Introduction to APACHE Spark™

Vahid Amiri
@vahidamiry
Vahidamiry.ir
Vahid.amiry@gmail.com

About Apache Spark

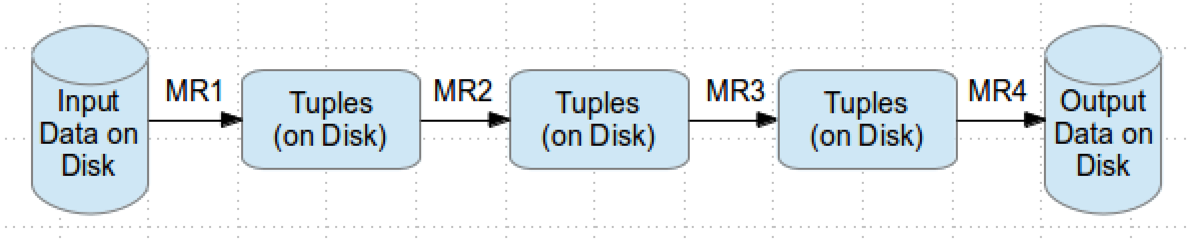
- Fast and general purpose cluster computing system
- 10x (on disk) - 100x (In-Memory) faster
- Most popular for running *Iterative Machine Learning Algorithms.*
- Provides high level APIs in
    - Java
    - Scala
    - Python
    - R
- http://spark.apache.org/

Why Spark ?

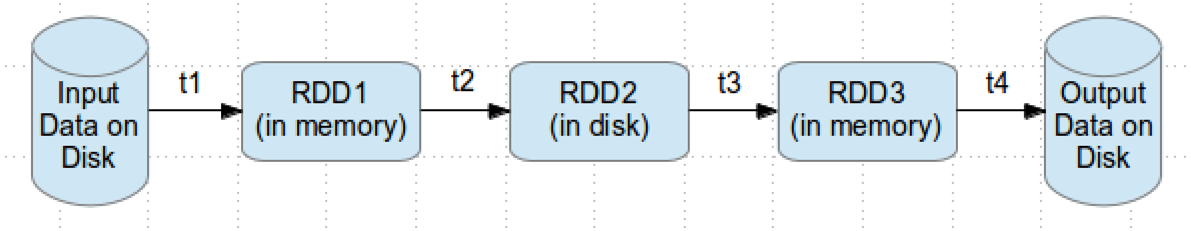
- Most of Machine Learning Algorithms are iterative because each iteration can improve the results

- With Disk based approach each iteration's output is written to disk making it slow

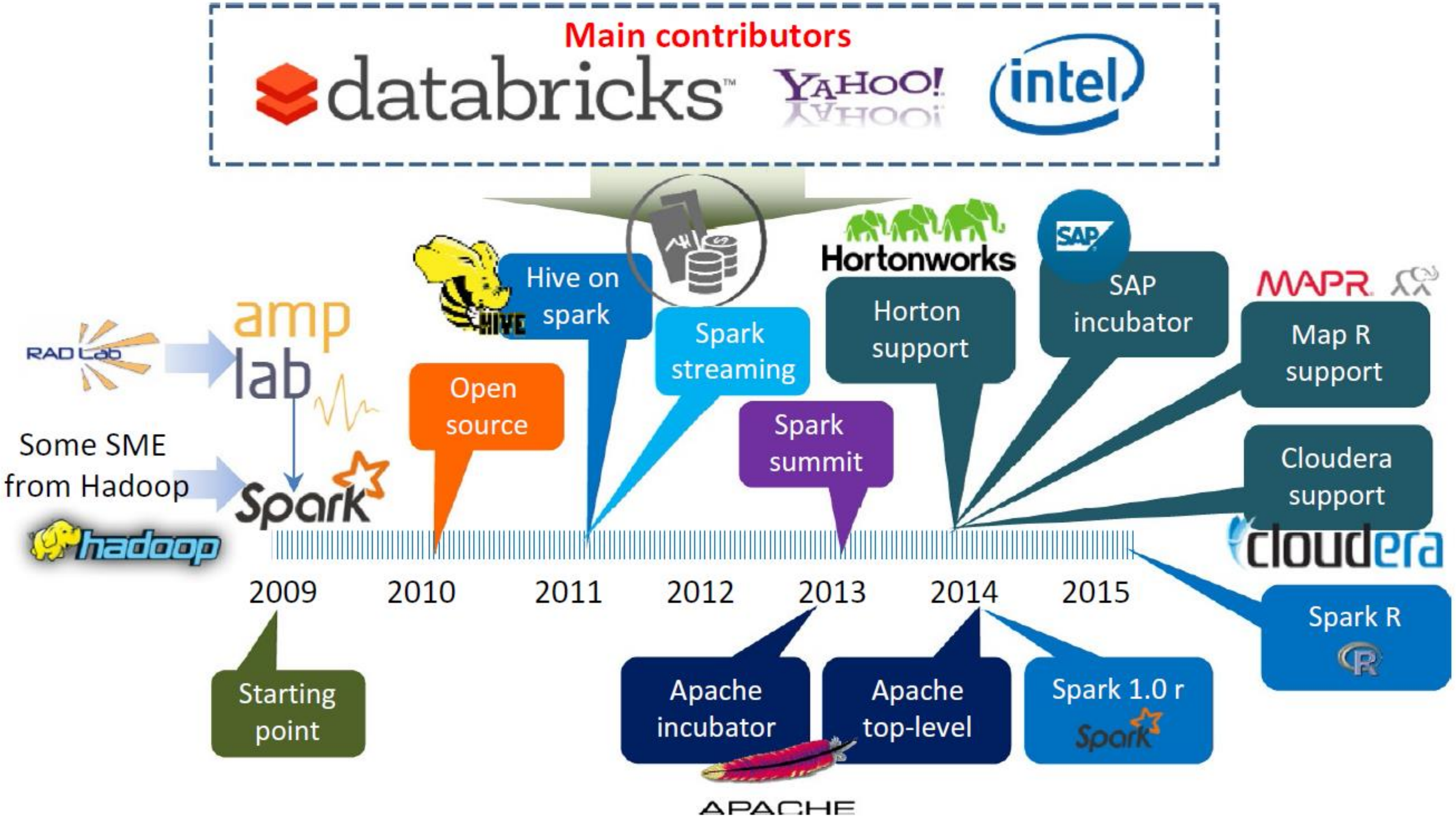
Mapreduce execution flow



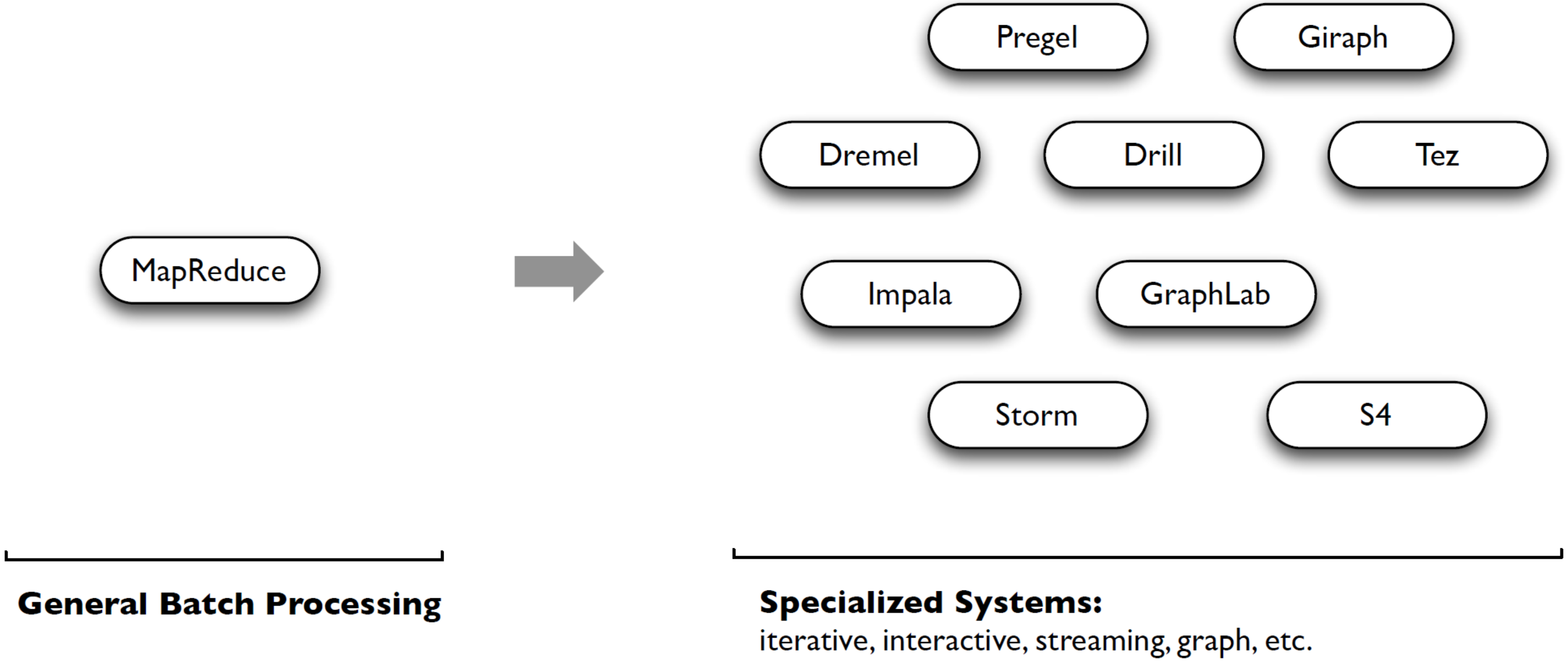
Spark execution flow



Spark history



A Brief History: *MapReduce*



MapReduce

General Batch Processing

Pregel

Giraph

Dremel

Drill

Tez

Impala

GraphLab

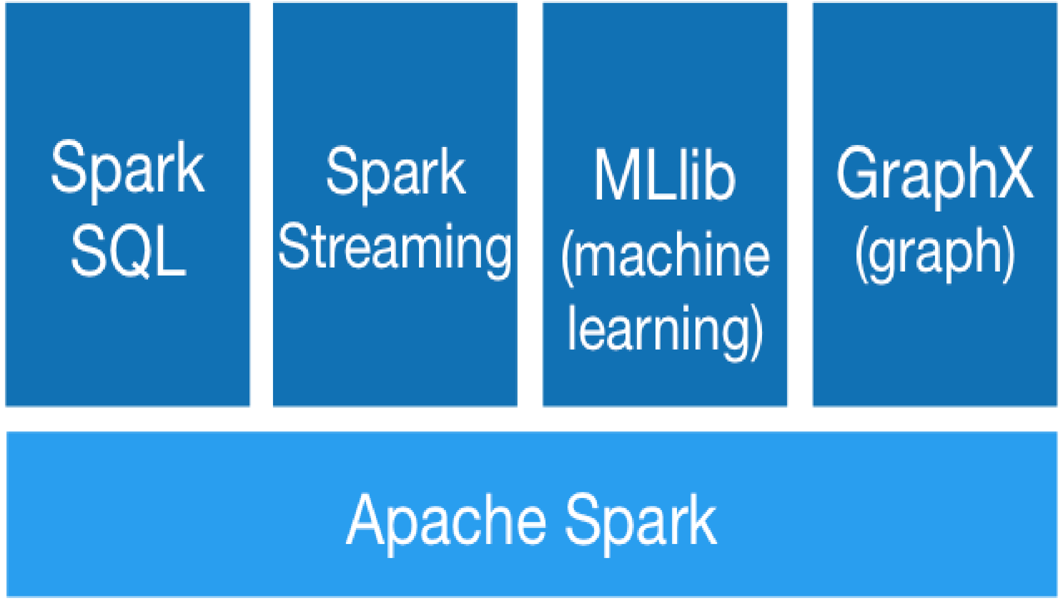
Storm

S4

Specialized Systems:
iterative, interactive, streaming, graph, etc.

