PySpark Core

RDD & Shared Variables

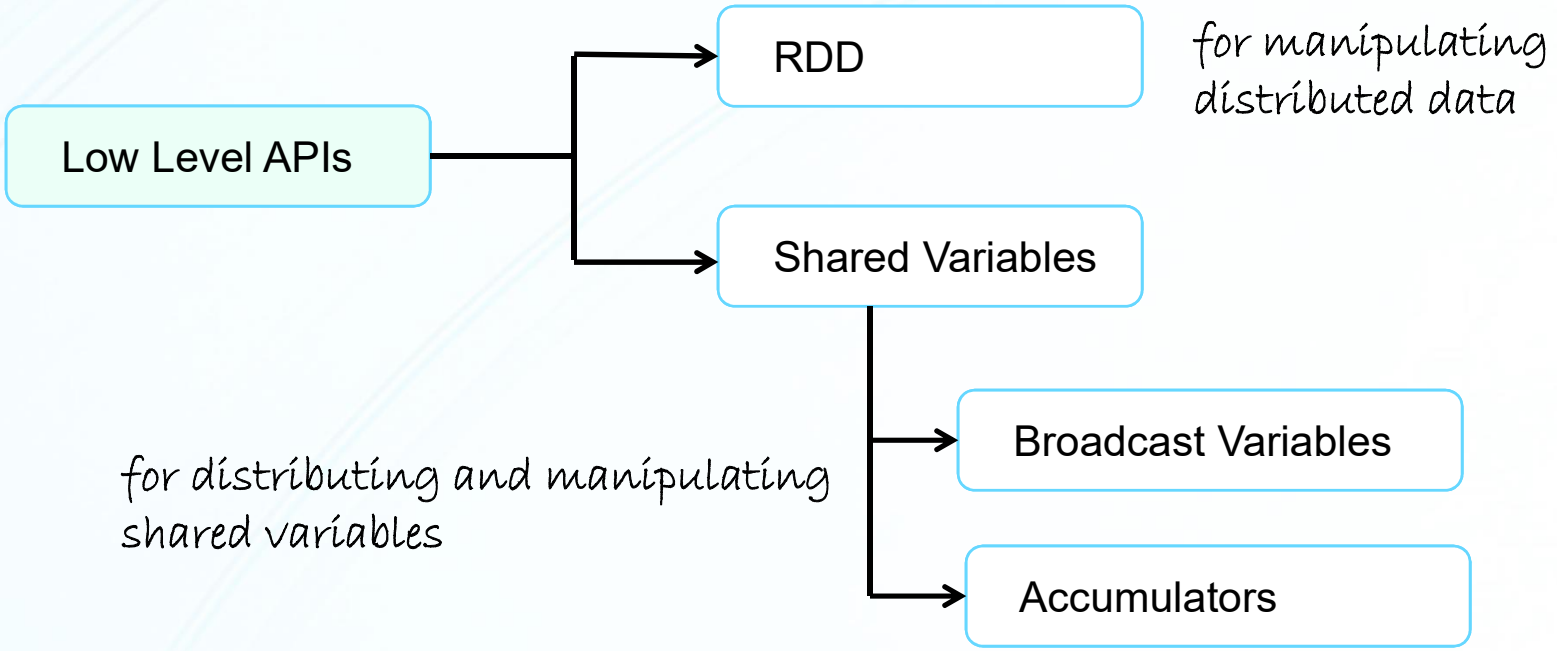
Spark Core – Low Level API

Agenda

In this module, we are going to look at the following topics:

- ✓ Low Level APIs
- ✓ Understanding RDDs
- ✓ Creating RDDs
- ✓ RDD Transformations
- ✓ RDD Actions
- ✓ Reading from & writing to files
- ✓ Spark Shared Variables
- ✓ Accumulators & Broadcast Variables

Low Level APIs



When to use Low Level APIs

- You should generally use the lower-level APIs in three situations:
  - You need some functionality that you cannot find in the higher-level APIs
    - Example: you need fine-grained control over the physical distribution of data (custom partitioning of data).
  - You need to maintain some legacy codebase written using RDDs
  - You need to do some custom shared variable manipulation

Spark Context

- A **`SparkContext`** is the entry point for low-level API functionality.
- You access it through the **`SparkSession`**, which is the tool you use to perform computation across a Spark cluster.
  - *`spark.sparkContext`*

A note about RDDs

- RDDs were the primary API in the Spark 1.X series and are still available in 2.X, but they are not as commonly used.
- However, virtually all Spark code you run, whether DataFrames or Datasets, compiles down to an RDD.
- All job execution is only described in terms of RDDs. Therefore, it is essential to have a good understanding of what an RDD is and how to use it.

