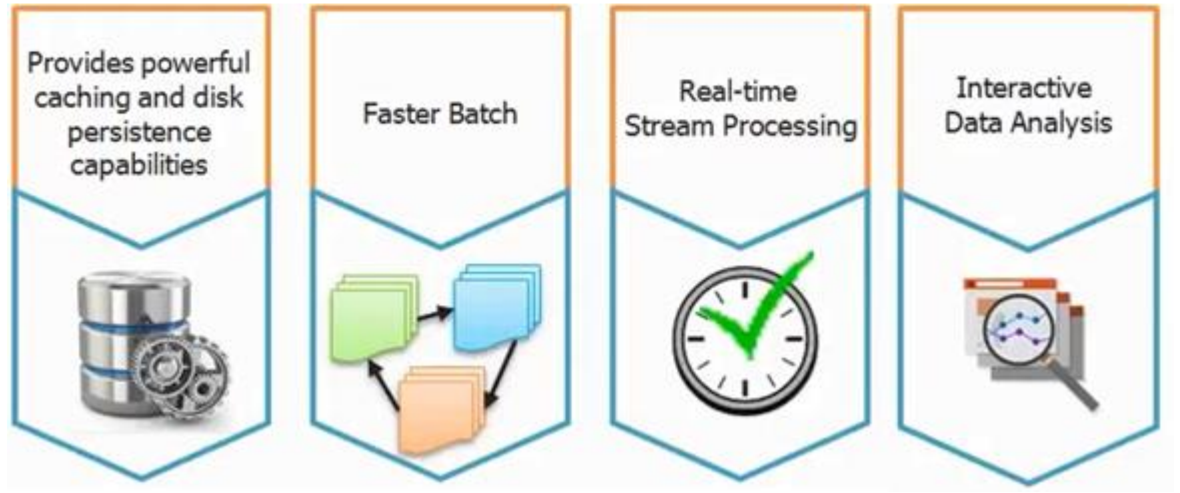
Apache Spark Core

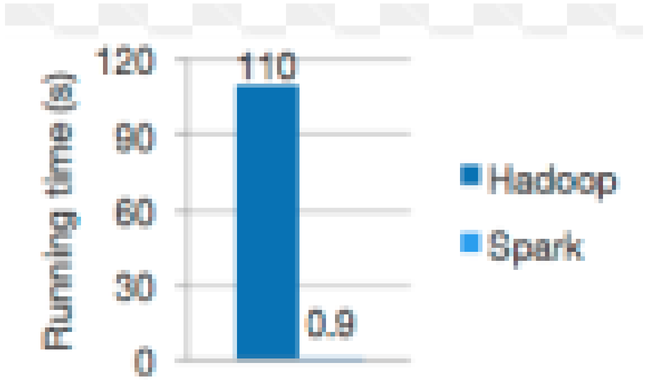
What is Spark

- Apache Spark is a powerful open source processing framework built for
  - Speed
  - Ease of Use
  - Sophisticated Analysis

Up to 10× faster on disk,
100× in memory
Efficient
2-5× less code
Usable

- Spark can be used in conjunction with Hadoop, to analyze data on the Hadoop File System (HDFS), or it can be run on its own.


Provides powerful caching and disk persistence capabilities
Faster Batch
Real-time Stream Processing
Interactive Data Analysis



- Spark is being adopted by major players like Amazon, eBay, and Yahoo (7000 node cluster)

Spark History

- Developed in Berkeley's AMPLab (the AMP stands for Algorithms, Machine and People), California on 2009 and open sourced in 2010 as Apache project
- Spark is written in Scala
- Latest Version: Apache Spark 3.0, as on Dec 2019

Version	Original Release Date	Latest Version	Release Date
0.5	12-06-2012	0.5.1	07-10-2012
0.6	14-10-2012	0.6.2	07-02-2013
0.7	27-02-2013	0.7.3	16-07-2013
0.8	25-09-2013	0.8.1	19-12-2013
0.9	02-02-2014	0.9.2	23-07-2014
1.0	30-05-2014	1.0.2	05-08-2014
1.1	11-09-2014	1.1.1	26-11-2014
1.2	18-12-2014	1.2.2	17-04-2015
1.3	13-03-2015	1.3.1	17-04-2015
1.4	11-06-2015	1.4.1	15-07-2015
1.5	09-09-2015	1.5.2	09-11-2015
1.6	04-01-2016	1.6.3	07-11-2016
2.0	26-07-2016	2.0.2	14-11-2016
2.1	28-12-2016	2.1.2	09-10-2017
2.2	11-07-2017	2.2.1	01-12-2017
2.3	28-02-2018	2.3.0	28-02-2018

