Module 2

SPARK

- Apache Spark is a **cluster computing platform** designed to **be *fast* and *generalpurpose***.

- On the speed side, **Spark extends the popular MapReduce model** to efficiently support more types of computations, including interactive queries and stream processing.

- On the generality side, **Spark is designed to cover a wide range of workloads** that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming.

- By supporting these **workloads in the same engine**, Spark makes it easy and inexpensive to *combine* different processing types, which is often necessary in production data analysis pipelines.

- In addition, it reduces the **management burden of maintaining separate tools.**

- Spark is designed to be **highly accessible**, offering simple APIs in Python, Java, Scala, and SQL, and rich built-in libraries.

- It also **integrates closely with other Big Data tools.**

- In particular, Spark can run in Hadoop clusters and access any Hadoop data source, including Cassandra.

A Unified Stack

- **The Spark project contains multiple closely integrated components.**

- At its core, **Spark is a "computational engine" that is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks across many worker machines, or a *computing cluster*.**

A philosophy of tight integration has several benefits.

- First, all libraries and higherlevel components in the stack benefit from improvements at the lower layers. For example, when Spark's core engine adds an optimization, SQL and machine learning libraries automatically speed up as well.

- Each time a new component is added to the Spark stack, every organization that uses Spark will immediately be able to try this new component.

- Finally, one of the largest advantages of tight integration is the ability to build applications that seamlessly combine different processing models. For example, in Spark you can write one application that uses machine learning to classify data in real time as it is ingested from streaming sources. Simultaneously, analysts can query the resulting data, also in real time, via SQL.

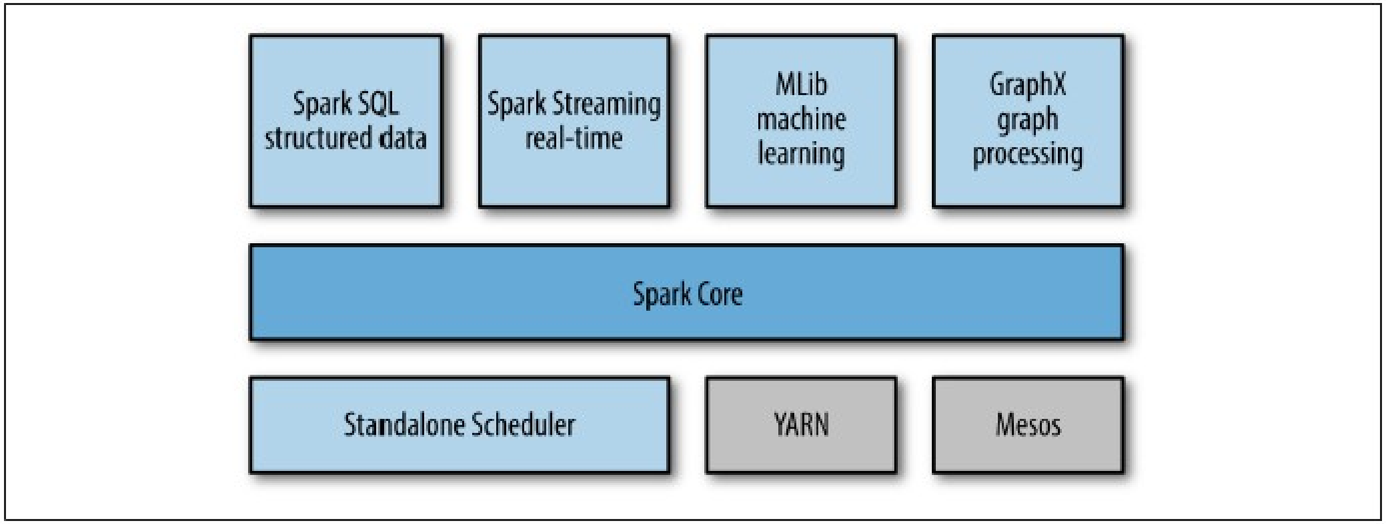


Figure 1-1. The Spark stack

Spark Core:

- Spark Core **contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more.**

- Spark Core is **also home to the API** that defines *resilient distributed datasets* (RDDs), which are Spark's main programming abstraction. RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel.

- Spark Core provides many APIs for building and manipulating these collections.

Spark SQL:

- Spark SQL is **Spark's package for working with structured data.** It allows querying data via SQL as well as the Apache Hive variant of SQL—called the Hive Query Language (HQL)—and it supports many sources of data, including Hive tables, Parquet, and JSON.

- Beyond providing a SQL interface to Spark, Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs in Python, Java, and Scala, all within a single application, **thus combining SQL with complex analytics.**

- This tight integration with the rich computing environment provided by Spark makes Spark SQL unlike any other open source data warehouse tool.

- Spark SQL was added to Spark in version 1.0.

- **Shark was an older SQL**-on-Spark project out of the University of California, Berkeley, that modified Apache Hive to run on Spark.

- It has now been replaced by Spark SQL to provide better integration with the Spark engine and language APIs.

Spark Streaming:

- Spark Streaming is **a Spark component that enables processing of live streams of data.** Examples of data streams include logfiles generated by production web servers, or queues of messages containing status updates posted by users of a web service.

- **Spark Streaming provides an API for manipulating data streams that closely matches the Spark Core's RDD API**, making it easy for programmers to learn the project and move between applications that manipulate data stored in memory, on disk, or arriving in real time.

- Underneath its API, Spark Streaming was designed to provide the **same degree of fault tolerance, throughput, and scalability as Spark Core.**

Mllib:

- Spark comes with a **library containing common machine learning (ML) functionality, called MLlib.**

- MLlib provides **multiple types of machine learning algorithms, including** classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import.

- It also provides some lower-level ML primitives, including a generic gradient descent optimization algorithm. All of these methods are designed to scale out across a cluster.

GraphX:

- GraphX **is a library for manipulating graphs (e.g., a social network's friend graph) and performing graph-parallel computations.**

- Like Spark Streaming and Spark SQL, GraphX extends **the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge.**

- GraphX also provides **various operators for manipulating graphs** (e.g., subgraph and mapVertices) and **a library of common graph algorithms** (e.g., PageRank and triangle counting).

Cluster Managers:

- Under the hood, Spark is designed to efficiently scale up from one to many thousands of compute nodes.

- To achieve this while maximizing flexibility, Spark can run over a variety of *cluster managers*, including Hadoop YARN, Apache Mesos, and a simple cluster manager included in Spark itself called the Standalone Scheduler.

- If you are just installing Spark on an empty set of machines, the Standalone Scheduler provides an easy way to get started; if you already have a Hadoop YARN or Mesos cluster, however, Spark's support for these cluster managers allows your applications to also run on them.

Background of YARN:

•Apache Hadoop is an ecosystem used for processing large amounts of data through the map reduce data processing model.

•This ecosystem was originally developed by Google Inc. and was handed over to the Apache Software Foundation in 2005.

•Hadoop supports distributed processing of large amounts of data through the core MapReduce processing mechanism.

•In 2012, the Hadoop project was upgraded to introduce a new architecture, known as YARN, which provides a more **generalpurpose data processing framework.**

•Hadoop 1, with the core MapReduce processing model, provides a way in which MapReduce programs can be written in various languages, such as Java, Python, Ruby-on-Rails, Pig, etc.

•However, the issue with Hadoop 1 is that the programs are run on the basis of the MapReduce processing model that cannot deal with all Big Data processing problems alone.

•The MapReduce processing model is a program-based model, which includes tools such as Pig and Hive to provide simplicity to data processing tasks.

•However, it does not solve all kinds of Big Data problems.

•Therefore, YARN was introduced to fill the gap resulted due to the incapability of the MapReduce model to fulfill all kinds of data processing requirements.

YARN Architecture:

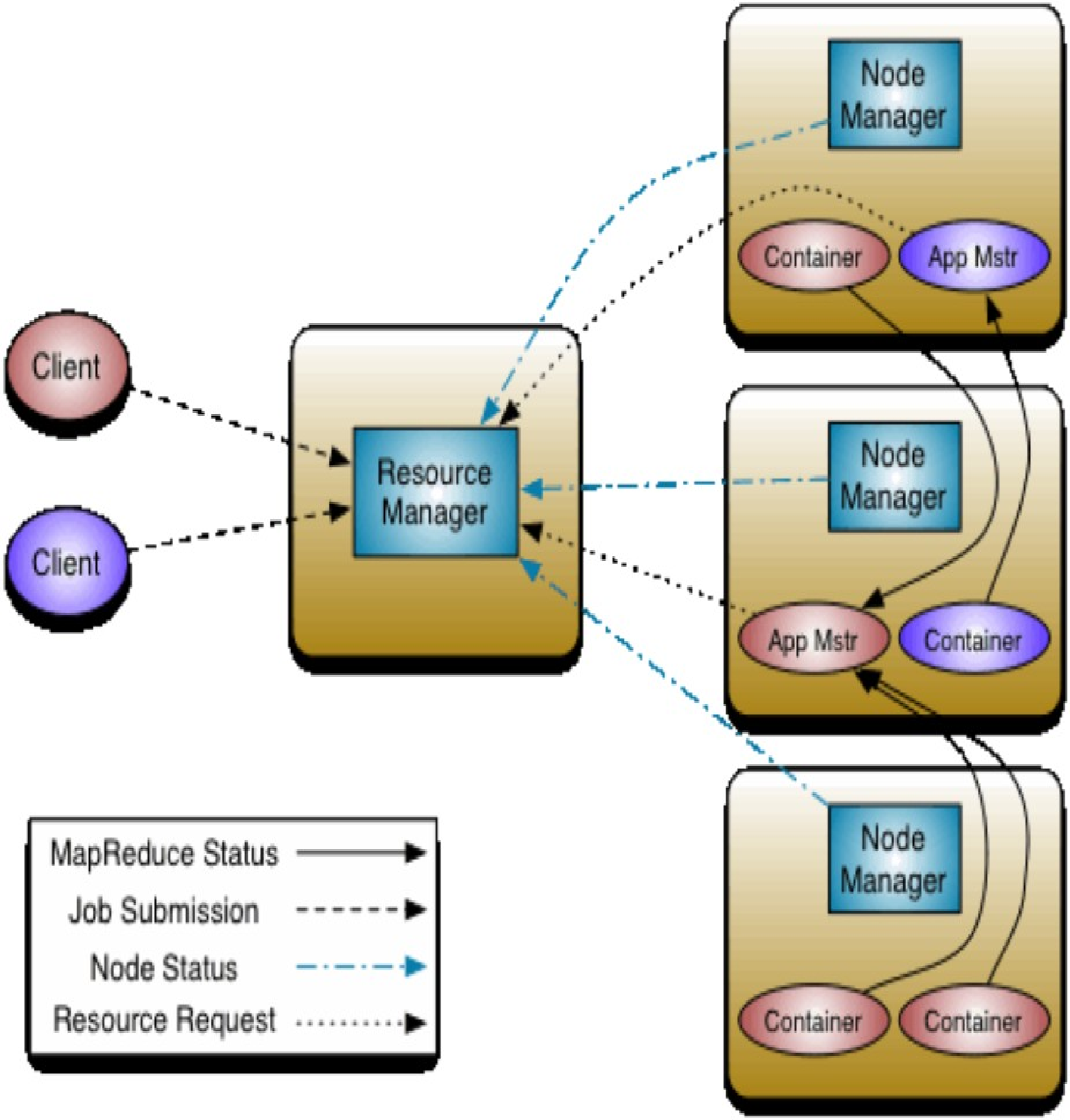
- The fundamental idea of YARN is to split the tasks of the JobTracker, ie, managing resources and scheduling/monitoring jobs to separate components.

- These components are known as Resource Manager and Application Master (also called Application Manager)

- In Hadoop 2. Resource Manager shares the overall responsibility of controlling and managing resources and the Application Manager component allows a cluster to handle multiple applications at a time.

