Tarbiat Modares University

# Introduction to APACHE Spark™

Amirreza Najafi – Fall 2021

## “Big Data and Distributed Computing at Google”

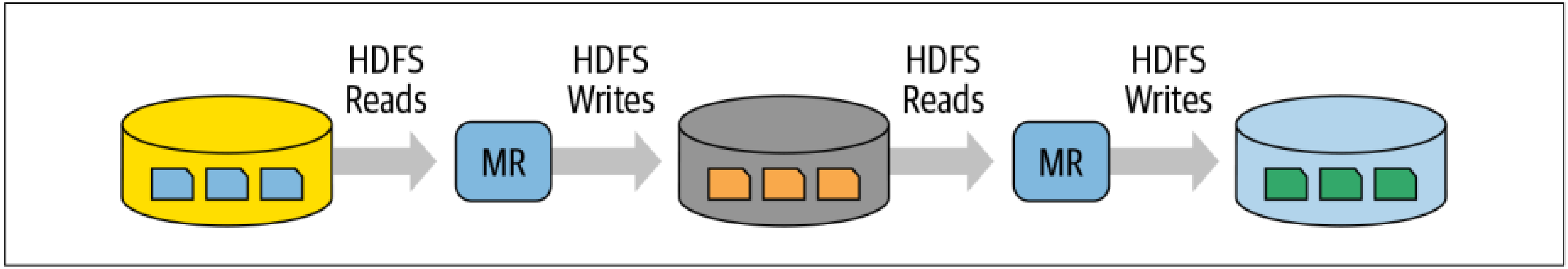
Neither traditional storage systems such as relational database management systems (RDBMSs) nor imperative ways of programming were able to handle the scale at which Google wanted to build and search the internet’s indexed documents. The resulting need for new approaches led to the creation of the *Google File System (GFS)*, *MapReduce (MR)*, and *Bigtable*.

## “Problems”

*First*, it was hard to manage and administer, with cumbersome operational complexity. *Second*, its general batch-processing MapReduce API was verbose and required a lot of boilerplate setup code, with brittle fault tolerance.

## “Problems”

*Third*, with large batches of data jobs with many pairs of MR tasks, each pair’s intermediate computed result is written to the local disk for the subsequent stage of its operation. This repeated performance of disk I/O took its toll: large MR jobs could run for hours on end, or even days.



## “Spark’s Early Years at AMPLab”

*Early papers* published on Spark demonstrated that it was 10 to 20 times faster than Hadoop MapReduce for certain jobs. Today, it’s *many orders of magnitude faster*.

## “Spark’s Idea”

The central thrust of the Spark project was to bring in ideas borrowed from Hadoop MapReduce, but to enhance the system: make it highly fault tolerant and embarrassingly parallel, support in-memory storage for intermediate results between iterative and interactive map and reduce computations, offer easy and composable APIs in multiple languages as a programming model, and support other workloads in a unified manner. We’ll come back to this idea of unification shortly, as it’s an important theme in Spark.

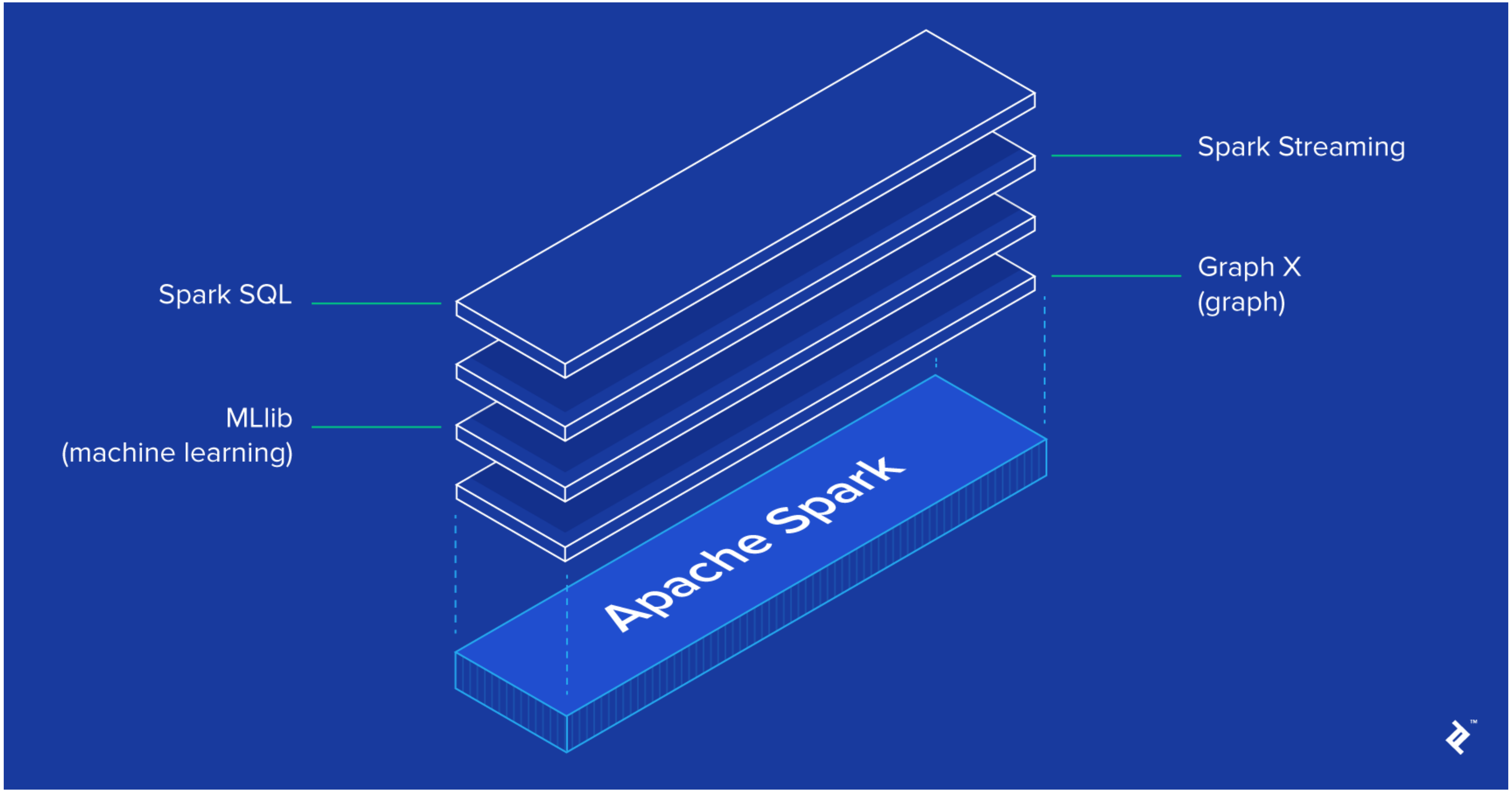
EASIER

FASTER

SMARTER

## “What Is Apache Spark?”

*Apache Spark* is a unified engine designed for large-scale distributed data processing, on premises in data centers or in the cloud.


Spark Streaming
Graph X
(graph)
Spark SQL
MLlib
(machine learning)
Apache Spark

## “Spark’s design philosophy?”

- Speed
- Ease of use
- Modularity
- Extensibility

## **“Speed”**

*First*, its internal implementation benefits immensely from the hardware industry’s recent huge strides in improving the price and performance of CPUs and memory.
*Second*, Spark builds its query computations as a directed acyclic graph (DAG).

## “Ease of use”

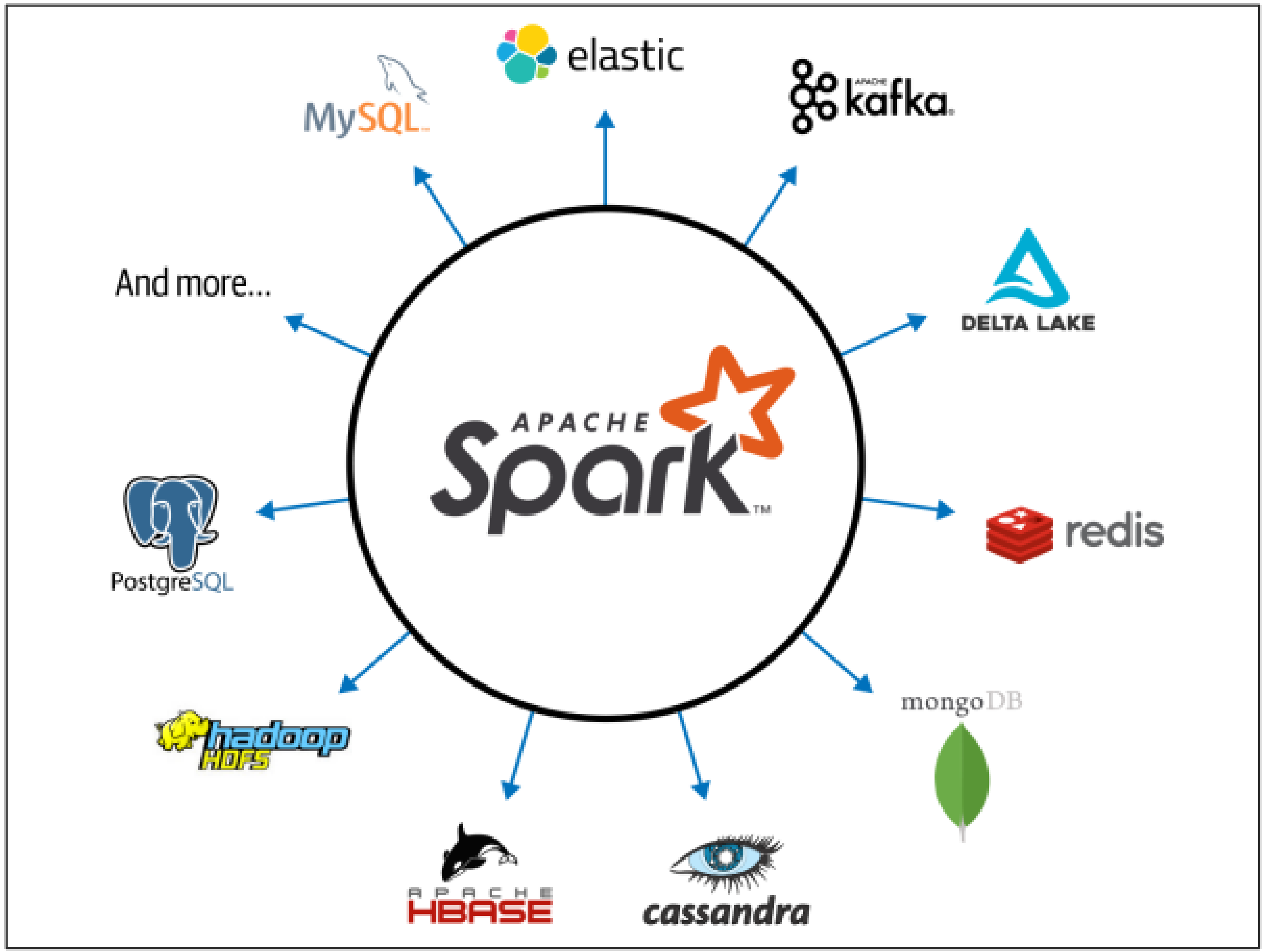
Spark achieves simplicity by providing a fundamental abstraction of a simple logical data structure called a Resilient Distributed Dataset (RDD) upon which all other higher-level structured data abstractions, such as DataFrames and Datasets, are con- structed. By providing a set of transformations and actions as operations, Spark offers a simple programming model that you can use to build big data applications in familiar languages.