Spark Stack

- Spark SQL
  - For SQL and unstructured data processing
- MLib
  - Machine Learning Algorithms
- GraphX
  - Graph Processing
- Spark Streaming
  - stream processing of live data streams



Languages

- APIs in Java, Scala, Python and R
- Interactive shells in Scala and Python

Which Language Should I Use?

- Standalone programs can be written in any, but console is only Python & Scala
- **Python developers:** can stay with Python for both
- **Java developers:** consider using Scala for console (to learn the API)

- Performance: Java / Scala will be faster (statically typed), but Python can do well for numerical work with NumPy

SparkContext

- First thing that a Spark program does is create a SparkContext object, which tells Spark how to access a cluster

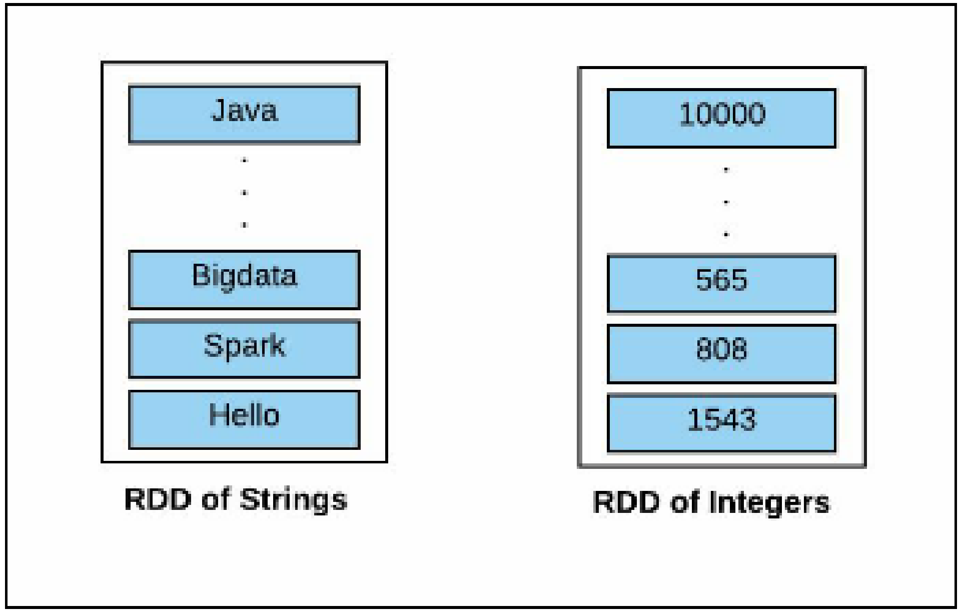
- In the shell for either Scala or Python, this is the sc variable, which is created automatically

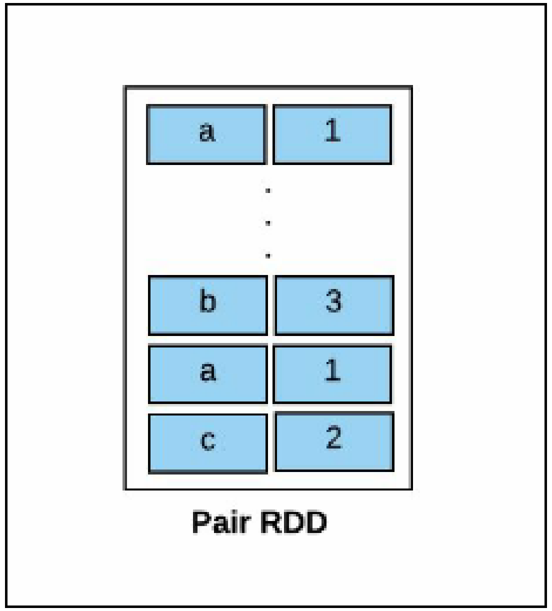
- Other programs must use a constructor to instantiate a new SparkContext

- Then in turn SparkContext gets used to create other variables

RDD

- **Resilient Distributed Datasets** (**RDDs**), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.





RDD — Resilient Distributed Dataset

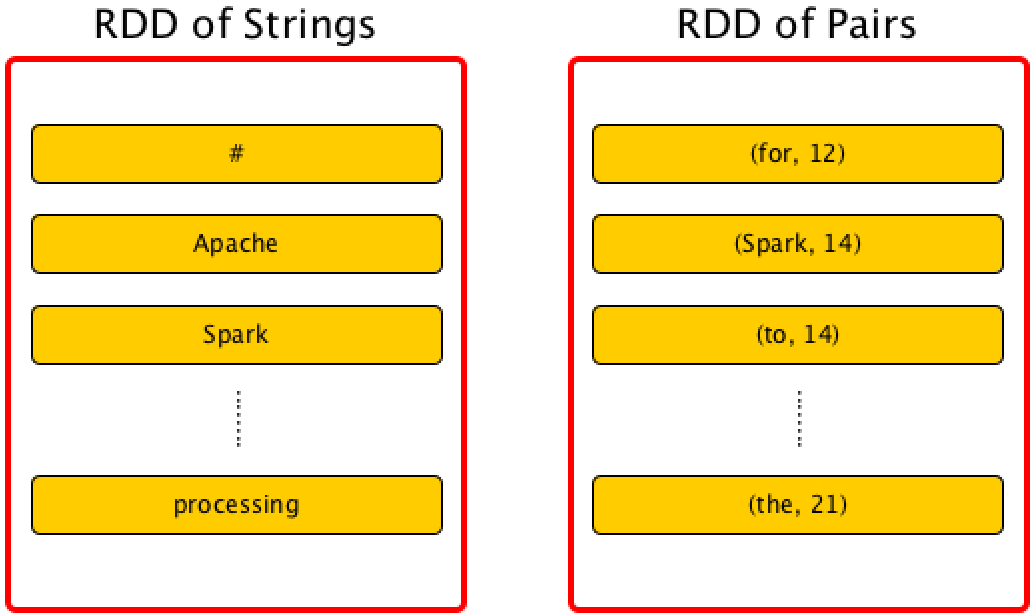
- Resilient Distributed Dataset (aka RDD) is the primary data abstraction in Apache Spark and the core of Spark
- A RDD is a resilient and distributed collection of records spread over one or many partitions.

scaladoc of org.apache.spark.rdd.RDD:

- A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable, partitioned collection of elements that can be operated on in parallel.

original paper about RDD:

- Resilient Distributed Datasets (RDDs) are a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.



RDD — Resilient Distributed Dataset
The features of RDDs:

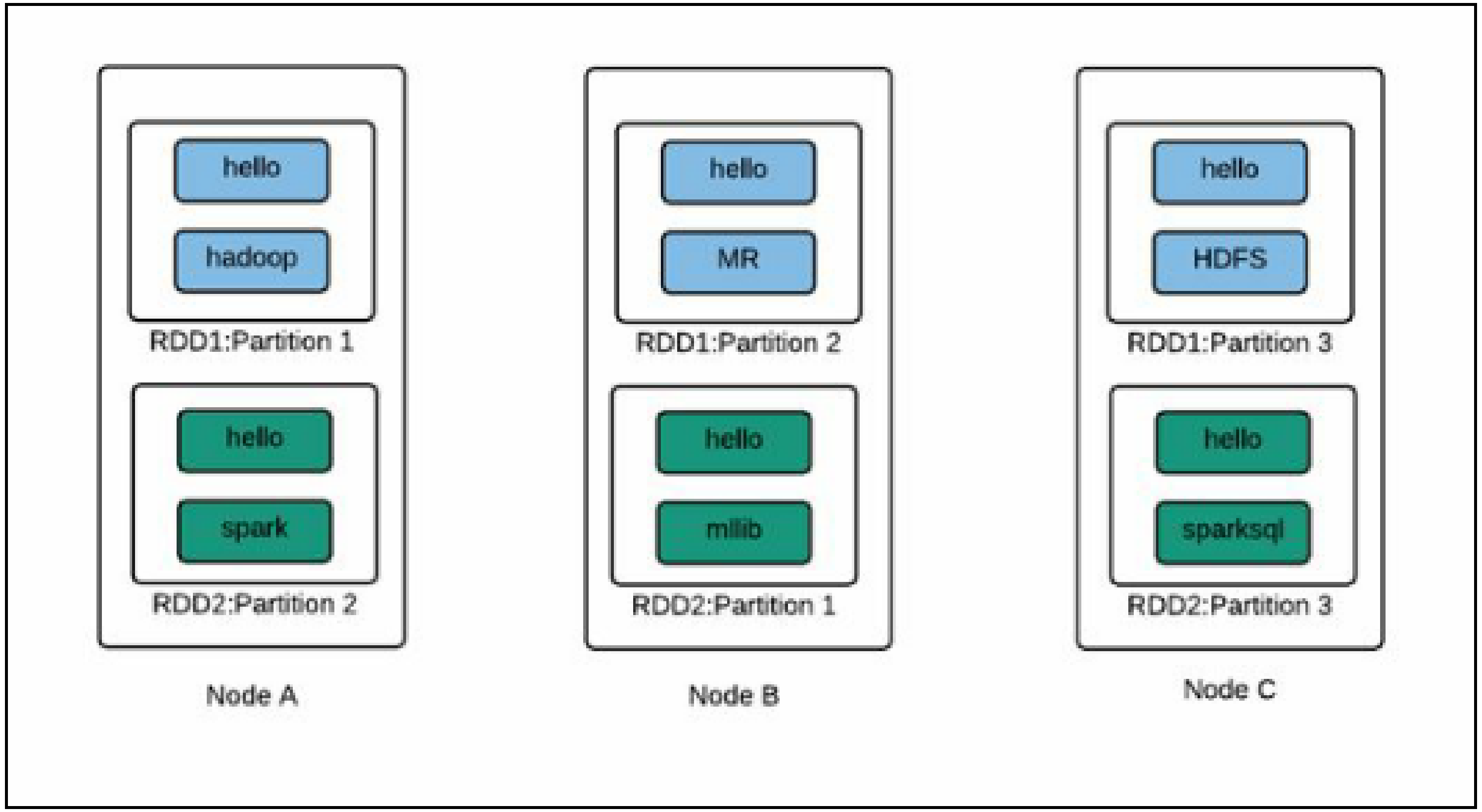
- **Resilient:** fault-tolerant with the help of RDD lineage graph and so able to recompute missing or damaged partitions due to node failures.
- **Distributed** with data residing on multiple nodes in a cluster.
- **Dataset** is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects (that represent records of the data you work with).

additional traits:

- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
- **Immutable or Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
- **Parallel**, i.e. process data in parallel.
- **Typed** — RDD records have types, e.g. Long in RDD[Long] or (Int, String) in RDD[(Int, String)]
- **Partitioned** — records are partitioned (split into logical partitions) and distributed across nodes in a cluster.
- **Location-Stickiness** — RDD can define placement preferences to compute partitions (as close to the records as possible).