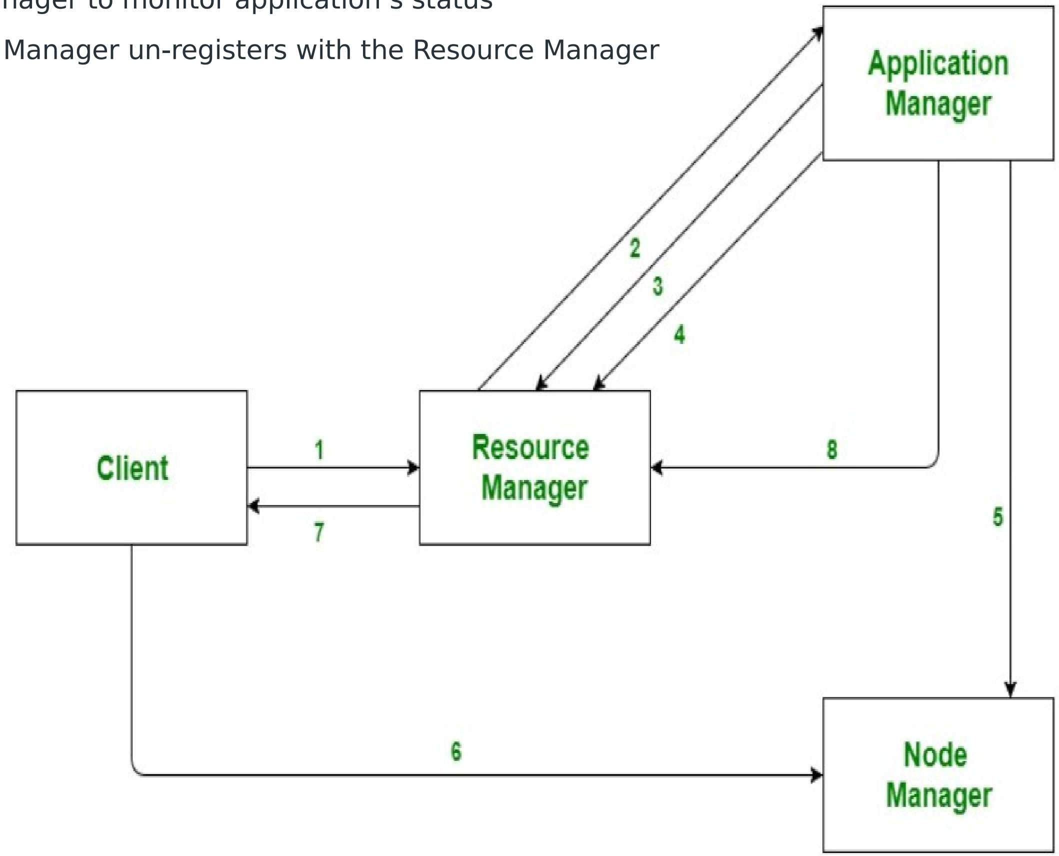
- Each application in the cluster will have its own Application Manager instance

- Resource Manager and per-node slave (also called NodeManager) together manage applications in a distributed manner by forming data-computation framework.


Node Manager
Container
App Mstr
Client
Client
Resource Manager
Node Manager
App Mstr
Container
Node Manager
Container
Container
MapReduce Status
Job Submission
Node Status
Resource Request

Application workflow in Hadoop YARN:

1. Client submits an application
2. The Resource Manager allocates a container to start the Application Manager
3. The Application Manager registers itself with the Resource Manager
4. The Application Manager negotiates containers from the Resource Manager
5. The Application Manager notifies the Node Manager to launch containers
6. Application code is executed in the container
7. Client contacts Resource Manager/Application Manager to monitor application’s status
8. Once the processing is complete, the Application Manager un-registers with the Resource Manager



ResourceManager:

•Resource Manager, in YARN architecture, is the supreme authority that controls all the **decisions related to resource management and allocation.**

•It has a Scheduler Application Programming Interface (API) that negotiates and schedules resources. However, the Scheduler API **does not monitor or track the status of applications.** Hence, no guarantee is offered by the Scheduler to restart the tasks that have failed due to application or hardware failure.

•This API schedules resources, such as CPU, memory, disk, and network, on **the basis of the resource requirements of applications.** The main purpose of introducing Resource Manager in YARN is to optimize the utilization of resources all the time by managing all the restrictions, which involve capacity guarantees, fairness in allocation of resources, etc. Thus, we can say that the YARN ResourceManager is responsible for almost all the tasks performed by the JobTracker in the MapReduce model.

•Resource Manager performs all its **tasks in integration with NodeManager and Application Manager components.** Resources on each node are allocated and managed by its respective NodeManager. The ResourceManager gives instructions to the NodeManager, which is responsible for managing resources available on the node it manages. Similarly for each application, there is an Application Manager instance that negotiates resources with the Resource Manager and, in association with NodeManager instances, starts the containers.

•In a cluster, applications can request resources at various levels, such as nodes, racks, etc. However, the resource allocation is done by the Scheduler **on the basis of the availability of resources** and the configured resource sharing policy in YARN.

•In short, we can say that the Scheduler uses the **abstract notion of a resource container** to perform its scheduling function on the basis of the amount of resources required for an application.

•The resource container incorporates various resource dimensions, including memory, CPU, disk, network, etc.

•Moreover, it is not concerned with monitoring of previously executed jobs, as this responsibility has been provided to the Job History Service (a component running on a separate node).

•Hence, the ResouceManager in YARN architecture is only responsible for resource scheduling and providing more scalability and flexibility.

Application Manager:

- Every instance of an application running within YARN is managed by an Application Manager, which is responsible for the negotiation of resources with the Resource Manager.
-
- Thus, we can say that Application Manager is responsible for negotiating for appropriate resource containers from the Scheduler, monitoring of their status, and checking the progress.

Integration of Resource Manager and Application Manager:

•With multiple applications residing within a cluster, Resource Manager **needs to have information about the resources needed for each application.**

•This information helps Resource Manager to negotiate for resources with Application Manager and **provide optimum resource utilization**.

•This concept of requesting for the appropriate resource is known **as ResourceRequest**. Application Manager sends ResourceRequest to Resource Manager for allocating the required resources by Resource Manager..

•**When Resource Manager approves a ResourceRequest of Application Manager, a container is created.**

Container:

•A container is nothing but a **set of physical resources on a single node. A container consists of memory (RAM), CPU cores, and disks.**

•Depending **upon the resources in a node, a node can have multiple containers** that are assigned to a specific Application Manager.

•A container thus represents a resource (memory, CPU, etc.) on a single node in a given cluster. **A container is supervised by the NodeManager and scheduled by the Resource Manager.**

•**Application Manager itself is launched in a container,** which is referred as container 0.

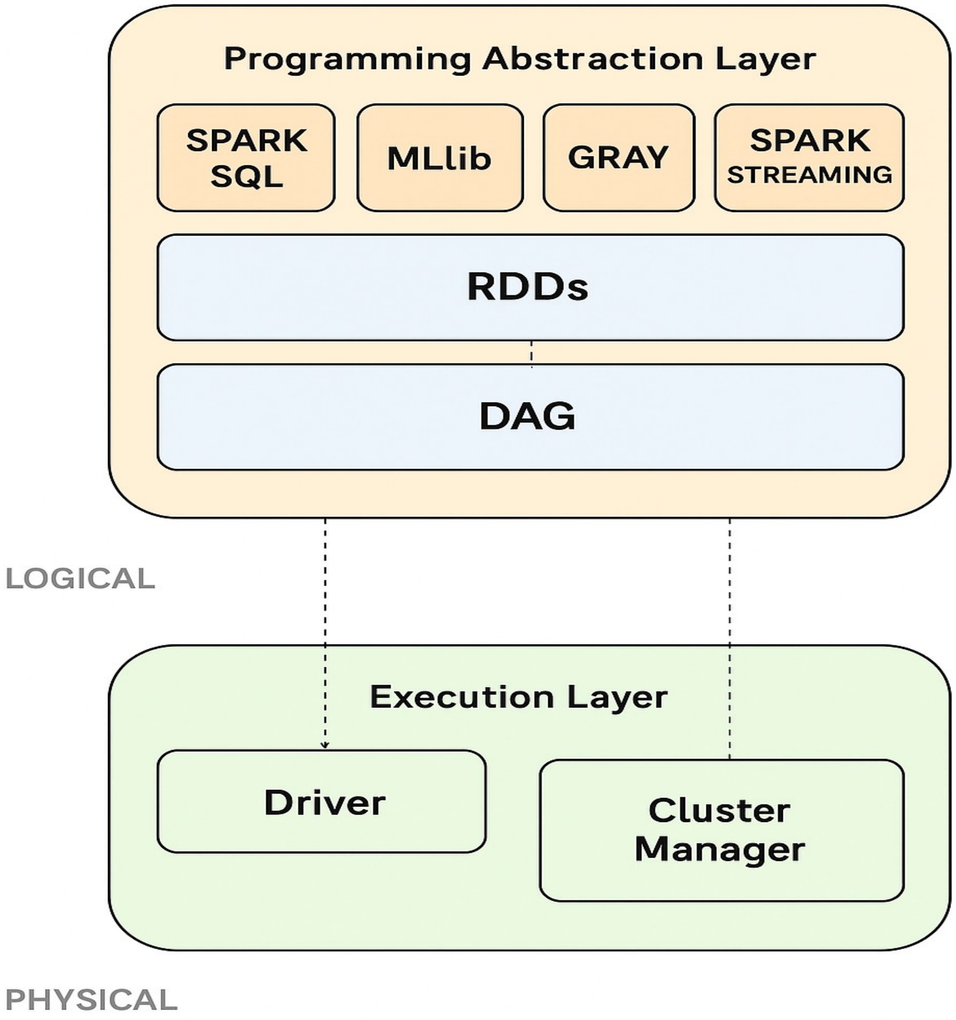
NodeManager:

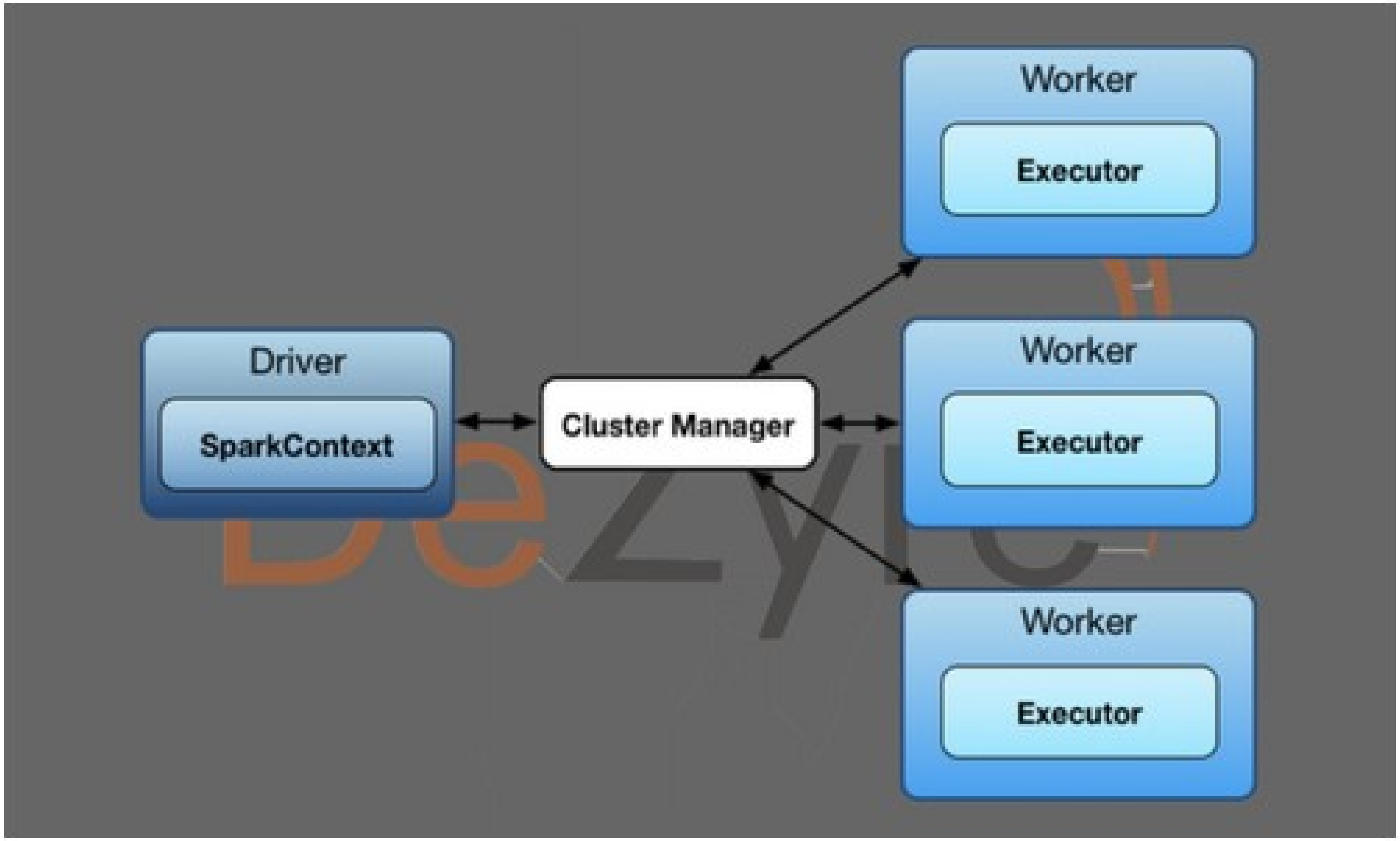
•Is the per-machine slave, which is responsible for **launching the applications containers (containers that result from a successful ResourceRequest), monitoring their resource usage (CPU, memory, disk, network, etc.), and reporting the status of the resource usage to the Resource Manager.**

•These services range from managing a container to monitoring resources and tracking the health of its node.

