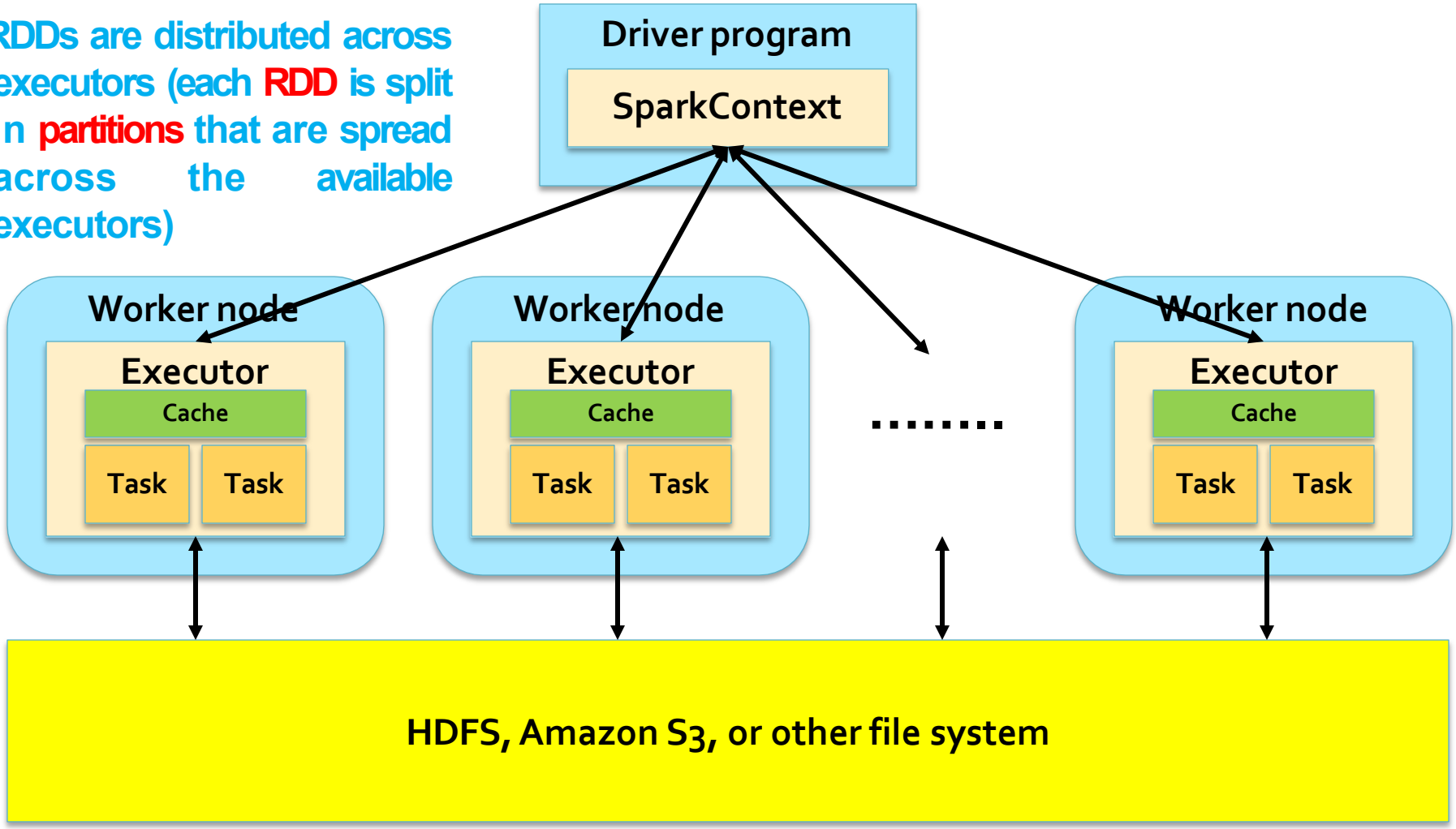
- The worker nodes of the cluster are used to run your application by means of **executors**
- Each executor runs on its partition of the RDD(s) the operations that are specified in the driver