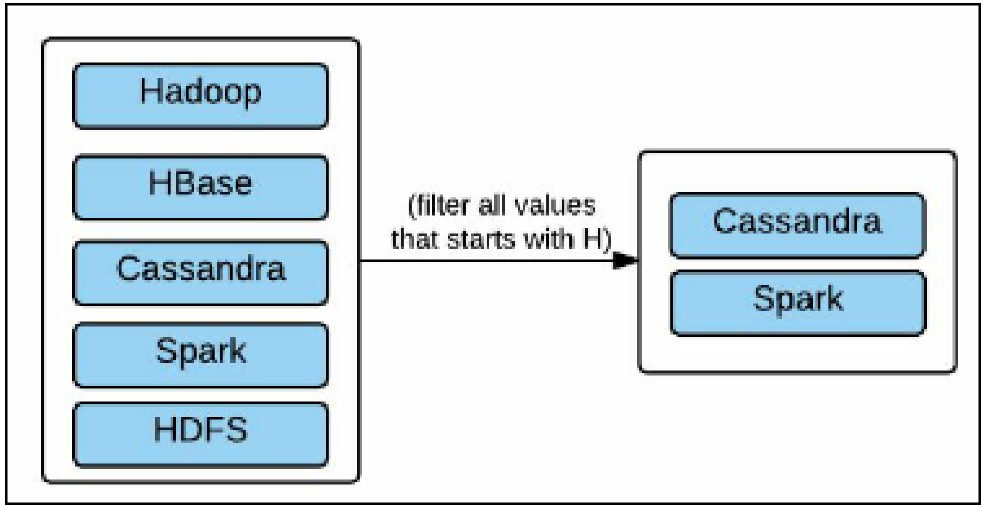
RDD

- RDDs have two types of operations:
  - Transformations
  - Actions



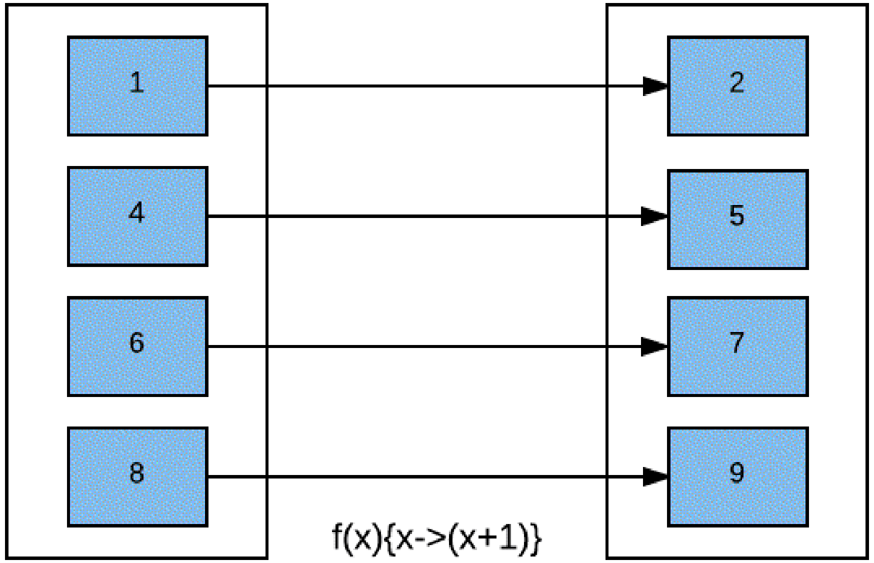
Transformation

- **Transformation:** If an operation on an RDD gives you another RDD, then it is a transformation.



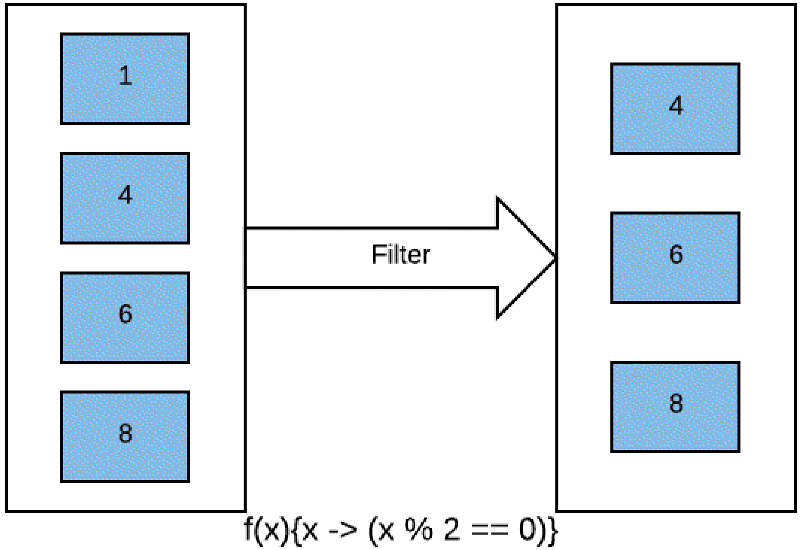
Map

- In a map transformation there is one-to-one mapping between elements of the source RDD and the target RDD



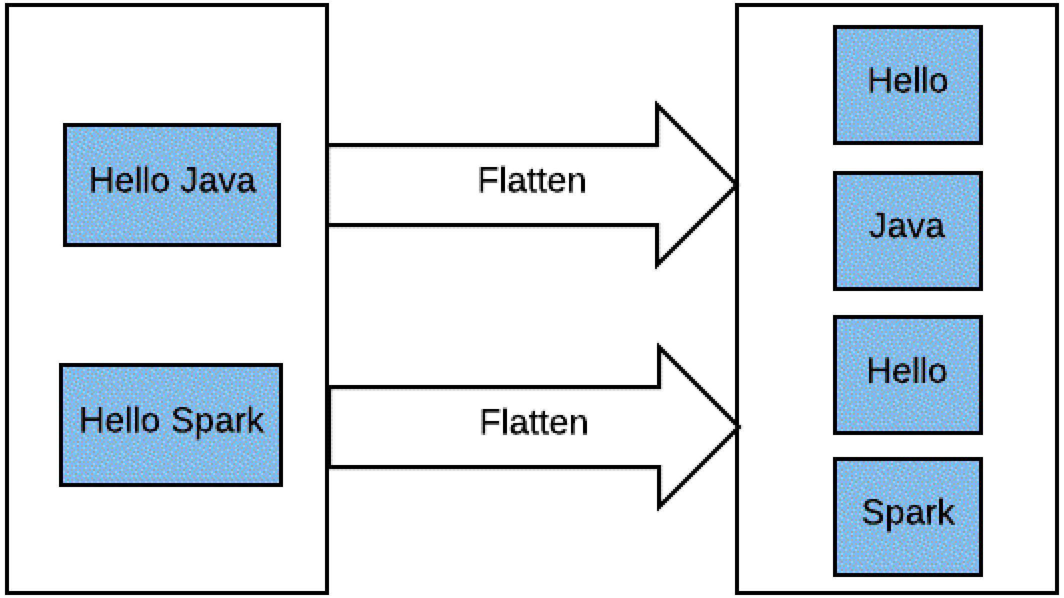
Filter

- With a filter transformation, the function is executed on all the elements of the source RDD and only those elements, for which the function returns true, are selected to form the target RDD.



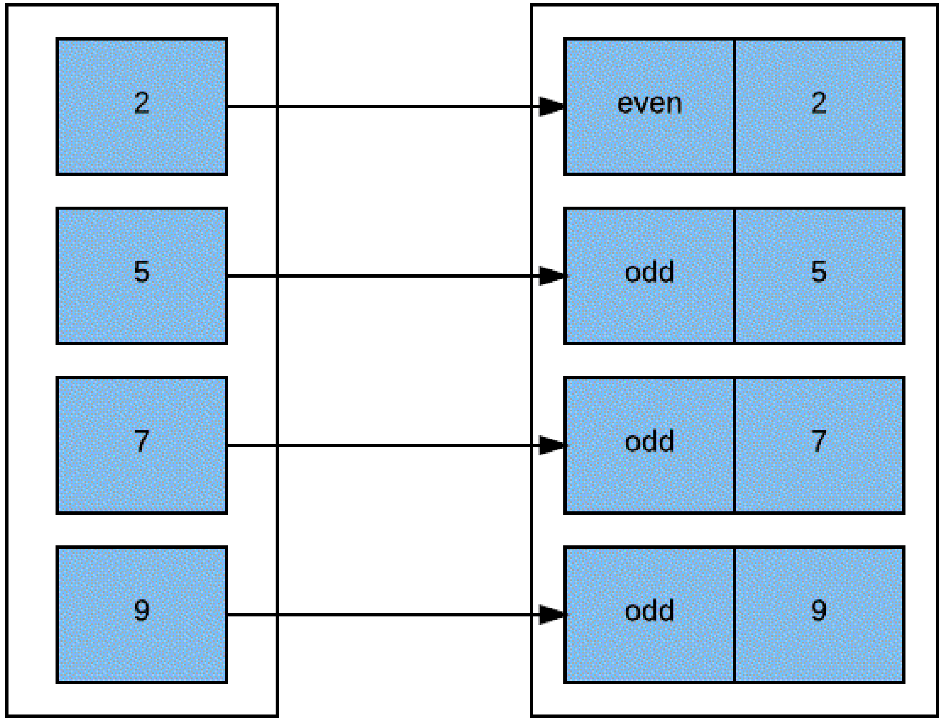
FlatMap

- In flatMap() transformation, an element of source RDD can be mapped to one or more elements of target RDD.



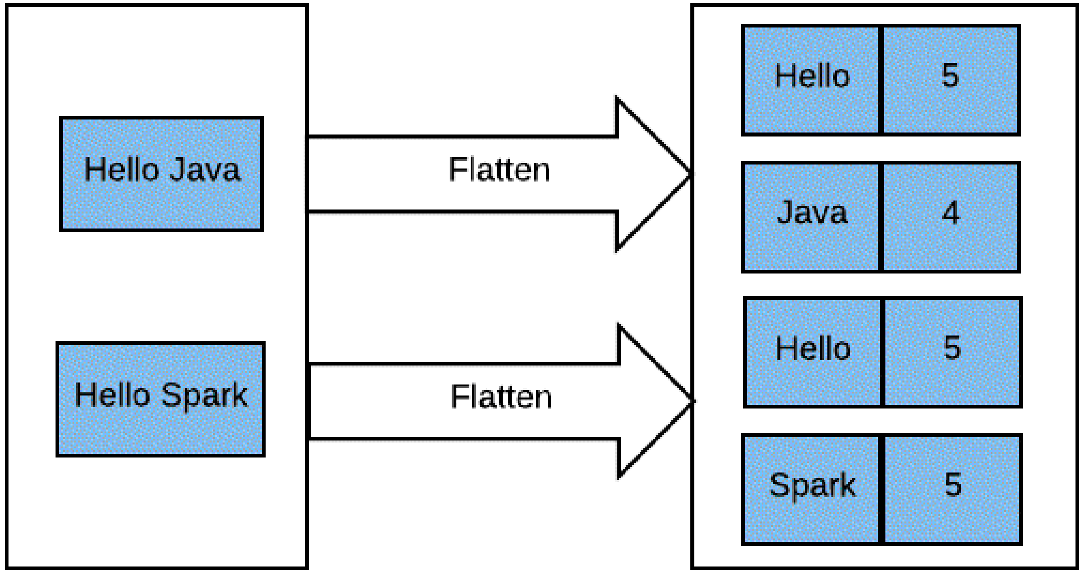
MapToPair

- It is similar to map transformation; however, this transformation produces PairRDD, that is, an RDD consisting of key and value pairs.



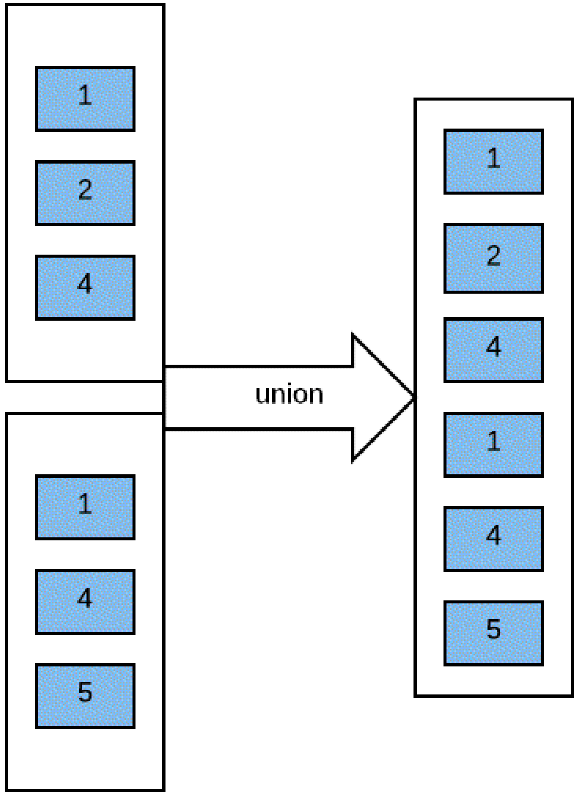
FlatMapToPair

- It is similar to flatMap(), however, the target RDD is a type of PairRDD, that is, an RDD consisting of key and value Pairs. Like mapToPair(), this operation is also specific to JavaRDDs.