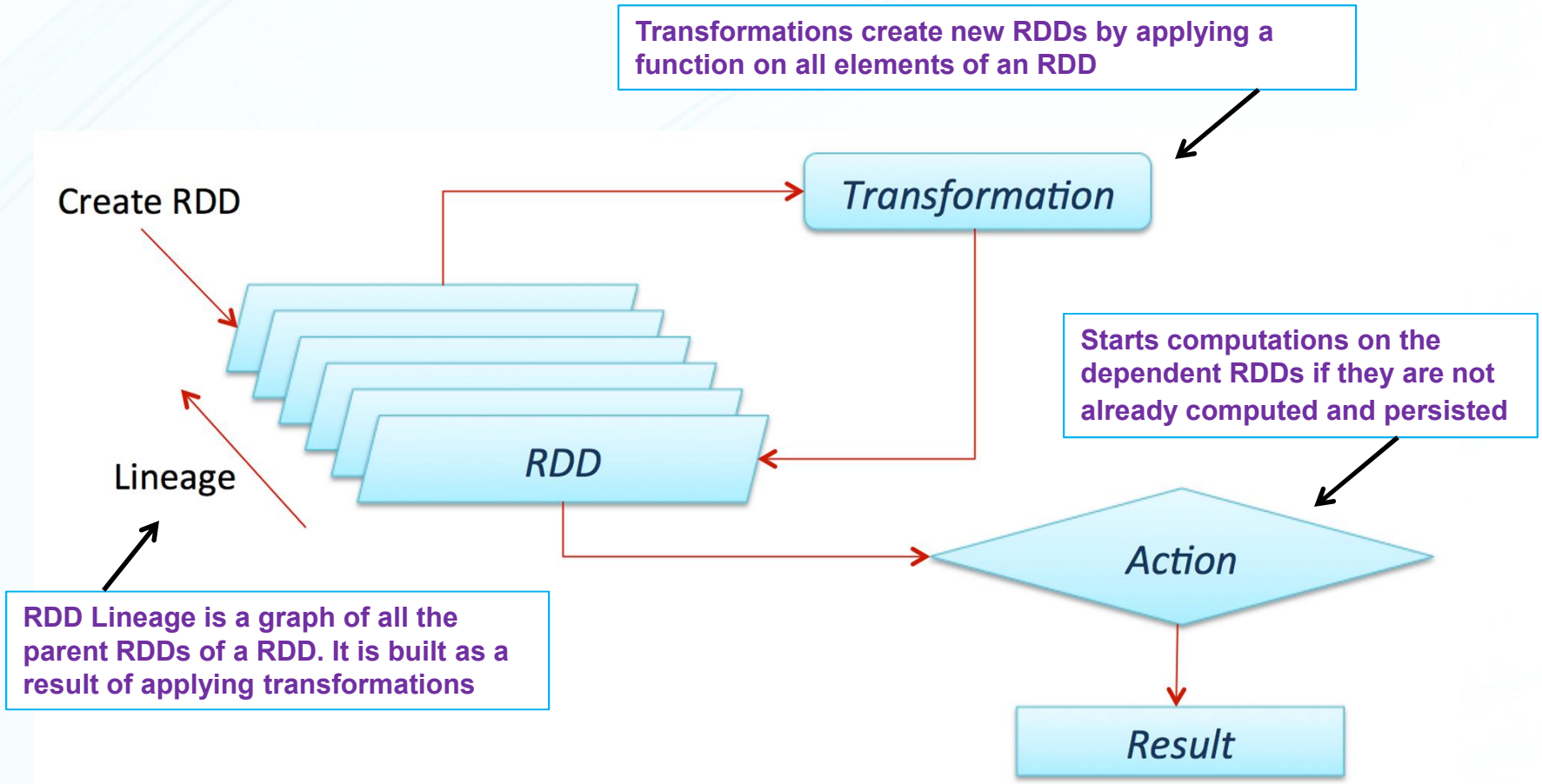
What is an RDD ?

- RDD is the fundamental data abstraction of Apache Spark
- RDD represents an immutable, partitioned collection of records that can be operated on in parallel
- RDD (Resilient Distributed Dataset)
  - **Resilient** – Fault-tolerant. (Lineage graph)
  - **Distributed** – Stored in memory across the cluster on multiple nodes
  - **Dataset** – Initial data can come from a file or created programmatically

What is an RDD ?

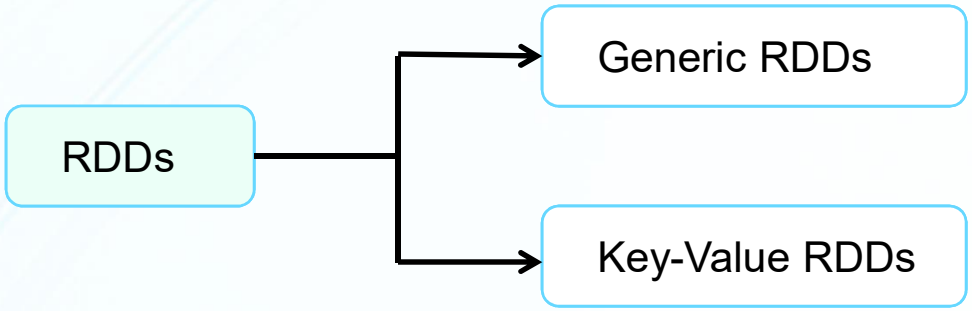
- Each RDD is characterized by five main properties:
  - A list of partitions
  - A function for computing each split
  - A list of dependencies on other RDDs
  - Optionally, a Partitioner for key-value RDDs
  - Optionally, a list of preferred locations on which to compute each split