## “Modularity”

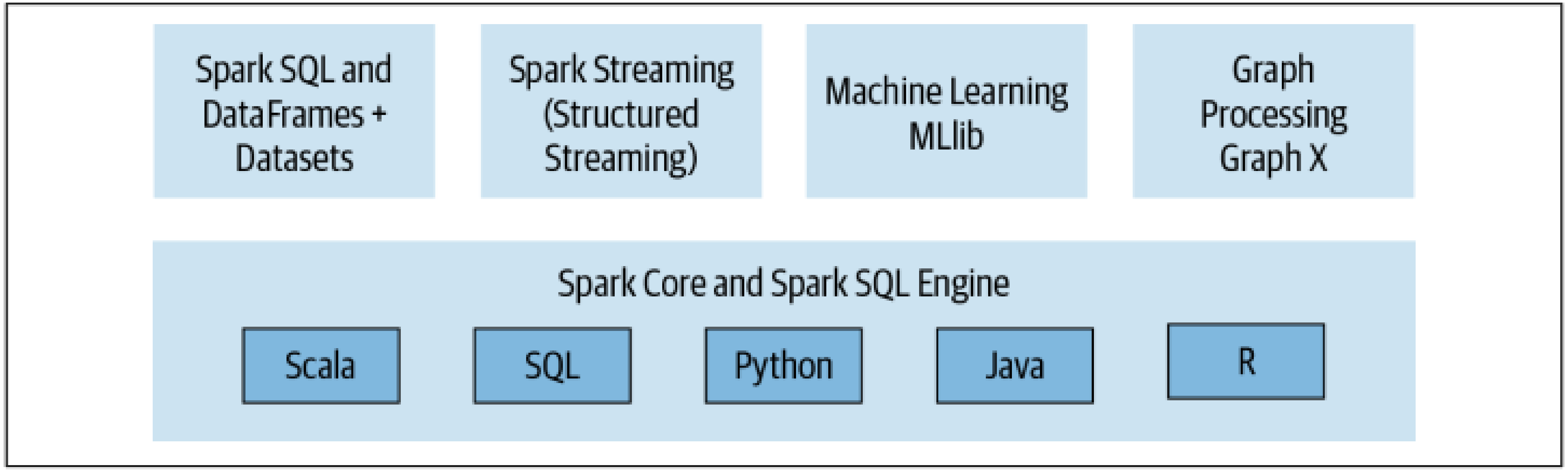
Spark operations can be applied across many types of workloads and expressed in any of the supported programming languages: Scala, Java, Python, SQL, and R. Spark offers unified libraries with well-documented APIs that include the following mod- ules as core components: Spark SQL, Spark Structured Streaming, Spark MLlib, and GraphX, combining all the workloads running under one engine. We’ll take a closer look at all of these in the next section.

## “Extensibility”

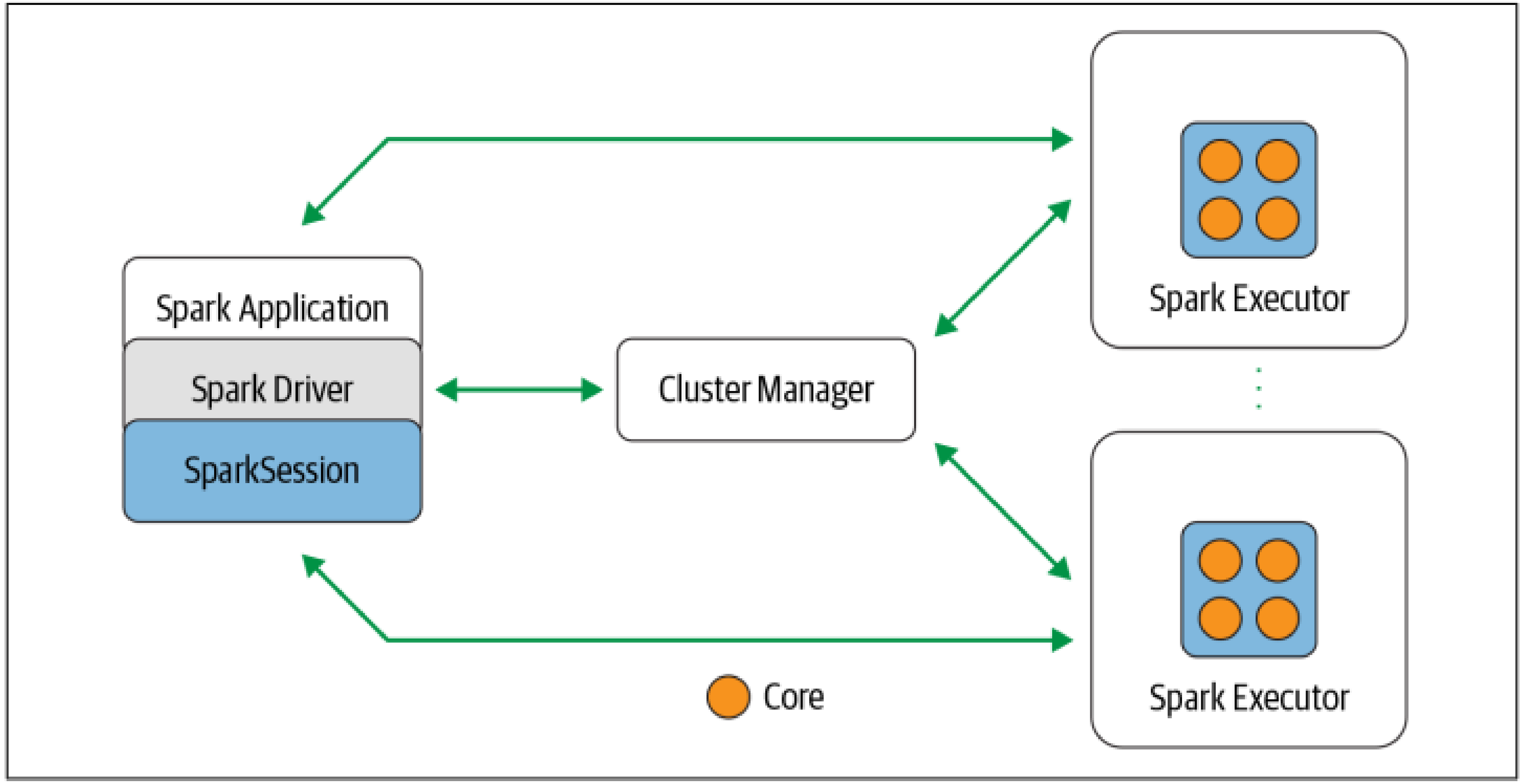
Spark focuses on its fast, parallel computation engine rather than on storage. Unlike Apache Hadoop, which included both storage and compute, Spark decouples the two. That means you can use Spark to read data stored in myriad sources—Apache Hadoop, Apache Cassandra, Apache HBase, MongoDB, Apache Hive, RDBMSs, and more—and process it all in memory.


elastic
MySQL
kafka
And more...
APACHE Spark
DELTA LAKE
redis
PostgreSQL
mongoDB
hadoop HDFS
APACHE HBASE
cassandra

## “Apache Spark Components as a Unified Stack”


Spark SQL and DataFrames + Datasets
Spark Streaming (Structured Streaming)
Machine Learning MLlib
Graph Processing Graph X
Spark Core and Spark SQL Engine
Scala
SQL
Python
Java
R

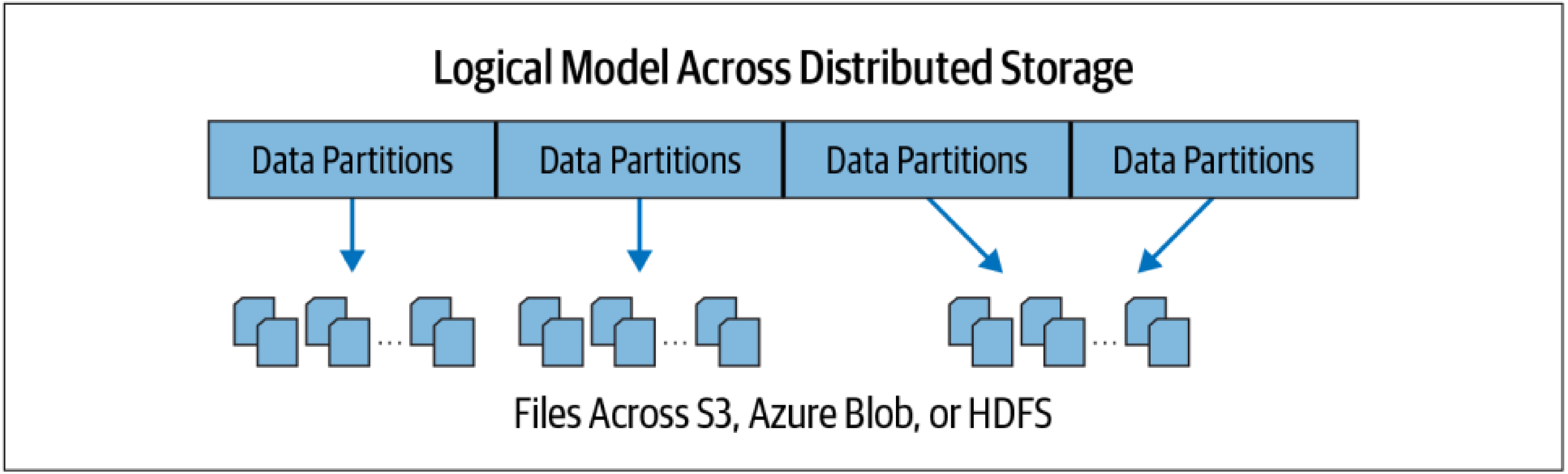
## “Apache Spark’s Distributed Execution”