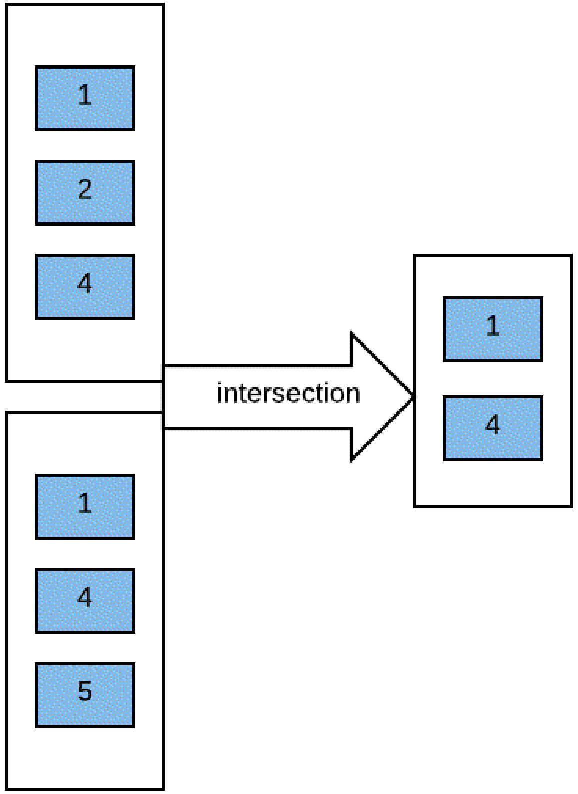
Union

- As the name suggests, this transformation unites two RDDs into one. The target consists of all the elements of source RDDs.



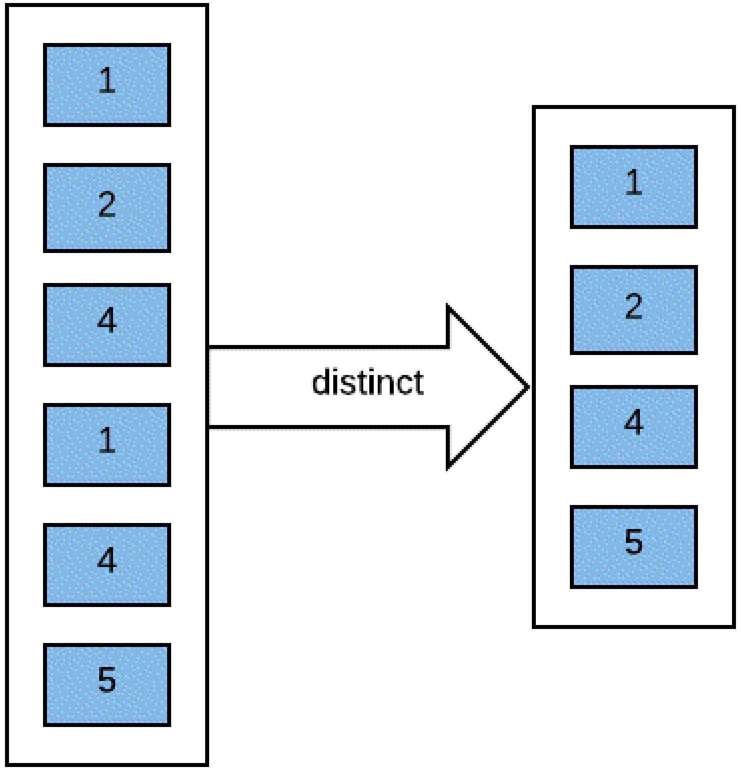
Intersection

- It produces an intersection of the elements of source RDDs, that is, the target RDD consists of the elements that are identical in source RDDs.



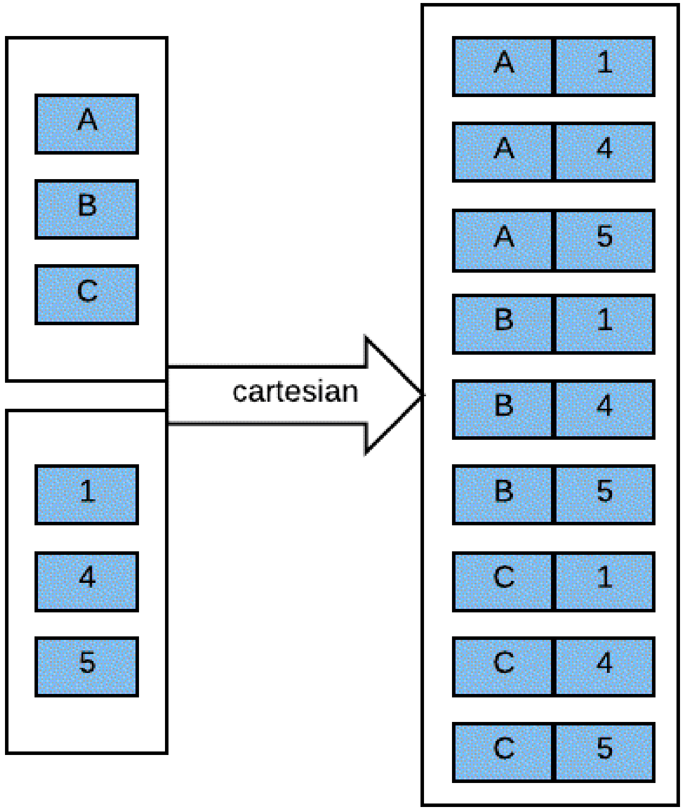
Distinct

- the distinct operation will return an RDD by fetching all the distinct elements of a source RDD



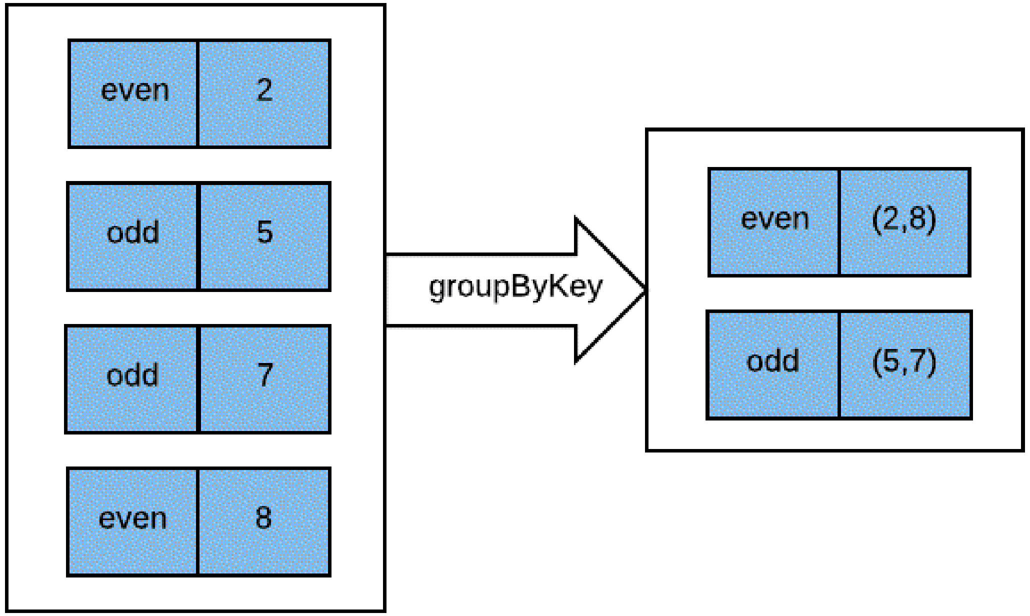
Cartesian

- Cartesian transformation generates a cartesian product of two RDDs



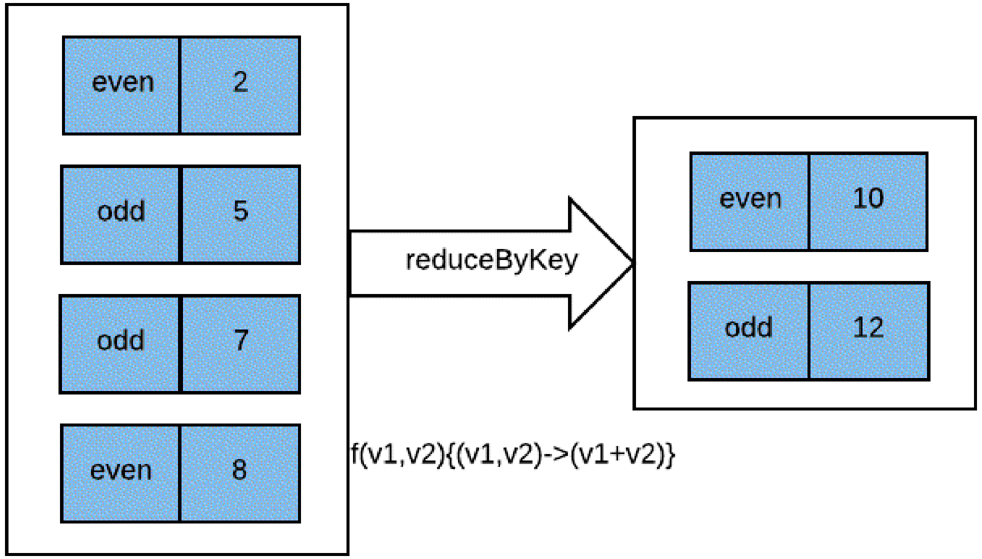
GroupByKey

- It is used to group all the values that are related to keys. It helps to transform a PairRDD consists of <key,value> pairs to PairRDD of <key,Iterable<value>>) pairs



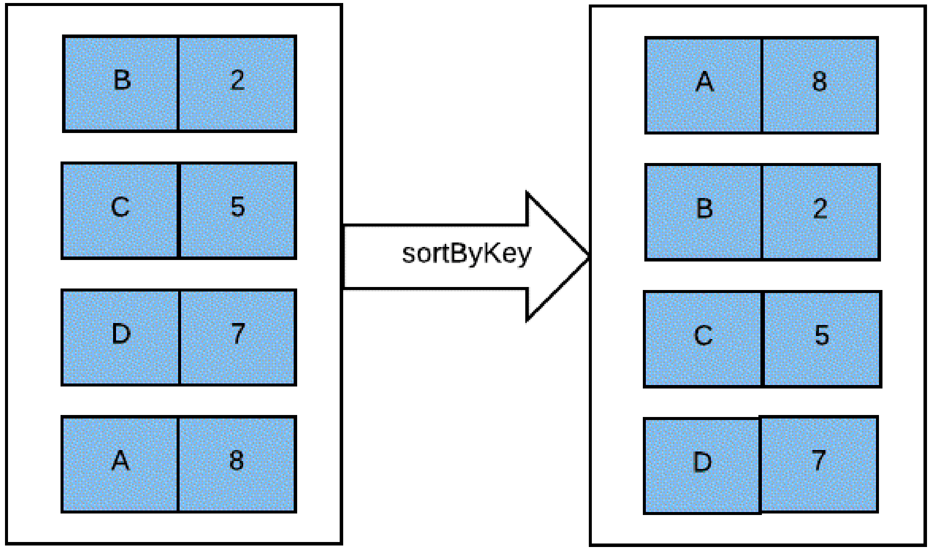
ReduceByKey

- A function is passed as an argument to this transformation, which helps to aggregate the values corresponding to a key
- For an RDD with multiple partitions, the reduceByKey transformation aggregates the values at partition level before shuffling, which helps in reducing the data travel of network. It is similar to the concept of combiner in Hadoop