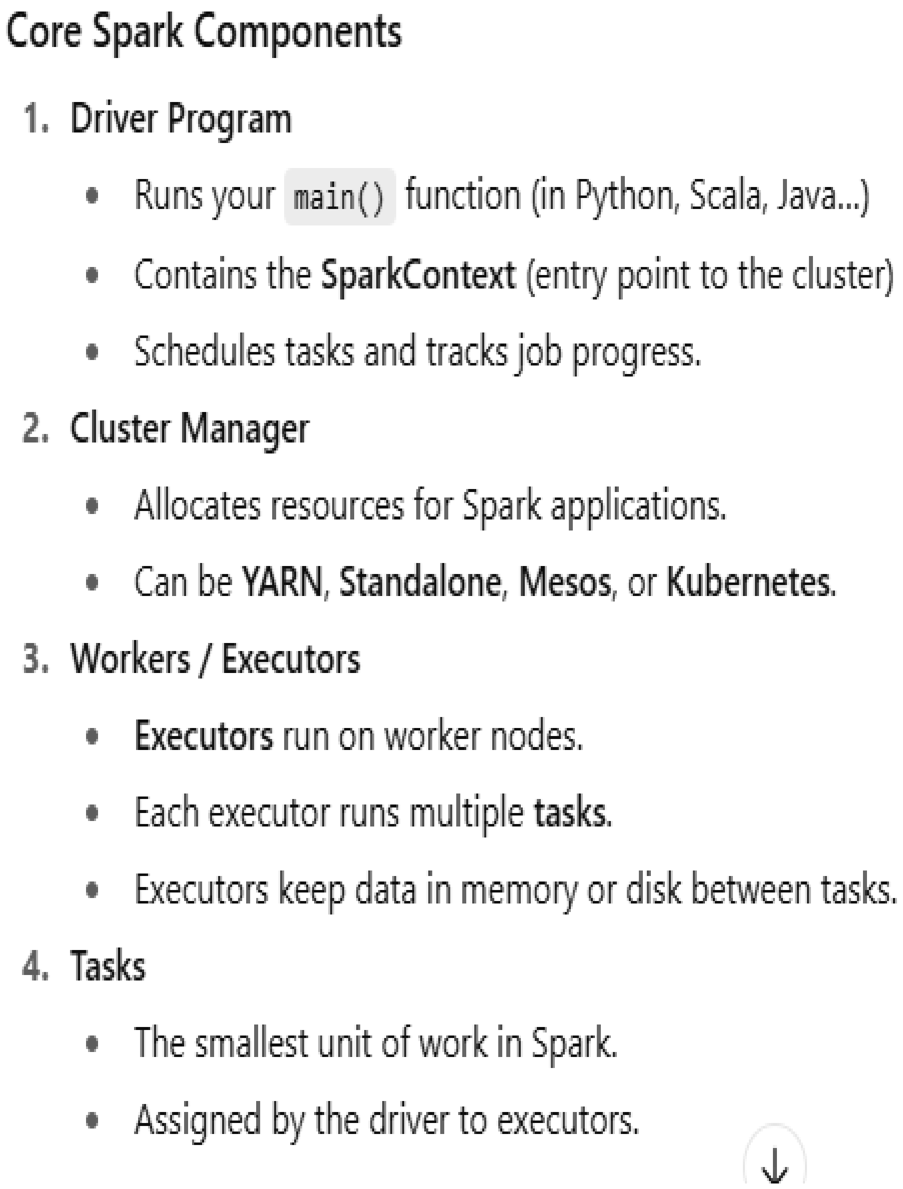
•NodeManager manages containers that represent per-node resources available for a particular application.

APACHE SPARK ARCHITECTURE



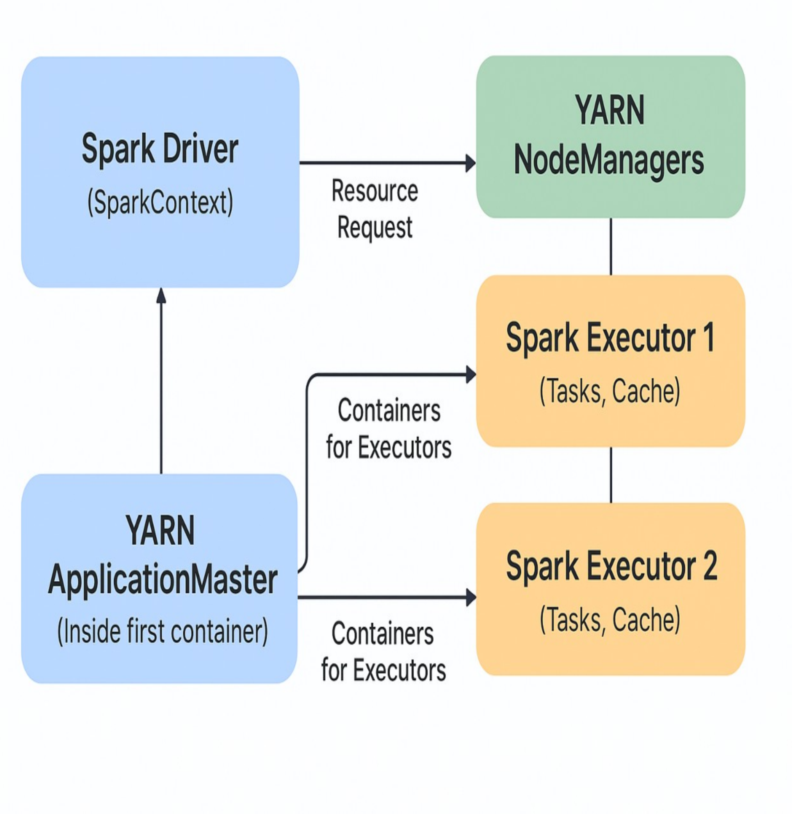


Spark Architecture Diagram – Overview of Apache Spark Cluster



Core Spark Components

1. **Driver Program**
   - Runs your `main()` function (in Python, Scala, Java...)
   - Contains the **SparkContext** (entry point to the cluster)
   - Schedules tasks and tracks job progress.
2. **Cluster Manager**
   - Allocates resources for Spark applications.
   - Can be **YARN**, **Standalone**, **Mesos**, or **Kubernetes**.
3. **Workers / Executors**
   - **Executors** run on worker nodes.
   - Each executor runs multiple **tasks**.
   - Executors keep data in memory or disk between tasks.
4. **Tasks**
   - The smallest unit of work in Spark.
   - Assigned by the driver to executors.



Resilient Distributed Datasets (RDD):

Definition:

An RDD is the core data structure in Apache Spark — it's an immutable, distributed collection of objects that can be processed in parallel across a cluster.

Resilient → Fault-tolerant. If part of the data is lost (e.g., a worker node fails), Spark can recompute it using lineage information.

Distributed → Data is split across multiple nodes in the cluster for parallel processing.

Dataset → Just means a collection of elements (like a list or table of records).

In terms of datasets, apache spark supports **two types of RDD's** – Hadoop Datasets which are created from the files stored on HDFS and parallelized collections which are based on existing Scala collections.

- Spark RDD's support **two different types of operations** – Transformations and Actions.

Directed Acyclic Graph (DAG):

Direct - Transformation is an action which transitions data partition state from A to B.
Acyclic -Transformation cannot return to the older partition

- DAG is a sequence of computations performed on data where each node is an RDD partition and edge is a transformation on top of data.

- The DAG abstraction helps eliminate the **Hadoop MapReduce** multistage execution model and provides performance enhancements over Hadoop.

A Directed Acyclic Graph (DAG) in Spark:
Represents the entire sequence of transformations for a job as a graph of stages
Spark analyzes the whole job upfront before execution
Groups operations together to minimize unnecessary stages
Keeps intermediate data in memory when possible

Apache Spark follows a master/slave architecture with two main daemons and a cluster manager –

Master Daemon – (Master/Driver Process)
Worker Daemon –(Slave Process)

- A spark cluster has a single Master and any number of Slaves/Workers. The driver and the executors run their individual Java processes and users can run them on the same horizontal spark cluster or on separate machines i.e. in a vertical spark cluster or in mixed machine configuration.

Spark Driver – Master Node of a Spark Application:

- It is the central point and the entry point of the Spark Shell (Scala, Python, and R). The driver program runs the main () function of the application and is the place where the Spark Context is created. Spark **Driver contains various components** – DAGScheduler, TaskScheduler, BackendScheduler and BlockManager **responsible for the translation of spark user code into actual spark jobs executed on the cluster.**

- The driver program that runs on the master node of the **spark cluster schedules the job execution and negotiates with the cluster manager**.

- It translates the RDD's **into the execution graph and splits the graph into multiple stages**.

- **Driver stores the metadata** about all the Resilient Distributed Databases and their partitions. Driver program converts a user application into smaller execution units known as tasks. **Tasks are then executed by the executors** i.e. the worker processes which run individual tasks.
-
- Driver exposes the information about the running spark application through a Web UI at port 4040.

- https://spark.apache.org/docs/latest/web-ui.html

Role of Executor in Spark Architecture

- Executor is a distributed agent **responsible for the execution of tasks.** Every spark applications has its own executor process.

- Executors usually **run for the entire lifetime of a Spark application** and this phenomenon is known as **"Static Allocation of Executors".**

- However, users can also opt for **dynamic allocations of executors** wherein they can add or remove spark executors dynamically to match with the overall workload.

- Executor **performs all the data processing.** Reads from and Writes data to external sources.

- Executor **stores the computation results data in-memory, cache or on hard disk drives.**

Interacts with the storage systems.

Role of Cluster Manager in Spark Architecture:

- An external **service responsible for acquiring resources on the spark cluster and allocating them to a spark job.**

- There are 3 different types of cluster managers a Spark application can leverage for the allocation and deallocation of various physical resources such as memory for client spark jobs, CPU memory, etc.

- **Hadoop YARN, Apache Mesos or the simple standalone spark cluster manager** either of them can be launched on-premise or in the cloud for a spark application to run.

- Choosing a cluster manager for any spark application **depends on the goals of the application** because all **cluster managers provide different set of scheduling capabilities.**

- To get started with apache spark, the standalone cluster manager is the easiest one to use when developing a new spark application.

What happens when a Spark Job is submitted?

- When a client submits a spark user application code, the driver implicitly converts the code containing transformations and actions into a logical directed acyclic graph (DAG).

- At this stage, the driver program also performs certain optimizations like pipelining transformations and then it converts the logical DAG into physical execution plan with set of stages.

- After creating the physical execution plan, it creates small physical execution units referred to as tasks under each stage. Then tasks are bundled to be sent to the Spark Cluster.

- The driver program then talks to the cluster manager and negotiates for resources. The cluster manager then launches executors on the worker nodes on behalf of the driver.

- At this point the driver sends tasks to the cluster manager based on data placement. Before executors begin execution, they register themselves with the driver program so that the driver has holistic view of all the executors.