Distributed execution of Spark



Distributed execution of Spark

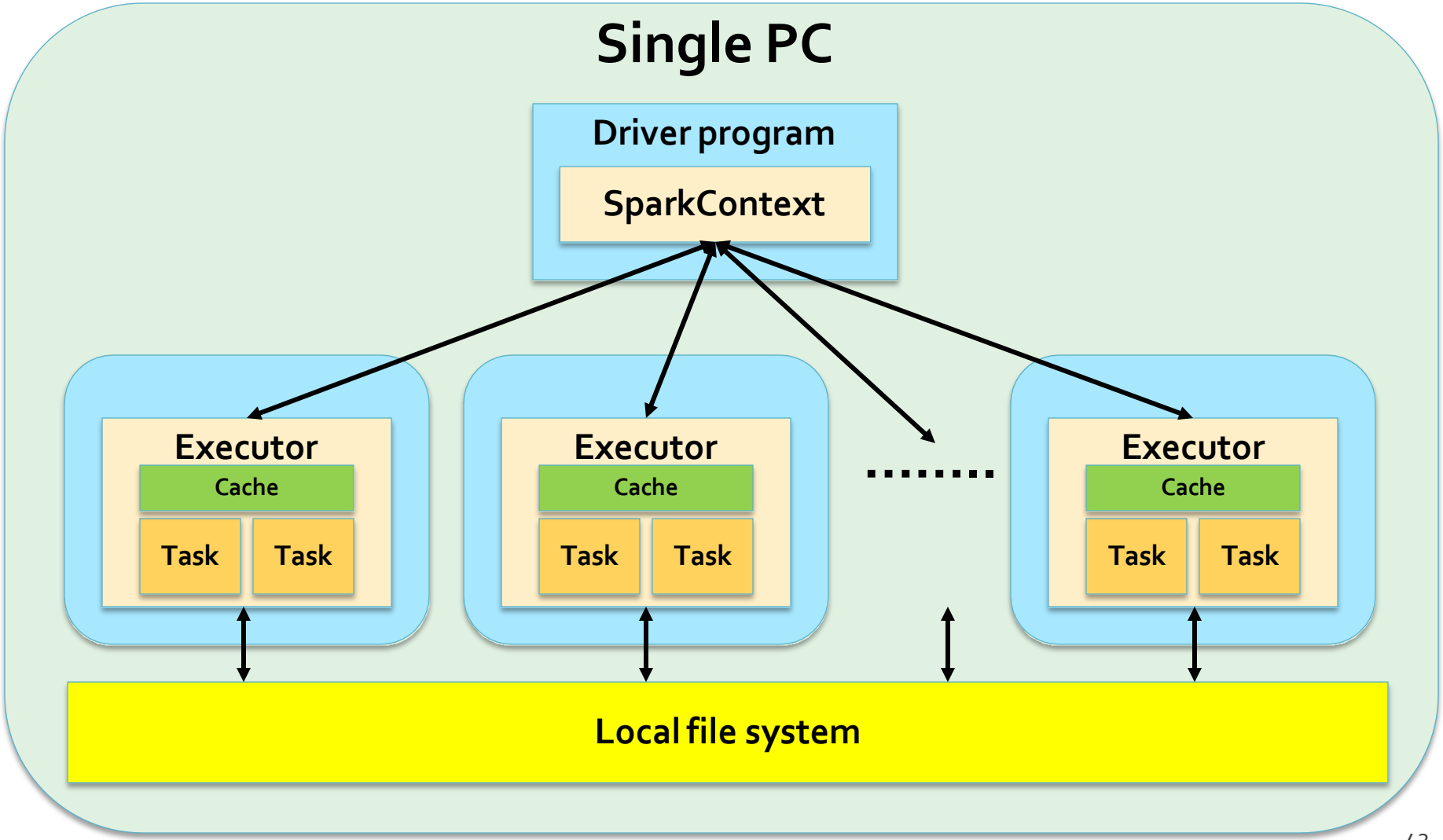
RDDs are distributed across executors (each RDD is split in partitions that are spread across the available executors)



Local execution of Spark

- Spark programs can also be executed locally
  - Local threads are used to parallelize the execution of the application on RDDs on a single PC
    - Local threads can be seen are "pseudo-worker" nodes
  - It is useful to develop and test the applications before deploying them on the cluster
  - A local scheduler is launched to run Spark programs locally

Local execution of Spark



Stages

- Stage
  - Each **job** gets **divided into** smaller sets of tasks called **stages**
  - The output of one stage is the input of the next stage(s)
    - Except the stages that compute (part of) the final result (i.e., the stages without output edges in the graph representing the workflow of the application)
      - The outputs of those stages is stored in HDFS or a database
  - The **shuffle operation** is always executed **between** two **stages**
    - Data must be grouped/repartitioned based on a grouping criteria that is different with respect to the one used in the previous stage
    - Similar to the shuffle operation between the map and the reduce phases in MapReduce
    - **Shuffle** is a **heavy operation**