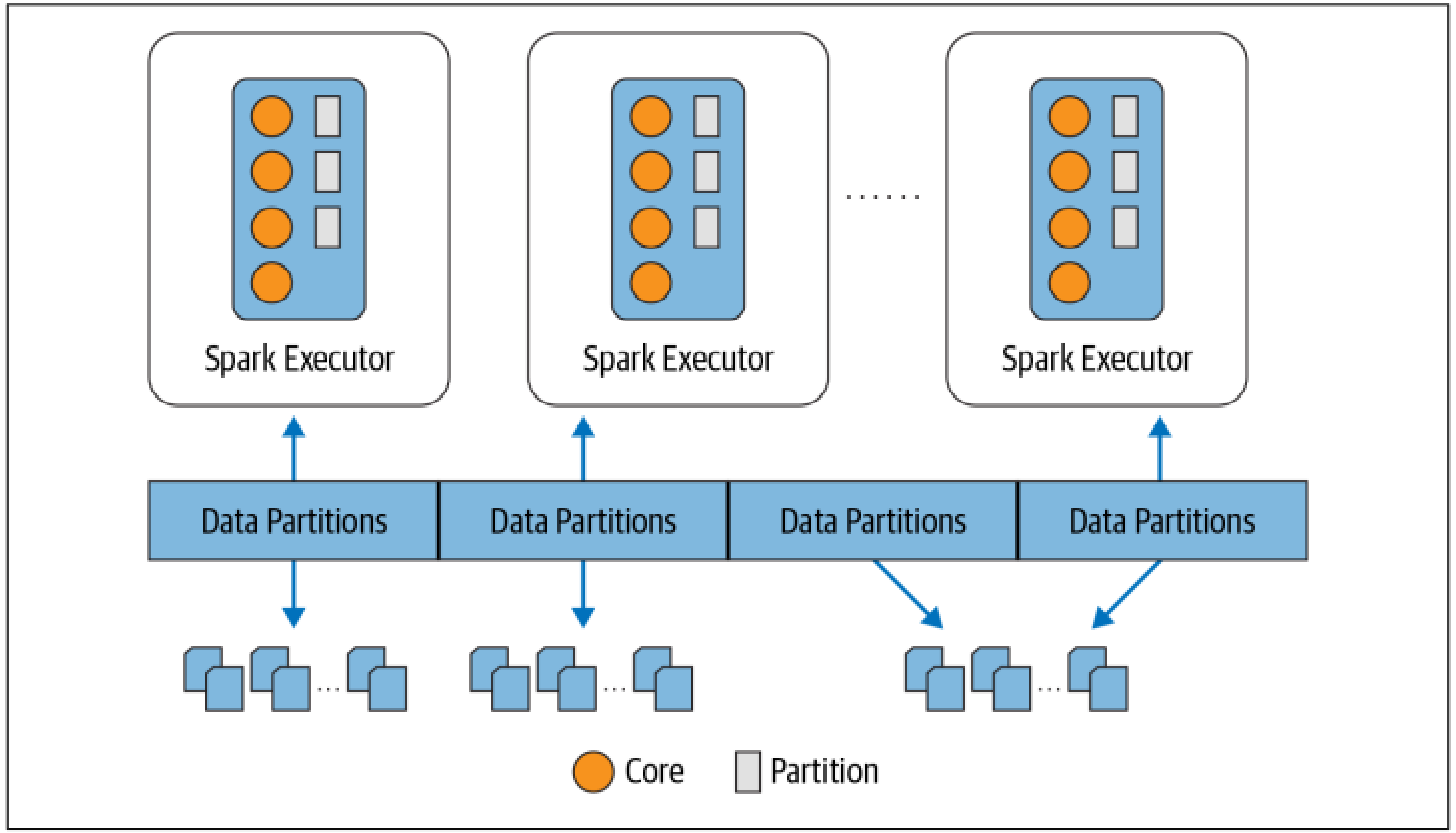
## “Deployment modes”

| Mode | Spark driver | Spark executor | Cluster manager |
|---|---|---|---|
| Local | Runs on a single JVM, like a laptop or single node | Runs on the same JVM as the driver | Runs on the same host |
| Standalone | Can run on any node in the cluster | Each node in the cluster will launch its own executor JVM | Can be allocated arbitrarily to any host in the cluster |
| YARN (client) | Runs on a client, not part of the cluster | YARN’s NodeManager’s container | YARN’s Resource Manager works with YARN’s Application Master to allocate the containers on NodeManagers for executors |
| YARN (cluster) | Runs with the YARN Application Master | Same as YARN client mode | Same as YARN client mode |
| Kubernetes | Runs in a Kubernetes pod | Each worker runs within its own pod | Kubernetes Master |

## “Distributed data and partitions”


Logical Model Across Distributed Storage
Data Partitions
Data Partitions
Data Partitions
Data Partitions
Files Across S3, Azure Blob, or HDFS

## “Distributed data and partitions”



## “Understanding Spark Application Concepts”

***Application***

A user program built on Spark using its APIs. It consists of a driver program and executors on the cluster.

***SparkSession***

An object that provides a point of entry to interact with underlying Spark functionality and allows programming Spark with its APIs. In an interactive Spark shell, the Spark driver instantiates a SparkSession for you, while in a Spark application, you create a SparkSession object yourself.

***Job***

A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g., save(), collect()).

***Stage***

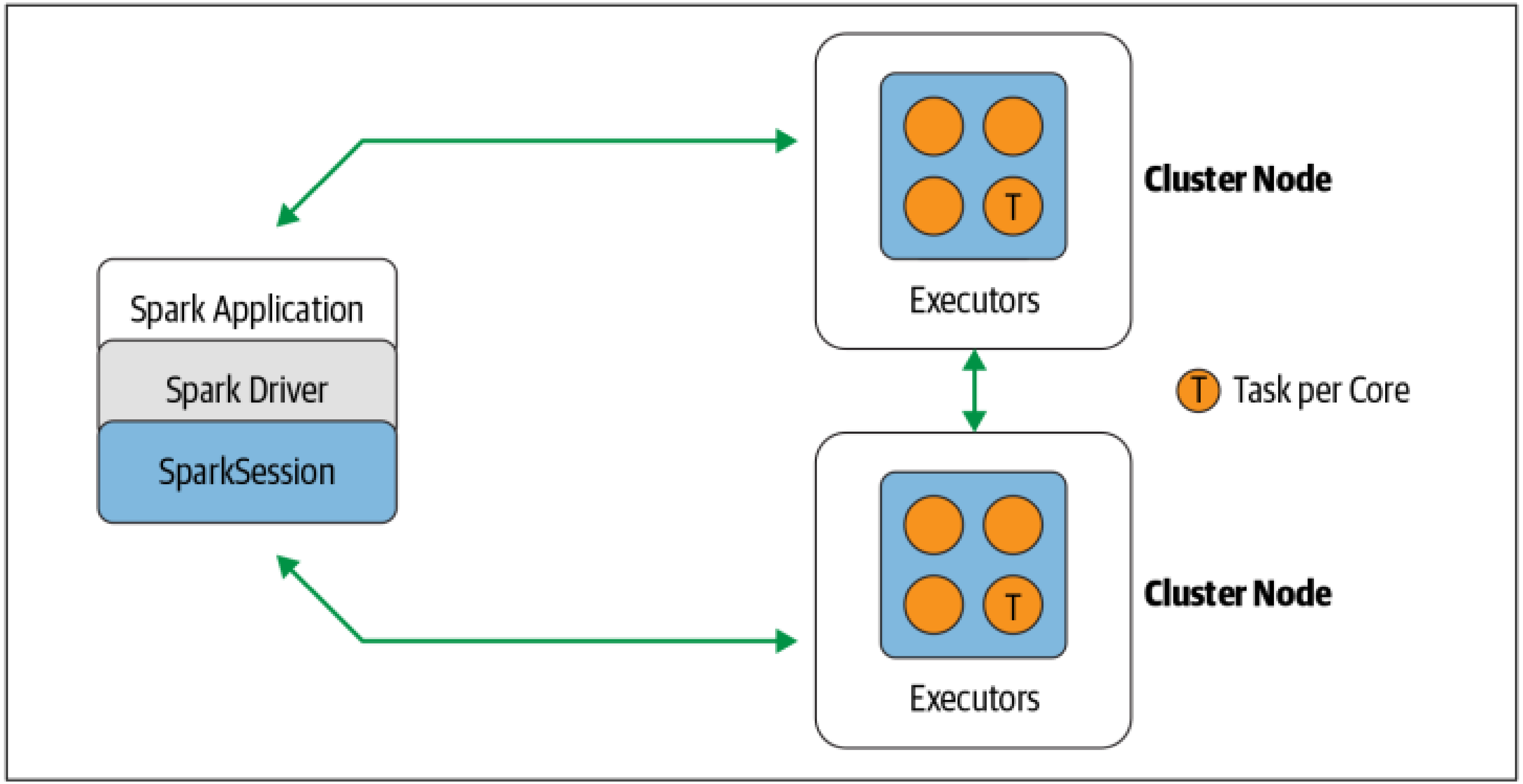
Each job gets divided into smaller sets of tasks called stages that depend on each other.

***Task***

A single unit of work or execution that will be sent to a Spark executor.

