Big Data Processing and Applications

Ioana G. CIUCIU
ioana.ciuciu@ubbcluj.ro

Lecture 8 – Introduction to Apache SPARK

Course structure

Date	Course/Week	Title
03.10.2025	1	Introduction to Data Science and Big Data (Part 1)
10.10.2025	2	Introduction to Data Science and Big Data (Part 2)
17.10.2025	3	Industrial standards for data mining projects. Big data case studies from industry – invited lecture from Bosch
24.10.2025	4	Data systems and the lambda architecture for big data
31.10.2025	5	Lambda architecture: batch layer
07.11.2025	6	Lambda architecture: serving layer
14.11.2025	7	Lambda architecture: speed layer
21.11.2025	8	Introduction to SPARK
28.11.2025	9	NoSQL Solutions for Big Data – invited lecture from UBB
05.12.2025	10	Presentation research essays
12.12.2025	11	Presentation research essays
19.12.2025	12	Data Ingestion
09.01.2026	13	Presentation research essays + Project Evaluation during Seminar
16.01.2026	14	Presentation research essays + Project Evaluation during Seminar

Slight modifications in the structure are possible

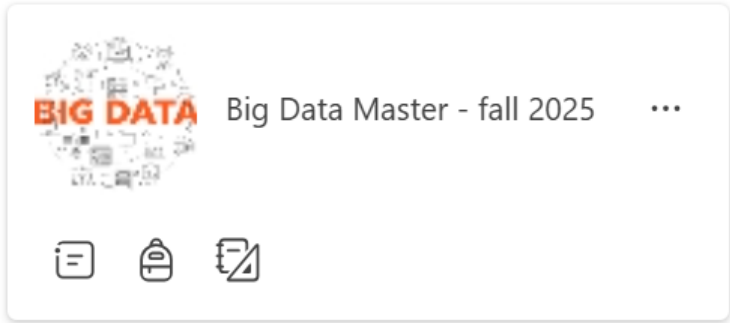
Semester Project

- Team-based (2-5 students with precise roles)
- Multidisciplinary: 3 CS Master Programes (HPC, SI, DS) + Master in Bioinformatics
- Real use cases: collaboration with local IT industry - TBA
- High degree of autonomy in selecting the topic and proposing the solution
- Implementation prototype
- Prototype demo & evaluation – student workshop (last seminar – weeks 13 & 14)
- *Best projects – disseminated in various events (TBD), scientifically disseminated (workshops/conferences/journals) AND/ OR possibility for a dissertation thesis*

Evaluation

- The final grade will be computed as follows:
  - 50% semester project (must be >= 5)
  - 50% research presentation or written exam (must be >= 5)

- Semester project
  - Details available on the course team
    - MS Team: **Big Data Master - fall 2025**
    - Access code: **j1exenq**



Agenda

Introduction to Apache SPARK

- What is Apache SPARK ?
- Spark Core
- Spark RDDs

What is Apache Spark?

- Apache Spark™ – a unified engine for large scale data analytics
  - standard tool for any developer or data scientist interested in big data
  - supports multiple widely used programming languages (Python, Java, Scala, and R)
  - includes libraries for diverse tasks ranging from SQL to streaming and machine learning
  - runs anywhere from a laptop to a cluster of thousands of servers

➔ an easy system to start with and scale-up to big data processing at incredibly large scale

What is Apache Spark?

- Apache Spark™ – a unified engine for large scale data analytics
  - Originates in a research project at AMPLab UC Berkeley
  - Led by Dr. Matei Zaharia (Romanian)
  - Kick-off paper:
    - "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing" (https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf)
  - Version 1.0 on May 2014 -> Version 4.0.1 (currently)
  - Open sourced (Databricks)
  - Written in Scala (using Functional Programming principles)

➔ an easy system to start with and scale-up to big data processing at incredibly large scale

What is Apache Spark?

- Apache Spark™ – a unified engine for large scale data analytics
  - Similar to Hadoop (a distributed, general-purpose computing platform)
  - Allows for keeping large amounts of data in memory
  - Offers tremendous performance improvements
    - Spark programs can be 100 times faster than their MapReduce counterparts!
  - Combines
    - Map-Reduce-like capabilities for batch programming
    - Real-time data-processing functions
    - SQL-like handling of structured data
    - Graph algorithms
    - Machine Learning

➔ an easy system to start with and scale-up to big data processing at incredibly large scale

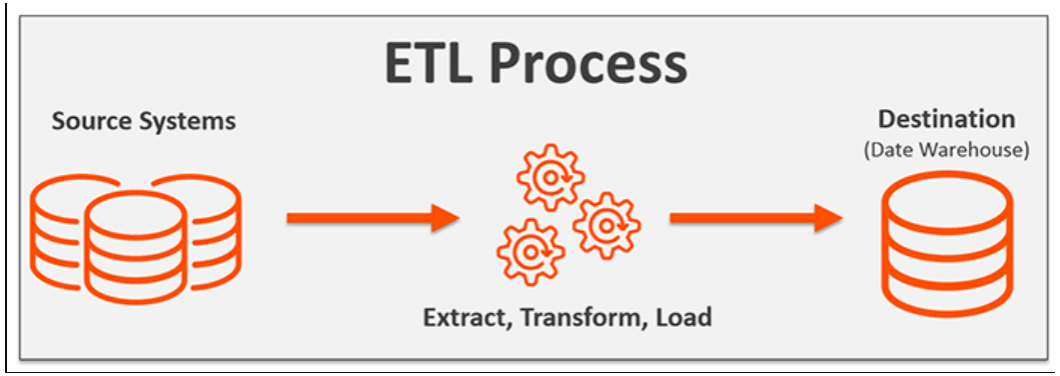
What is Apache Spark?

- The Spark Revolution
  - Three main challenges of distributed data processing:
    - Parallelization
    - Distribution
    - Fault tolerance
  - Solution Hadoop (HDFS, MapReduce)
  - Spark's core concept: in-memory execution model → speed the job execution 100 times as compared to e.g., Hadoop (MapReduce)
    - Biggest effect: on iterative algorithms (ML, graph algorithms, etc.)

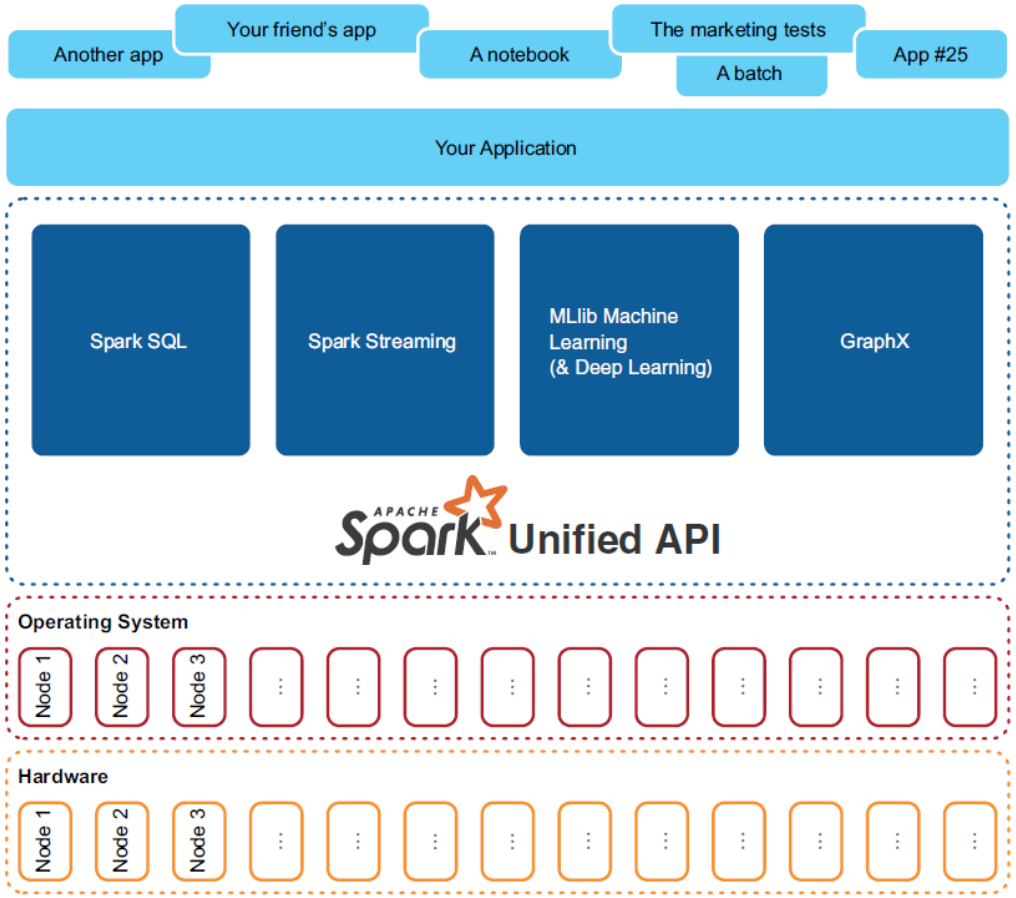
What is Apache Spark?

- Why an analytics framework?
- Every data processing/transformation in high scale needs to follow ETL/ELT process

- Spark is a key framework for Transform/Load
  - Data cleaning / transformation
  - Data aggregations
  - Data quality
  - Data preparation for being consumed



Apache Spark Components



Spark SQL - to run data operations, like traditional SQL jobs in an RDBMS. Spark SQL offers APIs and SQL to manipulate your data.