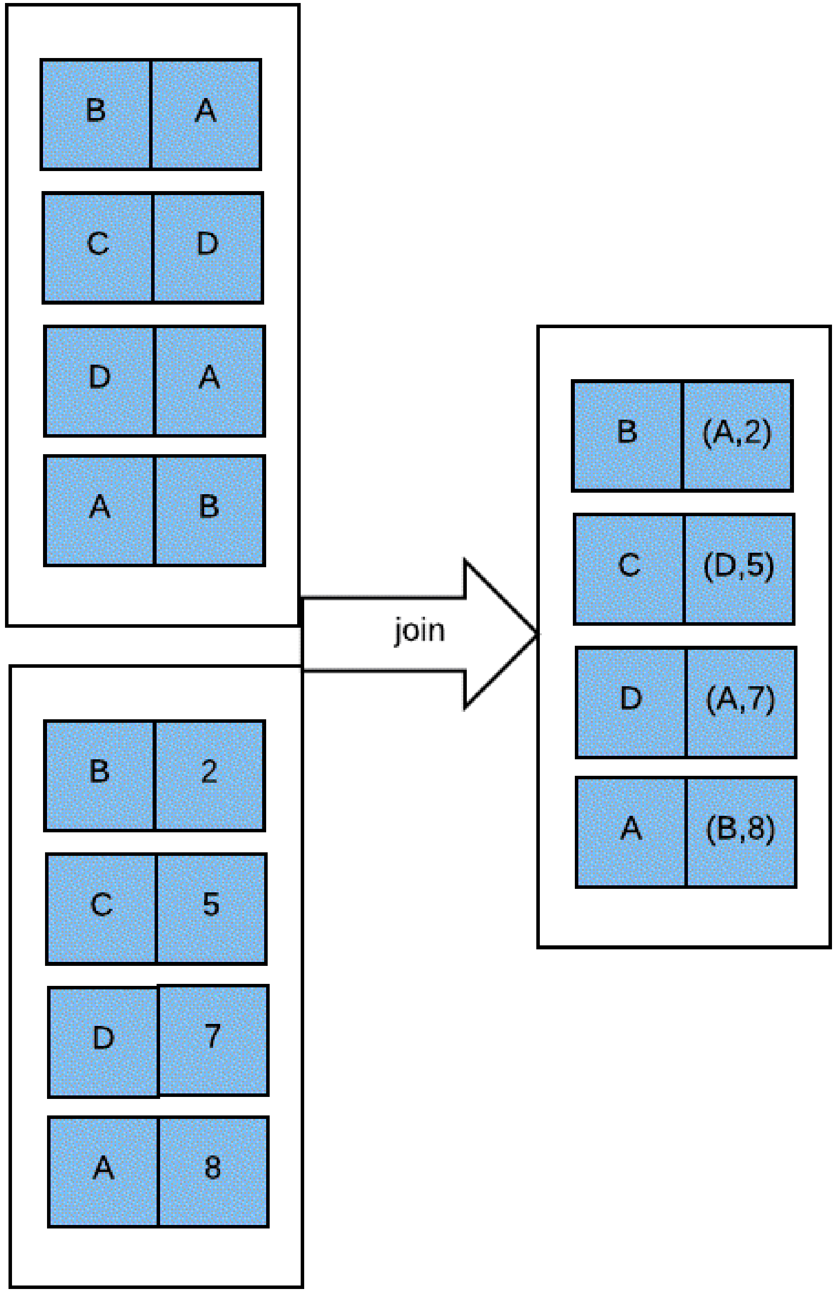
SortByKey

- It is used sort to a pair RDD by its key in ascending or descending order
- To execute sortByKey() on a PairRDD, Key should be of ordered type. In the Java world, key type should have implemented comparable, or else it throws an exception



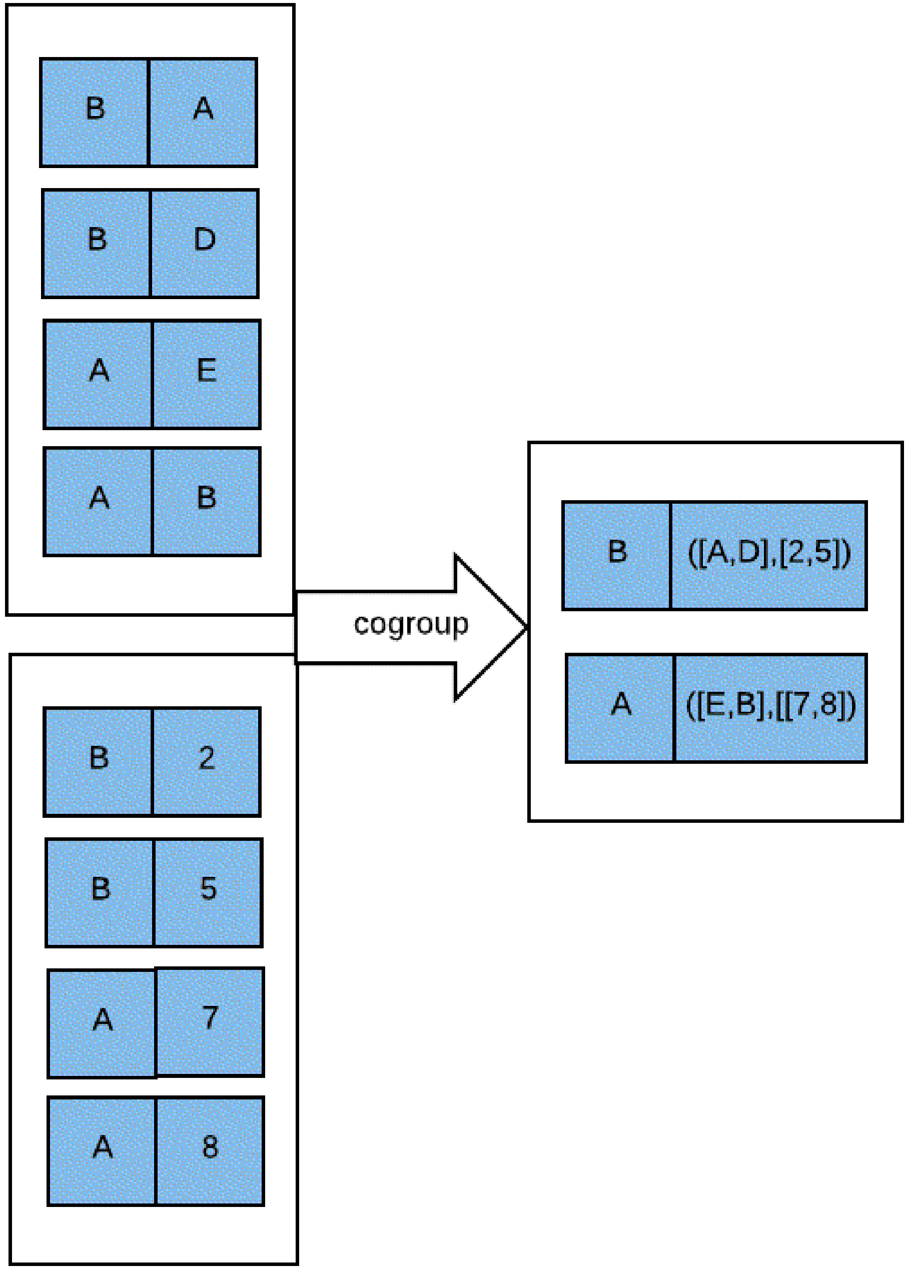
Join

- It is used to join pair RDDs. This transformation is very similar to Join operations in a database.
- we have two pair RDDs of <X,Y> and <X,Z> types . When the Join transformation is executed on these RDDs, it will return an RDD of <X,(Y,Z)> type.
- LeftOuterJoin, RightOuterJoin and FullOuterJoin are also supported which can be executed



CoGroup

- This operation also groups two PairRDD.
- we have two PairRDD of <X,Y> and <X,Z> types . When CoGroup transformation is executed on these RDDs, it will return an RDD of <X,(Iterable<Y>,Iterable<Z>)> type. This operation is also called groupwith.



Transformations

- There are two kinds of transformations:
  - narrow transformations
  - wide transformations

Narrow Transformations

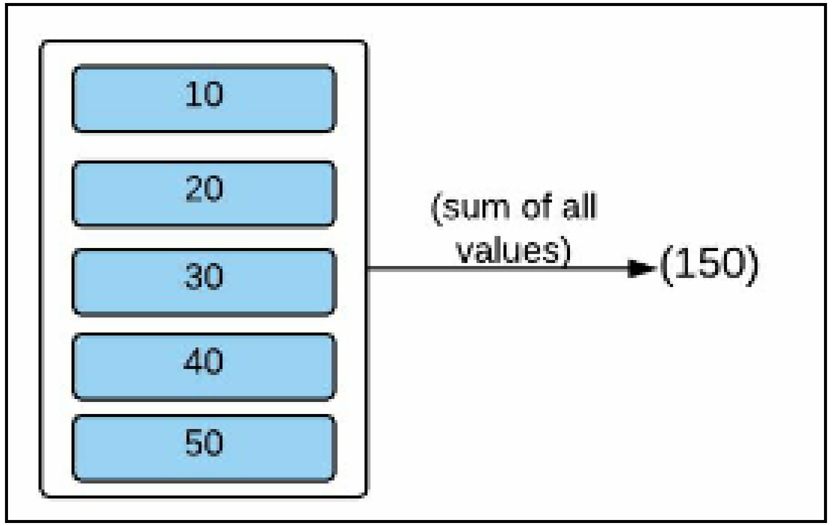
- Narrow transformations are the result of map, filter and such that is from the data from a single partition only, i.e. it is self-sustained.
- An output RDD has partitions with records that originate from a single partition in the parent RDD. Only a limited subset of partitions used to calculate the result.
- Spark groups narrow transformations as a stage which is called pipelining.

Wide Transformations

- Wide transformations are the result of groupByKey and reduceByKey. The data required to compute the records in a single partition may reside in many partitions of the parent RDD.
- Wide transformations are also called shuffle transformations as they may or may not depend on a shuffle.
- All of the tuples with the same key must end up in the same partition, processed by the same task. To satisfy these operations, Spark must execute RDD shuffle, which transfers data across cluster and results in a new stage with a new set of partitions

Action

- If an operation on an RDD gives you a result other than an RDD, it is called an action



RDD actions

- **isEmpty**
  - isEmpty returns a Boolean value as True, if the RDD contains no element at all.isEmpty returns a Boolean value as True, if the RDD contains no element at all.
- **collect**
  - Collect retrieves the data from different partitions of RDD into an array at the driver program!
- **collectAsMap**
  - The collectAsMap action is called on PairRDD to collect the dataset in key/value format at the driver program
- **Count**
  - The count method returns the number of element that exist in an RDD across partitions

RDD actions

- **countByKey**
  - countByKey is an extension to what the action count() does, it works on pair RDD to calculate the number of occurrences of keys in a pair RDD.
- **countByValue**
  - when countByValue() is applied on RDD , it counts the occurrence of elements having the specific value in that RDD and returns a Map having RDD elements as key and count of occurrence as value
  - When countByValue() is applied on Pair RDD it returns the count of each unique value in the RDD as a map of (value, count) pairs where value is the Tuple for RDD
- **Max**
  - The action max returns the largest value in the RDD where the comparison between the elements is decided by the comparator passed as an argument

RDD actions

- **Min**
  - The method min() computes the smallest value in the RDD.
- **First**
  - The first method returns the first element of the RDD.
- **Take**
  - The action take returns a list on n elements where n is the argument to be passed to the method take()
- **takeOrdered**
  - Elements of the RDD can be returned based on natural or custom ordering of the dataset using the method takeOrdered()

RDD actions

- **takeSample**
  - The takeSample method allows us to fetch a random list of elements from an RDD to the Driver program.
- **top**
  - The RDD action top returns a list of n elements based on natural or custom ordering of the elements in the RDD
- **reduce**
  - The reduce method passes the elements of the RDD to a function to be operated upon and then returns the computed output having the same data type as that of the input function.
- **Fold**
  - The fold() function aggregates the value of RDD elements by passing each element to the aggregate function and computing a resultant value of the same data type. However, the fold() function also requires a zero value that is utilized while initializing the call to the partitions for the RDD.