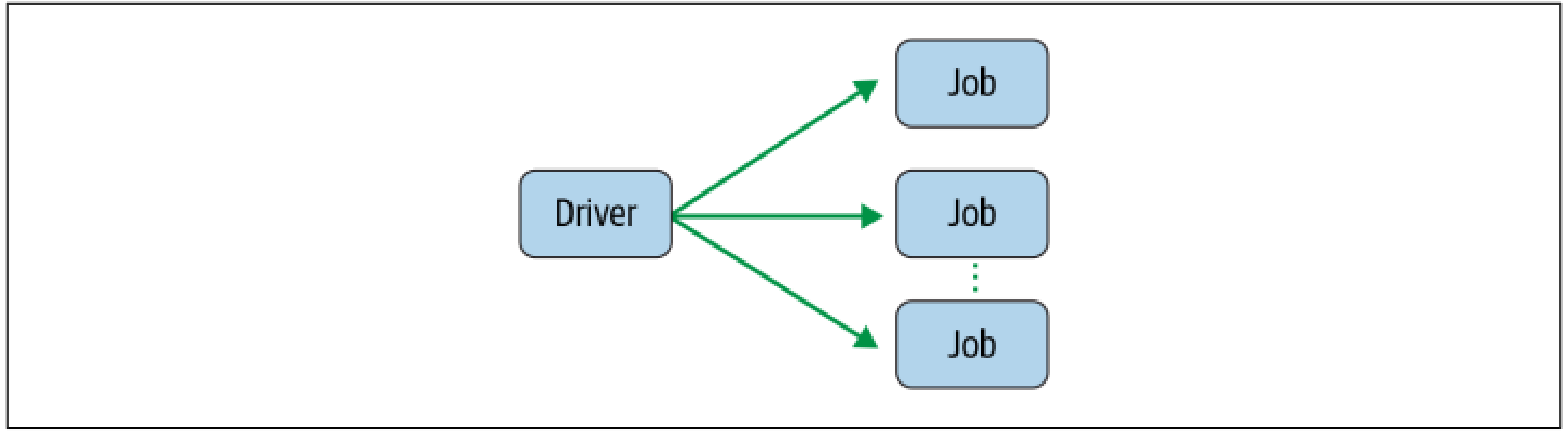
## “Spark Application and SparkSession”

At the core of every Spark application is the Spark driver program, which creates a SparkSession object. When you’re working with a Spark shell, the driver is part of the shell and the SparkSession object (accessible via the variable spark) is created for you, as you saw in the earlier examples when you launched the shells.


Spark Application
Spark Driver
SparkSession
Executors
T
Cluster Node
T Task per Core
Executors
T
Cluster Node

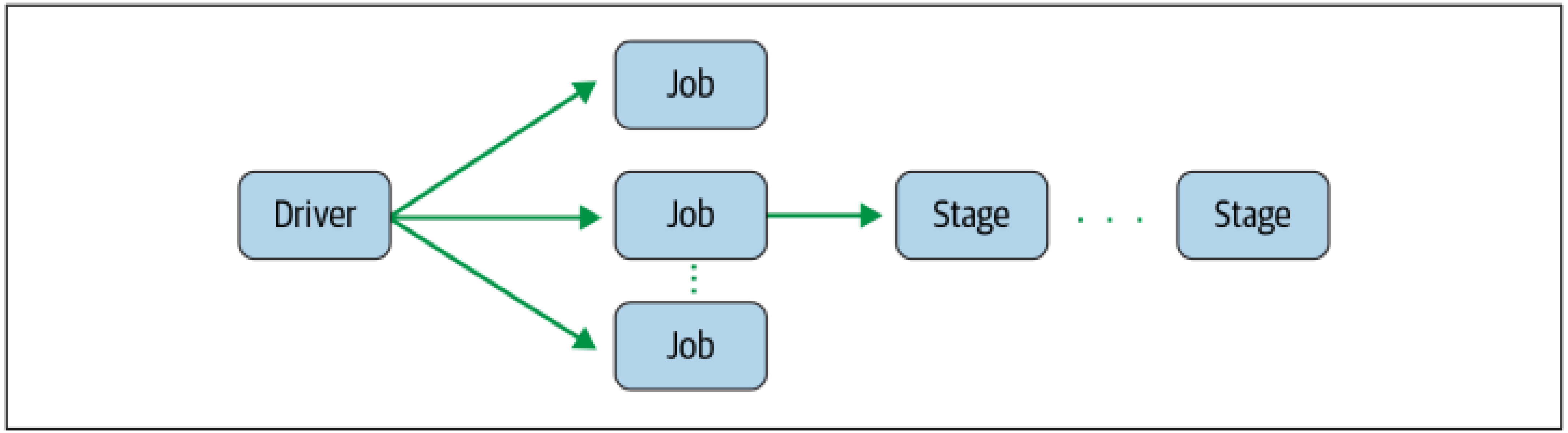
## “Spark Jobs”

The driver converts your Spark application into one or more Spark jobs. It then transforms each job into a DAG. This, in essence, is Spark’s execution plan, where each node within a DAG could be a single or multiple Spark stages.



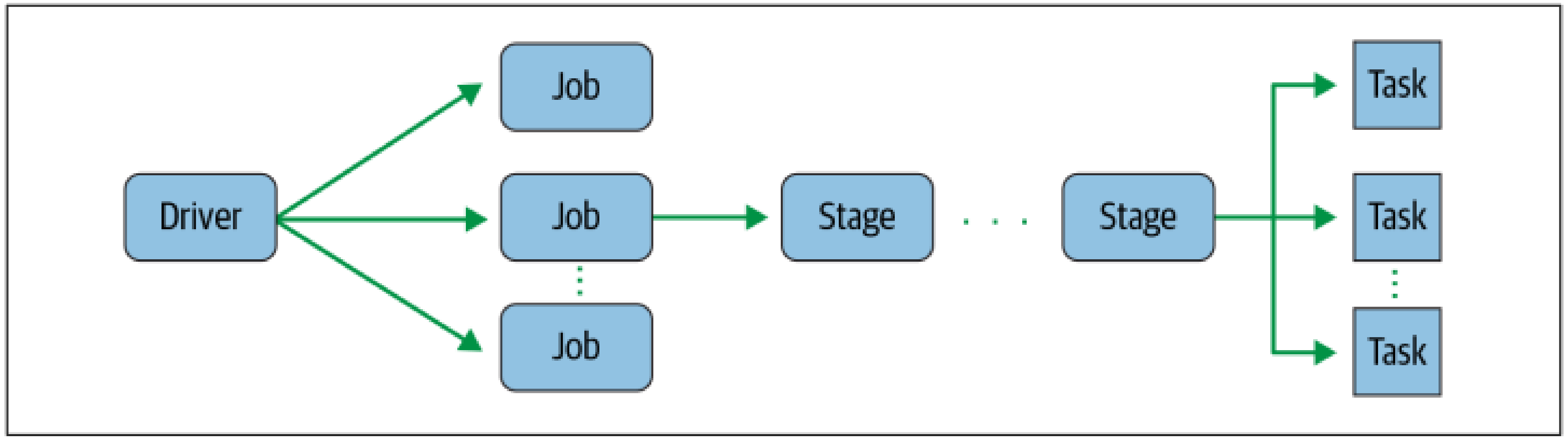
## “Spark Stages”

As part of the DAG nodes, stages are created based on what operations can be performed serially or in parallel.