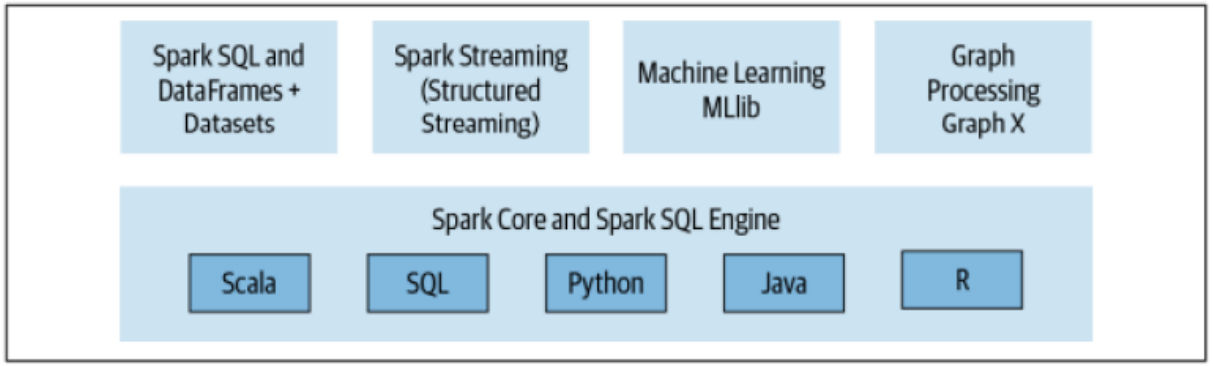
Spark Streaming, and specifically Spark structured streaming, to analyze streaming data. Spark's unified API will help you process your data in a similar way, whether it is streamed data or batch data.

Spark MLlib - for machine learning and recent extensions in deep learning.

GraphX - to exploit graph data structures.

Apache Spark Components

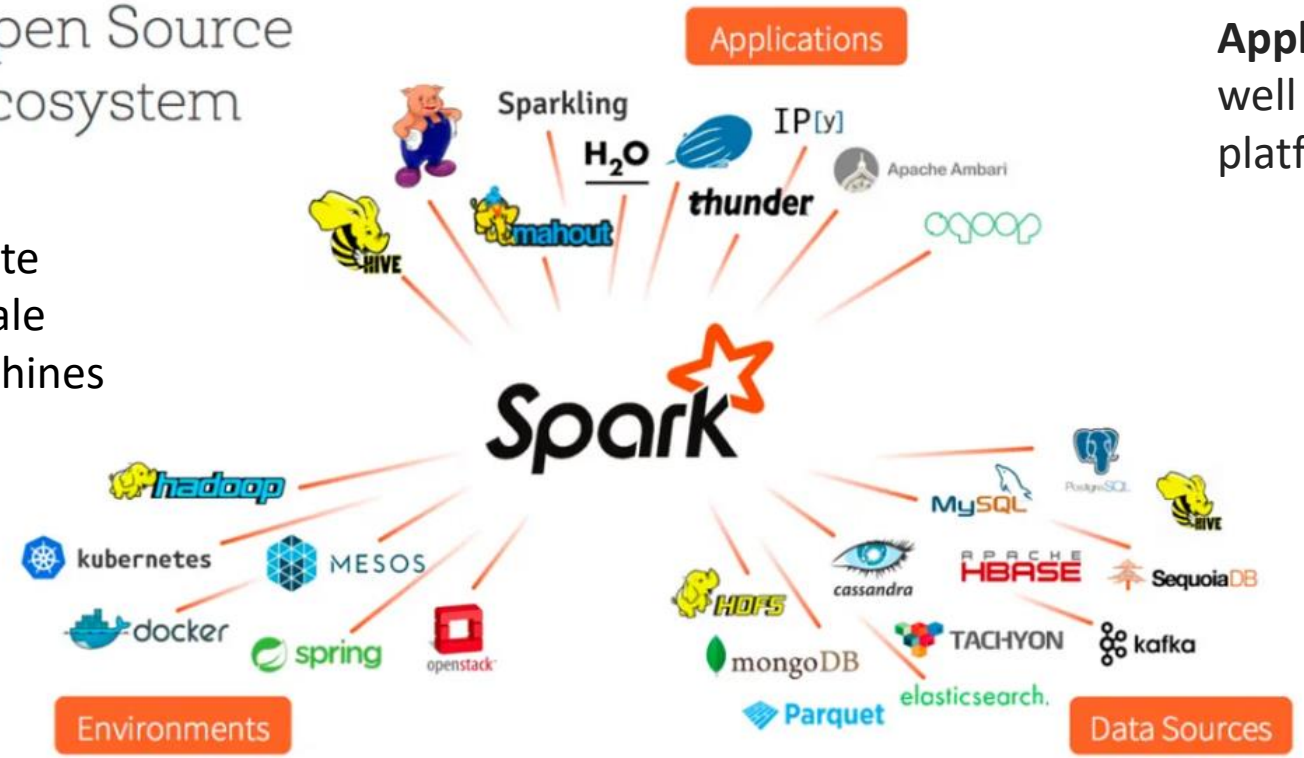
- General framework:
  - Core functionality (Internal processing & RDD)
  - Spark SQL (DataFrames), MLlib, GraphX, Streaming



- Multi-platform compatibility:
  - Standalone (client-mode)
  - On top cluster resources managers (Apache Mesos, YARN)
  - Containerised enviroments (Kubernetes)
  - Cloud-compatibility (Azure Databricks, AWS EMR, GCP)

- Key functionalities:
  - Distribute computation & data, in a fault tolerant and efficient way.
  - Multi-stage in-memory computations.
  - Cost-based Execution of code (Codegen) / Plans

Apache Spark Components



Ecosystem: Apache Spark integrates with your favorite frameworks, helping to scale them to thousands of machines

Applications: Spark integrates well with a variety of big data platforms and applications.

Environments: Spark can run anywhere and integrates well with other environments.

Data sources: Spark can read and write data from and to many data sources.

Apache Spark Components

- APIs

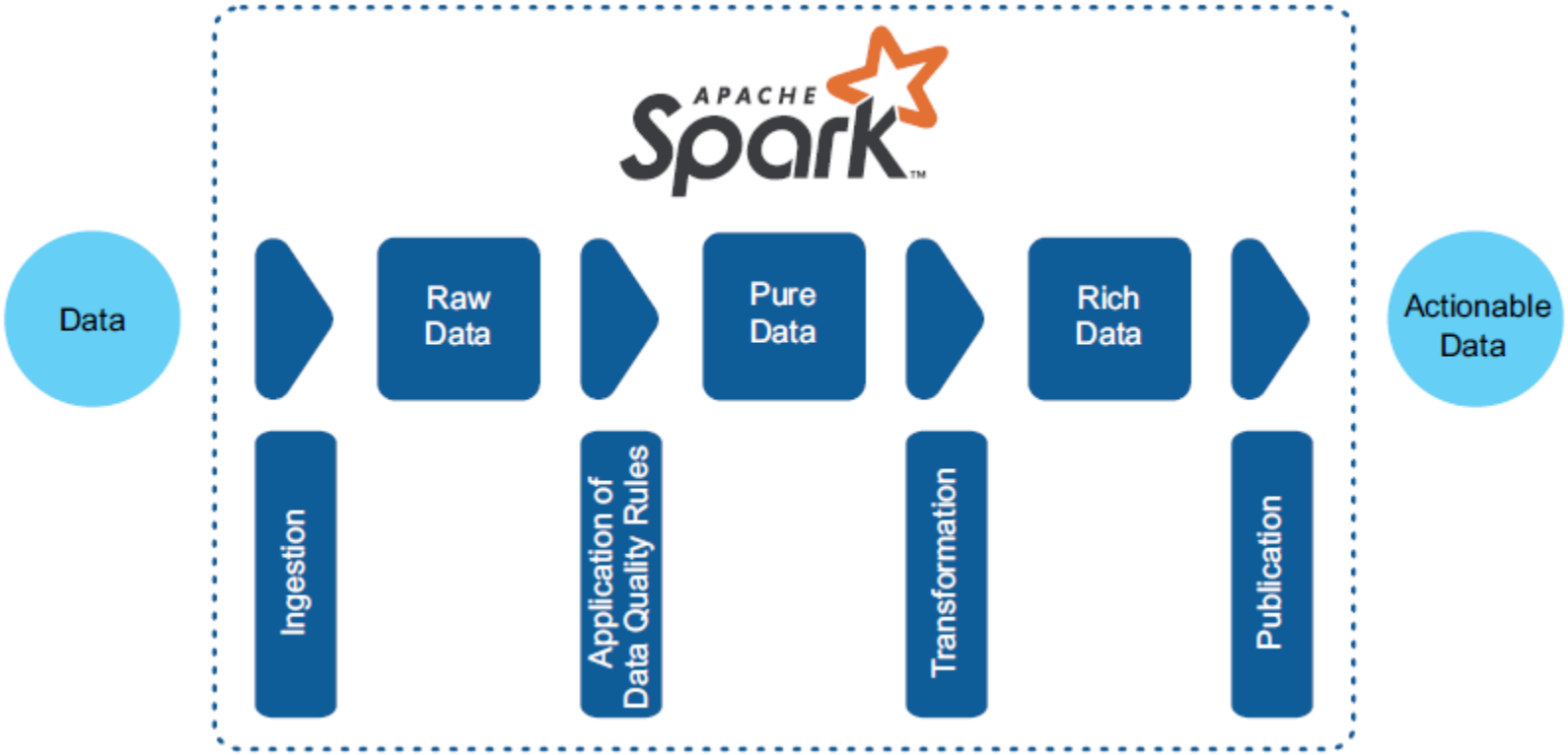
- Spark has multiple APIs and running approaches:
  - Scala / Python / Java / R
  - Standalone (client mode, own laptop) or Cluster mode

- Scala:
  - REPL (Read –Evaluate –Print –Loop) : Interactive Spark Shell
  - Compile / Assembly application into executable .jar and distribute to cluster (spark-submit)

- Python:
  - Interactive shell / Jupyternotebook
  - Use your own conda/venv
  - .pyfiles / .egg / .whlcan be submitted to a cluster

How can you use Spark?