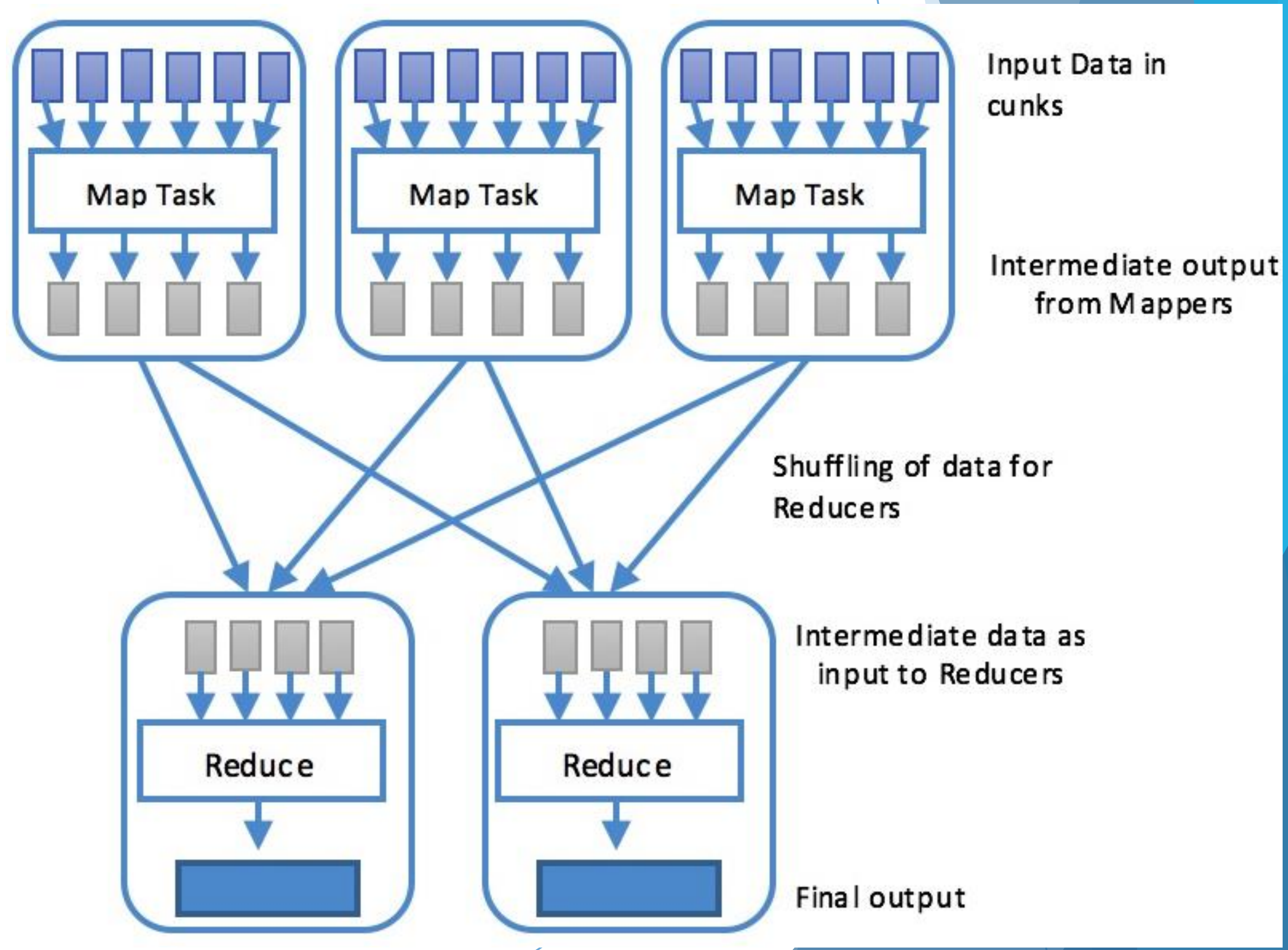
MapReduce and Spark

Similarities:

- Open Source processing framework
- Implemented in JVM based programming languages
- Provide fault tolerance and scalability

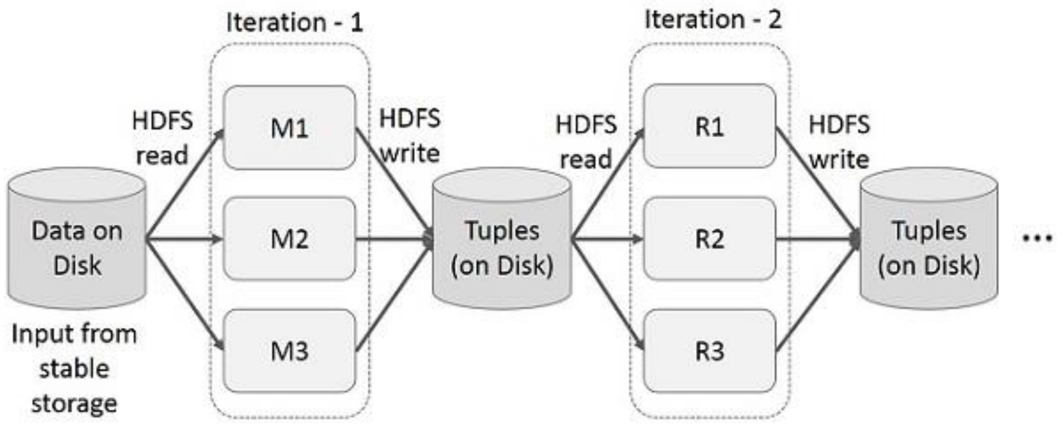
How MapReduce works?

- MR job contains set of mapper and reducer task which stores the intermediate results
- I/O operation take lot of time (For writing into the disk and reading from the disk)

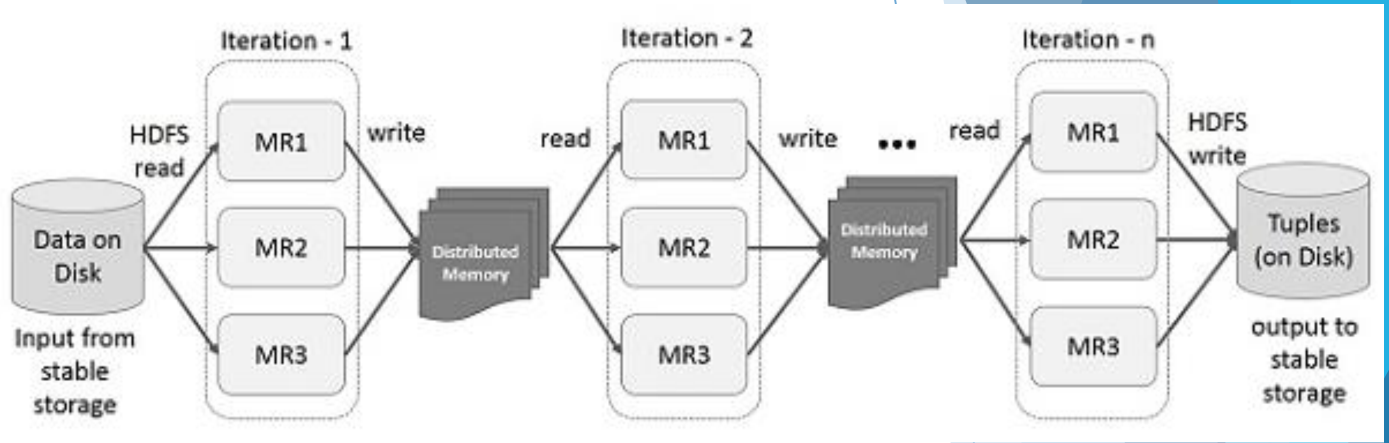

Input Data in cunks
Map Task
Map Task
Map Task
Intermediate output from Mappers
Shuffling of data for Reducers
Intermediate data as input to Reducers
Reduce
Reduce
Final output