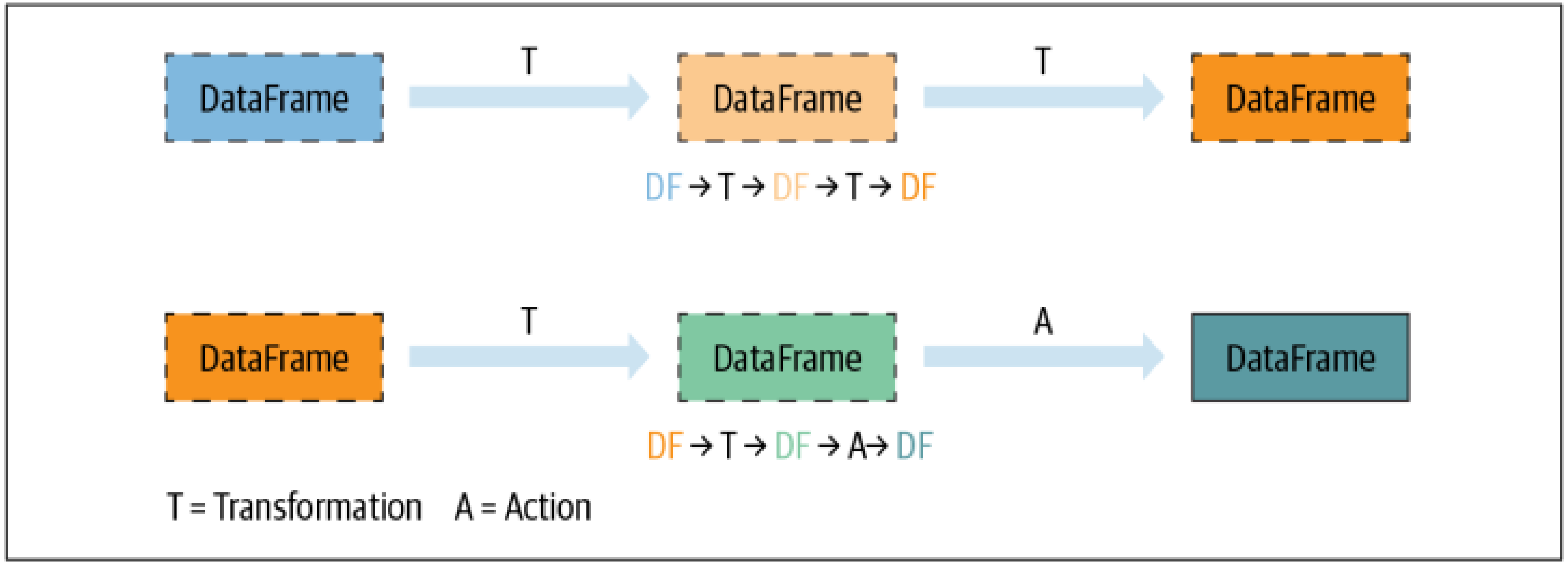


## “Spark Tasks”

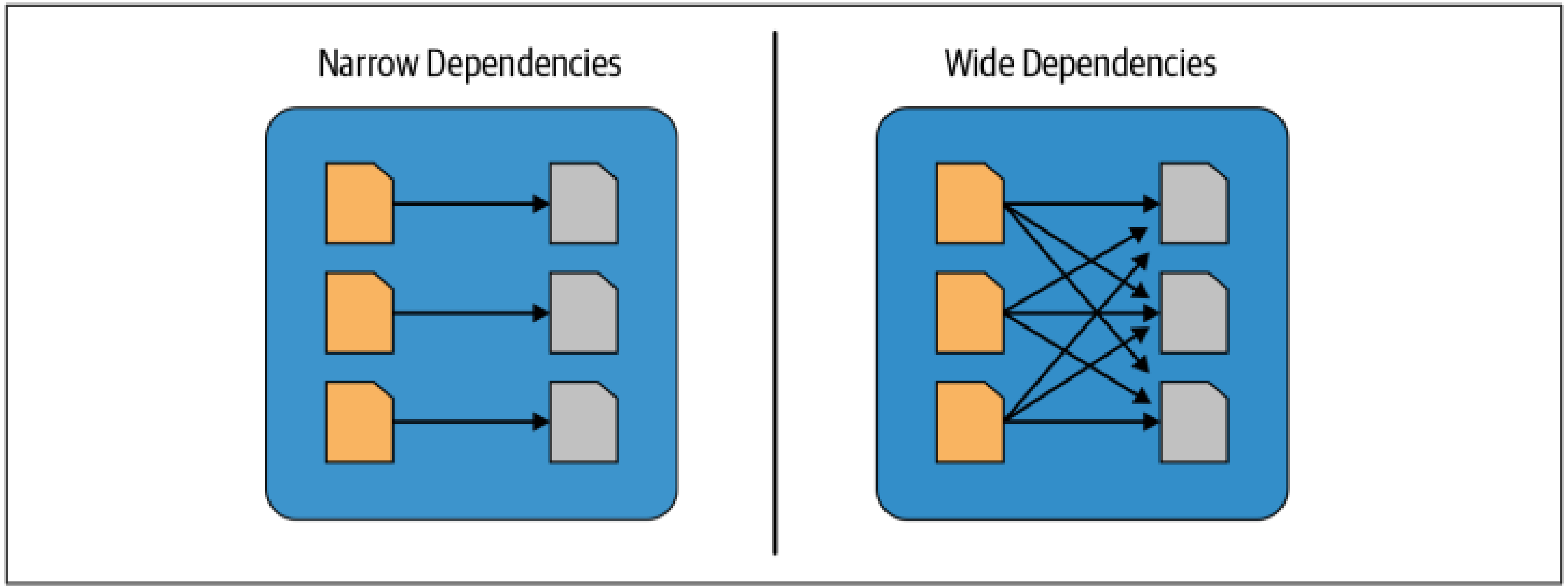
Each stage is comprised of Spark tasks (a unit of execution), which are then federated across each Spark executor; each task maps to a single core and works on a single partition of data



## “Transformations, Actions, and Lazy Evaluation”



## “Narrow and Wide Transformations”



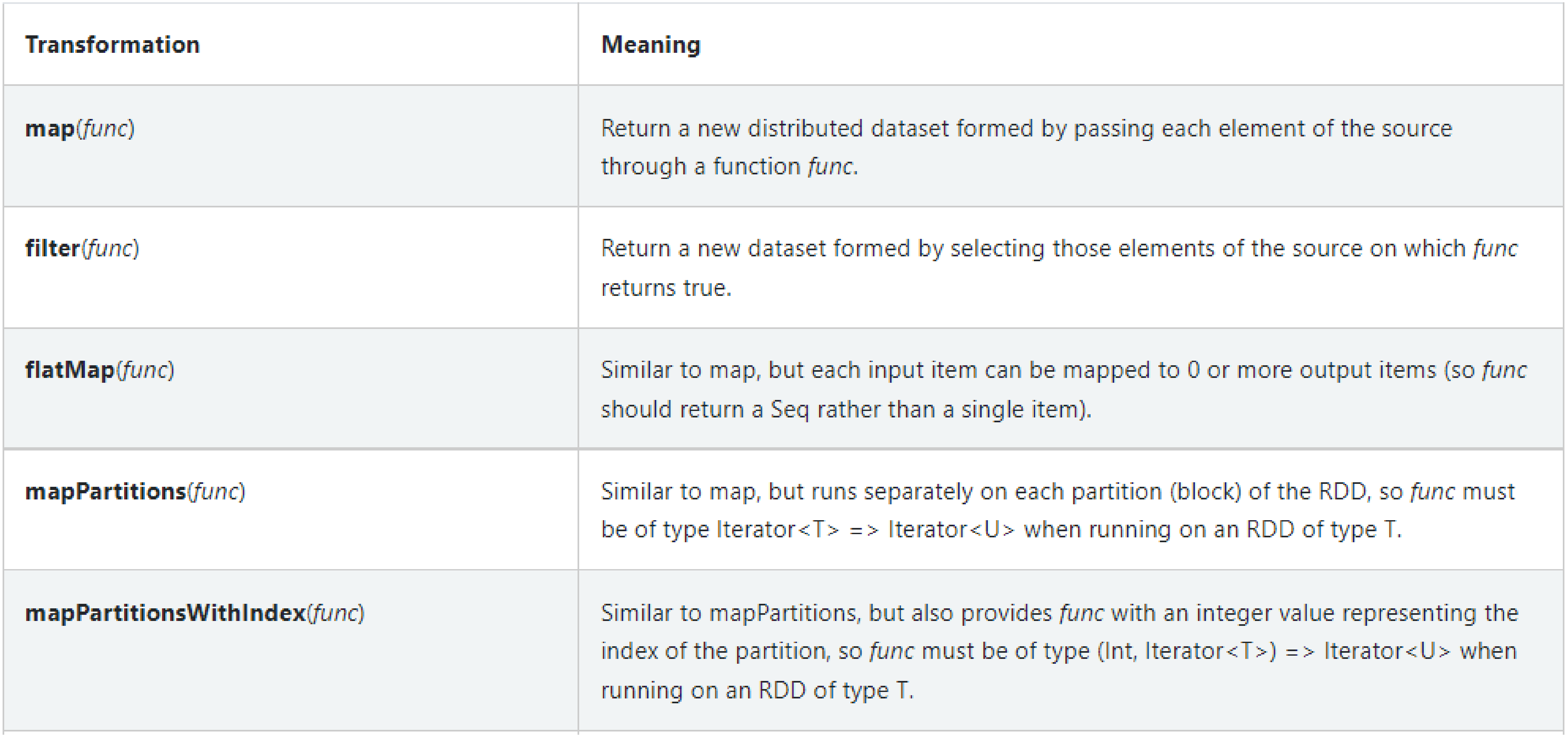
## “RDD, Resilient Distributed Datasets”

Spark revolves around the concept of a resilient distributed dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. There are two ways to create RDDs: parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

## “What’s Underneath an RDD?”

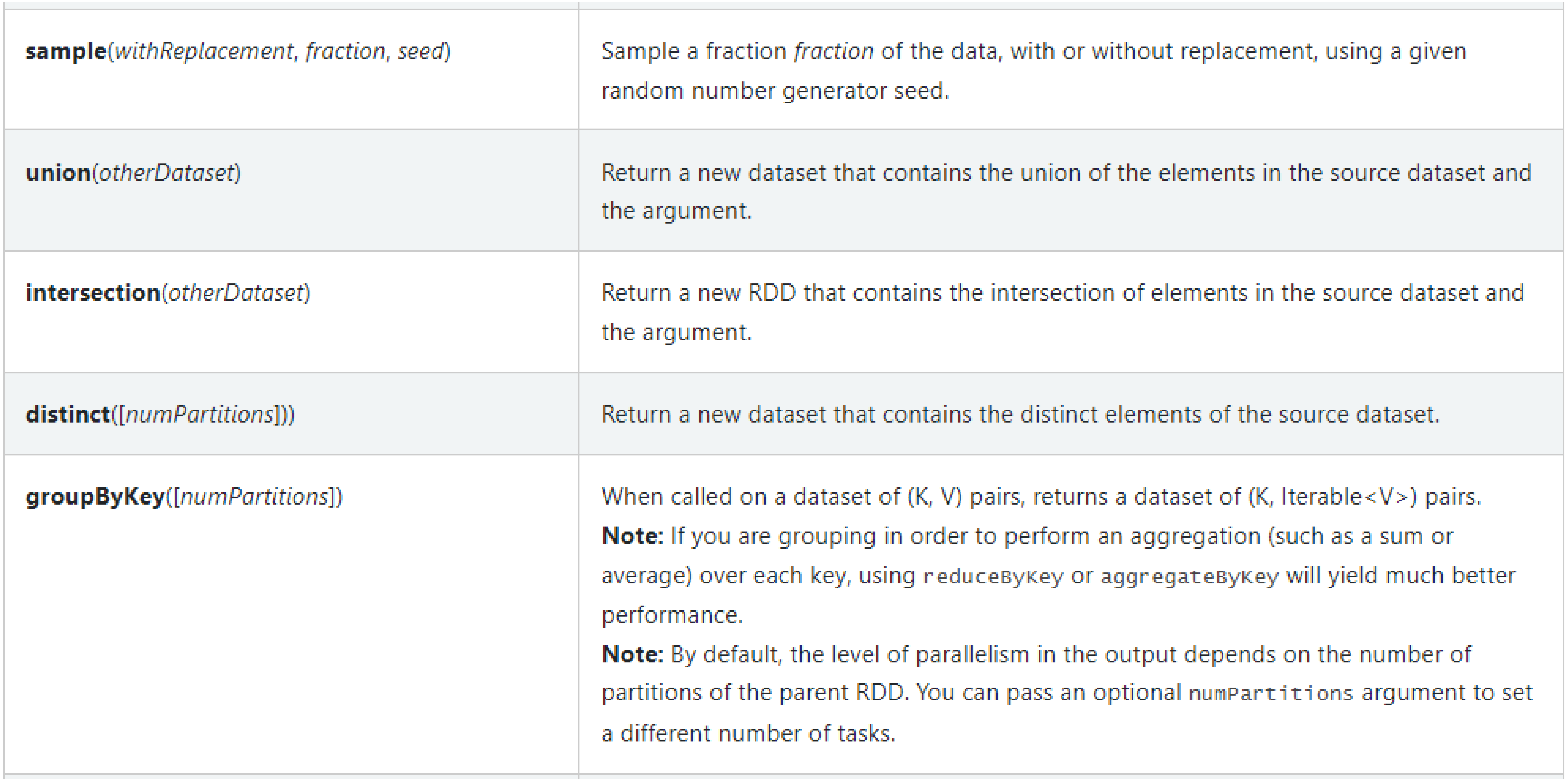
- Dependencies
- Partitions (with some locality information)
- Compute function: Partition => Iterator[T]