- Spark in a Big Data processing / engineering scenario

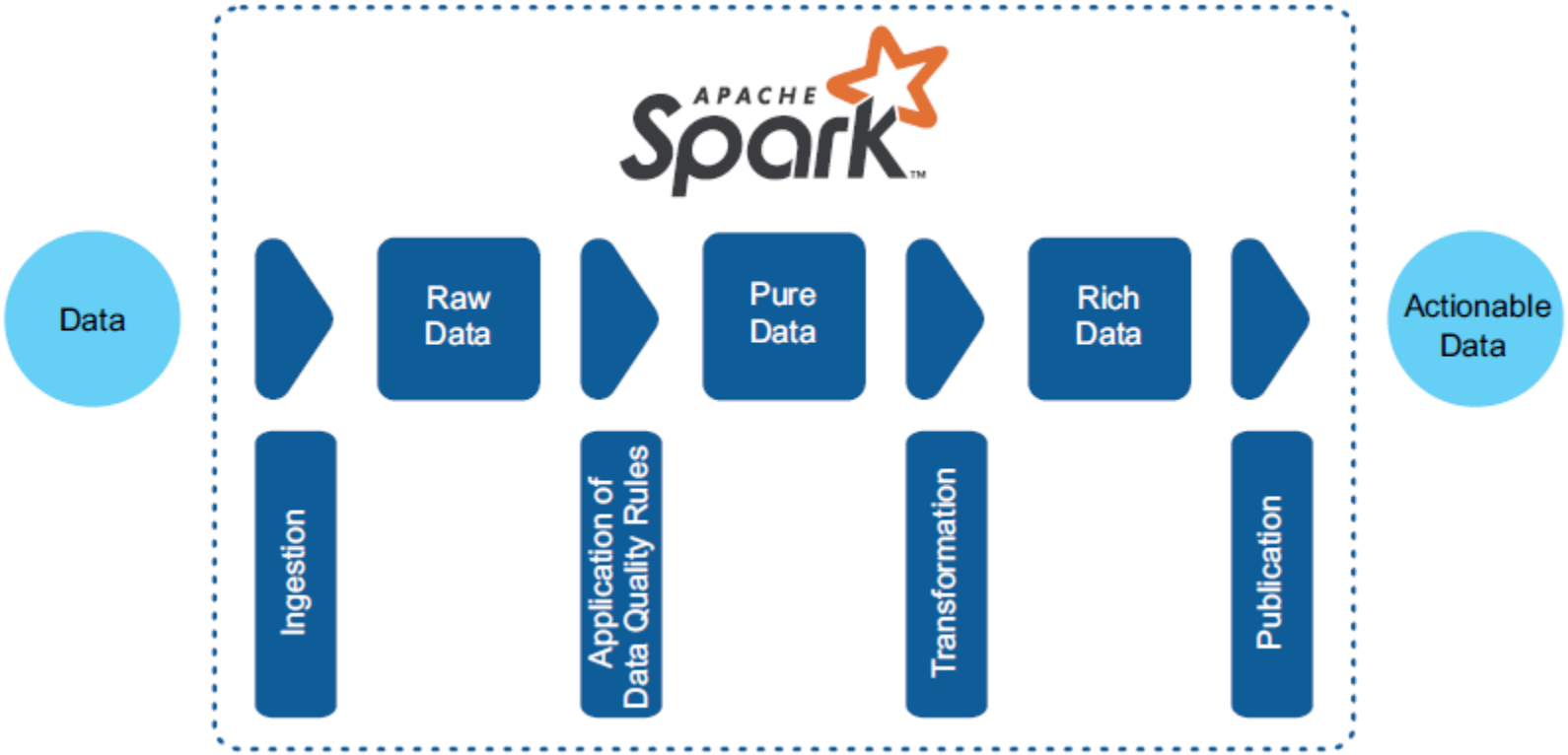


The **four** steps of a typical Spark (big data) scenario performed by data engineering are as follows:

1. Ingestion
2. Improvement of data quality (DQ)
3. Transformation
4. Publication

How can you use Spark?

- Spark in a Big Data processing / engineering scenario

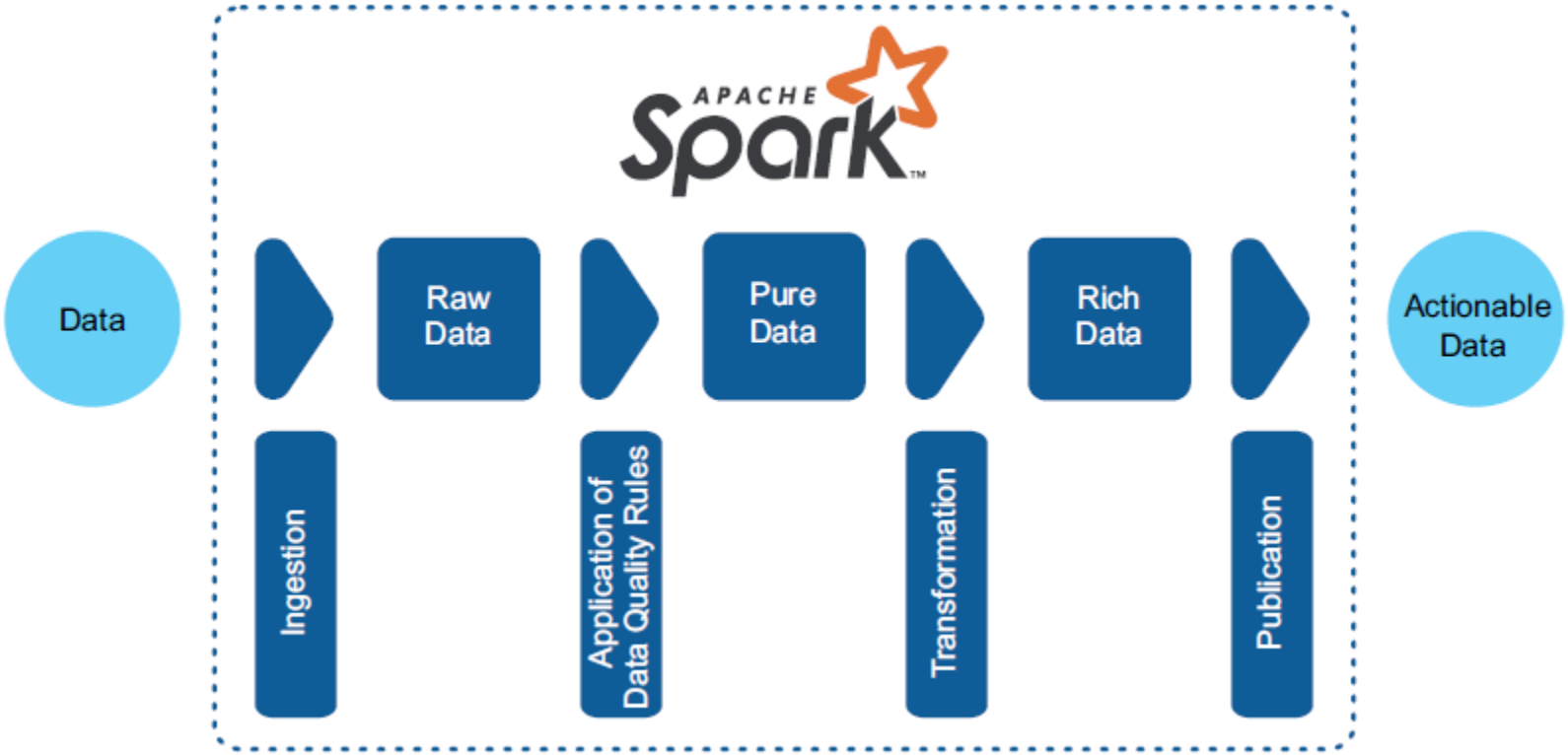


The **four** steps of a typical Spark (big data) scenario performed by data engineering are as follows:

1. **Ingestion:** Spark can ingest data from a variety of sources. If you can't find a supported format, you can build your own data sources. Data at this stage is called *raw data*. You can also find this zone named the *staging*, *landing*, *bronze*, or even *swamp zone*

How can you use Spark?