Iterative and Interactive – MR vs Spark

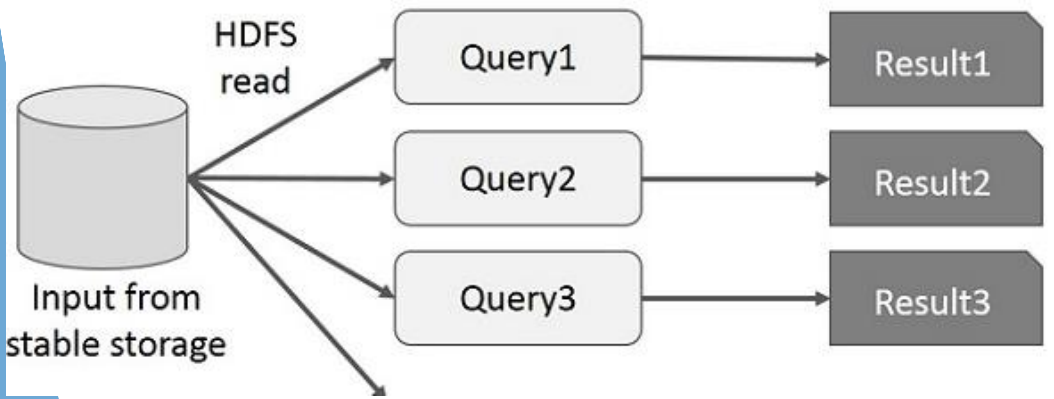
Iterative Operations on MapReduce



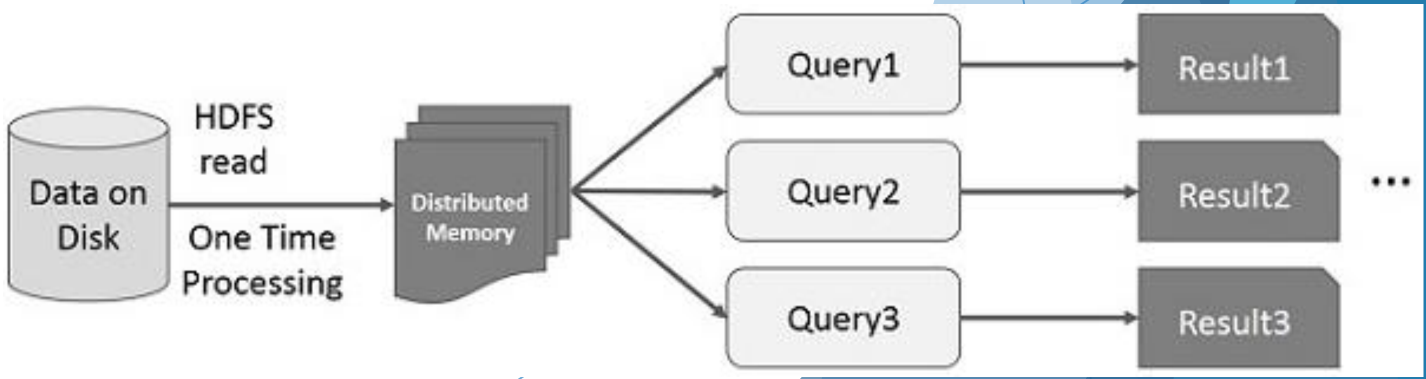
Iterative Operations on Spark RDD



Interactive Operations on MapReduce



Interactive Operations on Spark RDD



MapReduce vs Spark

Criteria	MapReduce	Spark
Performance	Hadoop MapReduce persists data to the disk after a map or reduce action	Apache Spark processes data in-memory
	Latency in few minutes	Latency in few seconds
Ease of Use	Hadoop MapReduce is written in Java and it is very difficult to program	Spark has comfortable APIs for Java, Scala and Python and includes Spark SQL for the SQL savvy
System	Lot of tools available but integration is not quite seamless, requiring lot of effort for their seamless integration	Unifies lot of interfaces like SQL, stream processing etc into single abstraction of RDD
Data Processing	Suitable for batch processing	Data can be cached in memory, great for iterative, interactive and batch processing
Fault Tolerance	Able to handle fault tolerance via persisting the results of each of phases	Exploits immutability of RDD to enable fault tolerance

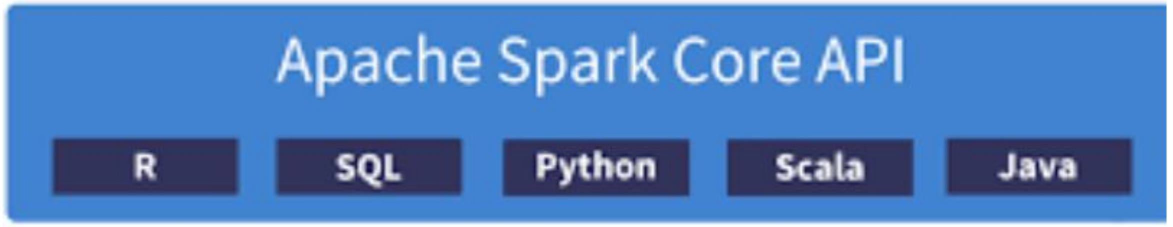
Layer Components

Spark Core:

- Spark Core is the base engine for large-scale parallel and distributed data processing.
- All other functionalities and extensions are built on top of Spark Core.
- It interact with storage systems

Spark API:

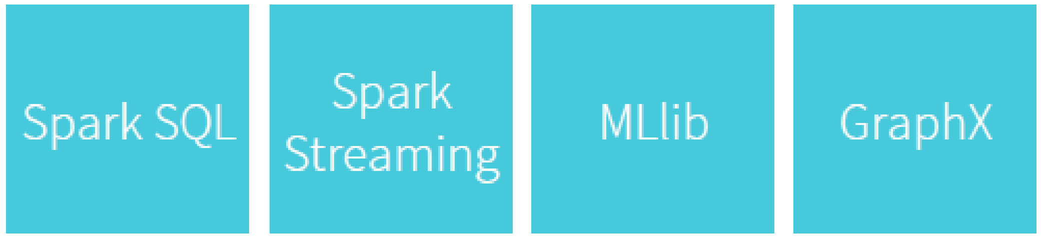
- Spark is written in Scala Programming Language and runs on Java Virtual Machine (JVM)
- The API provides the application developers to create Spark based applications using a standard API interface.
- Spark framework is polyglot – can be programmed in the below languages



Spark Ecosystem

Spark SQL:

- Provides SQL like interface over DataFrames to query data in CSV, JSON, Sequence and Parquet file formats



Spark Machine Learning (MLlib):

- Supports common learning algorithms like dimension reduction, clustering, classification, regression, collaborative filtering etc..