- Now executors start executing the various tasks assigned by the driver program.

- At any point of time when the spark application is running, the driver program will monitor the set of executors that run.

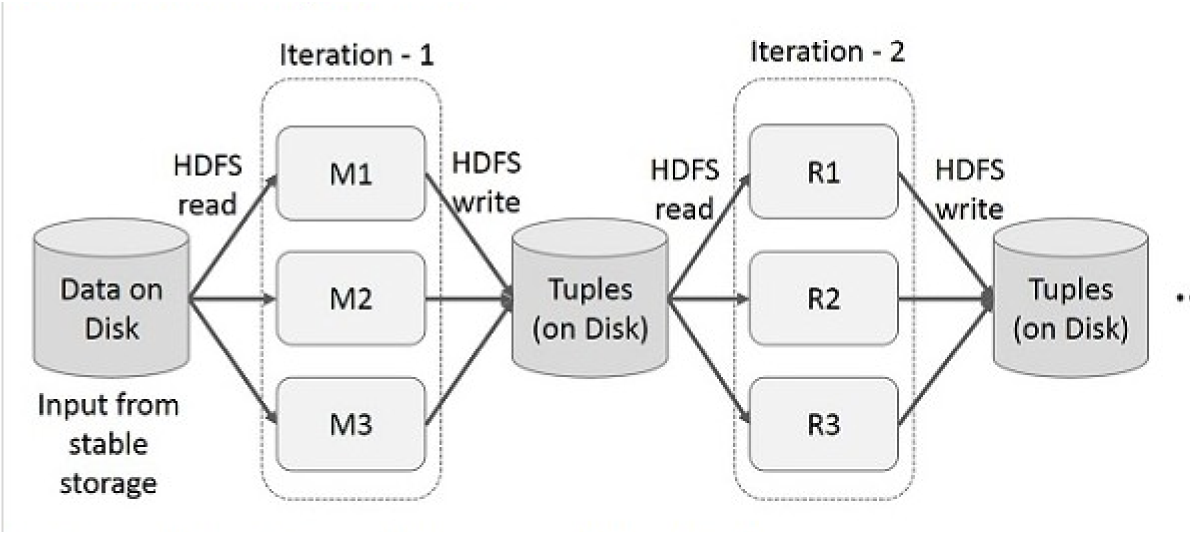
- Driver program in the spark architecture also schedules future tasks based on data placement by tracking the location of cached data.

- When driver programs main () method exits or when it call the stop () method of the Spark Context, it will terminate all the executors and release the resources from the cluster manager.

- The structure of a Spark program at higher level is - RDD's are created from the input data and new RDD's are derived from the existing RDD's using different transformations, after which an action is performed on the data. In any spark program, the DAG operations are created by default and whenever the driver runs the Spark DAG will be converted into a physical execution plan.

Data Sharing is Slow in MapReduce:

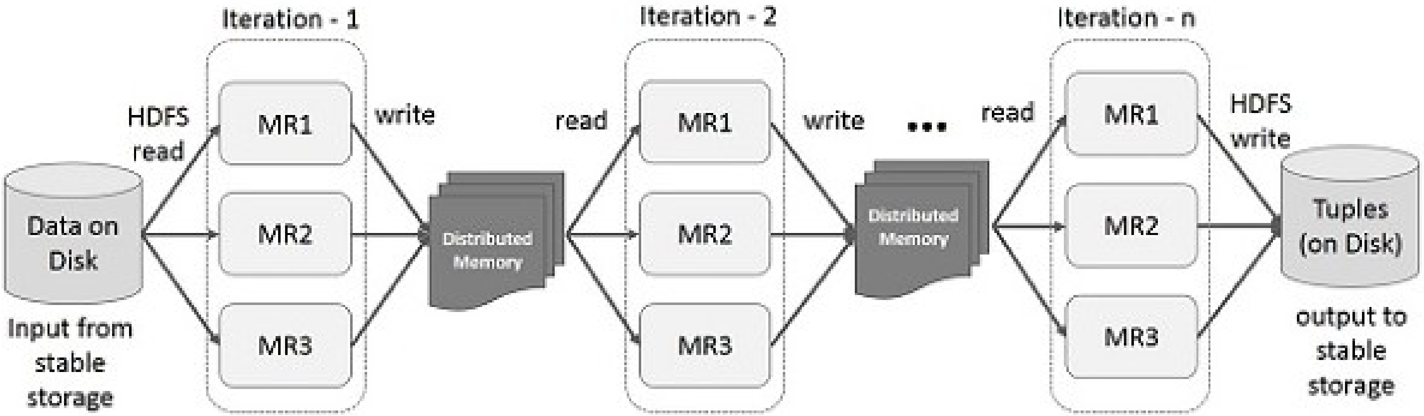
- Iterative Operations on MapReduce
- Reuse intermediate results across multiple computations in multi-stage applications. The following illustration explains how the current framework works, while doing the iterative operations on MapReduce. This incurs substantial overheads due to **data replication, disk I/O, and serialization, which makes the system slow.**



Iterative Operations on Spark RDD

The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of persistant storage (Disk) and make the system faster.

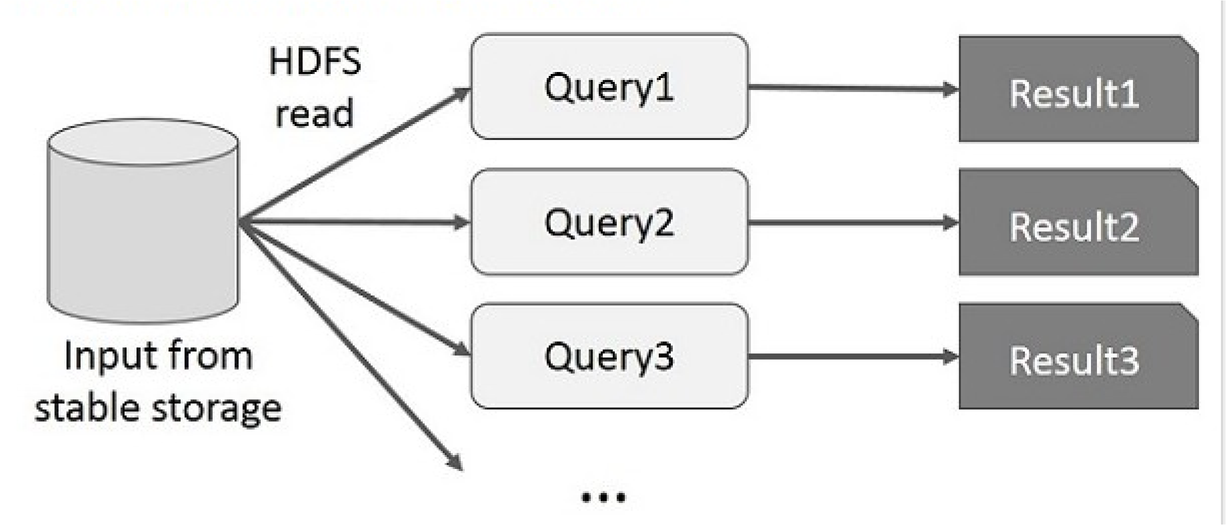
Note − If the Distributed memory (RAM) is not sufficient to store intermediate results (State of the JOB), then it will store those results on the disk.



Interactive Operations on MapReduce:

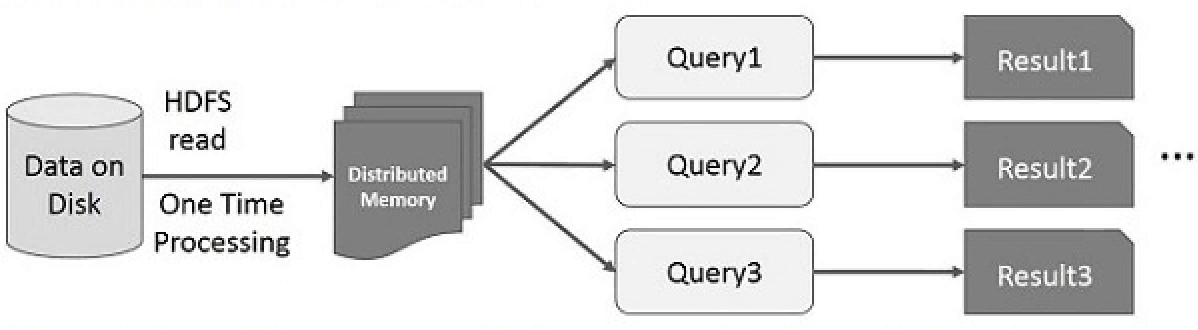
User runs ad-hoc queries on the same subset of data. Each query will do the disk I/O on the stable storage, which can dominate application execution time.

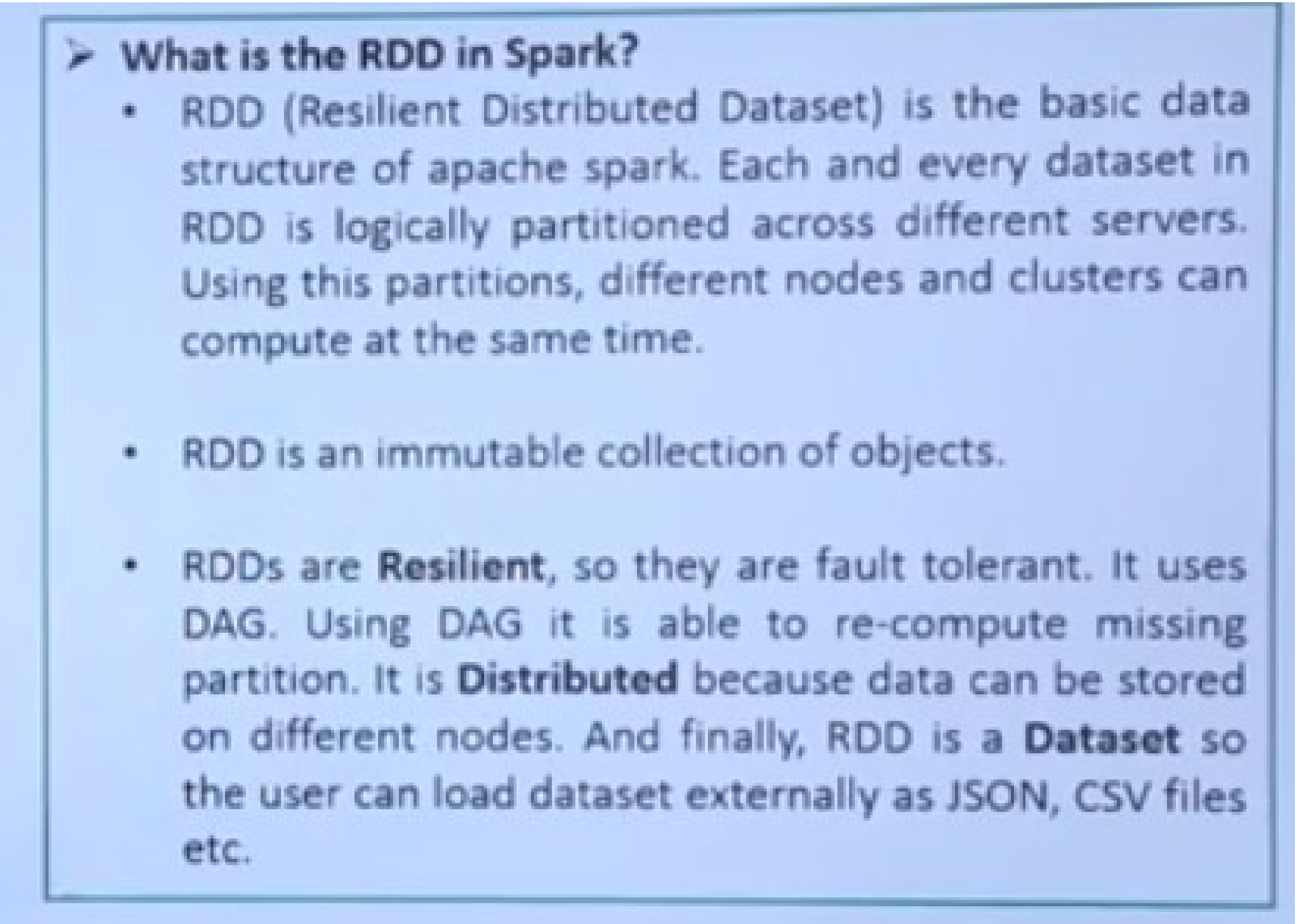
The following illustration explains how the current framework works while doing the interactive queries on MapReduce.