- Spark in a Big Data processing / engineering scenario

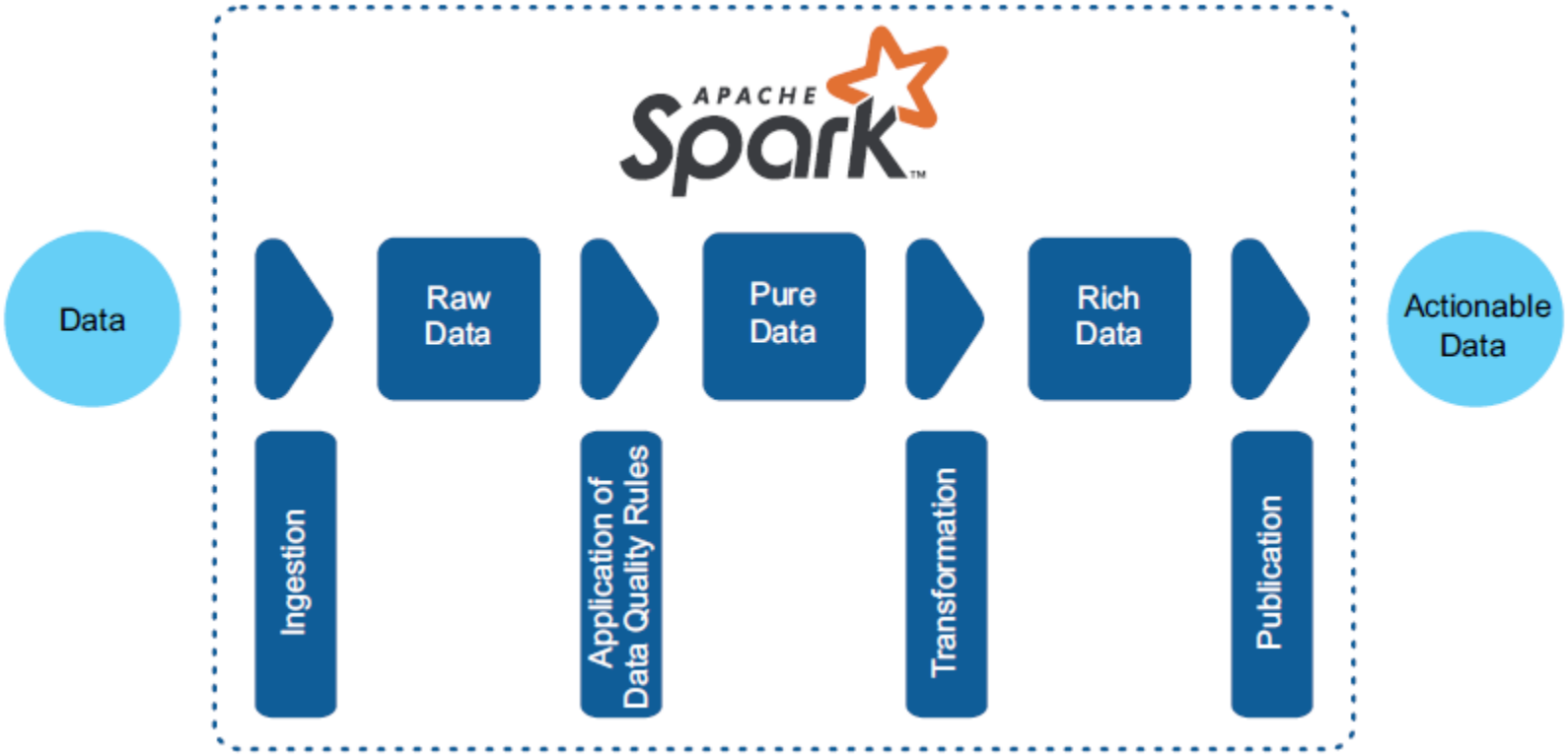


The **four** steps of a typical Spark (big data) scenario performed by data engineering are as follows:

2. Improvement of data quality (DQ) - Before processing your data, you may want to check the quality of the data itself. An example of DQ is to ensure that all birth dates are in the past. As part of this process, you can also elect to obfuscate some data: if you are processing Social Security numbers (SSNs) in a health-care environment, you can make sure that the SSNs are not accessible to developers or nonauthorized personnel. After your data is refined, this stage is called the *pure data zone*. You may also find this zone called the *refinery*, *silver*, *pond*, *sandbox*, or *exploration zone*.

How can you use Spark?

- Spark in a Big Data processing / engineering scenario

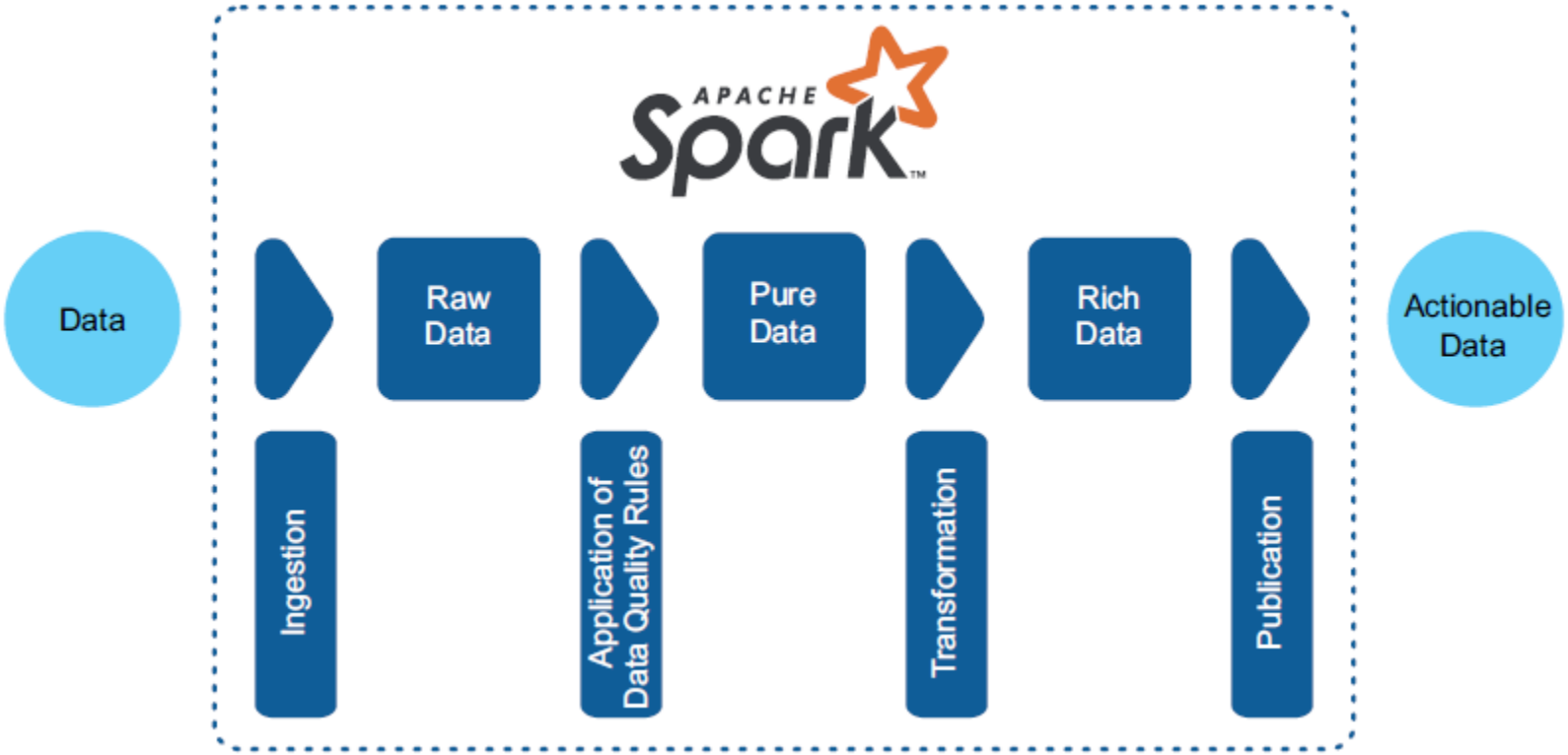


The **four** steps of a typical Spark (big data) scenario performed by data engineering are as follows:

3. Transformation: The next step is to process your data. You can join it with other datasets, apply custom functions, perform aggregations, implement machine learning, and more. The goal of this step is to get *rich data*, the fruit of your analytics work. This zone may also be called the *production*, *gold*, *refined*, *lagoon*, or *operationalization zone*.

How can you use Spark?