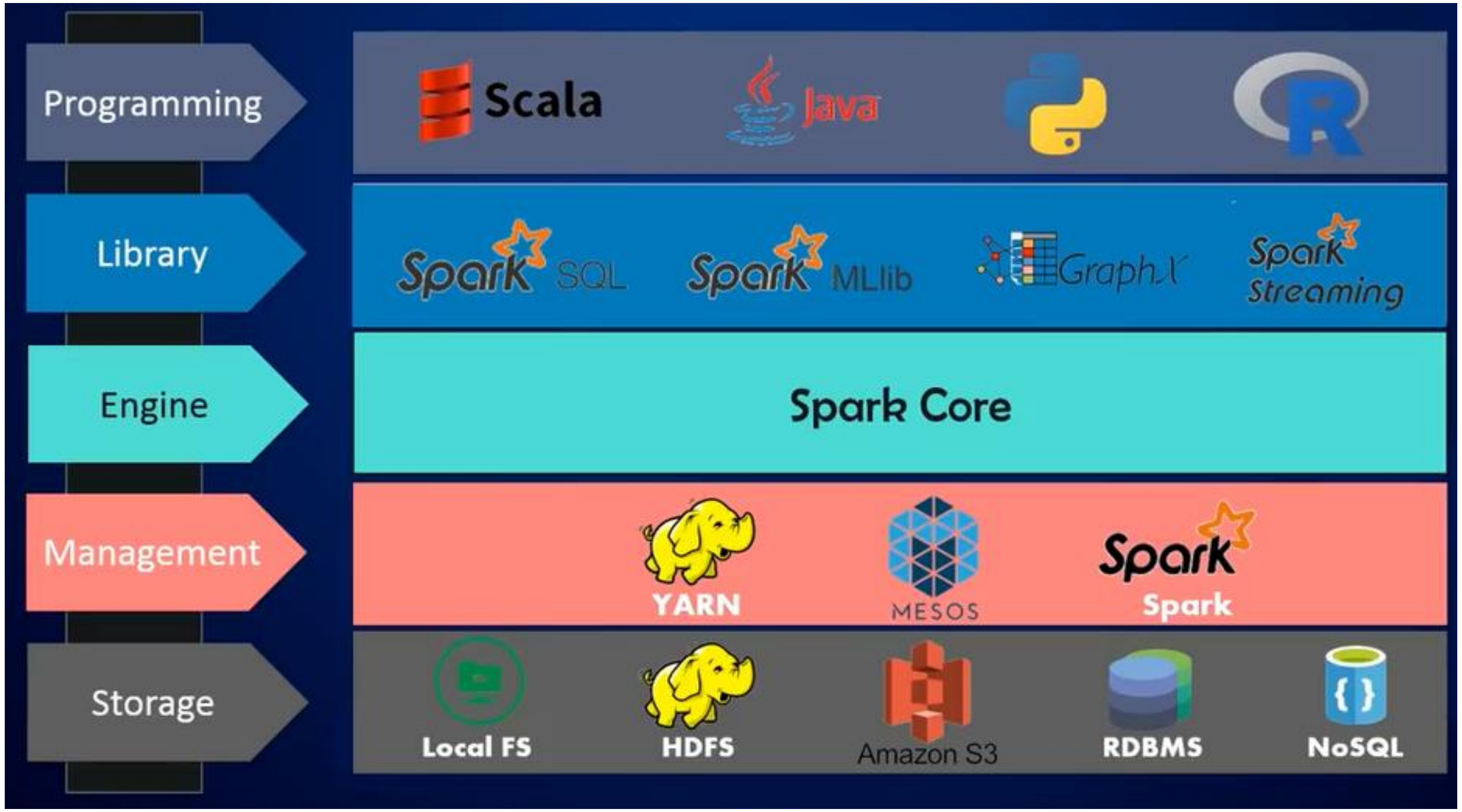
Spark Streaming:

- Adds capability of processing streaming data near to real time
- Capable of ingesting data in micro batches (in the form of micro RDDs) and performs transformation on series of micro batches(RDDs)

GraphX:

- **GraphX** is Apache Spark's API for graphs and graph-parallel computation.

Spark – Layered Architecture



- Dataframes is used for interact with other spark components
- Spark ML standardizes APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single pipeline

Cluster Architecture - Components

Application:

- User program built on Spark.