Interactive Operations on Spark RDD:

- This illustration shows interactive operations on Spark RDD. If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.





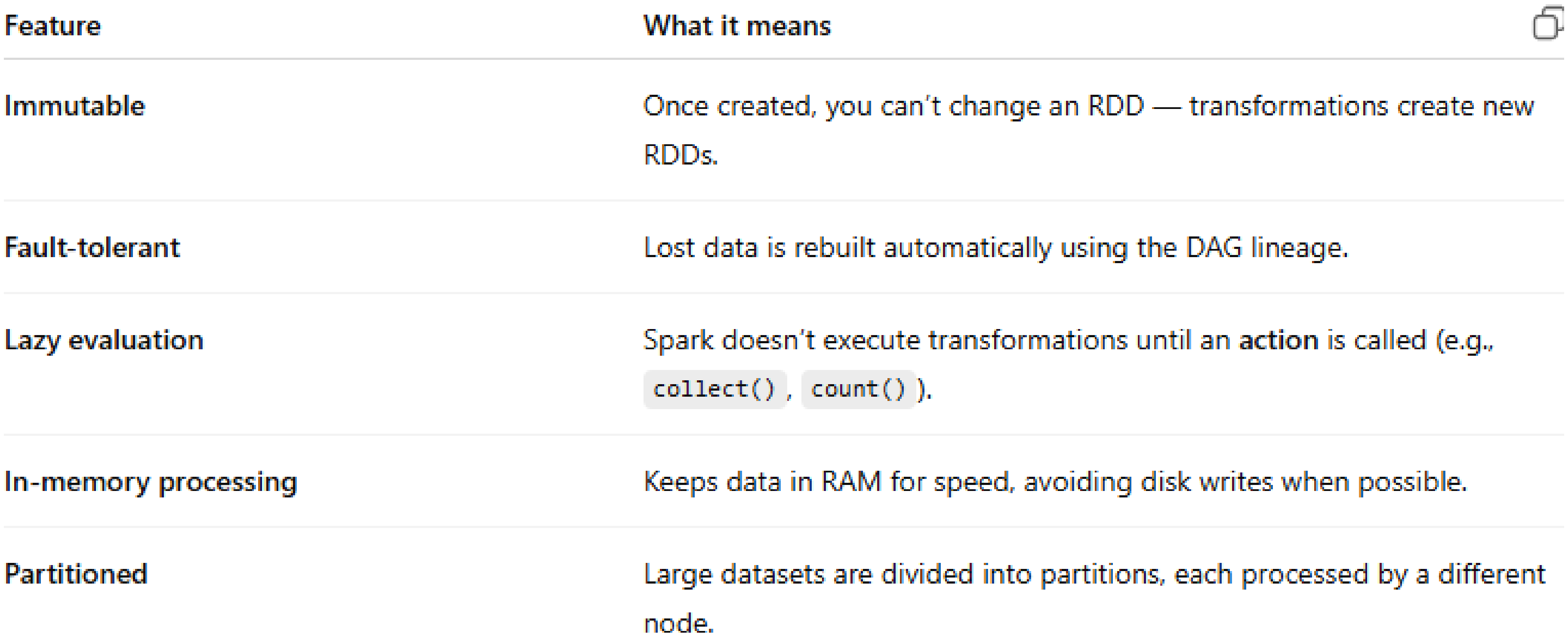
➢ **What is the RDD in Spark?**

- RDD (Resilient Distributed Dataset) is the basic data structure of apache spark. Each and every dataset in RDD is logically partitioned across different servers. Using this partitions, different nodes and clusters can compute at the same time.

- RDD is an immutable collection of objects.

- RDDs are **Resilient**, so they are fault tolerant. It uses DAG. Using DAG it is able to re-compute missing partition. It is **Distributed** because data can be stored on different nodes. And finally, RDD is a **Dataset** so the user can load dataset externally as JSON, CSV files etc.

Key Features of RDD



Feature	What it means
Immutable	Once created, you can't change an RDD — transformations create new RDDs.
Fault-tolerant	Lost data is rebuilt automatically using the DAG lineage.
Lazy evaluation	Spark doesn't execute transformations until an **action** is called (e.g., `collect()`, `count()`).
In-memory processing	Keeps data in RAM for speed, avoiding disk writes when possible.
Partitioned	Large datasets are divided into partitions, each processed by a different node.

- **How RDDs are created**
- From existing data: sc.parallelize([1, 2, 3, 4])
- From external storage: sc.textFile("hdfs://...")
- By transforming other RDDs

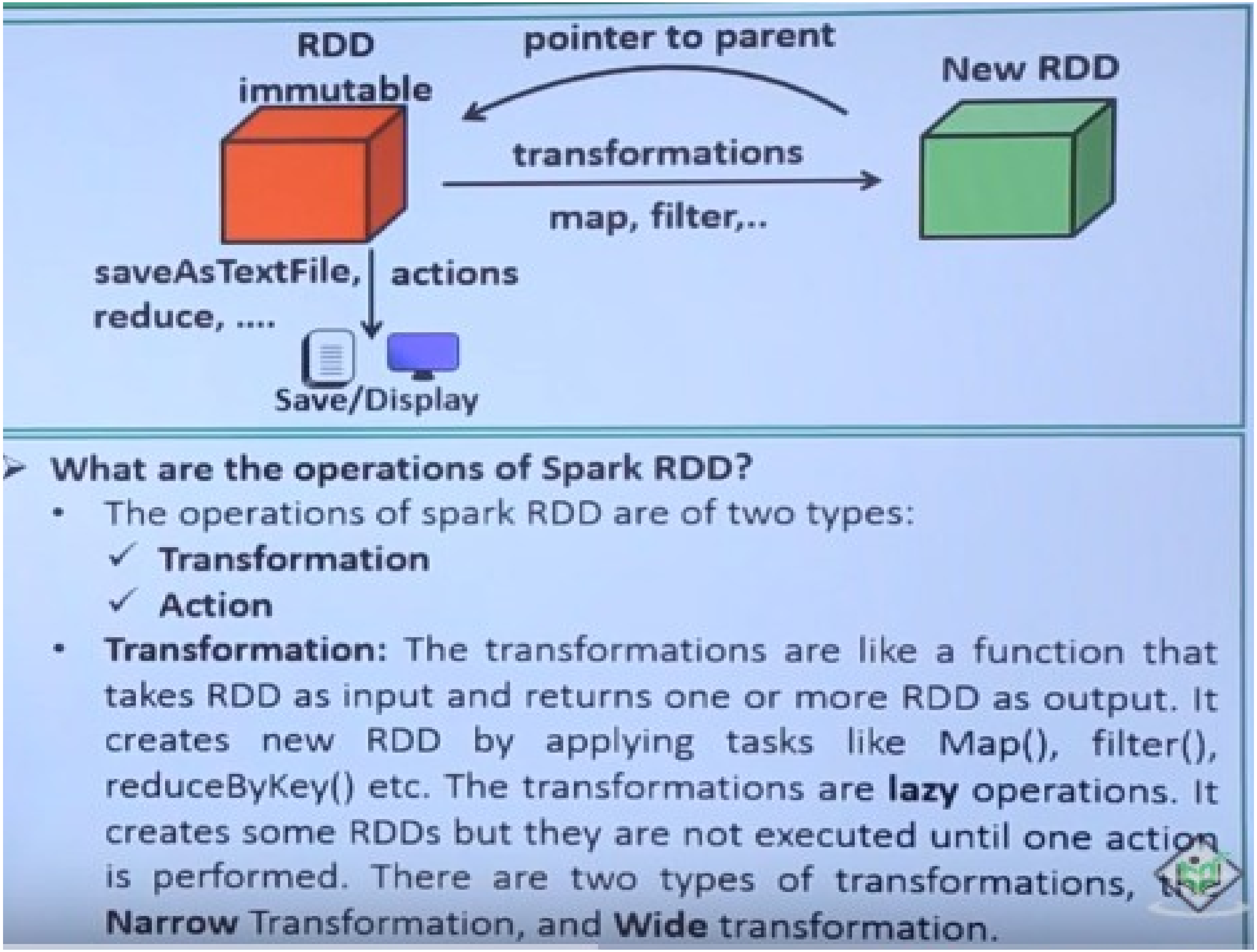
Two types of RDD operations

Transformations → Create new RDDs from existing ones

Examples: map(), filter(), flatMap(), groupByKey()

Actions → Trigger execution and return results

Examples: collect(), count(), reduce(), saveAsTextFile()



RDD

immutable

pointer to parent

New RDD

transformations

map, filter,..

saveAsTextFile,

reduce, ...

actions

Save/Display

- **What are the operations of Spark RDD?**
  - The operations of spark RDD are of two types:
    - ✓ **Transformation**
    - ✓ **Action**
  - **Transformation:** The transformations are like a function that takes RDD as input and returns one or more RDD as output. It creates new RDD by applying tasks like Map(), filter(), reduceByKey() etc. The transformations are **lazy** operations. It creates some RDDs but they are not executed until one action is performed. There are two types of transformations, the **Narrow** Transformation, and **Wide** transformation.

- Whether a given function is a transformation or an action, you can look at its **return type**: transformations return RDDs, whereas actions return some other data type.

- As an example, suppose that we have a logfile, *log.txt*, with a number of messages, and we want to select only the error messages. We can use the filter() transformation seen before.

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

- Note that the filter() operation does not mutate the existing inputRDD. Instead, it returns a pointer to an entirely new RDD. inputRDD can still be reused later in the program—for instance, to search for other words. In fact, let's use inputRDD again to search for lines with the word *warning* in them. Then, we'll use another transformation, union(), to print out the number of lines that contained either *error* or *warning*.

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

- Finally, as you derive new RDDs from each other using transformations, Spark keeps track of the set of dependencies between different RDDs, called the *lineage graph*. It uses this information to compute each RDD on demand and to recover lost data if part of a persistent RDD is lost.