RDD actions

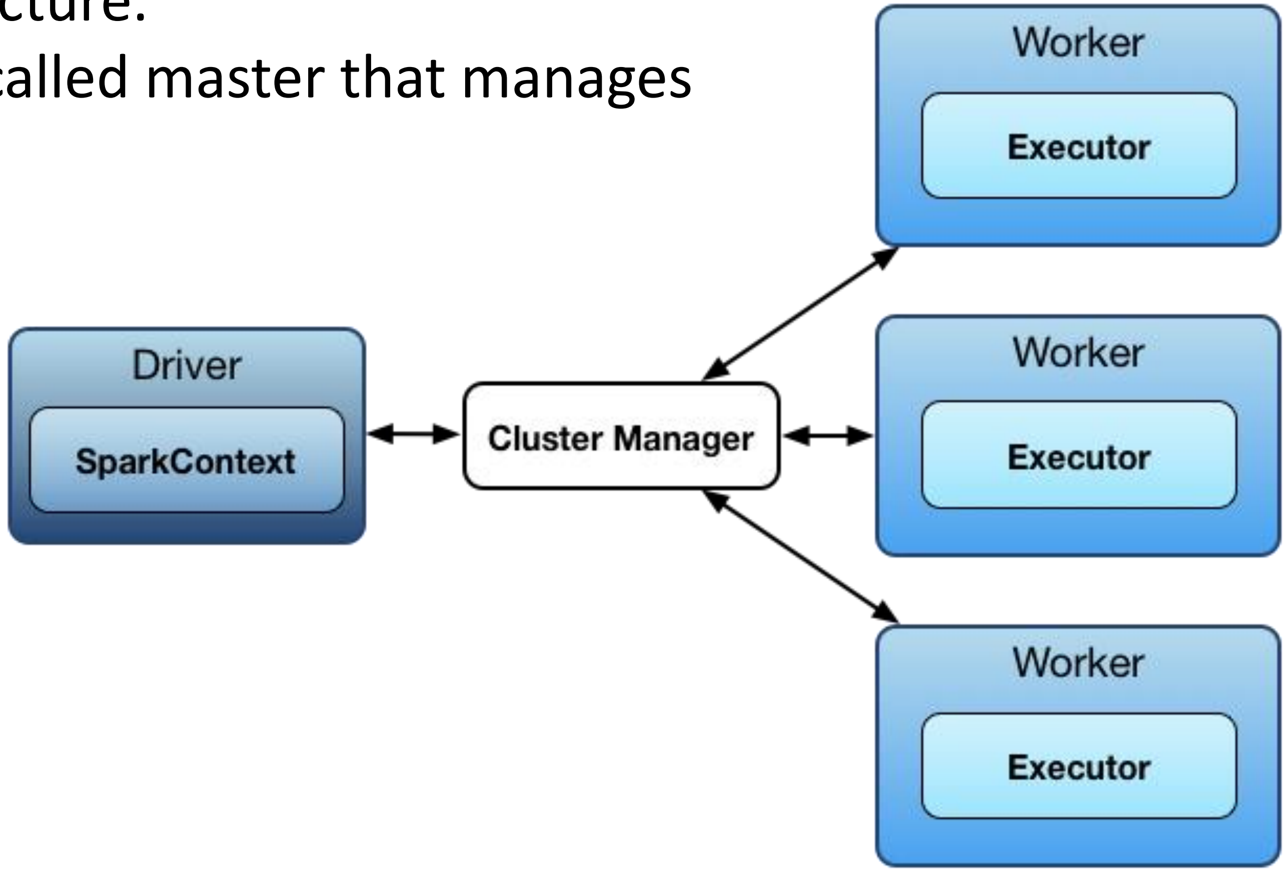
- **aggregate**
  - The distinctive features of the aggregate function is that the return type of the function need not be of the same type as are the elements of RDD being operated upon
- **forEach**
  - In order to iterate over all the elements of the RDD to perform some repetitive tasks without fetching the entire set of data to the Driver program can be achieved by using the foreach() function
- **saveAsTextFile**
  - The data of RDD can be saved to some local filesystem, HDFS, or any other Hadoop-supported filesystem using the action saveAsTextFile()

RDD actions

- **saveAsObjectFile**
  - In order to store and retrieve the data and metadata information about its data type saveAsObjectFile() can be used. It uses Java Serialization to store the data on filesystems and similarly the data can be read using the objectFile() method of SaprkContext

Spark Architecture

- Spark uses a master/worker architecture.
- driver talks to a single coordinator called master that manages workers in which executors run.



Driver program

- SparkContext is initialized in the Driver JVM. Spark driver can be considered as the master of Spark applications. The following are the responsibilities of Spark Driver program:
    - It creates the physical plan of execution of tasks based on the DAG of operations.
    - It schedules the tasks on the executors. It passes the task bundle to executors based. Data locality principle is used while passing the tasks to executors.
    - Spark driver tracks RDD partitions to executor mapping for executing future tasks.

Executor program

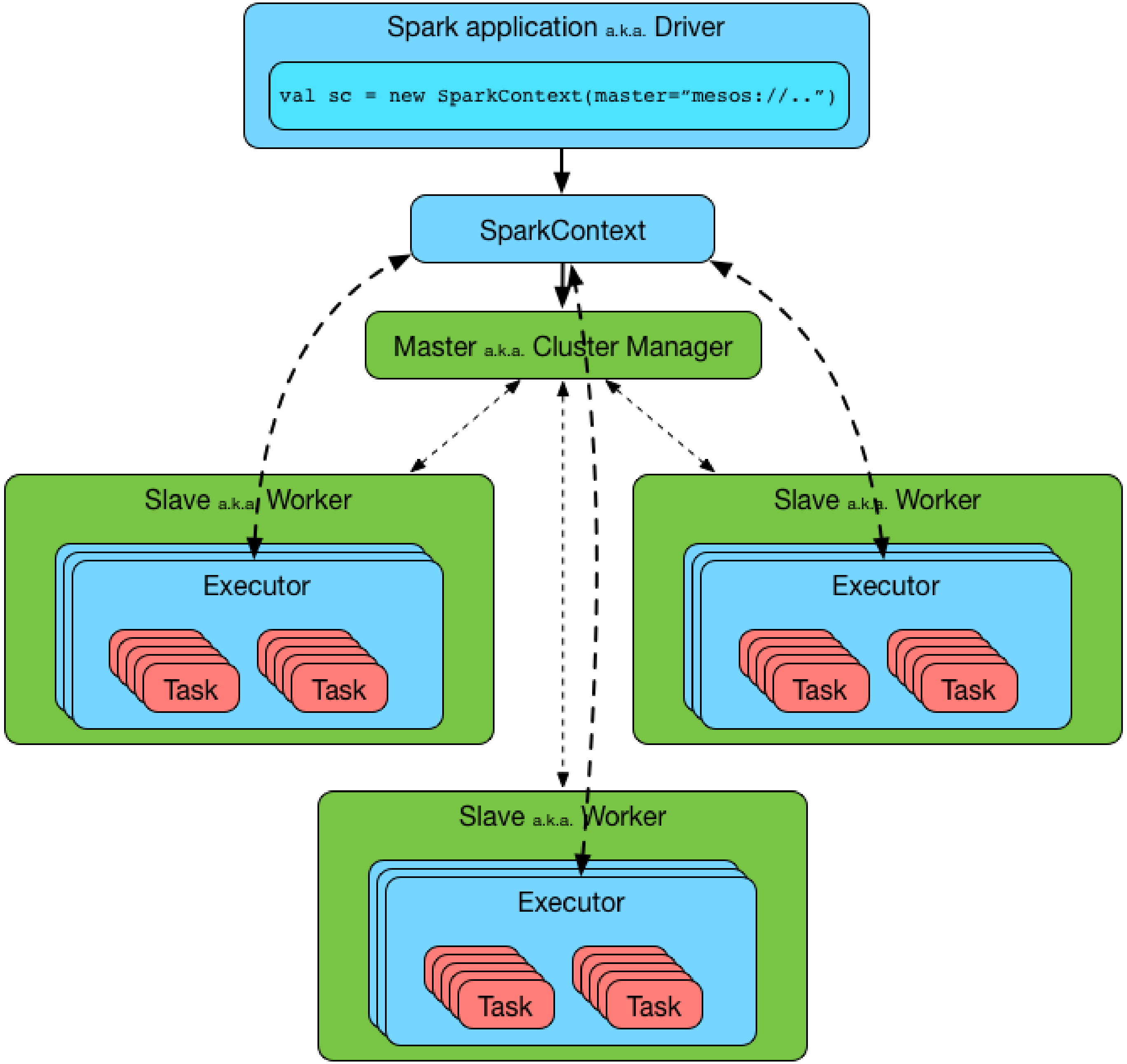
- Executors can be considered as slave JVMs for Driver processes. These are the JVM processes where actual application logic executes.
- Executors are initiated once at the beginning of the application execution and they run until the application finishes.
- Main responsibilities of executors:
    - Execute the tasks as directed by the Driver.
    - Cache RDD partitions in memory on which the tasks execute.
    - Report the tasks result to the driver.

Spark Architecture

- The driver and the executors run in their own Java processes

- run them all on the same (horizontal cluster) or separate machines (vertical cluster) or in a mixed machine configuration

Lines
Words
Pairwords
Wordcount
Collect



Driver

- A Spark driver is a JVM process that hosts SparkContext for a Spark application.
- master node in a Spark application
- It splits a Spark application into tasks and schedules them to run on executors.
- High-level control flow of work
- Your Spark application runs as long as the Spark driver.