Driver Program:

- It is a JVM that interprets java or scala code
- It creates spark context object
- Launch spark applications and asks Cluster Manager for resources
- Assign tasks to Executors

Spark Context:

- SparkContext represents the connection to a Spark execution environment
- Create a Spark context to use Spark features and services in your application

Cluster Architecture - Components

Cluster Manager:

- Allocating available resources for each applications
- Cluster manager gives the resources called executors to driver program

Worker Node:

- Worker Node is a node on which executors are launched

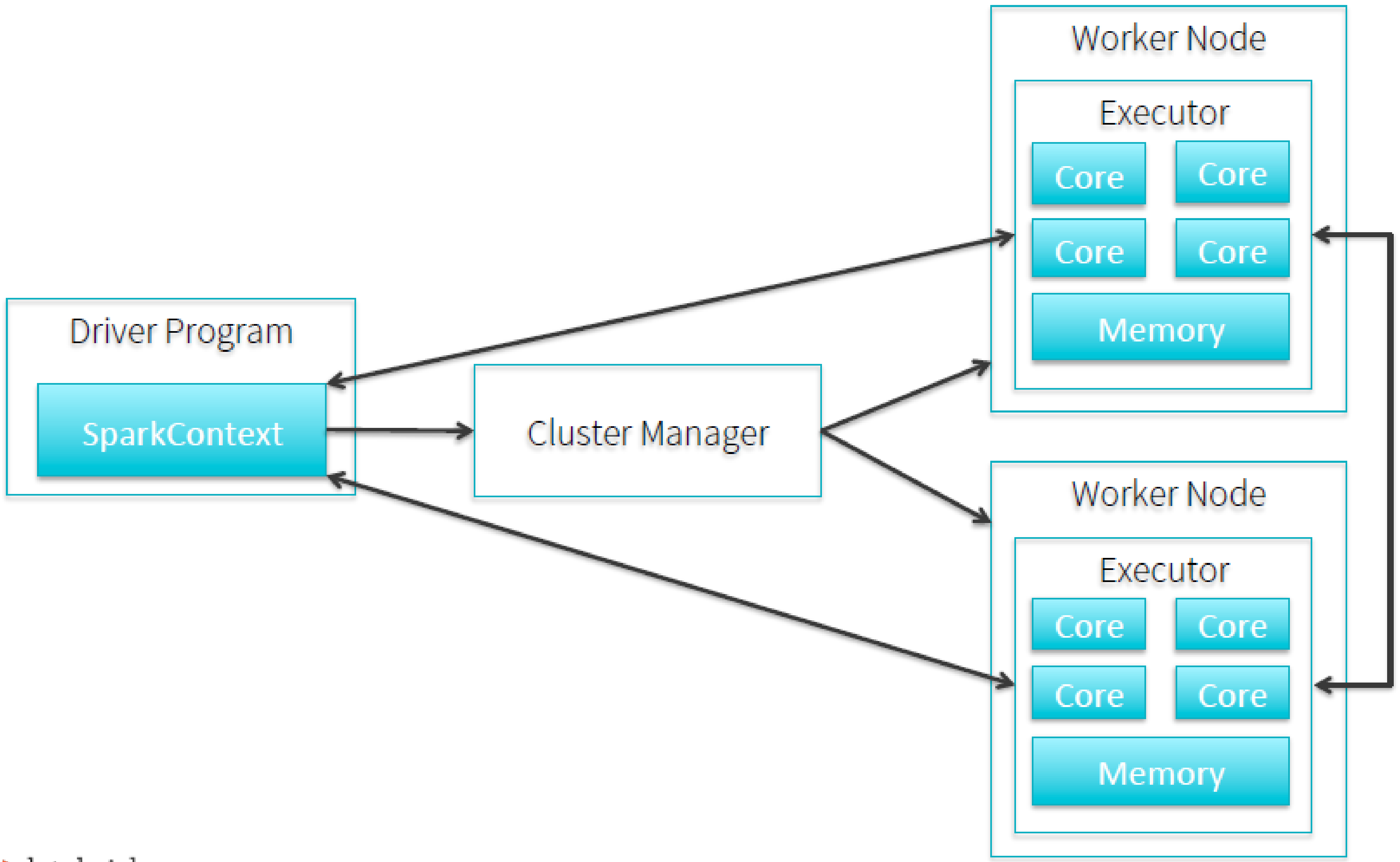
Executors:

- Executor run units of work called tasks
- It is set of cores or threads to run the tasks
- Executors are launched at per application basis and not shared by applications
- Process that run computations and store data for your application in worker node.
- Runs tasks and keeps data in memory or disk storage

Cores:

- Each executor runs a configurable number of threads called Cores
- Essentially, this is the number of slots available for running tasks

Spark - Cluster Architecture



Cluster Manager - Deployment Modes

- Spark can be deployed in 4 modes

1. Local Mode:

- Has no cluster manager
- Used for development , prototyping and debugging purposes
- Driver and Executor - run as threads in a single JVM

2. Standalone Mode:

- Spark's own cluster manager
- It has Spark Master which allocate Executors on Worker nodes
- Spark Worker which executes the task

3. YARN:

- Uses YARN's resource manager

4. Mesos:

- Spark uses Mesos cluster Manager
- Mesos - distributed OS

What is RDD

Resilient Distributed Datasets:

- ***Resilient*** because fault tolerant
- ***Distributed*** because distributed across the cluster
- ***Dataset*** because it holds data.

RDD Properties:

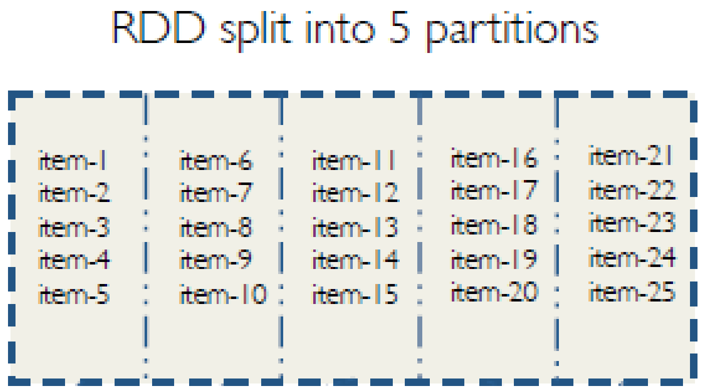
- Immutable
- Lazily Evaluated
- Cacheable

- Apache Spark lets you treat your input files almost like any other variable
- RDDs are automatically distributed across the network by means of Partitions.