Understanding RDD



Types of RDDs

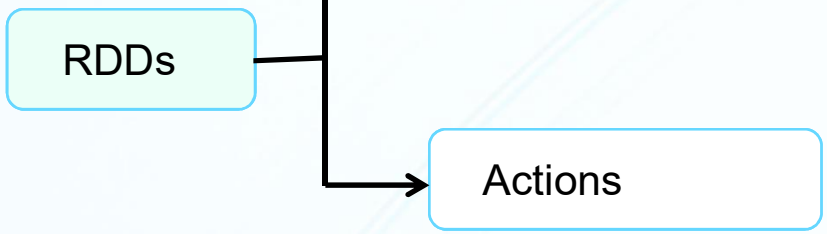
- From a user's perspective, we will mainly be using two types of RDDs



Creating RDDs

- RDDs can be created in three ways:
  - Using a dataset from external storage (such as HDFS)
    ```
    lines = sc.textFile("data.txt")
    ```
  - From an in-memory collection of objects (using **parallelize** method**)**
    ```
    distData = sc.parallelize( [1,2,3,4,5] )
    ```
  - As a transformation on an existing RDD
    ```
    words = lines.map(lambda line: line.split(","))
    ```

RDD Operations – Transformations & Actions

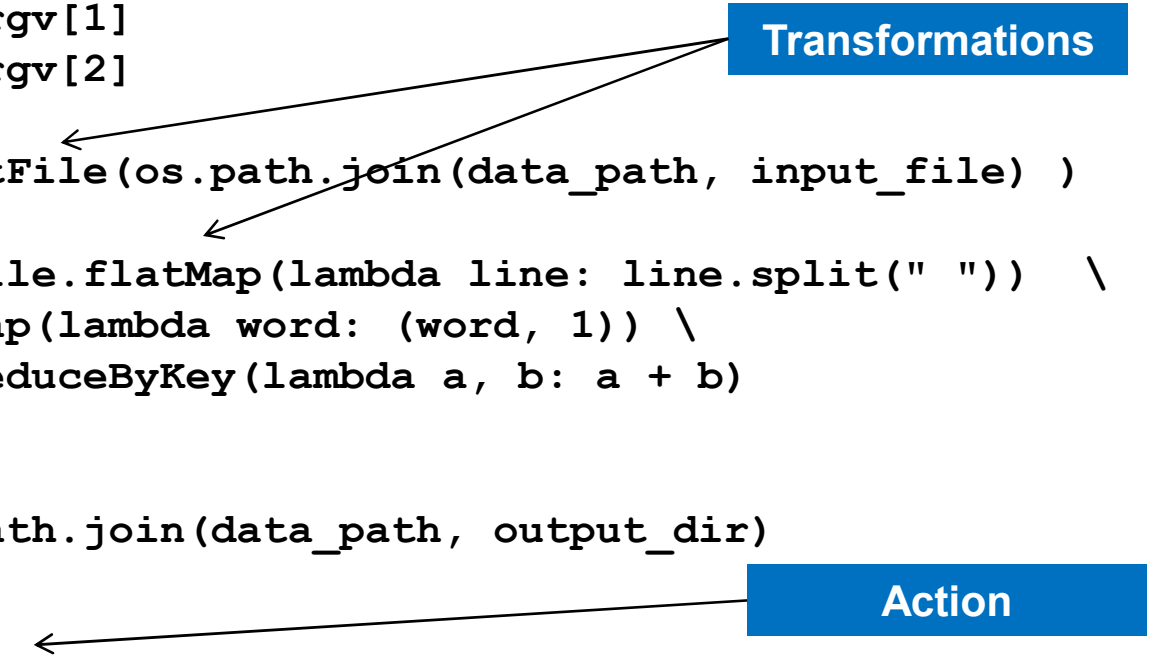


- Creates an RDD
- Lazily evaluated
- Constructs a lineage graph (DAG)

- Returns a final value to the driver
- Force evaluation of transformations

WordCount Program using RDDs

```
sc = SparkContext("local", "wordcount")

input_file = sys.argv[1]
output_dir = sys.argv[2]

text_file = sc.textFile(os.path.join(data_path, input_file) )

wordcount = text_file.flatMap(lambda line: line.split(" "))  \
                .map(lambda word: (word, 1)) \
                .reduceByKey(lambda a, b: a + b)


output_path = os.path.join(data_path, output_dir)

wordcount.saveAsTextFile(output_path)
```



Lazy Evaluation

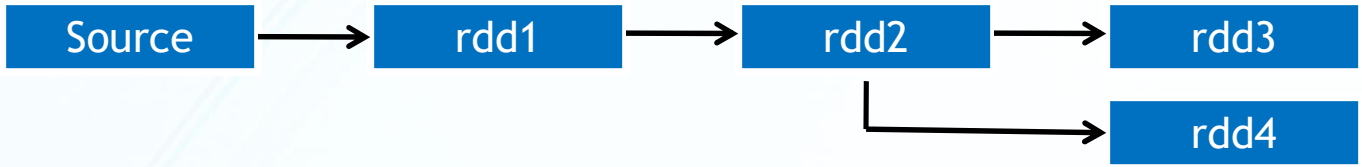
- Lazy evaluation Applies to both data loading and RDD operations
- Lazy evaluation means that when we call a transformation on an RDD the operation is not immediately performed. Spark internally records the meta-data to indicate that this operation is requested.
- Rather than thinking of an RDD as having specific data, it is best to think of each RDD as consisting of instructions on how to compute the data that is built up through transformations.