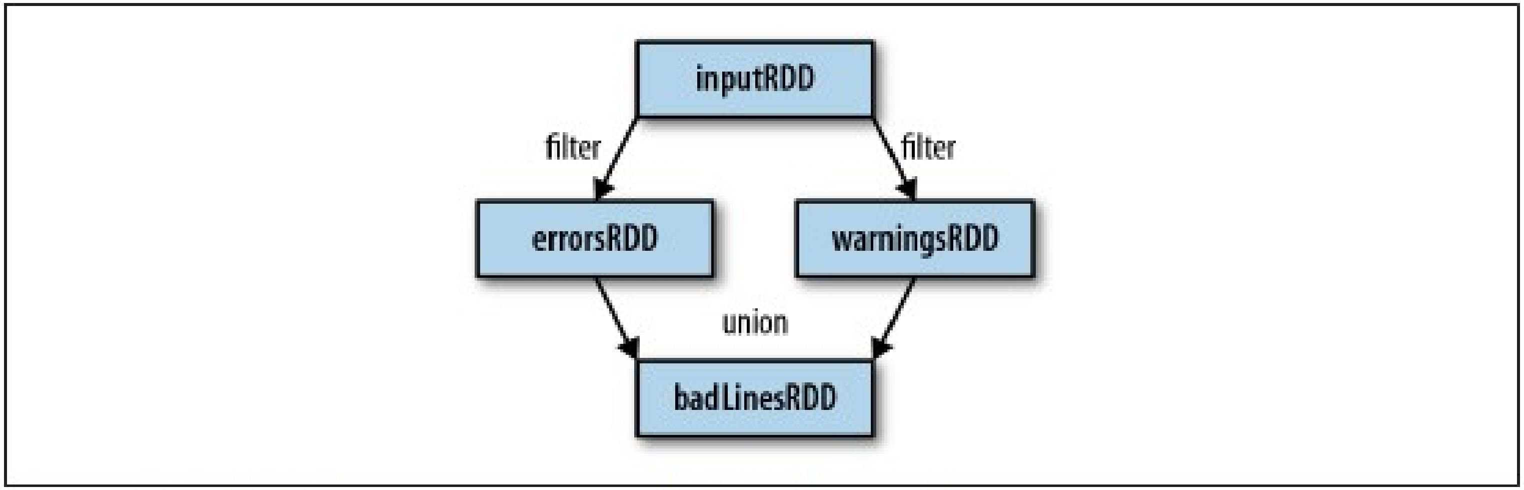
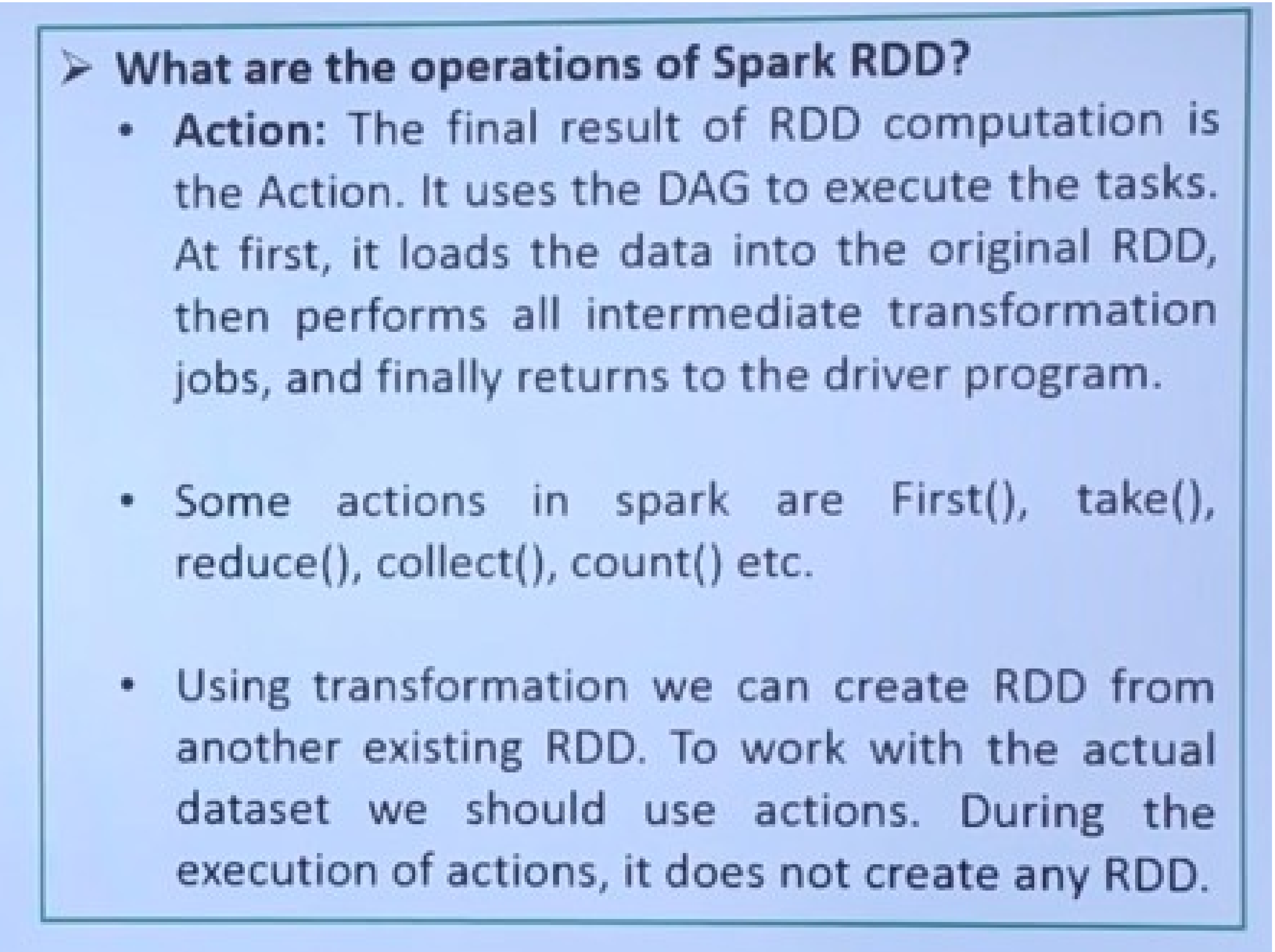


Figure 3-1. RDD lineage graph created during log analysis



➢ What are the operations of Spark RDD?

- **Action:** The final result of RDD computation is the Action. It uses the DAG to execute the tasks. At first, it loads the data into the original RDD, then performs all intermediate transformation jobs, and finally returns to the driver program.

- Some actions in spark are First(), take(), reduce(), collect(), count() etc.

- Using transformation we can create RDD from another existing RDD. To work with the actual dataset we should use actions. During the execution of actions, it does not create any RDD.

- Actions are the **second type of RDD operation**. They are the operations that **return a final value** to the driver program or **write data to an external storage system**. Actions force the evaluation of the transformations required for the RDD they were called on, since they need to actually produce output.

- Continuing the log example from the previous section, we might want to print out some information about the badLinesRDD. To do that, we'll use two actions, count(), which returns the count as a number, and take(), which collects a number of elements from the RDD.