Partition

- RDDs are divided into smaller chunks called Partitions
- These partitions of an RDD is distributed across all the nodes in the network.
- When you execute some action, a task is launched per partition
- More the number of partitions, the more the parallelism

- Spark automatically decides the number of partitions that an RDD has to be divided into but you can also specify the number of partitions when creating an RDD

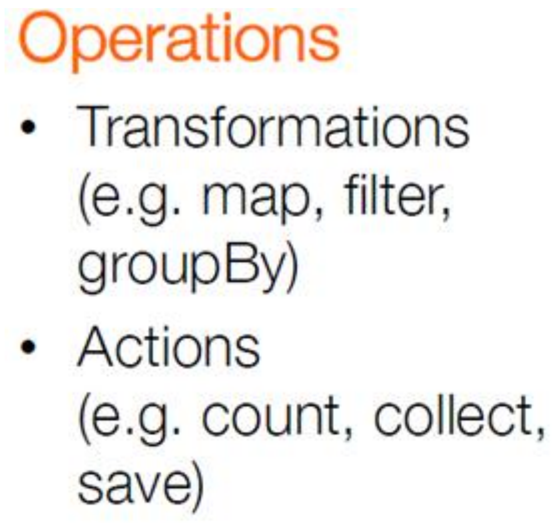


more partitions = more parallelism

RDD - Operations

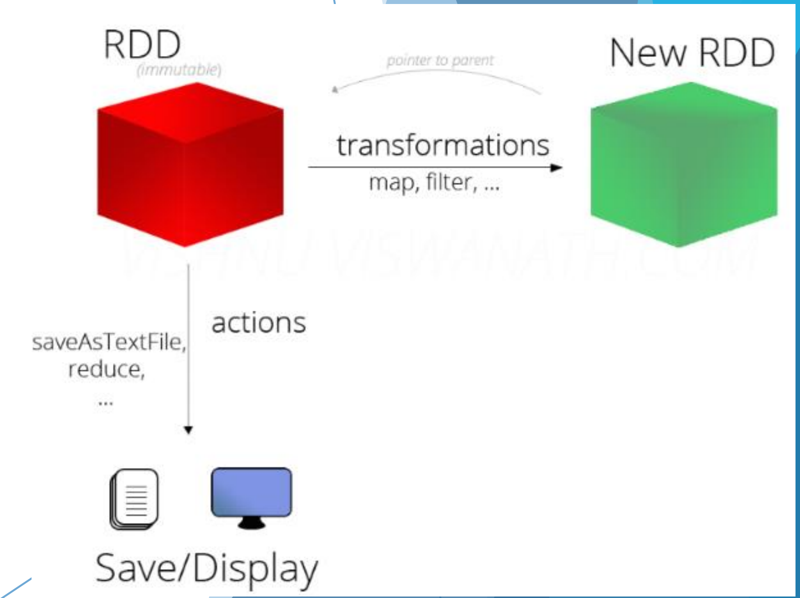
There are two types of operations that you can perform on an RDD

- Transformations
  - Transformation applies some function on a RDD and creates a new RDD
  - Data analytics operators are designed as transformations
  - Are evaluated lazily
  - Do not actually materialize data
  - They just construct the DAG
- Actions
  - Return a value to the application
  - Write data to stable storage
  - Force the DAG to be evaluated



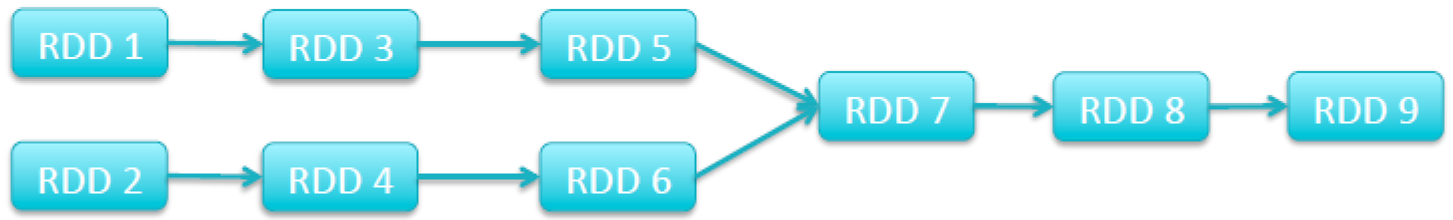
Operations

- Transformations (e.g. map, filter, groupBy)
- Actions (e.g. count, collect, save)



RDD – Lineage

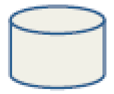
- A lineage keeps track of all the transformations to be applied on that RDD, including from where it has to read the data.
- It is Directed acyclic graph or DAG
- Series of transformations form DAG
- It helps in fault tolerance
- When an action is performed on a RDD, it executes it's entire lineage

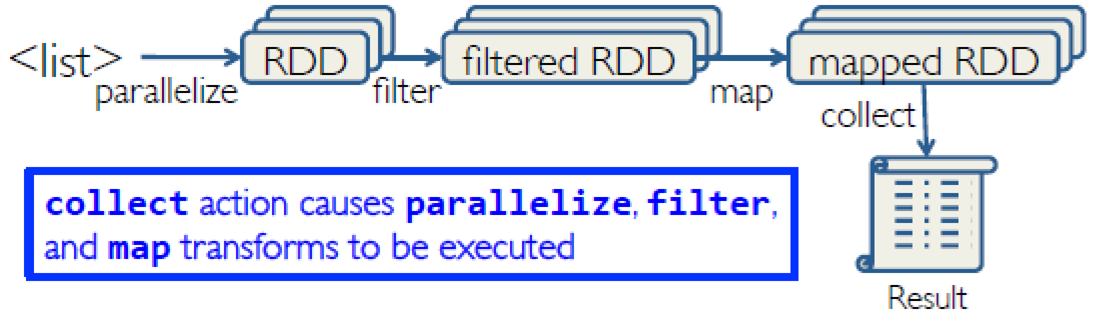


Lazy Evaluation:

- Affords opportunities for applying optimizations
- But runtime errors in transformations will not show up until an action is executed

RDD - Workflow

- Create an RDD from a data source:  <list>
- Apply transformations to an RDD: map filter
- Apply actions to an RDD: collect count



collect action causes **parallelize**, **filter**, and **map** transforms to be executed

- RDD can be created either from an external file or by parallelizing collections in your driver

```
val rdd = sc.textFile("/some_file",3)

val lines = sc.parallelize(List("this is","an example"))
```