- Spark in a Big Data processing / engineering scenario

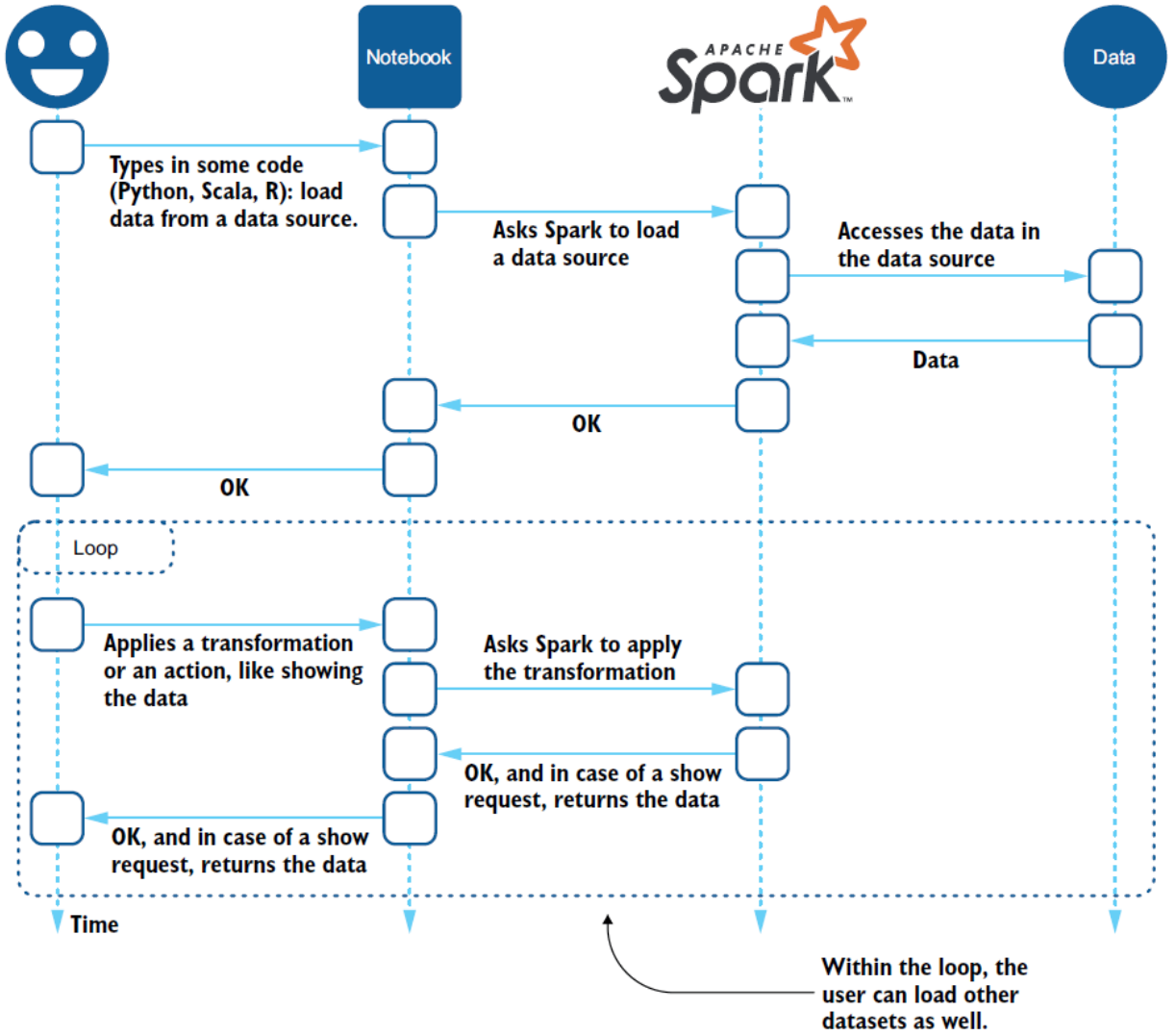


The **four** steps of a typical Spark (big data) scenario performed by data engineering are as follows:

4. Publication: As in an ETL process, you can finish by loading the data into a data warehouse, using a business intelligence (BI) tool, calling APIs, or saving the data in a file. The result is *actionable data* for your enterprise

How can you use Spark?

- Spark in a data science scenario
  - Slightly different approach than for data/software engineers
  - Focus is on the *transformation* part, in an interactive manner
  - DSs use notebooks (Jupyter, Zeppelin, IBM Watson Studio, Databricks Runtime)
  - DS projects will consume enterprise data, and therefore you may end up delivering data to the data scientists, off-loading their work (such as machine learning models) into enterprise data stores, or industrializing their findings.



What can you do with Spark?

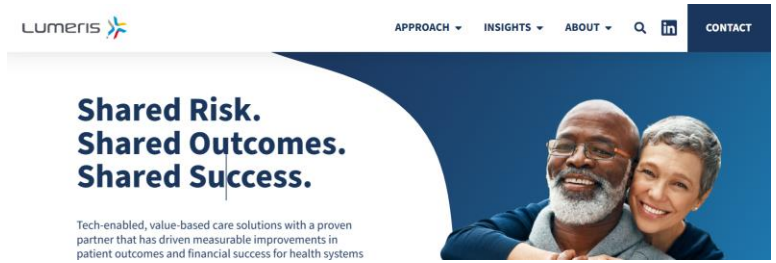
- Big Data projects
  - Data that cannot fit on a single computer
  - Require a cluster of computers
  - The need for a distributed OS, specialized in analytics

What can you do with Spark?

- Spark (real world) use cases
  - **UC1: *Spark predicts restaurant quality at North Carolina eateries***
    - Restaurant inspections by local health departments
    - Restaurants are graded based on these inspections (a 0 to 100 scale)
    - Grades measures indicators such as the cleanliness of the kitchen, how safely food is stored, etc.
    - Each county offers access to the restaurant's grade (no central location for accessing the information statewide)
    - NCEatery.com - a consumer-oriented website; lists restaurants with inspection grades over time
      - Aim: to centralize information and run predictive analytics on restaurants (ex: "Is this place I loved two years ago going downhill?")
      - Backend: Apache Spark ingests datasets of restaurants, inspections, and violations data coming from different counties
      - crunches the data, and publishes a summary on the website
      - During the crunching phase, several data quality rules are applied, as well as machine learning to try to project inspections and scores.
      - Spark processes 1.6 × 1021 datapoints and publishes about 2,500 pages every 18 hours using a small cluster.
      - This ongoing project is in the process of onboarding more NC counties.

What can you do with Spark?

- Spark (real world) use cases
  - **UC2: *Spark allows fast data transfer for Lumeris***
    - *Lumeris* - an information-based health-care services company, based in St. Louis, Missouri
    - It has traditionally helped health-care providers get more insights from their data
    - Need: IT system needed a boost to accommodate more customers and drive more powerful insights from the data it had
    - Data engineering processes:
      - Apache Spark ingests thousands of comma-separated values (CSV) files stored on Amazon Simple Storage Service (S3),
      - Builds health-care-compliant HL7 FHIR resources,
      - Saves them in a specialized document store where they can be consumed by both the existing applications and a new generation of client applications.
    - This technology stack allows *Lumeris* to continue its growth, both in terms of *processed data* and *applications*

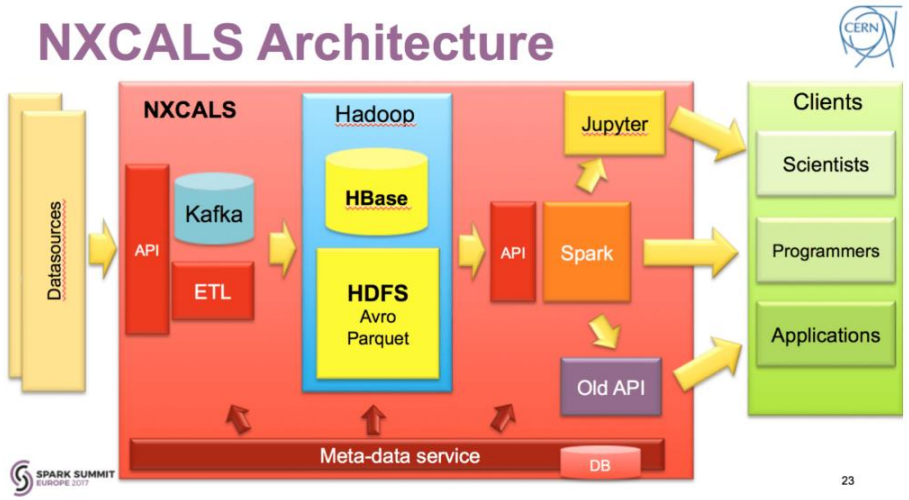


Source: https://www.lumeris.com/

What can you do with Spark?

- Spark (real world) use cases
  - **UC3: *Spark analyzes equipment logs for CERN***

    - CERN, or the European Organization for Nuclear Research, was founded in 1954
    - It is home to the Large Hadron Collider (LHC), a 27-kilometer ring located 100 meters under the border between France and Switzerland, in Geneva
    - The giant physics experiments run there generate 1 petabyte (PB) of data per second
      - After significant filtering, the data is reduced to 900 GB per day
    - CERN team designed the Next CERN Accelerator Logging Service (NXCALS) around Spark
      - On an on-premises cloud running OpenStack with up to 250,000 core
      - Consumers of the architecture:
        - Scientists (through custom applications and Jupyter notebooks)
        - Developers,
        - Applications
    - Ambition: to onboard even more data and increase the overall velocity of data processing



Source: https://www.databricks.com/blog/2017/12/14/the-architecture-of-the-next-cern-accelerator-logging-service.html

Anatomy of a Spark Application

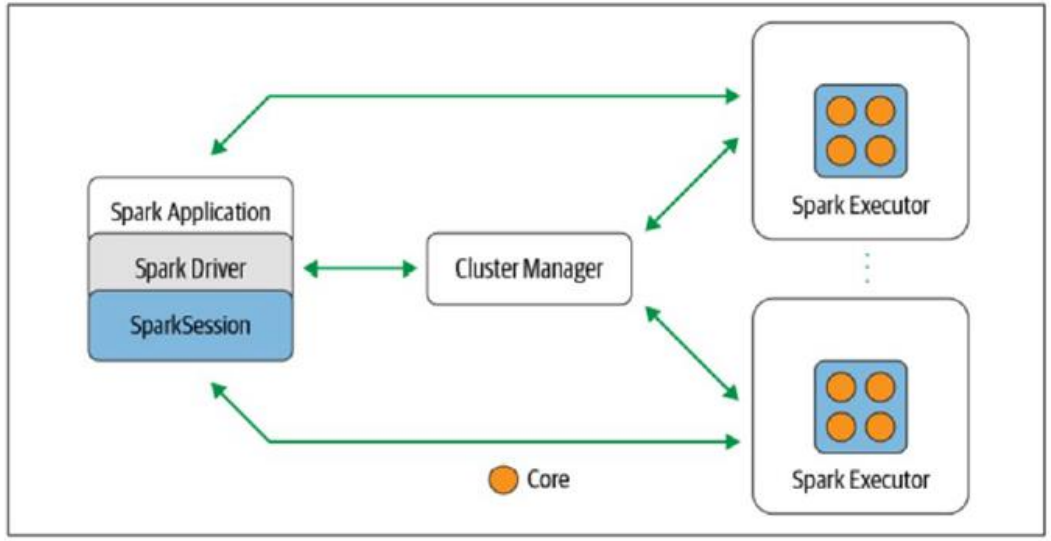
- One **driver**, multiple **executors**

Spark Driver runs the Spark application, which creates a SparkSession upon start-up.

The SparkSession connects to a **cluster-manager** which allocates resources.

Spark acquires executors on nodes in the cluster.

Spark Driver sends your application code to the executors and SparkSession sends the tasks.