Lineage graph & Logical execution plan

- RDD Lineage is a graph of all the parent RDDs of an RDD. It is built as a result of applying transformations to RDDs and creates a **logical execution plan**.
- Logical Execution Plan starts with the earliest RDDs (those with no dependencies on other RDDs or reference cached data) and ends with the RDD that produces the result of the action that has been called to execute.
- A logical plan, i.e. a DAG, is materialized and executed when SparkContext is requested to run a Spark job

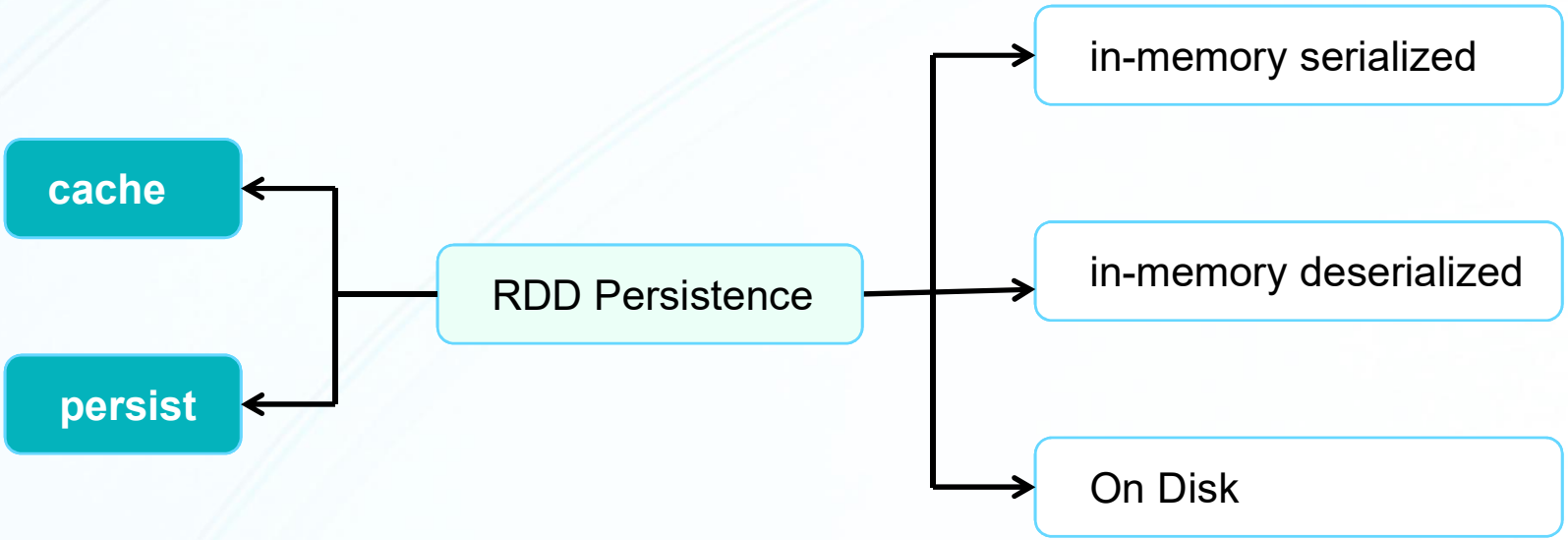
Lineage graph & Logical execution plan

```
val rdd1 = sc.textFile("data/serverlog-sample-1.txt")
val rdd2 = rdd1.map(line => line.split(" "))
val rdd3 = rdd2.map(words=>(words(0),1)).reduceByKey((x,y)=>x+y)
val rdd4 = rdd2.count
val rdd3_lineage = rdd3.toDebugString
```




```
(1) ShuffledRDD[5] at reduceByKey at rdd_lineage_1.scala:19 []
 +-(1) MapPartitionsRDD[4] at map at rdd_lineage_1.scala:18 []
    |  MapPartitionsRDD[3] at filter at rdd_lineage_1.scala:16 []
    |  MapPartitionsRDD[2] at map at rdd_lineage_1.scala:15 []
    |  data/serverlog-sample-1.txt MapPartitionsRDD[1] at textFile at rdd_lineage_1.scala:13 []
    |  data/serverlog-sample-1.txt HadoopRDD[0] at textFile at rdd_lineage_1.scala:13 []
```