RDD - Construction

RDD - Workflow

Read

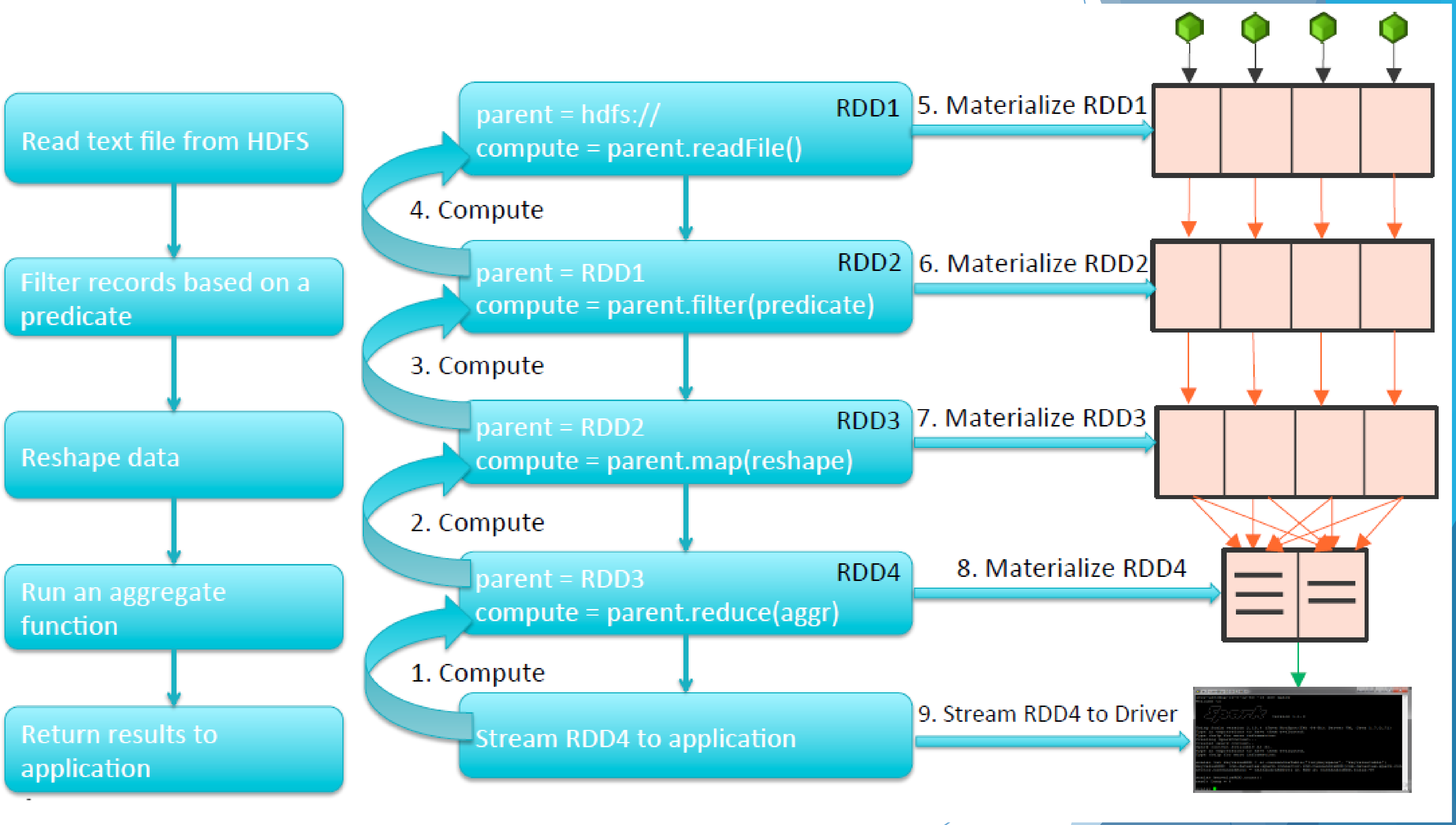
- Construct "Base" RDD by reading data from stable storage

Transform

- Apply a series of transformations constructing the Lineage DAG

Write (Call an Action)

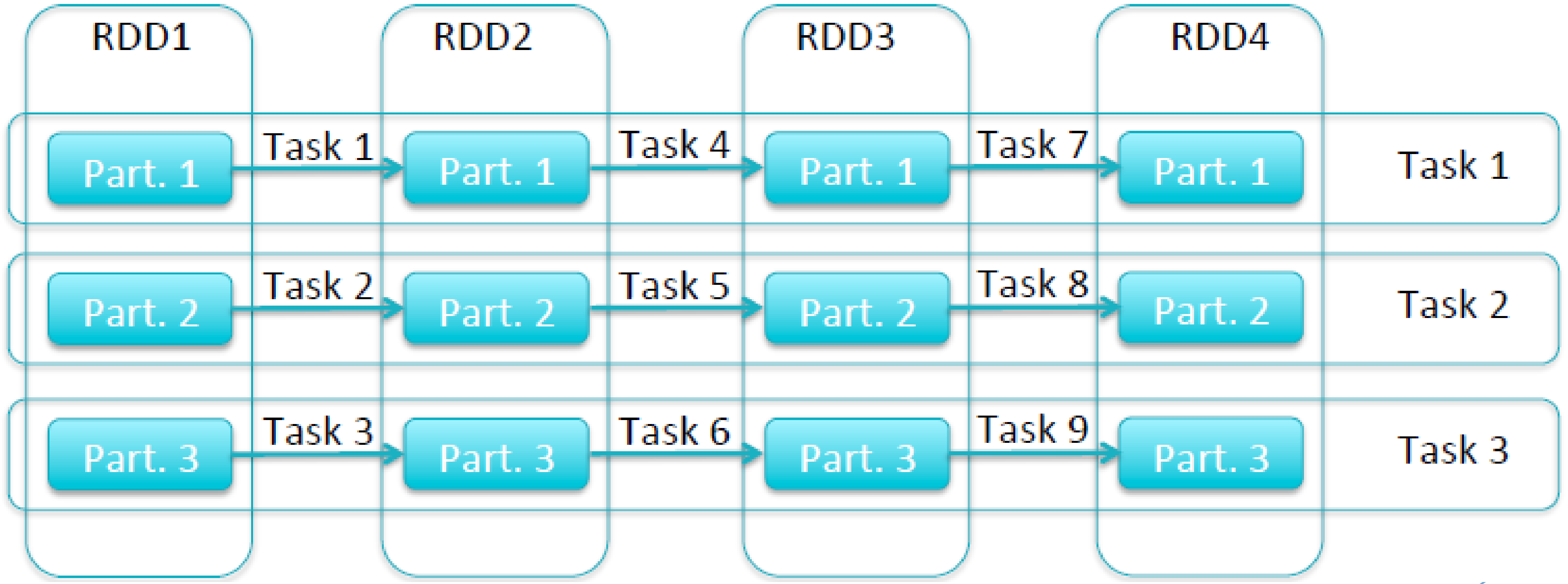
- Evaluate the DAG materializing the RDDs
- Write the result back to the application or to stable storage