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
print line
```

Lazy Evaluation

- As you read earlier, transformations on RDDs are lazily evaluated, **meaning that Spark will not begin to execute until it sees an action.**

- Lazy evaluation means that when we call a transformation on an RDD (for instance, calling map**()), the operation is not immediately performed**.

- Instead, **Spark internally records metadata to indicate that this operation has been requested.**

- Rather than thinking of an RDD as containing specific data, it is best to think of each RDD as consisting of instructions on how to compute the data that we build up through transformations.

- Loading data into an RDD is lazily evaluated in the same way transformations are. So, when **we call sc.textFile(), the data is not loaded until it is necessary. As with transformations, the operation (in this case, reading the data) can occur multiple times.**

- Spark uses lazy evaluation to reduce the number of passes it has to take over our data
by grouping operations together.

Persistence (Caching)

- Spark RDDs are lazily evaluated, and sometimes we may wish to use the same RDD multiple times.

- If we do this naively, Spark will recompute the RDD and all of its dependencies each time we call an action on the RDD.

- This can be especially expensive for iterative algorithms, which look at the data many times. Another trivial example would be doing a count and then writing out the same RDD.

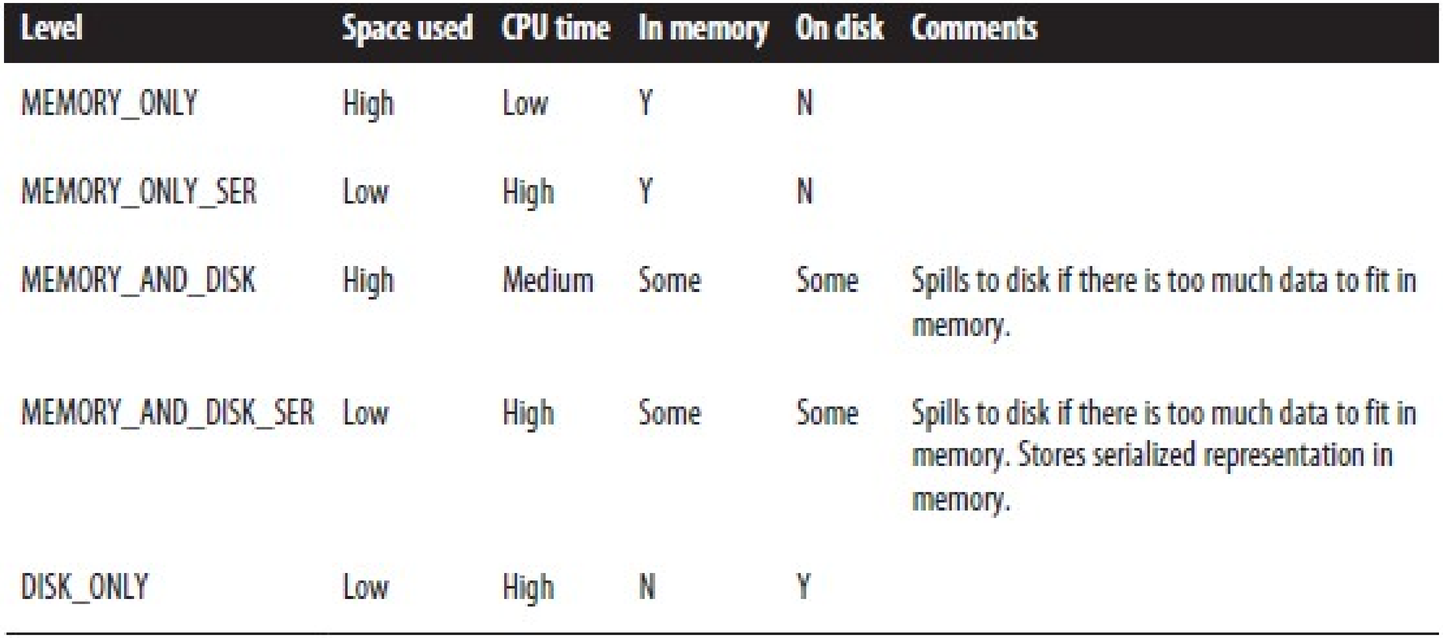
```
val result = input.map(x => x*x)
println(result.count())
println(result.collect().mkString(","))
```

- To avoid computing an RDD multiple times, we can ask Spark to *persist* the data. When we ask Spark to persist an RDD, the nodes that compute the RDD store their partitions.

- If a node that has data persisted on it **fails, Spark will recompute** the lost partitions of the data when needed. We can also **replicate our data on multiple nodes** if we want to be able to handle node failure without slowdown.

- Spark has many levels of persistence to choose from based on what our goals are, as you can see in Table

- In Scala and Java, the default persist() will store the data in the JVM heap as unserialized objects. In Python, we always serialize the data that persist stores, so the default is instead stored in the JVM heap as pickled objects. When we write data out to disk or off-heap storage, that data is also always serialized.

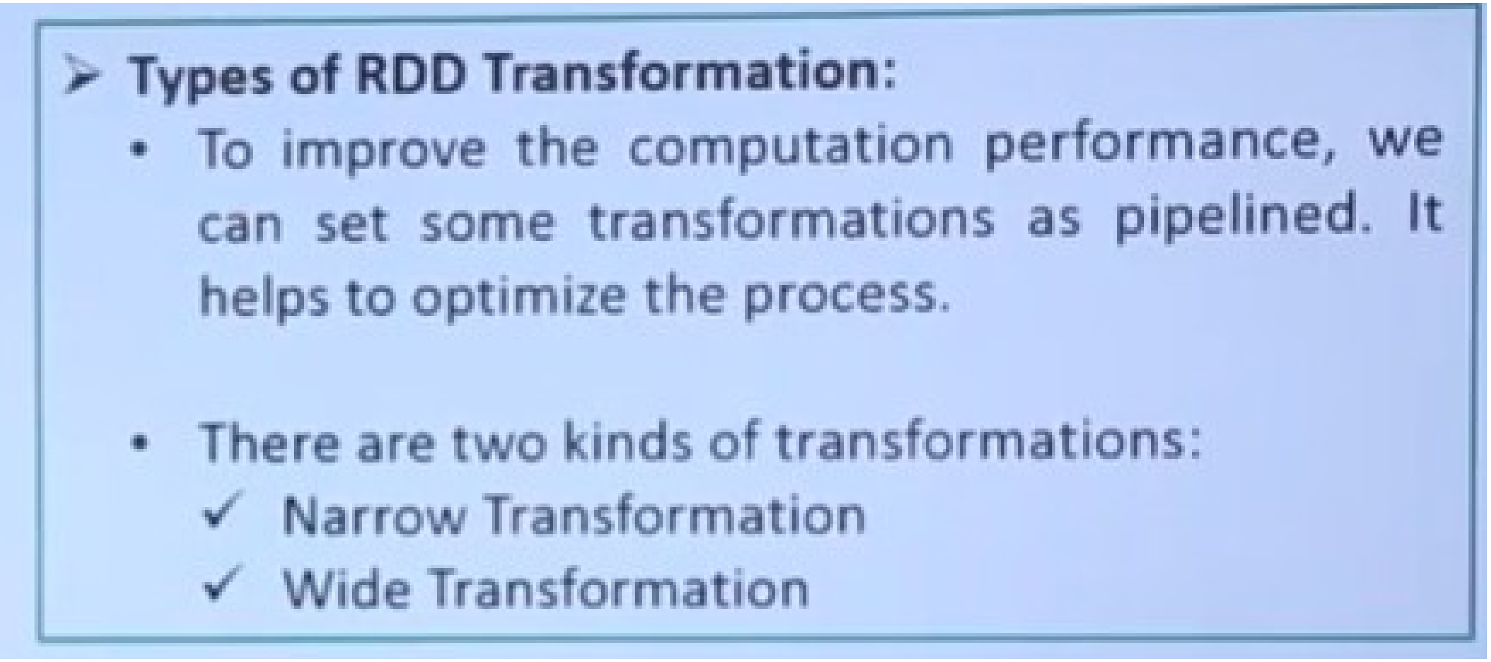


Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

- If you attempt **to cache too much data to fit in memory**, Spark will automatically evict old partitions using a **Least Recently Used (LRU) cache policy**.

- For the **memory only storage levels, it will recompute** these partitions the next time they are accessed, while for the **memory-and-disk ones, it will write them out to disk**. In either case, this means that you don’t have to worry about your job breaking if you ask Spark to cache too much data.

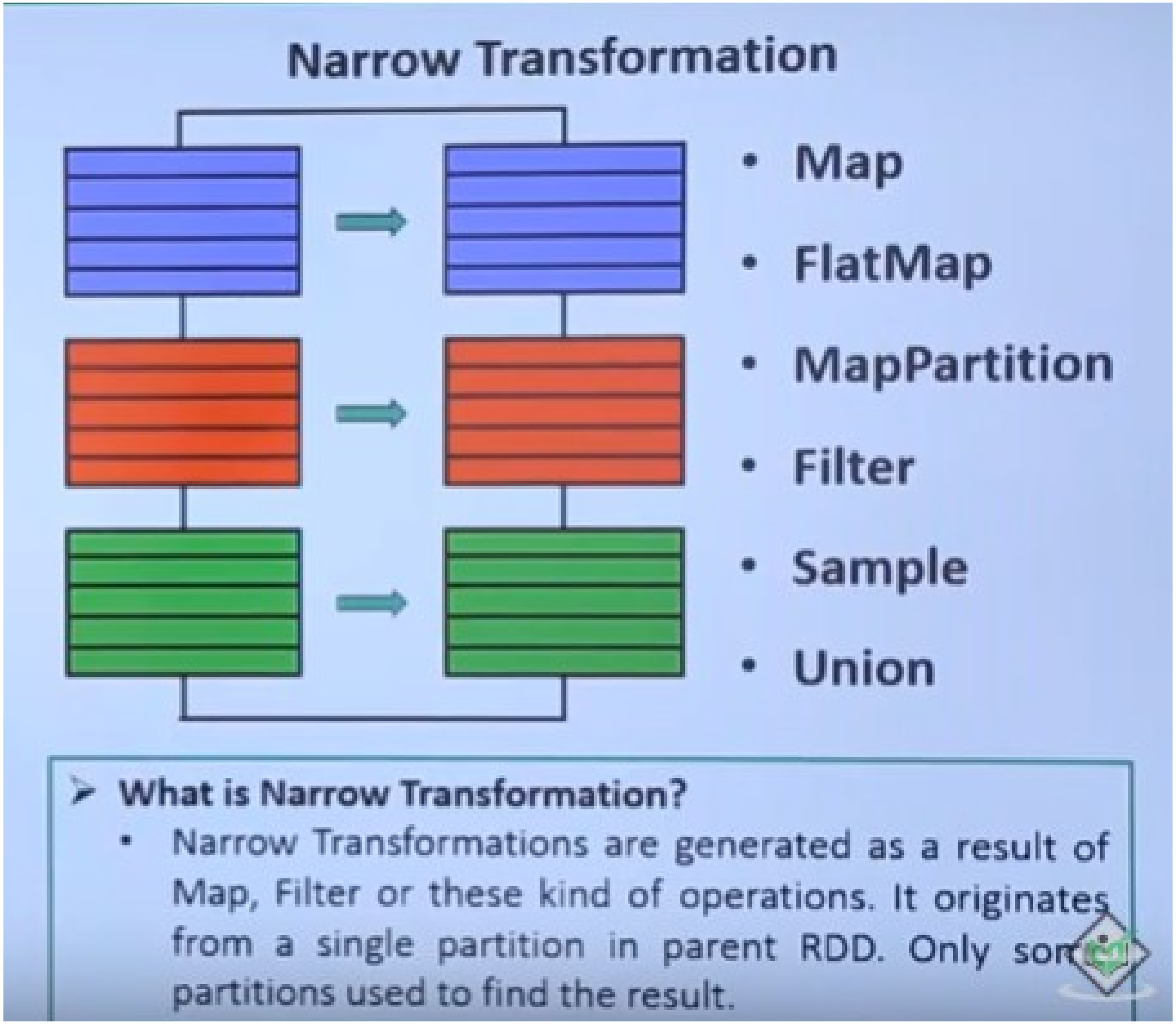
- However, caching unnecessary data can lead to eviction of useful data and more recomputation time.



➢ **Types of RDD Transformation:**