RDD Persistence and Memory Management



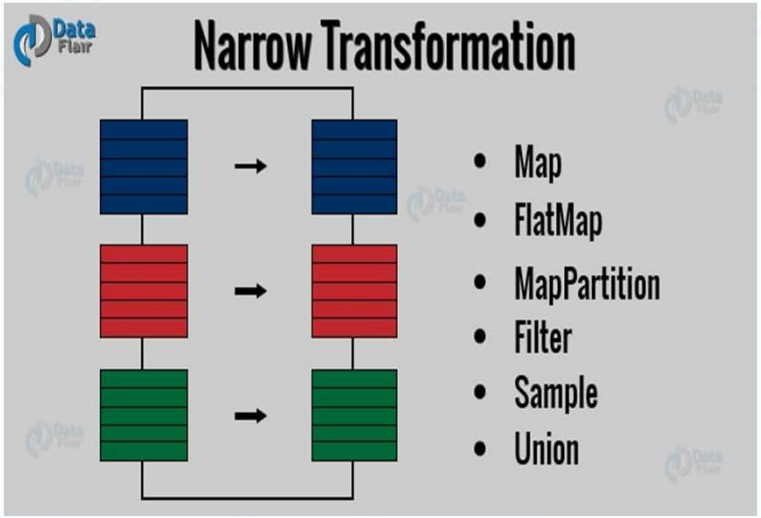

```
f = sc.parallelize(range(1, 100), 5)
f.cache()
f.persist(pyspark.StorageLevel.MEMORY_ONLY)
f.unpersist()
```

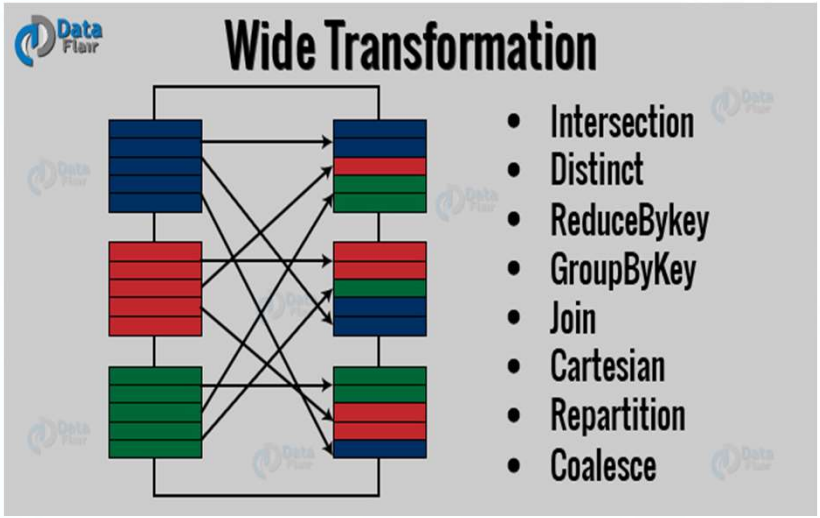
RDD Persistence - Storage Levels

Storage Level	Meaning
MEMORY_ONLY	DEFAULT - 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.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions on disk that don't fit on memory, and read them from there when they're needed.
DISK_ONLY	Store the RDD partitions only on disk
MEMORY_ONLY_SER	Store RDD as serialized java objects(one byte array per partition)
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of re-computing them on the fly each time they're needed.
MEMORY_ONLY_2, MEMORY_AND_DISK_2	Same as the levels above, but replicate each partition on two cluster nodes.

RDD Transformations

RDD Transformations





RDD Transformations

Element-wise transformations on a single RDD			
map	filter	flatMap	distinct
partitionBy	repartition	mapPartition	mapPartitionWithIndex
coalesce	pipe		

Element-wise transformations on a two RDDs			
union	intersection	subtract	cartesion

Transformations on pair RDDs			
reduceByKey	groupByKey	combineByKey	mapValues
flatMapValues	keys, values	aggregateByKey	sortByKey

Transformations on a two pair RDDs			
subtractByKey	join	rightOuterJoin	leftOuterJoin
fullOuterJoin	cogroup		

RDD Actions

RDD Actions			
collect	count	countByValue	first
take	takeOrdered	takeSample	top
fold	aggregate	forEach	Reduce
saveAsTextFile	saveAsSequenceFile	saveAsObjectFile	

RDD Transformations

map(*func*)

- Return a new RDD formed by passing each element of the source through a function *func*.

```
sc.parallelize([1,2,3]).map(lambda x: x*x)
```

RDD Transformations

`filter`(*`func`*)

- Return a new RDD formed with elements on which *func* returns true.

```
sc.parallelize([1,2,3]).filter(lambda x: x%2 == 0)
```

RDD Transformations

distinct

- Removes duplicates from the RDD

```
sc.parallelize([4,5,4,5]).distinct()
```

RDD Transformations

`flatMap`(*`func`*)

- Similar to map, but each input item can be mapped to 0 or more output items.
- *func* should return an **`iterator`** rather than a single item.

```
input = sc.textFile(inputFile)
words = input.flatMap(lambda line: line.split(" "))
```

RDD Transformations

`mapPartitions`(*`func`*)

- Similar to map, but runs separately on each partition of the RDD,
- *func* must be of type **`Iterator<T> => Iterator<U>`** when running on an RDD of type T.

```
rdd = sc.parallelize([1, 2, 3, 4], 2)
def f(iterator): yield sum(iterator)
rdd.mapPartitions(f).collect()
```

RDD Transformations

`mapPartitionsWithIndex`(*`func`*)

- Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition,
- *func* must be of type **`(Int, Iterator<T>) => Iterator<U>`** when running on an RDD of type T.