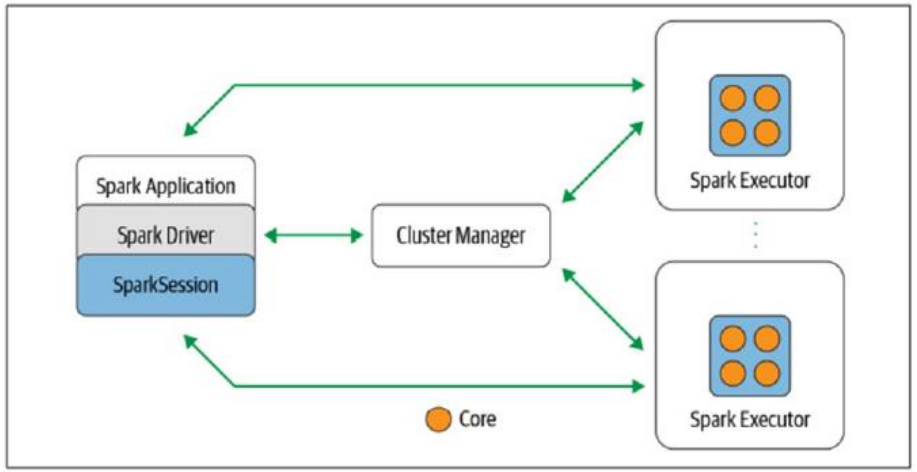
Anatomy of a Spark Application

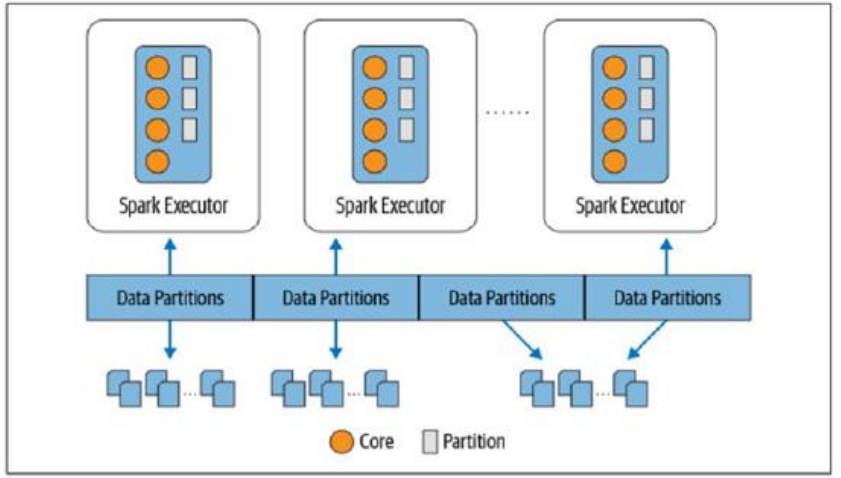
- One **driver**, multiple **executors**

Executors - are the entities where the processing, shuffling and distributed persistence take place.

Eventually, all the executors return their results to the driver.

Both driver and executors are JVM instances. A worker node (physical machine) can contain multiple executors.

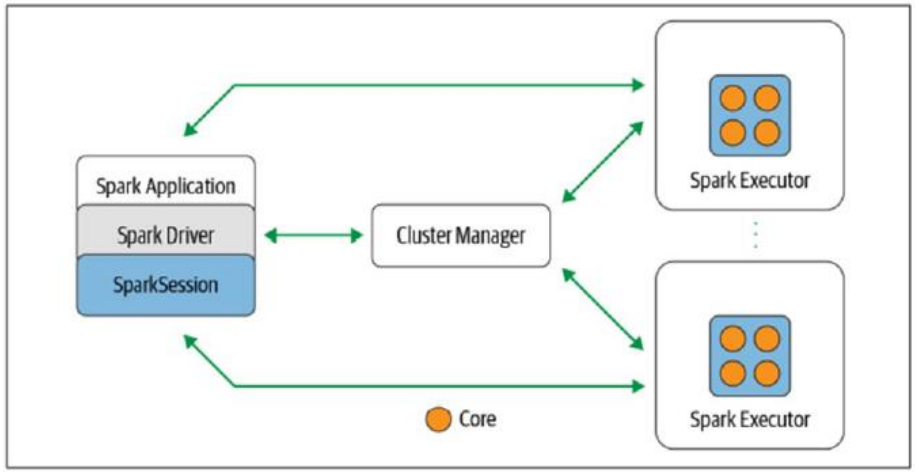




Anatomy of a Spark Application

SparkSession:

- Main entry point between the app and the cluster
- 1 active instance per app in JVM
- A Spark application is divided into *“job”, “stage”, ”task”*

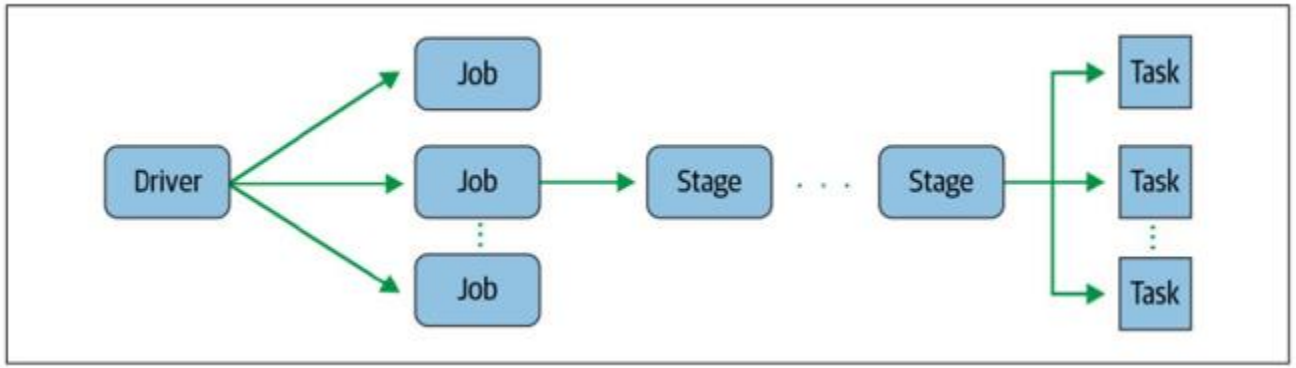


Job:

- is sequence of *transformations* initiated by *action* on data (e.g. count)
- consists of one or more *stages*

Stage:

- is group of *tasks* separated by shuffle (of data)



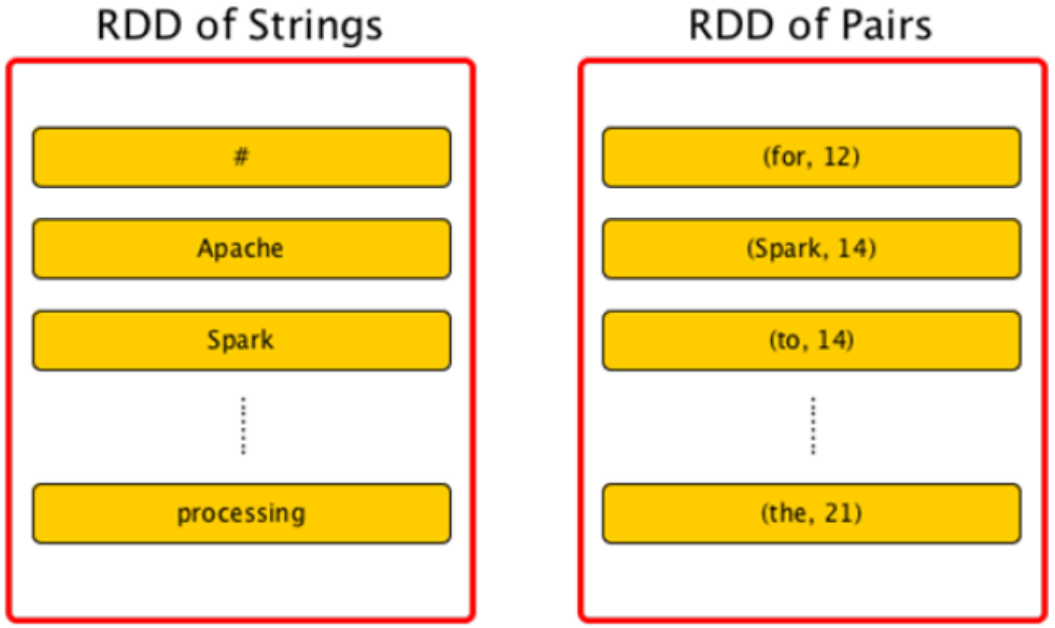
Task:

- is a unit of work to be sent to executors for either performing computation and/or cache data.

Sparc Core - RDD

- In Spark, data are represented as RDDs

- **R**esilient **D**istributed **D**atasets
  - Immutable and partitioned collection of **typed** objects.
  - Reside (mostly) in memory.
  - Distributed across the nodes.
  - Can be cached or stored temporarily on disk and may depend on zero or more RDDs.



Sparc Core - RDD

- **RDD Operations**

- **Transformations:**
- Create new RDDs from existing ones.
- Log dependencies (lineage of RDDs in graphs) and generate chain of parents.
- “Guides” spark on how to retrieve data and what to do with it.

- **Actions:**
- Performs the transformations.
- Computes and outputs the results.