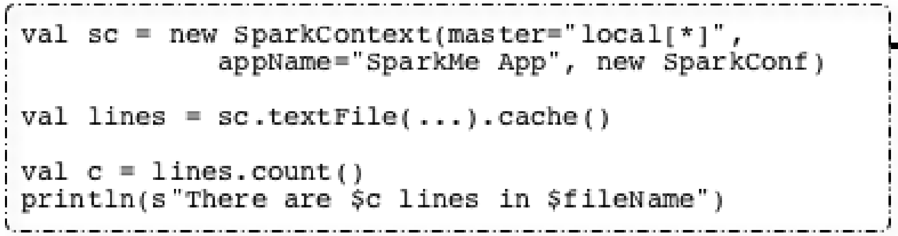
Settings

- spark.driver.memory
- spark.driver.cores
- spark.driver.port

…

SparkContext-Entry Point to Spark Core

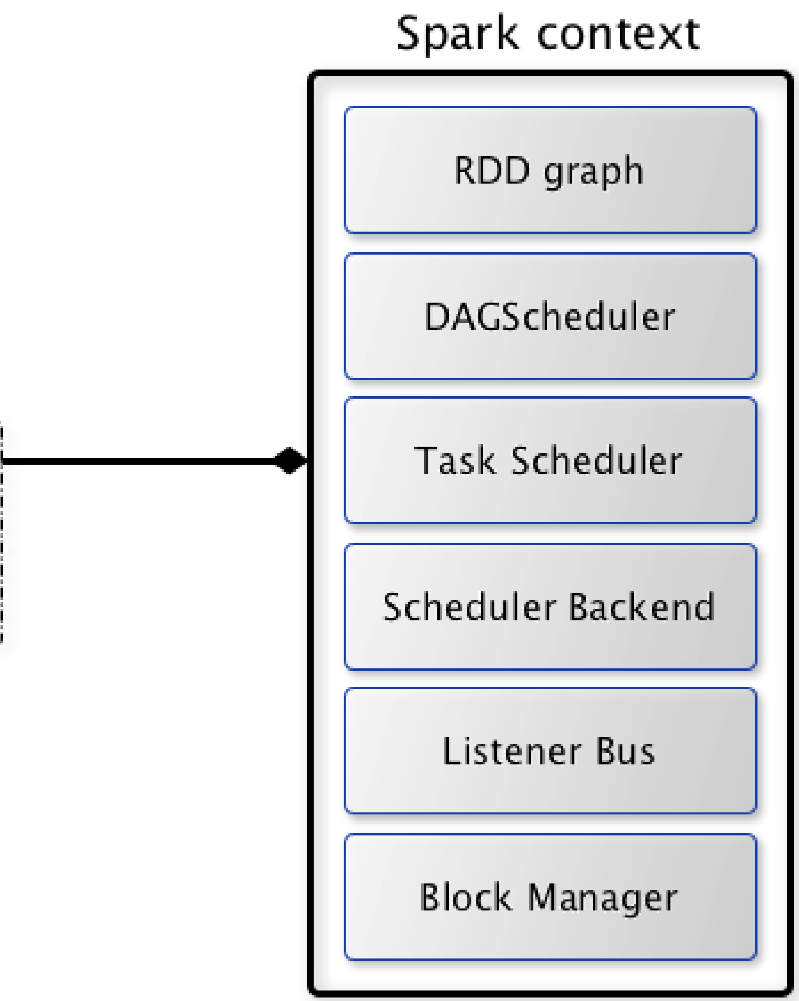
- Spark context is the heart of a Spark application
- Spark context sets up internal services and establishes a connection to a Spark execution environment.
- Once a SparkContext is created you can use it to create RDDs, accumulators and broadcast variables, access Spark services and run jobs (until SparkContext is stopped).



```
val sc = new SparkContext(master="local[*]",
               appName="SparkMe App", new SparkConf)

val lines = sc.textFile(...).cache()

val c = lines.count()
println(s"There are $c lines in $fileName")
```



SparkContext offers the following functions:

- Getting current status of a Spark application
- Setting Configuration
- Creating Distributed Entities
- Accessing services, e.g. TaskScheduler, LiveListenerBus, BlockManager, SchedulerBackends, ShuffleManager and the optional ContextCleaner.
- Running jobs synchronously
- Submitting jobs asynchronously
- Cancelling a job
- Cancelling a stage
- Assigning custom Scheduler Backend, TaskScheduler and DAGScheduler

Master URLs

Spark supports the following master URLs:

- local, local[N] and local[*] for Spark local
- local[N, maxRetries] for Spark local-with-retries
- local-cluster[N, cores, memory] for simulating a Spark cluster of N executors (threads), cores CPUs and memory locally (aka Spark local-cluster)
- spark://host:port,host1:port1,… for connecting to Spark Standalone cluster(s)
- mesos:// for Spark on Mesos cluster
- yarn for Spark on YARN

You can specify the master URL of a Spark application as follows:

- spark-submit's --master command-line option
- spark.master Spark property
- When creating a SparkContext (using setMaster method)
- When creating a SparkSession (using master method of the builder interface)

Deploy Mode

- Deploy mode specifies the location of where driver executes in the deployment environment.
- You can control the deploy mode of a Spark application using spark-submit's --deploy-mode command-line option or spark.submit.deployMode Spark property.

Deploy mode can be one of the following options:

- client (default) - the driver runs on the machine that the Spark application was launched.
- cluster - the driver runs on a random node in a cluster.

Deployment Environments — Run Modes

Spark Deployment Environments (aka Run Modes):

- Local
- Clustered
  - Spark Standalone
  - Spark on Hadoop YARN
  - Spark on Apache Mesos
- A Spark application is composed of the driver and executors that can run locally (on a single JVM) or using cluster resources (like CPU, RAM and disk that are managed by a cluster manager).

Spark standalone

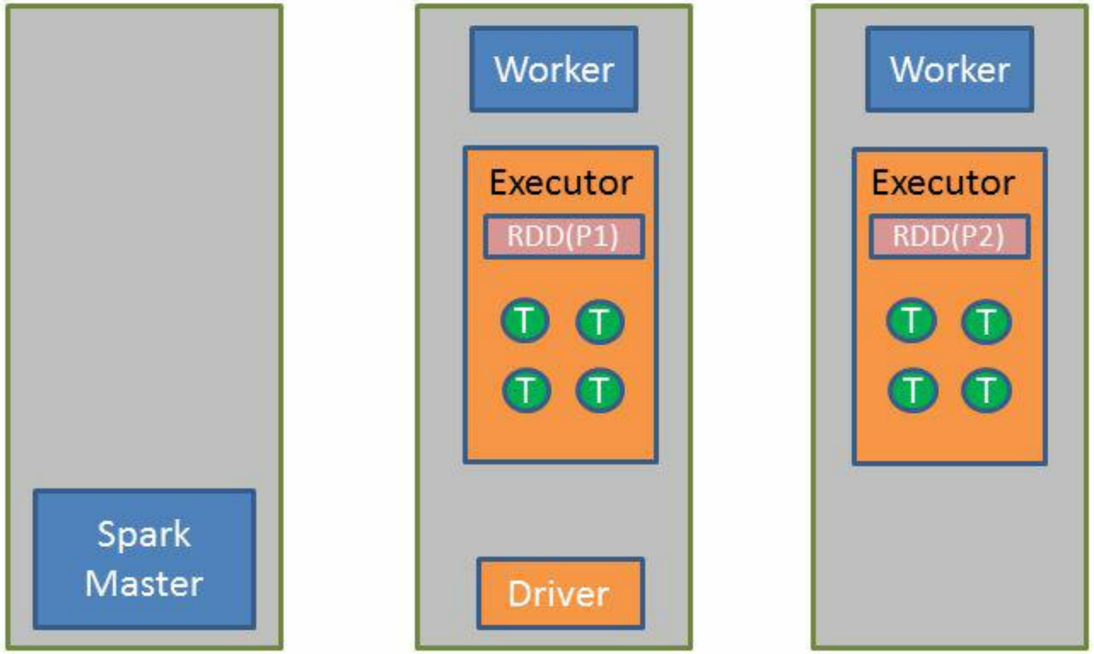
- master-slave architecture
- Spark master works a scheduler for the submitted Spark applications

- start Spark master:

```
$cd $SPARK_HOME
sbin/start-master.sh
```

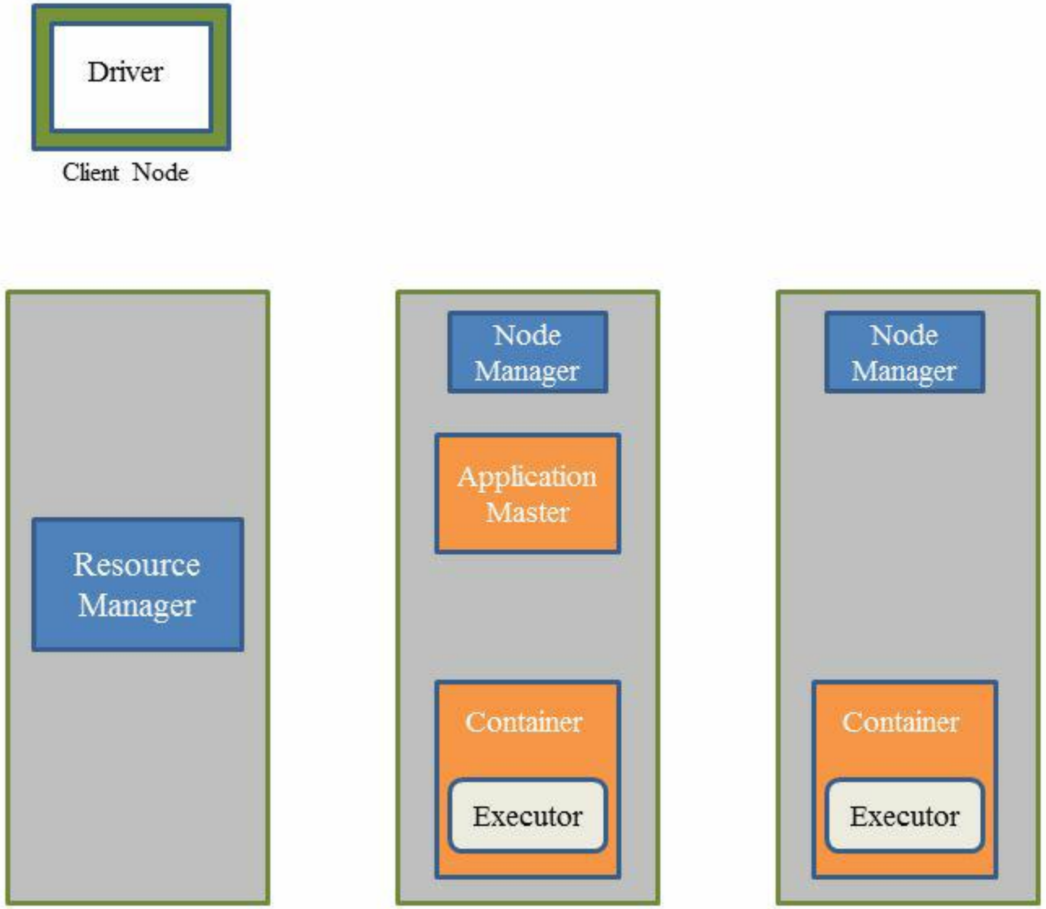
- start Spark slaves:

```
cd $SPARK_HOME
sbin/start-slave.sh spark://spark-master:7077
```



YARN client

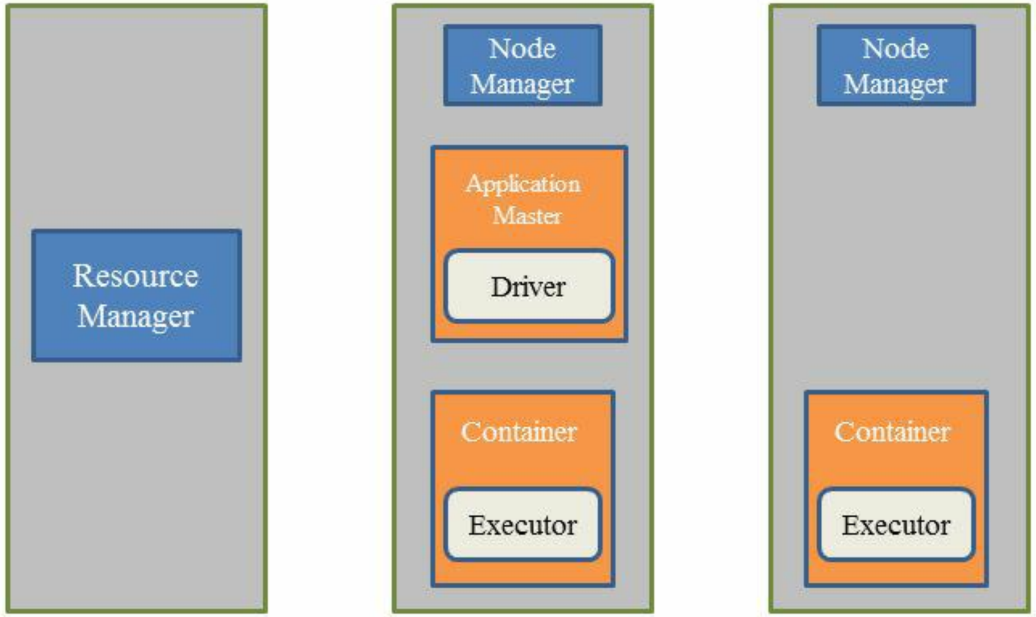
- In YARN client mode, Spark Driver runs in client nodes. The client submits requests to RM, RM launches applications for the application on the cluster, which negotiates resources with RM.
- In YARN, the AM and containers are scheduled by RM, however, tasks that run the application are scheduled by Spark Driver itself



```
cd $SPARK_HOME
./bin/spark-shell -master yarn -deploy-mode client --class org.apache.spark.examples.SparkPi
examples/jars/spark-examples_2.11-2.1.1.jar
```