RDD Transformations

sortBy(*func*)

- Sorts the elements of an RDD
- Specify a function to extract a value from the objects in your RDDs and then sort based on that

```
val words = sc.textFile(inputFile)
words.sortBy( lambda a: len(a) )
```

RDD Transformations

randomSplit

- Splits an RDD into an Array of RDDs
- Accepts an Array of weights and a random seed as parameters

```
input = sc.textFile(inputFile)
rddSplits = words.randomSplit([0.5, 0.5])
```

RDD Transformations

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.

```
rpRdd = srcRdd.repartition(3)
```

RDD Transformations

coalesce(*numPartitions*)

- Decrease the number of partitions in the RDD to numPartitions.
- Useful for running operations more efficiently after filtering down a large dataset.

```
clscRdd = srcRdd.coalesce(3)
```

RDD Transformations

partitionBy(*partitioner*)

- Operates on RDDs consisting of (k, v) pairs.
- For each element of this RDD, the partitioner is used to compute a hash function and the RDD is partitioned using this hash value.

```
rdd.partitionBy(new HashPartitioner(2))
rdd.partitionBy(new RangePartitioner(2, rdd))
```

RDD Transformations - set operations

union(*otherDataset*)

- Returns a new RDD that contains the union of the elements in the source dataset and the argument.

intersection(*otherDataset*)

- Returns a new RDD that contains the intersection of elements in the source dataset and the argument.

```
rddUnion = rdd1.union(rdd2)
rddIntrsect = rdd1.intersection(rdd2)
```

RDD Transformations - set operations

subtract(*otherDataset*)

- Returns new RDD with elements of argument removed from source RDD.

cartesian(*otherDataset*)

- Returns a new RDD with the cartesian product of two RDDs

```
rddSub  = rdd1.subtract(rdd2)
rddCart = rdd1.cartesian(rdd2)
```

RDD Transformations

groupByKey([*numPartitions*])

- When called on an RDD of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
- If you are grouping in order to perform an aggregation over each key using **reduceByKey** or **aggregateByKey** will yield much better performance.
- 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.

```
rdd2 = rdd1.groupByKey(2)
```

RDD 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.
- Number of reduce tasks is configurable through an optional second argument.

```
rdd2 = rdd1.reduceByKey( lambda x,y: x + y )
```

RDD Transformations

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.

```
rdd2 = rdd1.sotyByKey()
rdd2 = rdd1.sotyByKey(ascending = False)
rdd2 = rdd1.sotyByKey(ascending = False, 3)
```

RDD Transformations

aggregateByKey(*zeroVal*)(*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.

```
rdd2 = rdd1.aggregateByKey(zero_val, seq_op, comb_op)
```

RDD Transformations

`combineByKey(createCombFn, combinerFn, mergerFn)`

- When using **`combineByKey`**, values are merged into one value at each partition, and then, each partition value is merged into a single value.
- The **`combineByKey`** function takes 3 functions as arguments:
  - A function that creates a combiner.
  - A function that takes a value and merges it with previous collection.
  - A function that combines the merged values together

RDD Transformations

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**.

```
join = rdd1.join(rdd2)
leftOuterJoin = rdd1.leftOuterJoin(rdd2)
rightOuterJoin = rdd1.rightOuterJoin(rdd2)
fullOuterJoin = rdd1.fullOuterJoin(rdd2)
```

RDD Transformations

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.

```
cogroupRdd = rdd1.cogroup(rdd2)
```

RDD Actions

RDD Actions

`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.

RDD Actions

- `count()`
  - Return the number of rows in the RDD.
- `countByKey()`
  - Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
- `countByValue()`
  - Counts how many times each value occurred in the RDD.

RDD Actions

- `first()`
  - Return the first element of the dataset.
- `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.

RDD Actions

- **takeOrdered(*n*, *[ordering]*)**
  - Return the first *n* elements of the RDD using either their natural order or a custom comparator.

```
rdd = sc.parallelize( [1,2,3,4,5,6] )
ord = rdd.takeOrdered(6, key=lambda x: -x)
```

RDD Actions

- **`reduce`**(*`func`*)
  - Aggregate the elements of the dataset using a reduce function.
  - The function should be commutative and associative so that it can be computed correctly in parallel.

RDD Actions

- `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.
  - **Note**: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior.

Saving to Files

RDD Actions – Saving to files

- **saveAsTextFile**(*path*)
  - Write the elements of the dataset as a text file (or set of text files) in a given directory in the local file system, 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.

RDD Actions – Saving to files

- **saveAsSequenceFile**(*path*)
  - 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).

RDD Actions – Saving to files

- **saveAsObjectFile**(*path*)
  - Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().