## “Transformations”



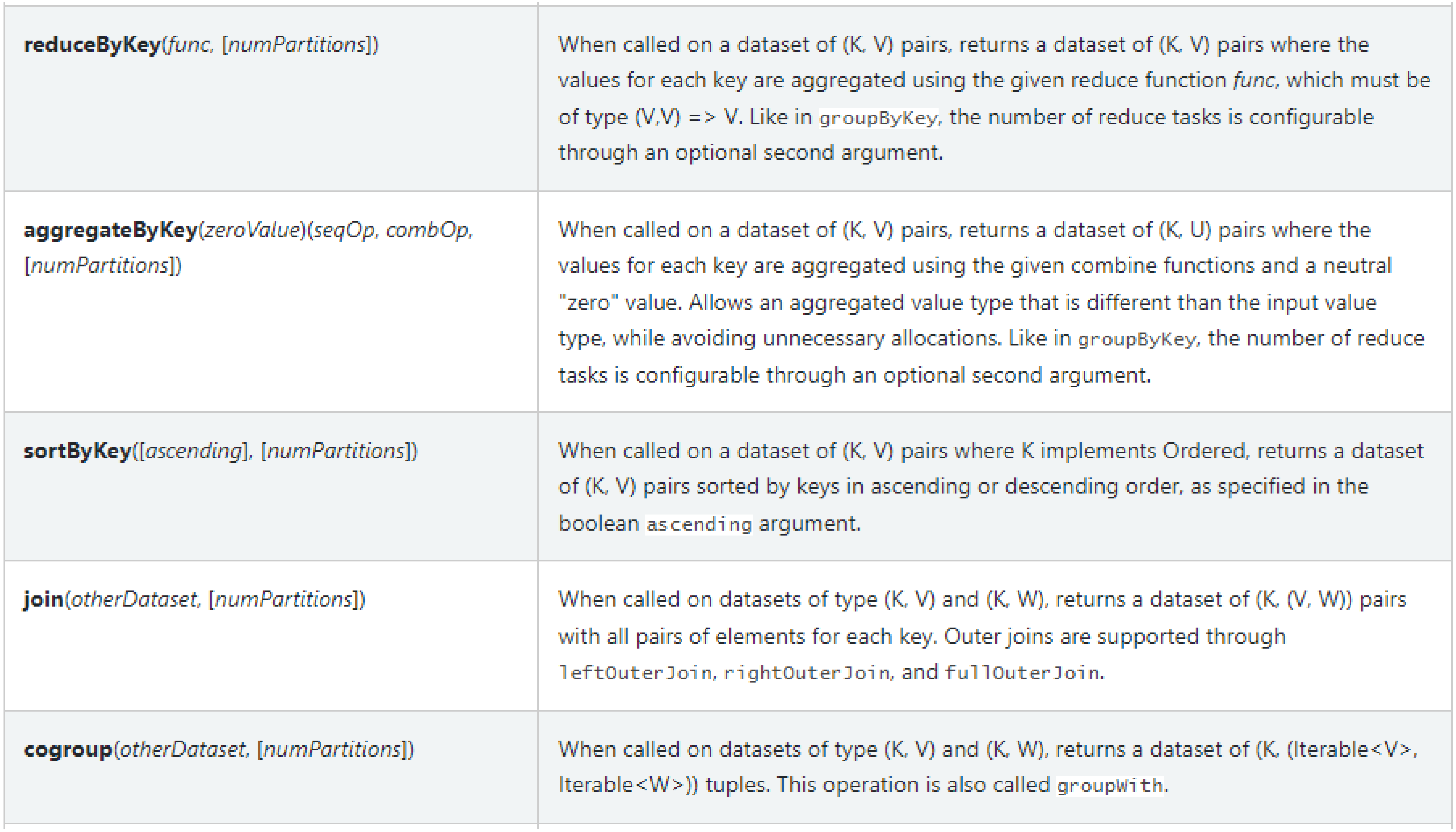
| Transformation | Meaning |
|---|---|
| **map**(*func*) | Return a new distributed dataset formed by passing each element of the source through a function *func*. |
| **filter**(*func*) | Return a new dataset formed by selecting those elements of the source on which *func* returns true. |
| **flatMap**(*func*) | Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item). |
| **mapPartitions**(*func*) | Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T. |
| **mapPartitionsWithIndex**(*func*) | Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T. |

## “Transformations”



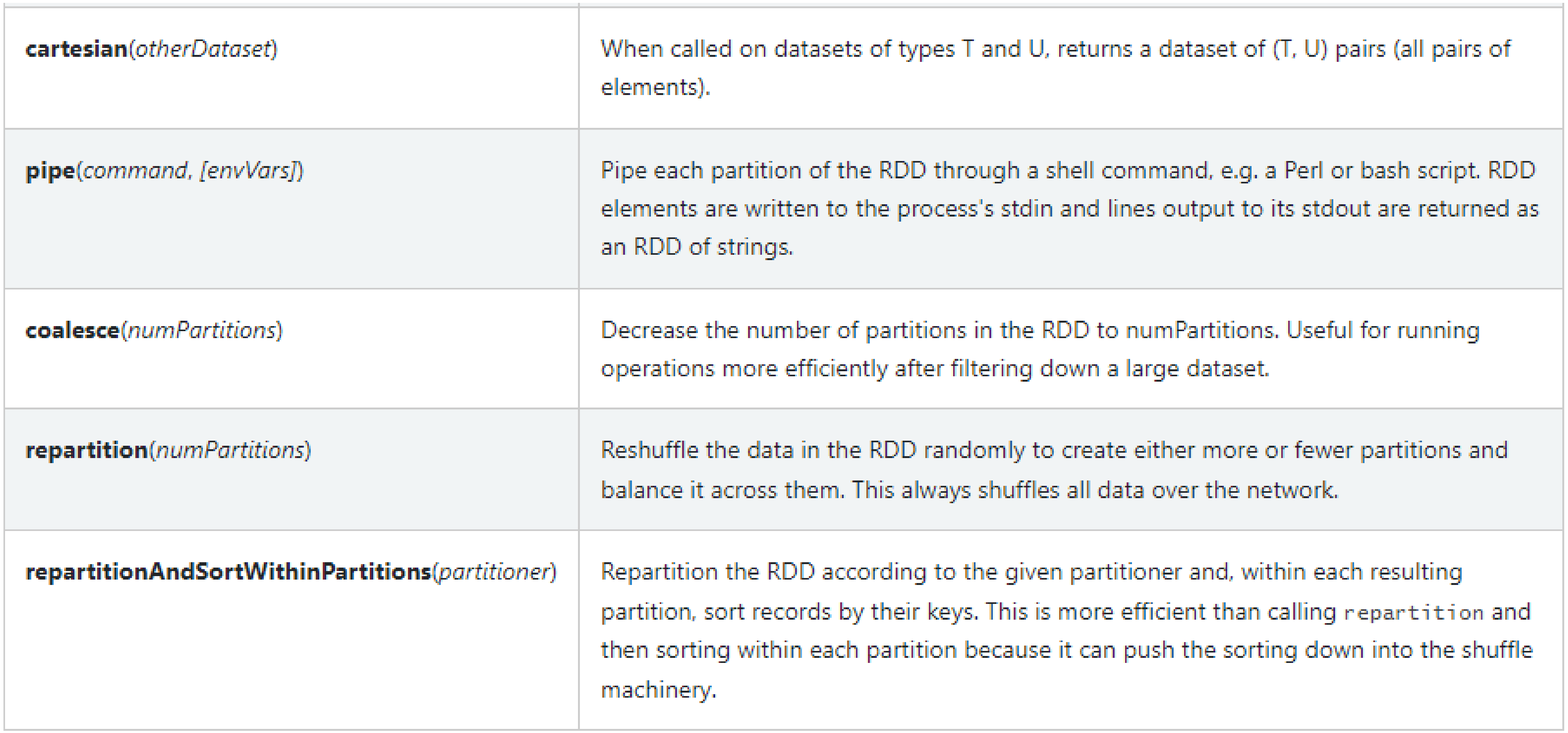
| | |
|---|---|
| **sample**(*withReplacement*, *fraction*, *seed*) | Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed. |
| **union**(*otherDataset*) | Return a new dataset that contains the union of the elements in the source dataset and the argument. |
| **intersection**(*otherDataset*) | Return a new RDD that contains the intersection of elements in the source dataset and the argument. |
| **distinct**([*numPartitions*])) | Return a new dataset that contains the distinct elements of the source dataset. |
| **groupByKey**([*numPartitions*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks. |