Spark Programs: Examples

Spark program: Count line

- Count the number of lines of the input file
  - The name of the file is set to “myfile.txt”
- Print the results on the standard output

Spark program: Count line

```
from pyspark import SparkConf, SparkContext

if __name__== "__main__":

    #Create a configuration object and
    #set the name of the application
    conf = SparkConf().setAppName("Spark Line Count")

    # Create a Spark Context object
    sc = SparkContext(conf=conf)

    # Store the path of the input file in inputfile
    inputFile= "myfile.txt"
```

Spark program: Count line

```
# Build an RDD of Strings from the input textual file
 # Each element of the RDD is a line of the input file
linesRDD = sc.textFile(inputFile)

# Count the number of lines in the input file
# Store the returned value in the local variable numLines
numLines = linesRDD.count()
# Print the output in the standard output
print("NumLines:", numLines)

# Close the Spark Context object
sc.stop()
```

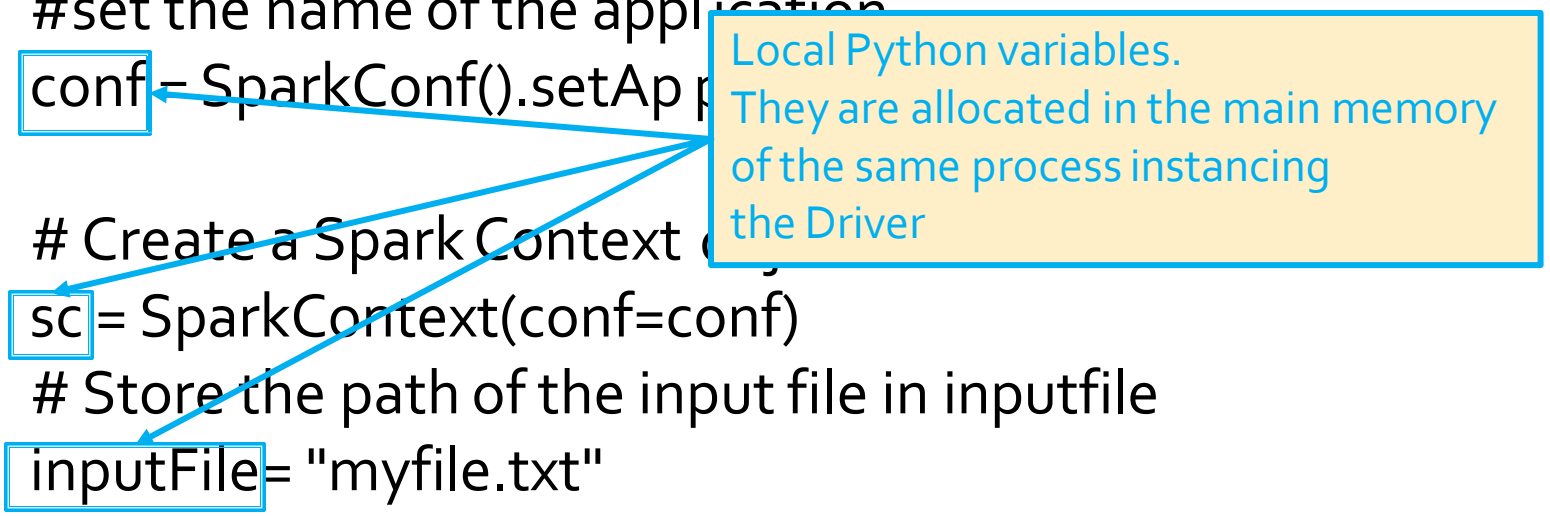
Spark program: Count line

```
from pyspark import SparkConf, SparkContext

if __name__ == "__main__":

    #Create a configuration object and
    #set the name of the appl ication
    conf = SparkConf().setAp p

    # Create a Spark Context
    sc = SparkContext(conf=conf)
    # Store the path of the input file in inputfile
    inputFile= "myfile.txt"
```



Spark program: Count line

```
# Build an RDD of Strings from the input textual file
# Each element of the RDD is a line of the input file
linesRDD = sc.textFile(input File)

# Count the number of lines
# Store the returned value i
numLines = linesRDD.count()
# Print the output in the standard output
print("NumLines:", numLines)

# Close the Spark Con
sc.stop()
```

RDD.
It is allocated/stored in the main memory or in the local disk of the executors of the worker nodes

Local Python variable.
It is allocated in the main memory of the same process instancing the Driver