Sparc Core - RDD

- **RDD Operations**

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

[Zaharia]

Sparc Core - RDD

- **Lazy execution**
  - Transformations are not executed immediately but **only when** an action is triggered, when necessary
    - By default, each transformed RDD may be recomputed each time you run an action on it
    - You may also *persist* an RDD in memory (`persist` / `cache` methods): Spark will keep the elements around on the cluster for much faster access the next time you query it
    - There is also support for persisting RDDs on disk (or replicated across multiple nodes)
  - Steps are kept in the lineage of RDDs
  - Save computation overhead
  - Reduced number of queries
  - Fault tolerance in practice

Sparc Core - RDD

- **Lazy execution**
  - Division by zero in lazy evaluations

```
scala> val xs: List[Int] = Range(1, 10000).toList

scala> val rdd = spark.sparkContext.parallelize(xs)

scala> val divByZero = rdd.map(x => x / 0)
divByZero: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[9] at map at <console>:28

scala> divByZero.collect
[Stage 2:======================>                                  (6 + 9) /
15]20/11/13 16:54:14 WARN scheduler.TaskSetManager: Lost task 14.0 in stage 2.0 (TID
10 executor 5): java.lang.ArithmeticException: / by zero
20/11/13 16:54:14 ERROR scheduler.TaskSetManager: Task 7 in stage 2.0 failed 4 times;
aborting job
org.apache.spark.SparkException: Job aborted due to stage failure: Task 7 in stage 2.0
failed 4 times, most recent failure: Lost task 7.3 in stage 2.0 (TID 20, executor 3):
java.lang.ArithmeticException: / by zero
```

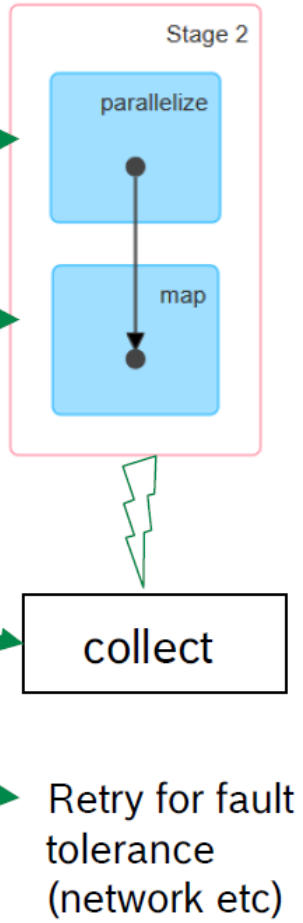
- Stage 0: Creation of list at driver
- Stage 1: Distribute rdd across executors
- Stage 2: Apply map() function

Sparc Core - RDD

- **Lazy execution**
  - Division by zero in lazy evaluations

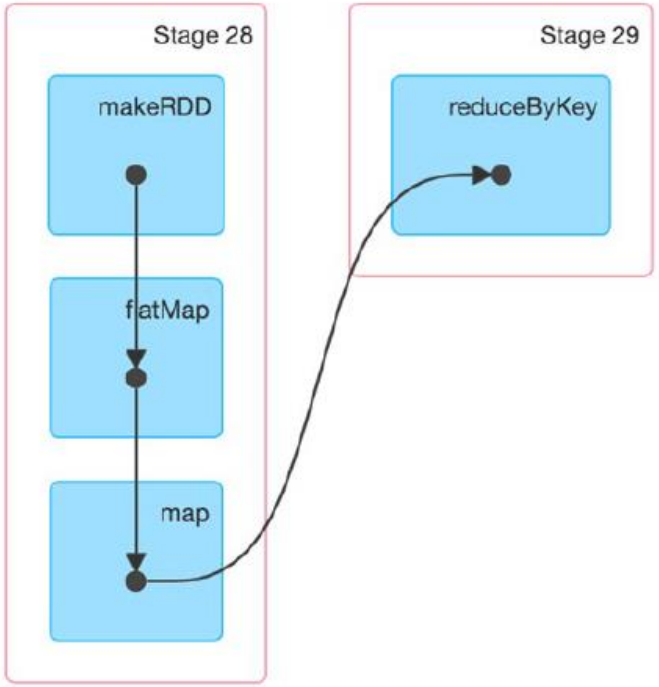
```
scala> val xs: List[Int] = Range(1, 10000).toList

scala> val rdd = spark.sparkContext.parallelize(xs)

scala> val divByZero = rdd.map(x => x / 0)
divByZero: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[9] at map at <console>:28

scala> divByZero.collect
[Stage 2:======================>                                  (6 + 9) /
15]20/11/13 16:54:14 WARN scheduler.TaskSetManager: Lost task 14.0 in stage 2.0 (TID
10 executor 5): java.lang.ArithmeticException: / by zero
20/11/13 16:54:14 ERROR scheduler.TaskSetManager: Task 7 in stage 2.0 failed 4 times;
aborting job
org.apache.spark.SparkException: Job aborted due to stage failure: Task 7 in stage 2.0
failed 4 times, most recent failure: Lost task 7.3 in stage 2.0 (TID 20, executor 3):
java.lang.ArithmeticException: / by zero
```

Direct Acyclic Graph (DAG)



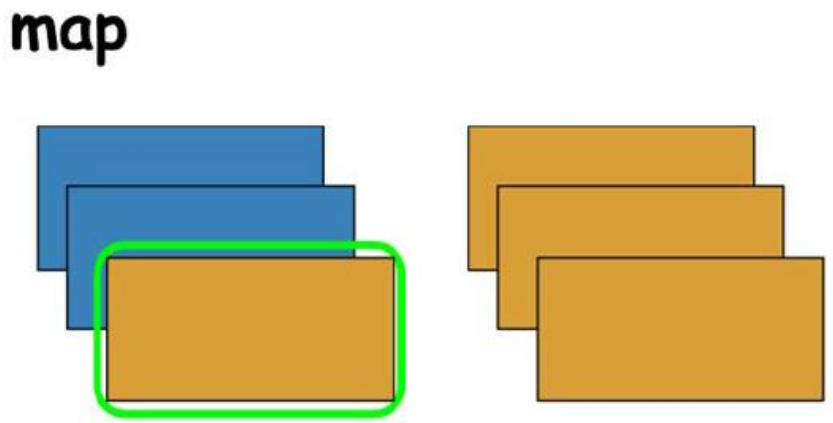
Spark Core - RDD

- **D**irected **A**cyclic **G**raphs (DAG)
  - Finite directed graph without cyclical orientation / loops
- Efficient fault recovery of RDDs using lineage
  - Log one operation to apply to many elements
  - Recompute lost partitions on failure
  - No cost if nothing fails
- Can start from entry point or any persisted/saved (intermediate)
  - from which any partition is **reconstructed** in case of failure
  - automatic recovery
- Define the optimal sequence of tasks/steps to be executed within a single job



Spark Core - RDD

- Basic Transformations

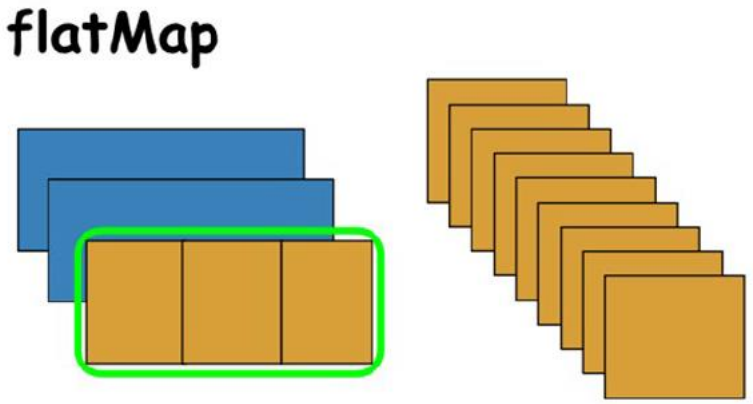


In [2]:
```
# map
x = sc.parallelize([1,2,3]) # sc = spark context, parallelize creates an RDD from the passed object
y = x.map(lambda x: (x,x**2))
print(x.collect())  # collect copies RDD elements to a list on the driver
print(y.collect())
```

```
[1, 2, 3]
[(1, 1), (2, 4), (3, 9)]
```

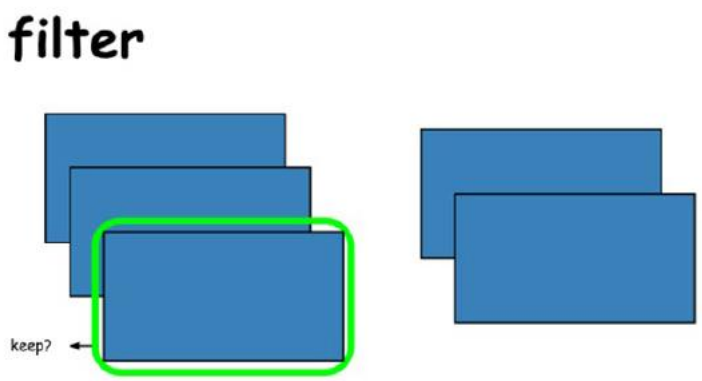
Spark Core - RDD

- Basic Transformations




```
In [3]: # flatMap
        x = sc.parallelize([1,2,3])
        y = x.flatMap(lambda x: (x, 100*x, x**2))
        print(x.collect())
        print(y.collect())
```

```
[1, 2, 3]
[1, 100, 1, 2, 200, 4, 3, 300, 9]
```



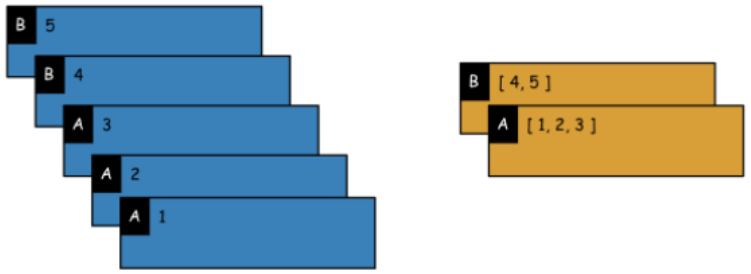

```
In [7]: # filter
        x = sc.parallelize([1,2,3])
        y = x.filter(lambda x: x%2 == 1)  # filters out even elements
        print(x.collect())
        print(y.collect())
```

```
[1, 2, 3]
[1, 3]
```