Reading from Files

Reading from files

- When we load a single text file as an RDD, each input line becomes an element in the RDD.
- We can also load multiple whole text-files into a Pair RDD, with the key being the name and the value being the content of each file.

```
input = sc.textFile("/path/of/the/file.txt")
```

```
input = sc.wholeTextFiles("/path/to/directory")
```

Spark Shared Variables

Spark Closures

In Spark, a **closure** constitutes all the variables and methods which must be visible for the executor to perform its computations on the RDD. This closure is serialized and sent to each executor.

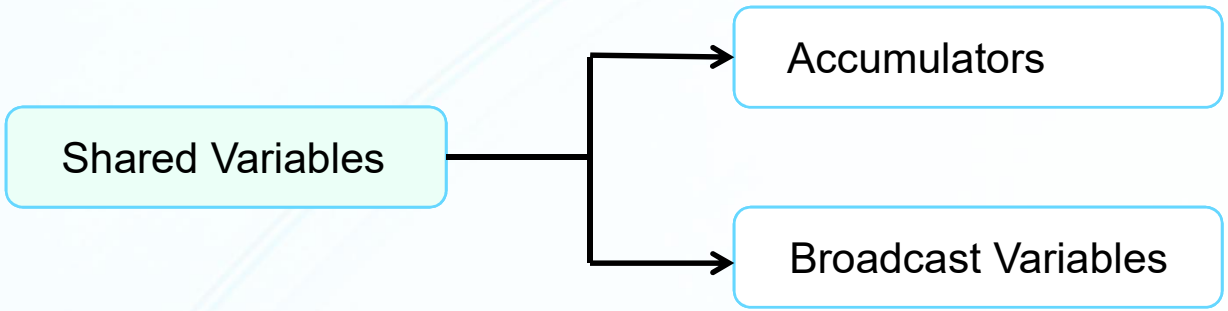
Consider the following example:

The variables within the closure sent to each executor are now copies and thus, when **counter** is referenced within the foreach function, it's no longer the **counter** on the driver node.

```
c = 0
rdd = sc.parallelize(data)

// Wrong: Don't do this!!
rdd.foreach(lambda x: c += x)
println("Counter:" + c)
```

To ensure well-defined behavior in these sorts of scenarios one should use an **Accumulator**. Accumulators in Spark are used specifically to provide a mechanism for safely updating a variable when execution is split up across worker nodes in a cluster.

Spark Distributed Shared Variables



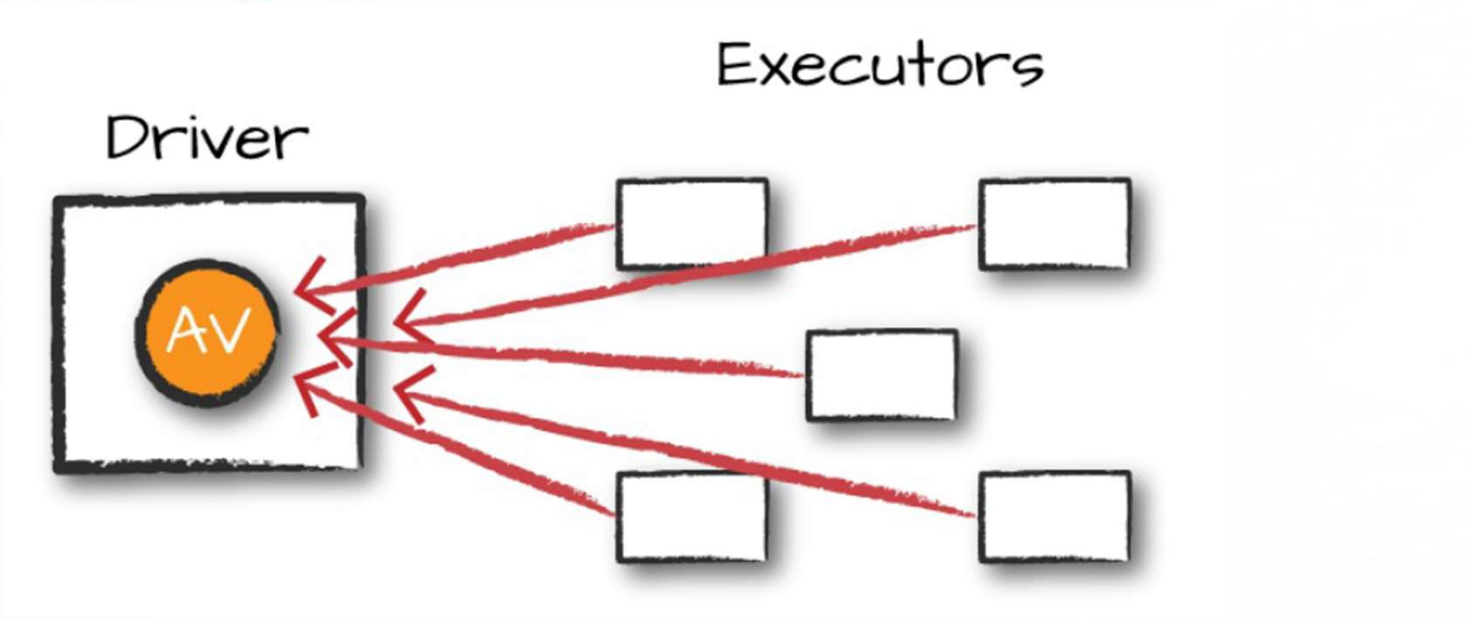
Distributed Shared Variables

- Normally when we pass functions such as `map` or `filter` to Spark, they can use variables defined outside of them in the driver program, but each task running on the cluster gets a new copy of each variable, and updates from these copies are not propagated back to the driver.