- To improve the computation performance, we can set some transformations as pipelined. It helps to optimize the process.

- There are two kinds of transformations:
  - ✓ Narrow Transformation
  - ✓ Wide Transformation

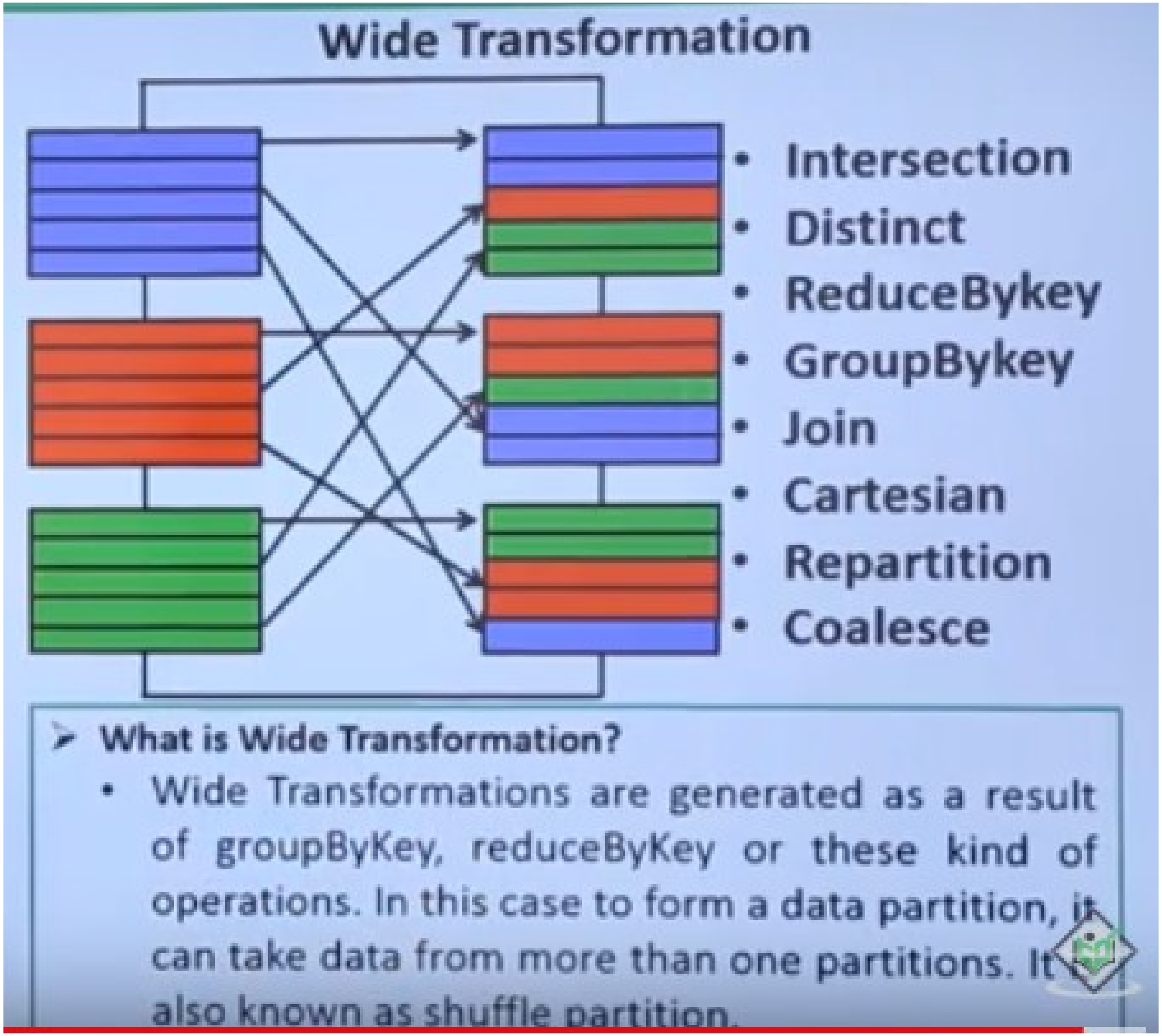


Narrow Transformation

- Map
- FlatMap
- MapPartition
- Filter
- Sample
- Union

> What is Narrow Transformation?

- Narrow Transformations are generated as a result of Map, Filter or these kind of operations. It originates from a single partition in parent RDD. Only som partitions used to find the result.



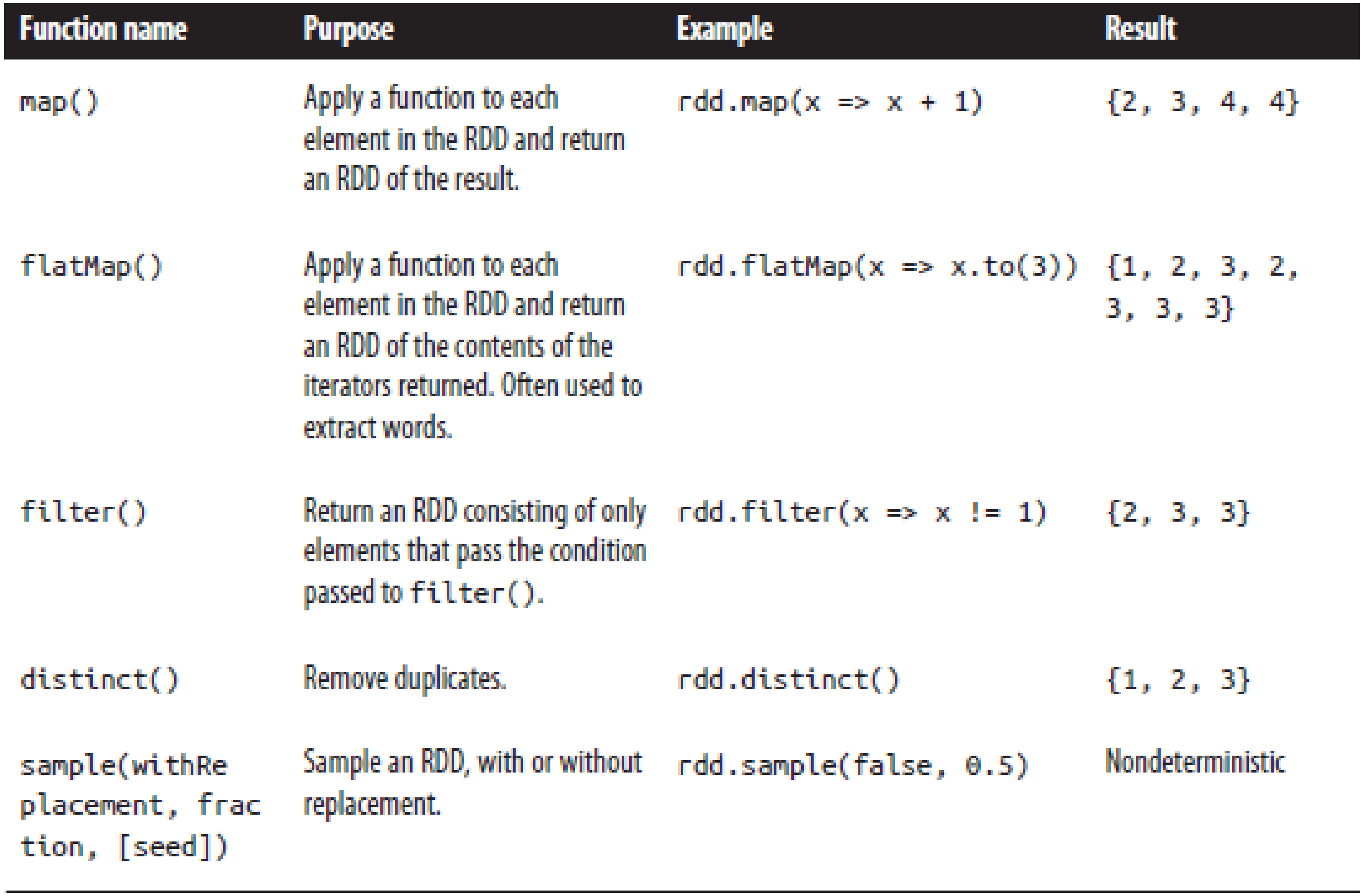
Wide Transformation

- Intersection
- Distinct
- ReduceBykey
- GroupBykey
- Join
- Cartesian
- Repartition
- Coalesce

➢ **What is Wide Transformation?**

- Wide Transformations are generated as a result of groupByKey, reduceByKey or these kind of operations. In this case to form a data partition, it can take data from more than one partitions. It is also known as shuffle partition.

Table 3-2. Basic RDD transformations on an RDD containing {1, 2, 3, 3}



Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

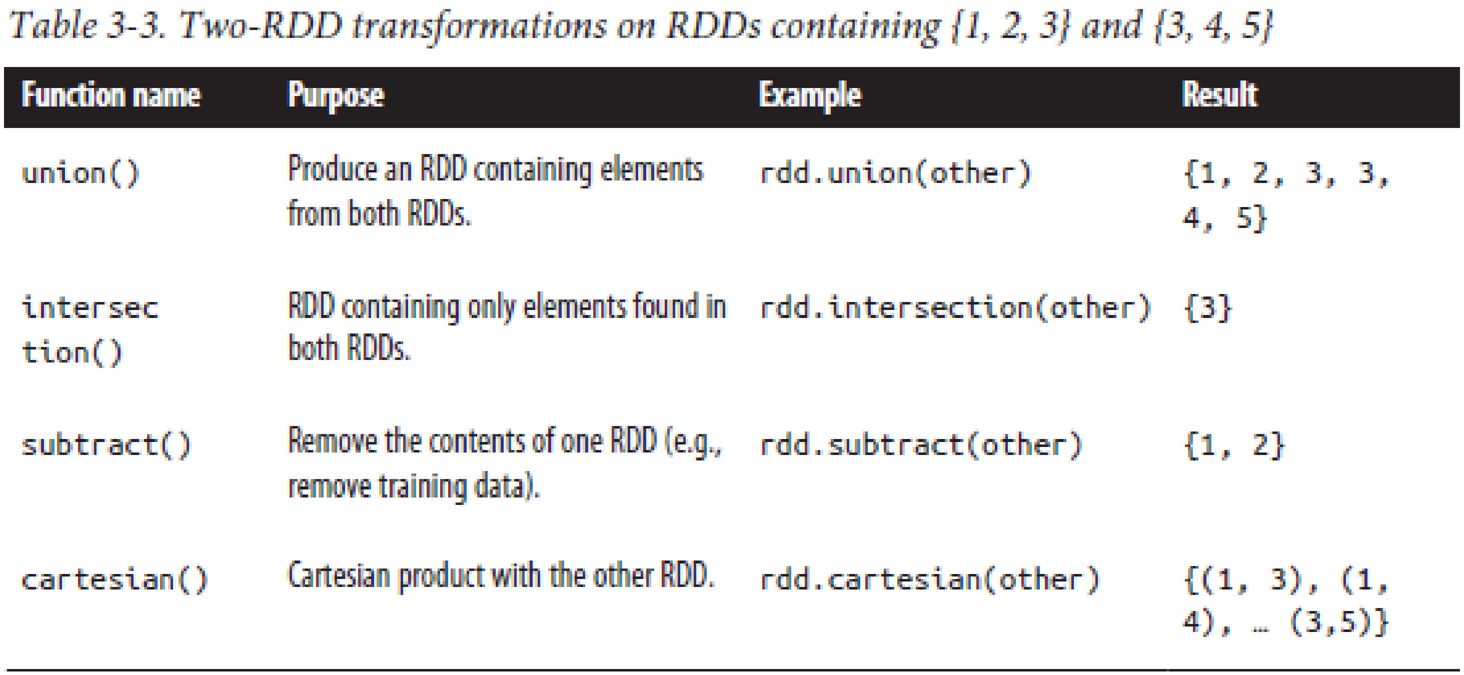


Table 3-3. Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
`union()`	Produce an RDD containing elements from both RDDs.	`rdd.union(other)`	`{1, 2, 3, 3, 4, 5}`
`intersection()`	RDD containing only elements found in both RDDs.	`rdd.intersection(other)`	`{3}`
`subtract()`	Remove the contents of one RDD (e.g., remove training data).	`rdd.subtract(other)`	`{1, 2}`
`cartesian()`	Cartesian product with the other RDD.	`rdd.cartesian(other)`	`{(1, 3), (1, 4), … (3,5)}`

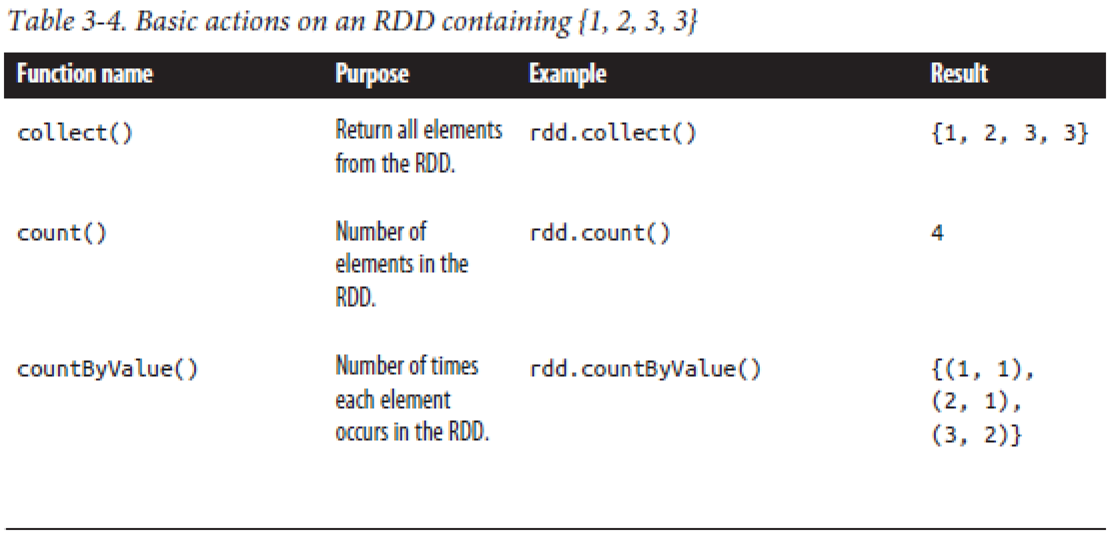
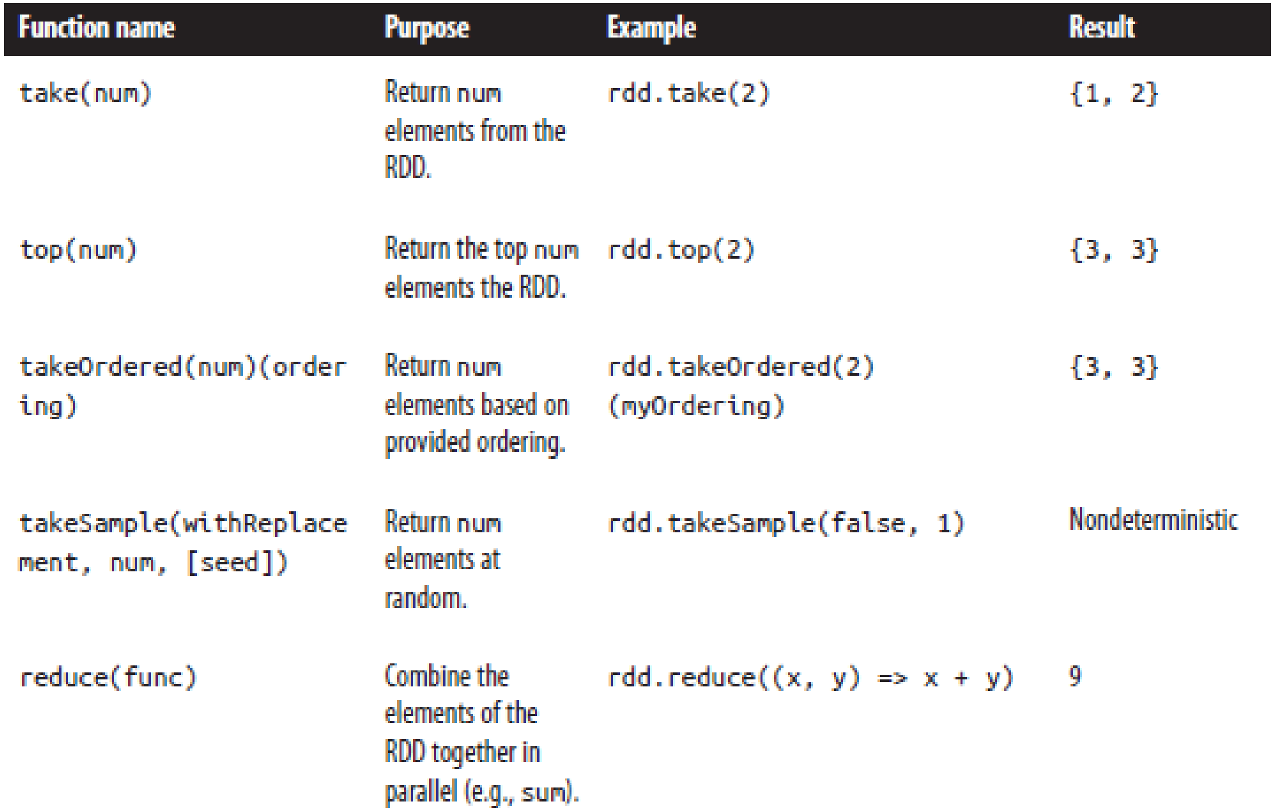


Table 3-4. Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`

`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1),`
			`(2, 1),`
			`(3, 2)}`



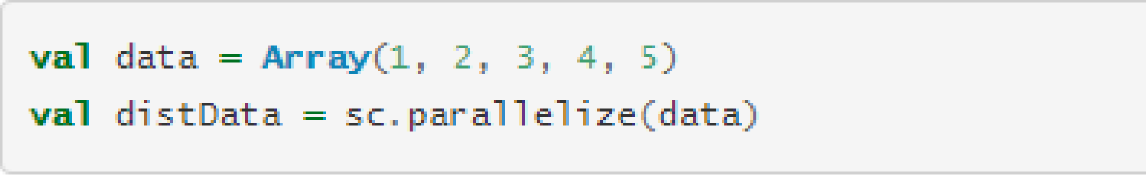
Function name	Purpose	Example	Result
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return `num` elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplacement, num, [seed])`	Return `num` elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) => x + y)`	9

Parallelized Collections

- Parallelized collections are created by calling SparkContext's parallelize method on an existing collection in your driver program (a Scala Seq). The elements of the collection are copied to form a distributed dataset that can be operated on in parallel. For example, here is how to create a parallelized collection holding the numbers 1 to 5:

- Once created, the distributed dataset (distData) can be operated on in parallel. For example, we might call distData.reduce((a, b) => a + b)to add up the elements of the array.

- One important parameter for parallel collections is the number of *partitions* to cut the dataset into. Spark will run one task for each partition of the cluster. Typically you want 2-4 partitions for each CPU in your cluster. Normally, Spark tries to set the number of partitions automatically based on your cluster. However, you can also set it manually by passing it as a second parameter to parallelize (e.g. sc.parallelize(data, 10)).



```scala
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```