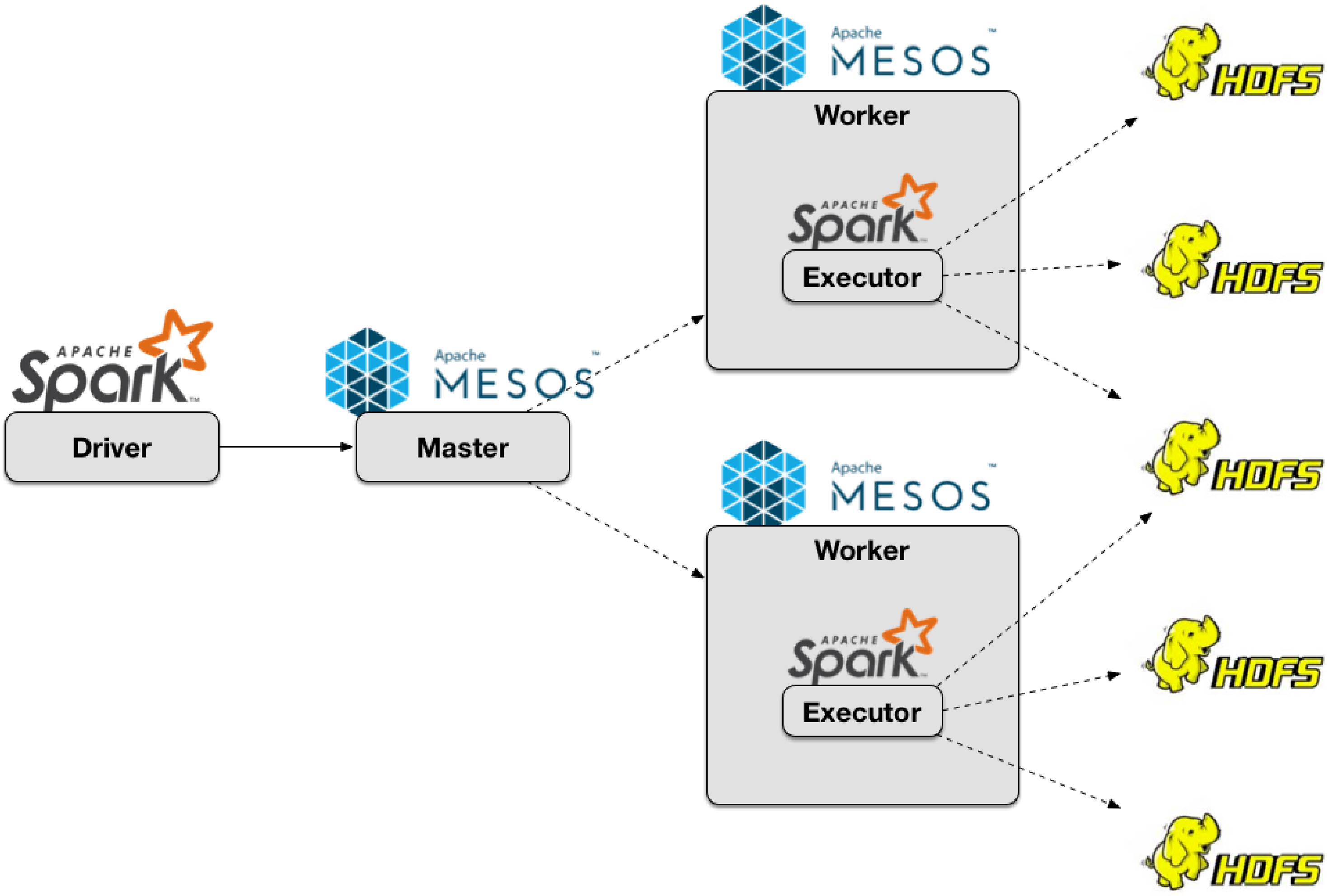
YARN cluster

- In a YARN cluster, a client connects to RM, submits the request, and exits. RM launches the AM of the Spark application on one of the NM and the AM negotiates resources with the AM and then launches the driver thread. NMs launch containers that connect to AM. After containers initialize, AM launches executors in containers, which connect to the Driver in the AM to run the application.



```
cd $SPARK_HOME
./bin/spark-submit --master yarn --deploy-mode cluster --class org.apache.spark.examples.SparkPi
examples/jars/spark-examples_2.11-2.1.1.jar
```

Apache Mesos



persistence and cache

- Spark provides two different options for persisting the intermediate RDD, that is, cache() and persist()
  - The cache() method persists the data unserialized in the memory

Example: Mining Console Logs

- Load error messages from a log into memory, then interactively search for patterns

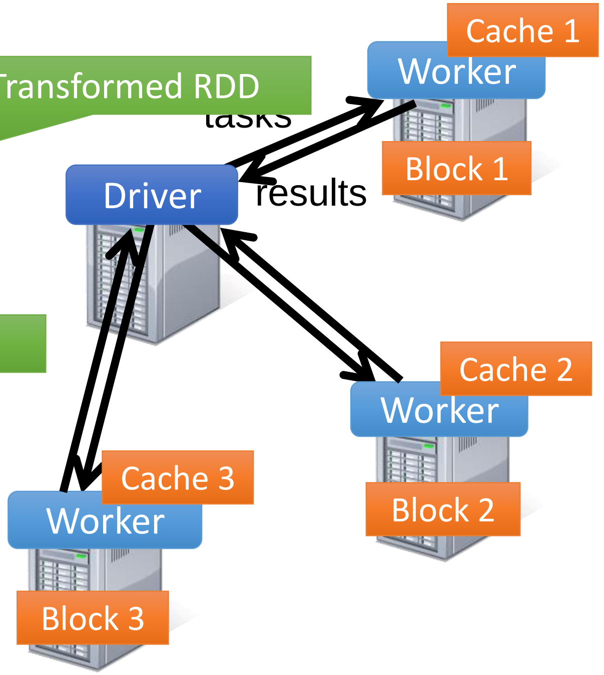
```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(‘\t’)[2])
messages.cache()

messages.filter(lambda s: “foo” in s).count()
messages.filter(lambda s: “bar” in s).count()
. . .
```

Base RDD

Transformed RDD

Action



Storage Level

Storage Level	Memory Footprint	Computation Time	Replication	Serialized Objects	Memory Usage	Disk
MEMORY_ONLY	HIGH	LOW	NO	NO	YES	NO
MEMORY_ONLY_2	HIGH	LOW	YES	NO	YES	NO
MEMORY_ONLY_SER	LOW	HIGH	NO	YES	YES	NO
MEMORY_ONLY_SER_2	LOW	HIGH	YES	YES	YES	NO
DISK_ONLY	LOW	HIGH	NO	YES	NO	YES
DISK_ONLY_2	LOW	HIGH	YES	YES	NO	YES
MEMORY_AND_DISK	HIGH	MEDIUM	NO	PARTIAL	PARTIAL	PARTIAL
MEMORY_AND_DISK_2	HIGH	MEDIUM	YES	PARTIAL	PARTIAL	PARTIAL
MEMORY_AND_DISK_SER	LOW	HIGH	NO	YES	PARTIAL	PARTIAL
MEMORY_AND_DISK_SER_2	LOW	HIGH	YES	YES	PARTIAL	PARTIAL
OFF_HEAP	LOW	HIGH	NO	YES	OFF-HEAP MEMORY	NO

Guiding Principles

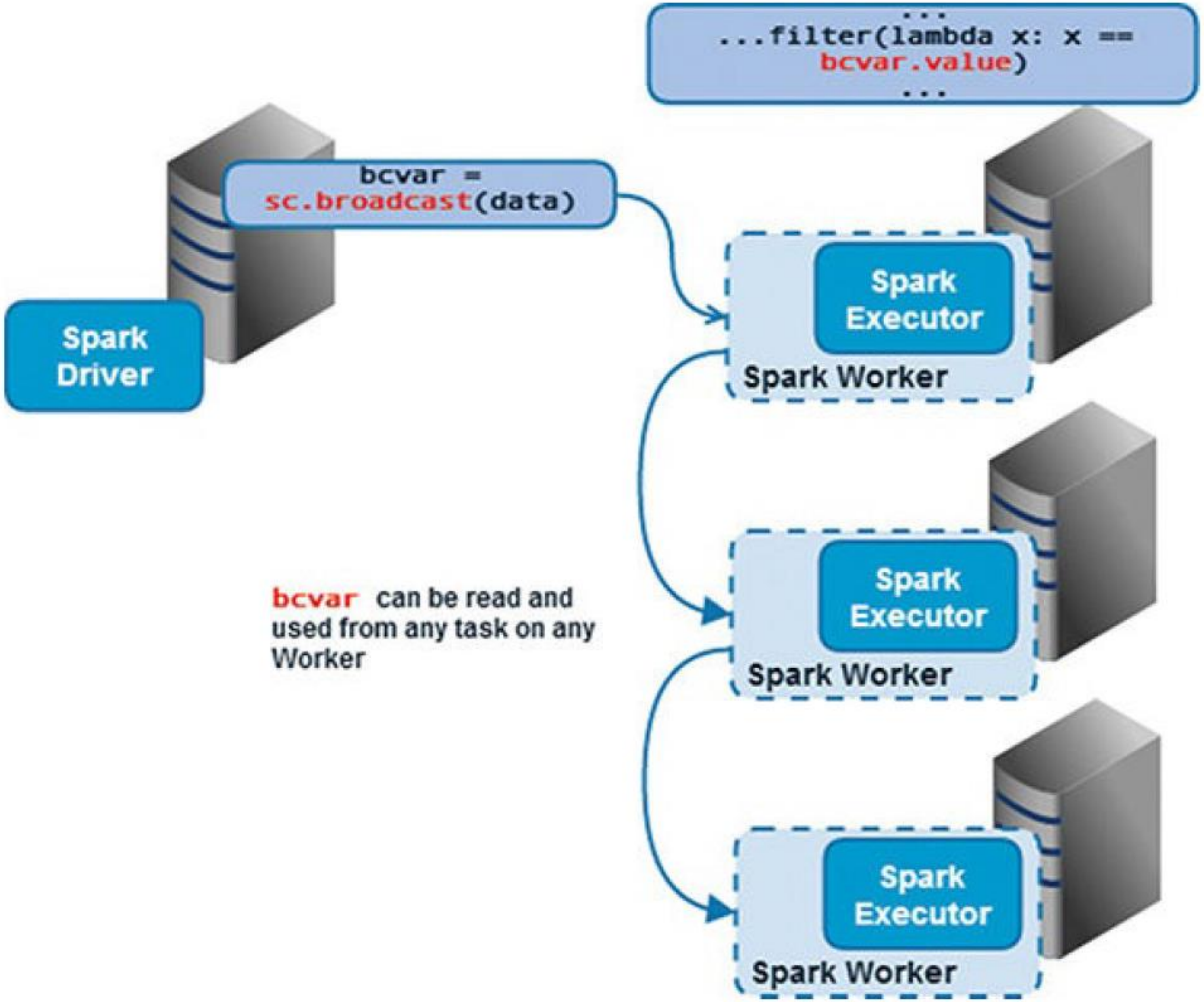
- Prefer MEMORY_ONLY storage as long as complete RDD can be persisted in memory.

- Try MEMORY_ONLY_SER if memory required for RDD persistence is slightly more than available, however, this option increases the operation time as some computation gets consumed in serializing and de-serializing the data.

- Spilling the RDD data on disk may sometimes be a far more expensive operation than re-computing the partition itself.

- Data replication should be chosen for use cases where extremely high throughput is required such that the time lost in recomputation of lost partitions be avoided at all cost

Broadcast Variables



```
...
...filter(lambda x: x ==
bcvar.value)
...
```

```
bcvar =
sc.broadcast(data)
```

bcvar can be read and used from any task on any Worker