- Spark's shared variables – '**accumulators**' and '**broadcast variables**' relax this restriction for two common types of communication patterns – aggregation of results & broadcasts.

Accumulators

- *Accumulators* let you add together data from all the tasks into a shared result (e.g., to implement a counter so you can see how many of your job's input records failed to parse)



Accumulators

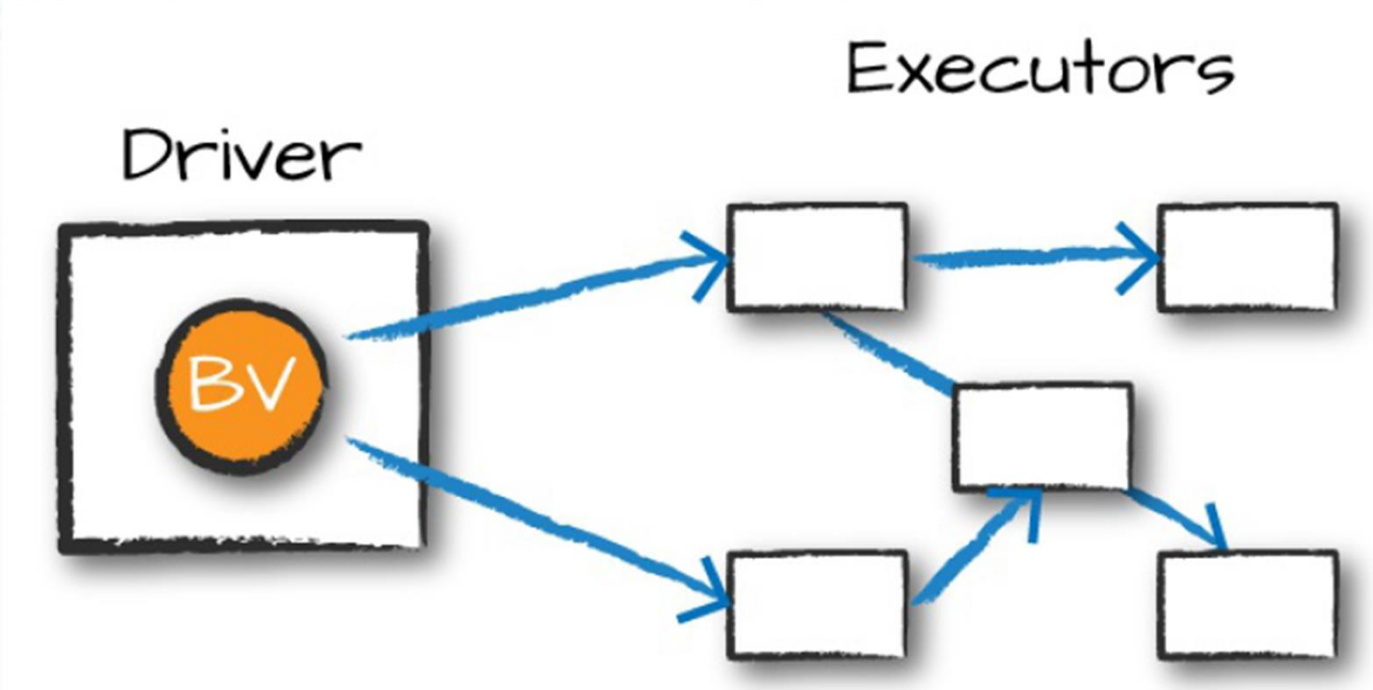
- Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel.
- Spark natively supports accumulators of numeric types. Programmers can add support for new types.
- A numeric accumulator can be created by calling **`sc.longAccumulator()`** or **`sc.doubleAccumulator()`** to accumulate values of type Long or Double, respectively. Tasks running on a cluster can then add to it using the **`add`** method. However, they cannot read its value. Only the driver program can read the accumulator’s value, using its **`value`** method.

```
accum = sc.accumulator(0)

sc.parallelize([1, 2, 3, 4])
  .foreach(lambda x: accum.add(x))
```

Broadcast Variables

- Broadcast variables are a way you can share an immutable value efficiently around the cluster without encapsulating that variable in a function closure.
- This is where broadcast variables come in. Broadcast variables are shared, immutable variables that are cached on every machine in the cluster instead of serialized with every single task.
- The canonical use case is to pass around a large lookup table that fits in memory on the executors and use that in a function

Broadcast Variables



Broadcast Variables

- Broadcast variables are created from a variable v by calling **`sc.broadcast(v)`**. The broadcast variable is a wrapper around v, and its value can be accessed by calling the value method.

```
broadcastVar = sc.broadcast([1, 2, 3])
broadcastVar.value
```