Spark Core - RDD

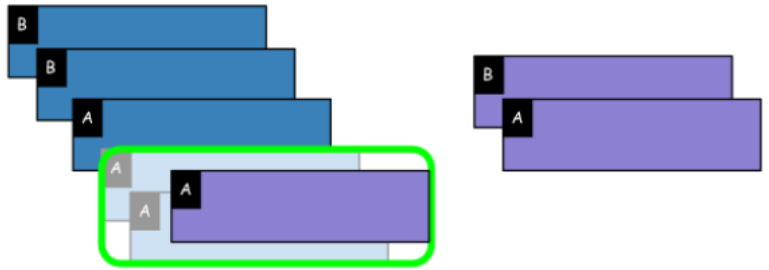
- Basic Transformations

groupByKey




```
In [54]: # groupByKey
         x = sc.parallelize([('B',5),('B',4),('A',3),('A',2),('A',1)])
         y = x.groupByKey()
         print(x.collect())
         print([(j[0],[i for i in j[1]]) for j in y.collect()])

         [('B', 5), ('B', 4), ('A', 3), ('A', 2), ('A', 1)]
         [('A', [3, 2, 1]), ('B', [5, 4])]
```

reduceByKey

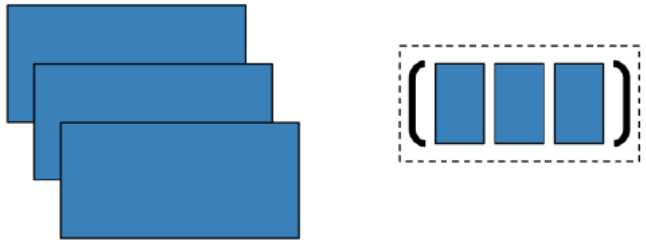



```
In [44]: # reduceByKey
         x = sc.parallelize([('B',1),('B',2),('A',3),('A',4),('A',5)])
         y = x.reduceByKey(lambda agg, obj: agg + obj)
         print(x.collect())
         print(y.collect())

         [('B', 1), ('B', 2), ('A', 3), ('A', 4), ('A', 5)]
         [('A', 12), ('B', 3)]
```

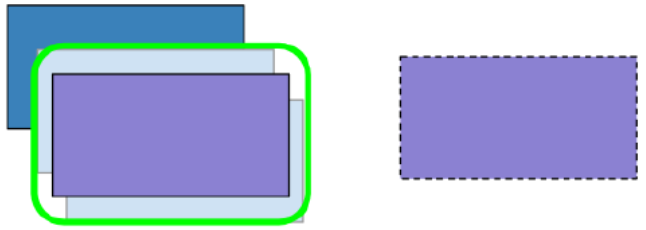
Spark Core - RDD

- Basic Actions

collect



reduce



```
In [21]: # collect
         x = sc.parallelize([1,2,3])
         y = x.collect()
         print(x)   # distributed
         print(y)   # not distributed
```

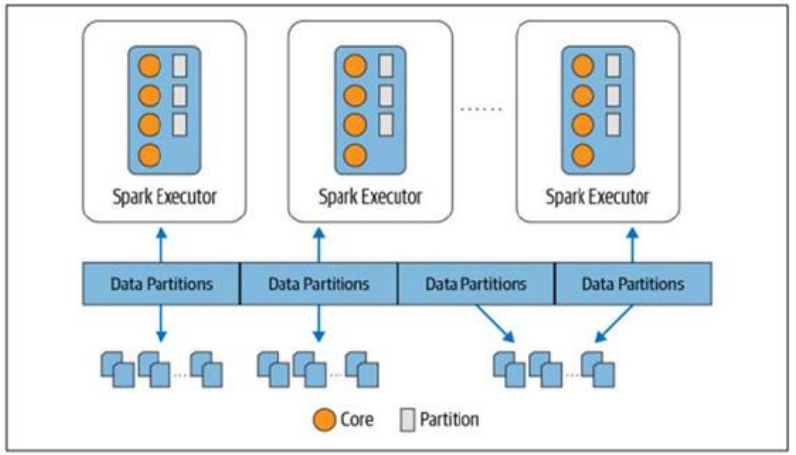
```
ParallelCollectionRDD[84] at parallelize at PythonRDD.scala:42
[1, 2, 3]
```

```
In [22]: # reduce
         x = sc.parallelize([1,2,3])
         y = x.reduce(lambda obj, accumulated: obj + accumulated)  # computes a cumulative sum
         print(x.collect())
         print(y)
```

```
[1, 2, 3]
6
```

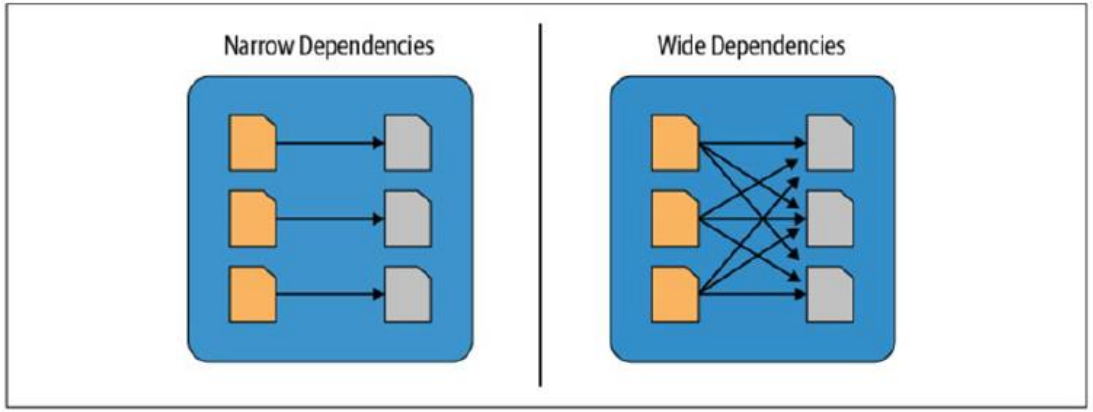
Spark Core - RDD

- Partitioning
- RDDs are to be considered like a continuous block, but data in RDDs is split into ***partitions***
- Partitioning in Spark ensures **parallelism** in the job
- Default: 200 partitions spread on executors
  - Level of partitioning dependent on data skewness/cardinality.
  - Fault tolerance is achieved also through partitioning. (if an executor fails to process it, the partition is sent to another)
  - **Default partitioning:** Splits in equally sized partitions, agnostic of data features.
  - **Range partitioning:** Split data based on ordered keys
  - **Hash partitioning:** Calculates a hash over each item *key* and then produces the modulo of this hash to determine the new partition



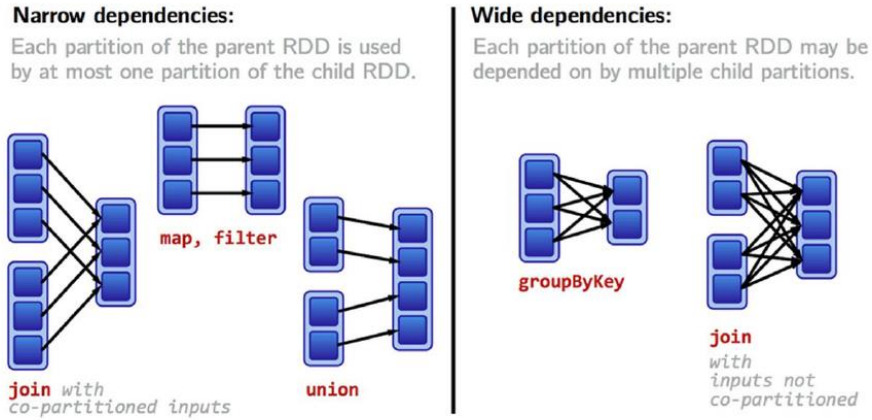
Spark Core - RDD

- Partitioning
- **Narrow Transformation (co-partitioned data)**
  - `map,flatMap,mapValues`
  - `filter`
  - `sample`
  - `union`
  - ….
- **Wide transformations (non co-partitioned data)**
  - `groupByKey`
  - `reduceByKey`
  - `sortByKey`
  - `distinct`
  - `join`
  - …



Spark Core - RDD

- Shuffling
- Wide transformations introduce **Shuffling**
- When operations need to calculate results using a common characteristic (e.g. a common key), the data needs to be close (even same node)
- The process of re-arranging data so that similar records end up in the same partitions is called **shuffling**
- Very expensive operation! Network traffic dependency, possible “data spill” on disk
- **Shuffle** combines data from all partitions to generate new key-value pairs (expensive operation)



Take home – Intro to Spark

- Spark: tool for (Big) data processing and analytics
- Spark for data engineers and data scientists
- Spark components
- Spark Use cases
- Spark RDDs and examples

Bibliography

- Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, University of California, Berkeley
- Bill Chambers, Matei Zaharia, **Spark: The Definitive Guide** Published by O'Reilly Media
- Petar Zecevi´c, Marko Bonaci, Spark in Action, Manning Publishing
- Jean-Georges Perrin, Spark in Action, 2nd Ed., Manning Publishing
- Jonathan Rioux, Data Analysis with Python and PySpark, Manning Publishing
- Dimitrios Kyriazopoulos, Ph.D., Introduction to Apache Spark
- Spark doc: https://spark.apache.org/docs/latest/
- Databricks:
  - –https://pages.databricks.com/gentle-intro-spark.html
  - –https://pages.databricks.com/data-scientist-spark-guide.html
  - –https://pages.databricks.com/data-engineer-spark-guide.html