# “Transformations”



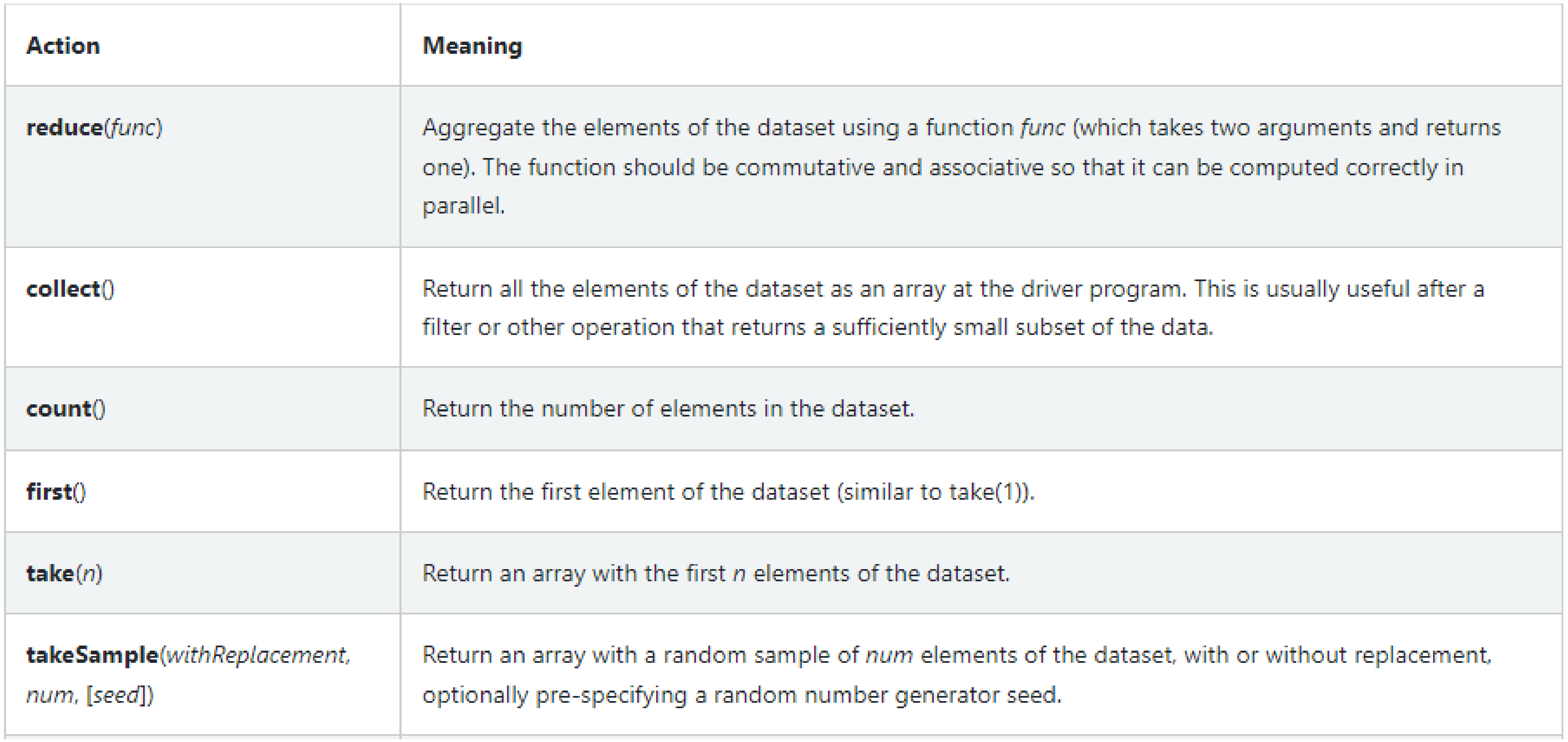
| | |
|---|---|
| **reduceByKey**(*func*, [*numPartitions*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument. |
| **aggregateByKey**(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument. |
| **sortByKey**([*ascending*], [*numPartitions*]) | When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument. |
| **join**(*otherDataset*, [*numPartitions*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`. |
| **cogroup**(*otherDataset*, [*numPartitions*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called `groupWith`. |

## “Transformations”



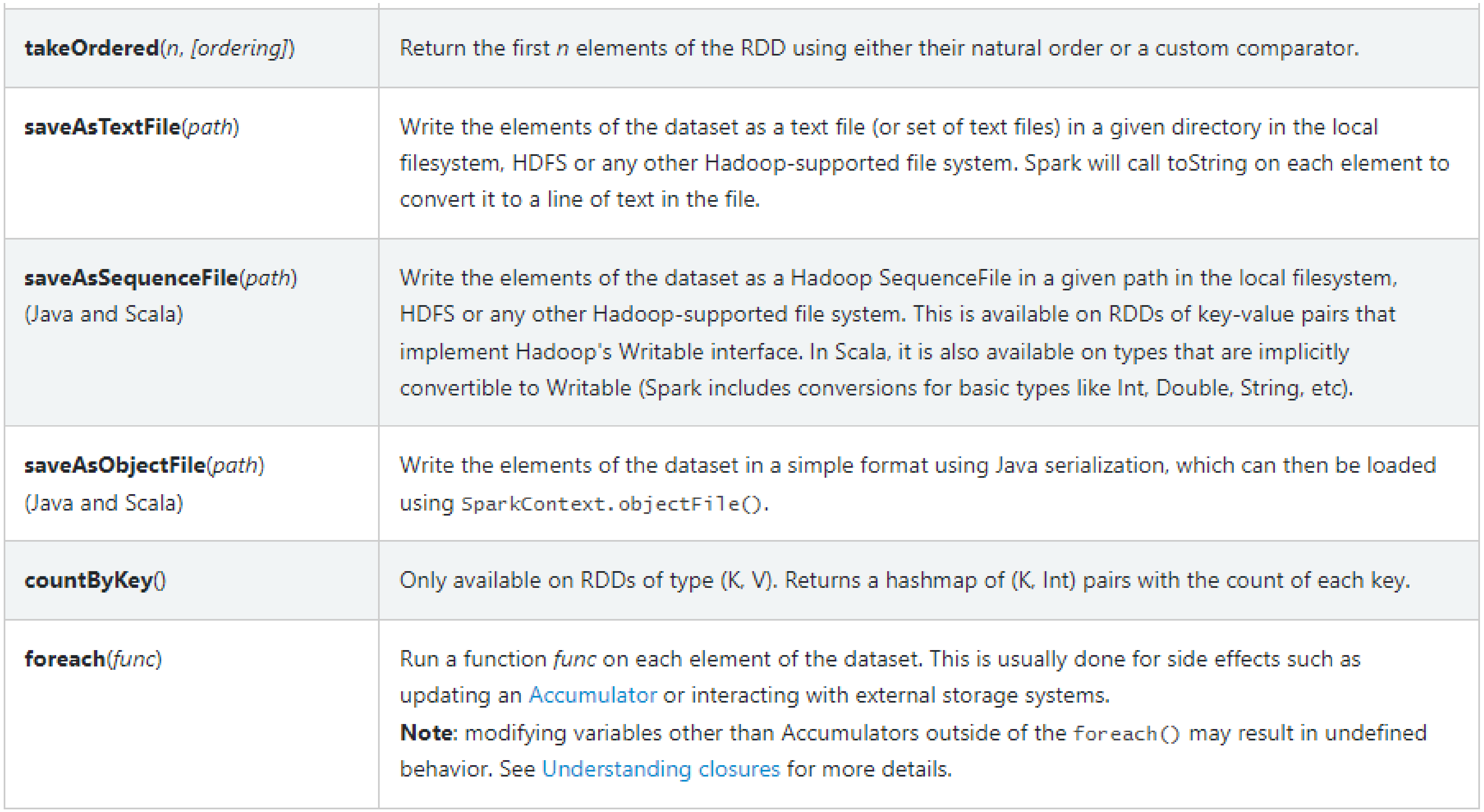
| | |
|---|---|
| **cartesian**(*otherDataset*) | When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements). |
| **pipe**(*command, [envVars]*) | Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings. |
| **coalesce**(*numPartitions*) | Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset. |
| **repartition**(*numPartitions*) | Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network. |
| **repartitionAndSortWithinPartitions**(*partitioner*) | Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling `repartition` and then sorting within each partition because it can push the sorting down into the shuffle machinery. |

## “Actions”



| **Action** | **Meaning** |
|---|---|
| **reduce**(*func*) | Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel. |
| **collect**() | Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data. |
| **count**() | Return the number of elements in the dataset. |
| **first**() | Return the first element of the dataset (similar to take(1)). |
| **take**(*n*) | Return an array with the first *n* elements of the dataset. |
| **takeSample**(*withReplacement*, *num*, [*seed*]) | Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed. |

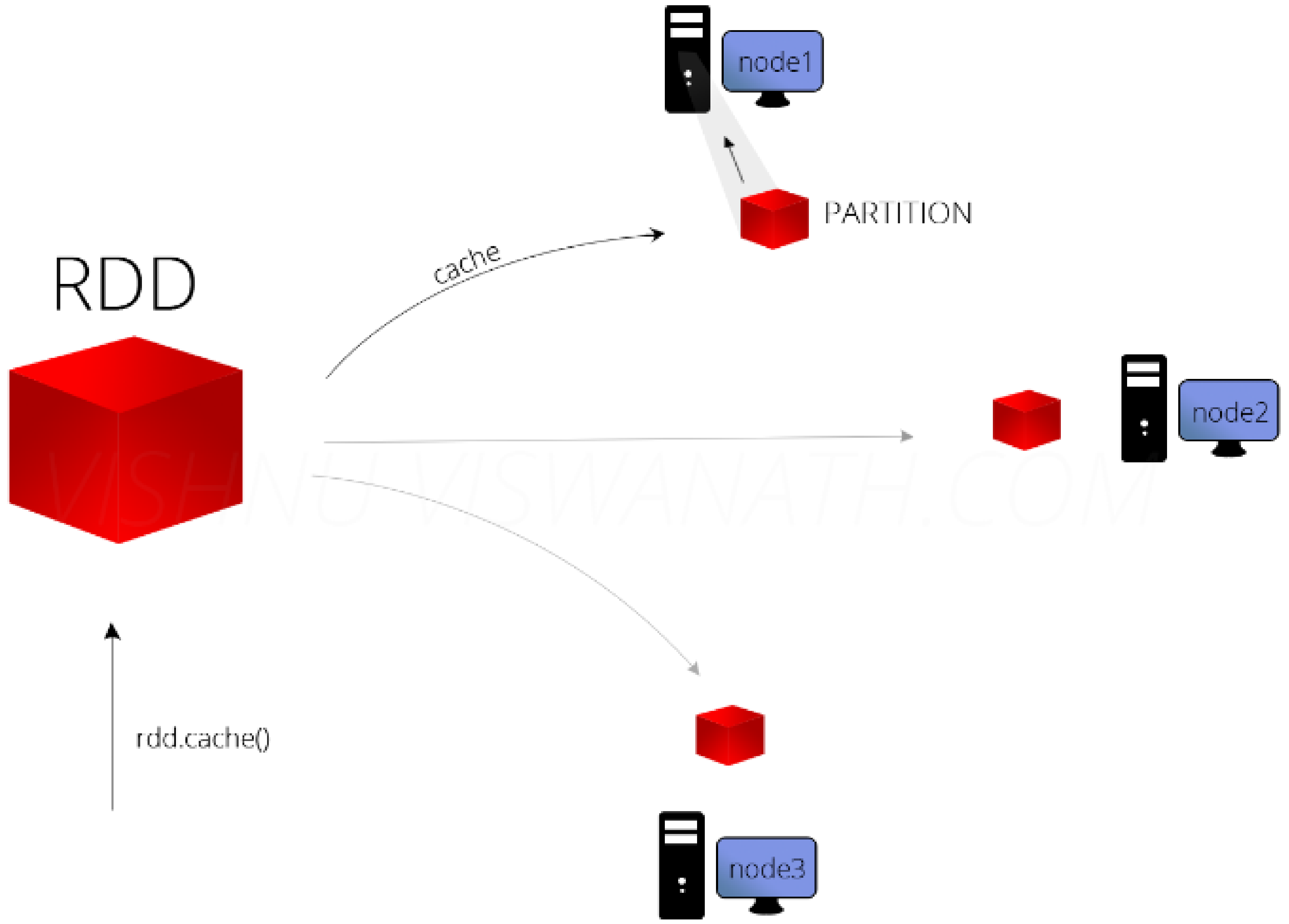
# “Actions”



| | |
|---|---|
| **takeOrdered**(*n*, *[ordering]*) | Return the first *n* elements of the RDD using either their natural order or a custom comparator. |
| **saveAsTextFile**(*path*) | Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file. |
| **saveAsSequenceFile**(*path*)<br>(Java and Scala) | Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc). |
| **saveAsObjectFile**(*path*)<br>(Java and Scala) | Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`. |
| **countByKey**() | Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key. |
| **foreach**(*func*) | Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems.<br>**Note**: modifying variables other than Accumulators outside of the `foreach()` may result in undefined behavior. See Understanding closures for more details. |

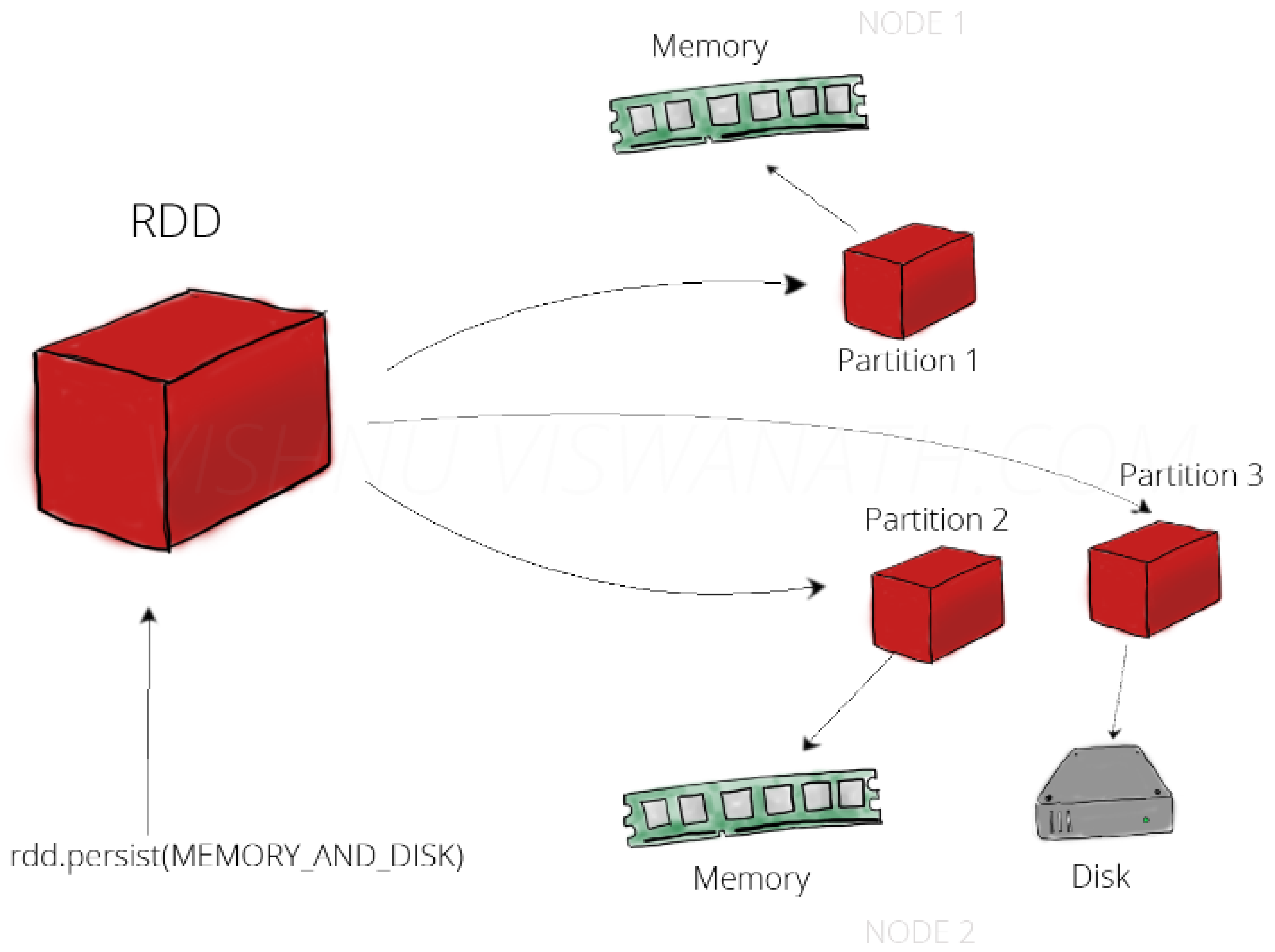
## “Caching”

If you find yourself performing some queries or transformations on a dataset repeatedly, we should consider caching the dataset which can be done by calling the cache method on the dataset.

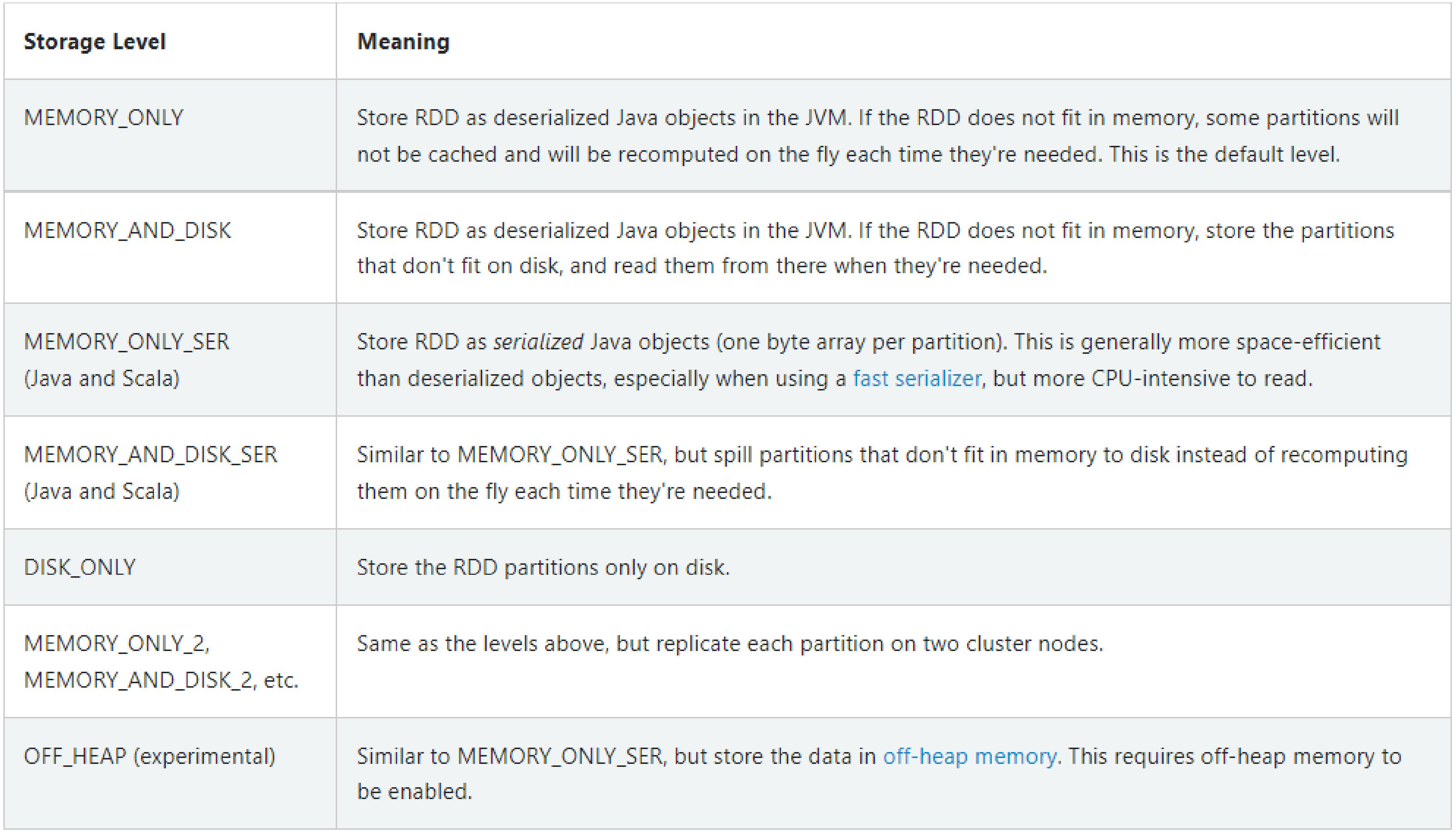

RDD
cache
PARTITION
node1
node2
node3
rdd.cache()

## “Persist”

Sometimes we would like to call actions on the same RDD multiple times. If we do this naively, RDDs and all of its dependencies are recomputed are recomputed each time an action is called on the RDD. This can be very expensive, especially for some iterative algorithms, which would call actions on the same dataset many times.


NODE 1
Memory
RDD
Partition 1
Partition 3
Partition 2
rdd.persist(MEMORY_AND_DISK)
Memory
Disk
NODE 2

# “Storage Level”



| **Storage Level** | **Meaning** |
|---|---|
| MEMORY_ONLY | Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level. |
| MEMORY_AND_DISK | Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed. |
| MEMORY_ONLY_SER (Java and Scala) | Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read. |
| MEMORY_AND_DISK_SER (Java and Scala) | Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed. |
| DISK_ONLY | Store the RDD partitions only on disk. |
| MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc. | Same as the levels above, but replicate each partition on two cluster nodes. |
| OFF_HEAP (experimental) | Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled. |