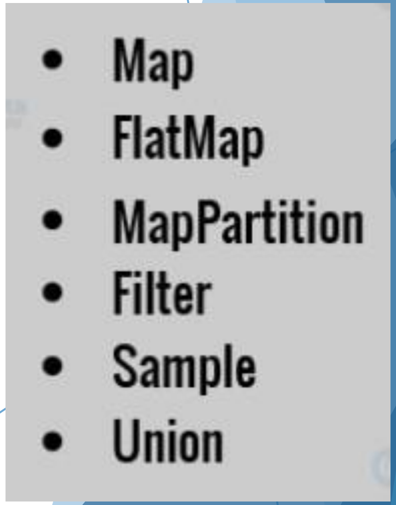


RDD data garbage collected once the Action is complete

RDD – Dependencies - Narrow

- Each parent partition contributes data to a single child partition
- Ex: A filter operator
- A sequence of operations involving narrow dependencies can be pipelined



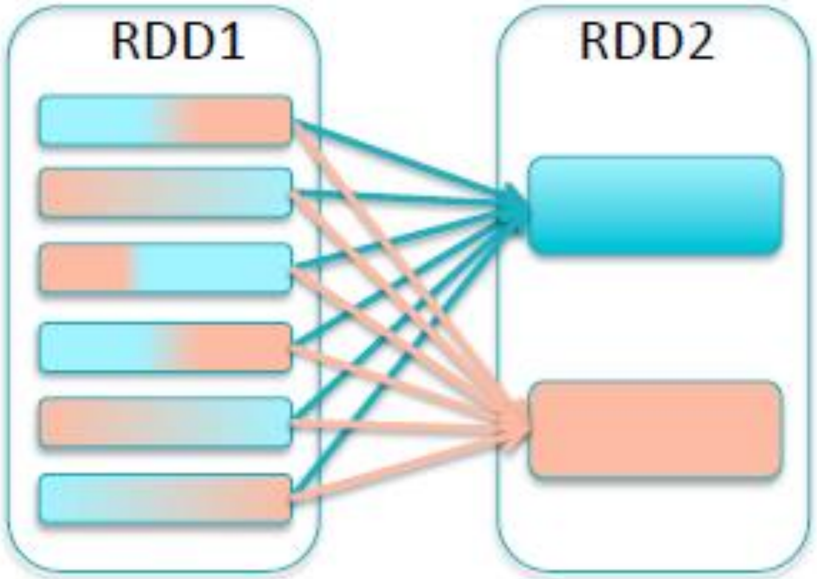


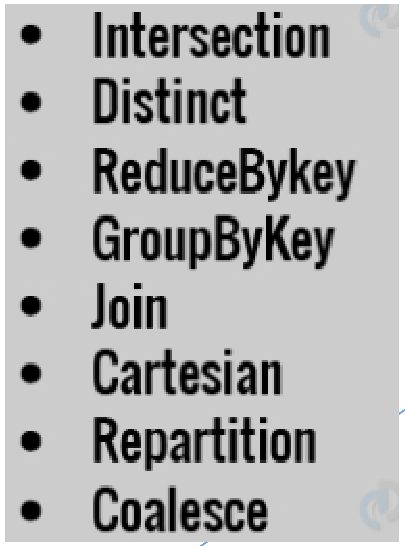
- Map
- FlatMap
- MapPartition
- Filter
- Sample
- Union

RDD – Dependencies - Wide

- Each parent partition contributes data to multiple child partitions
- Requires a *shuffle*
- Ex: Aggregation operators like group by

- An expensive operation in a distributed system
- Limits performance of the overall application





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

RDD - Caching

- RDDs in the lineage are garbage collected once the action is complete

- Expensive RDDs:
- RDDs created as a result of an expensive shuffle
- RDDs created after an expensive read and extensive filtering process

- For all such RDDs, it may be prudent to cache them
- RDD can be cached in memory by calling **rdd.cache()**.
- Caching stores the computed result of the RDD in the memory thereby eliminating the need to recompute it every time
- If there is not enough memory in the cluster, you can tell spark to use disk also for saving the RDD by using the method *persist()*.

- Caching can improve the performance of your application to a great extent
- Caches managed using an LRU algorithm

Memory Management

Persistence	description	
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM	`RDD.cache() == RDD.persist(MEMORY_ONLY)`
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM and spill to disk	`RDD.persist(MEMORY_AND_DISK)`
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition)	`RDD.persist(MEMORY_ONLY_SER)`
MEMORY_AND_DISK_SER	Spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed	`RDD.persist(MEMORY_AND_DISK_SER)`
DISK_ONLY	Store the RDD partitions only on disk	`RDD.persist(DISK_ONLY)`
MEMORY_ONLY_2, MEMORY_AND_DISK_2	Same as the levels above, but replicate each partition on two cluster nodes	`RDD.persist(MEMORY_ONLY_2)` `RDD.persist(MEMORY_AND_DISK_2)`
OFF_HEAP	Store RDD in serialized format in Tachyon	`RDD.persist(OFF_HEAP)`

RDD - Types

Base RDD:

- Created from SparkContext
  - textFile
  - newAPIHadoopFile
  - objectFile
  - sequenceFile
  - wholeTextFiles
  - range
  - Parallelize

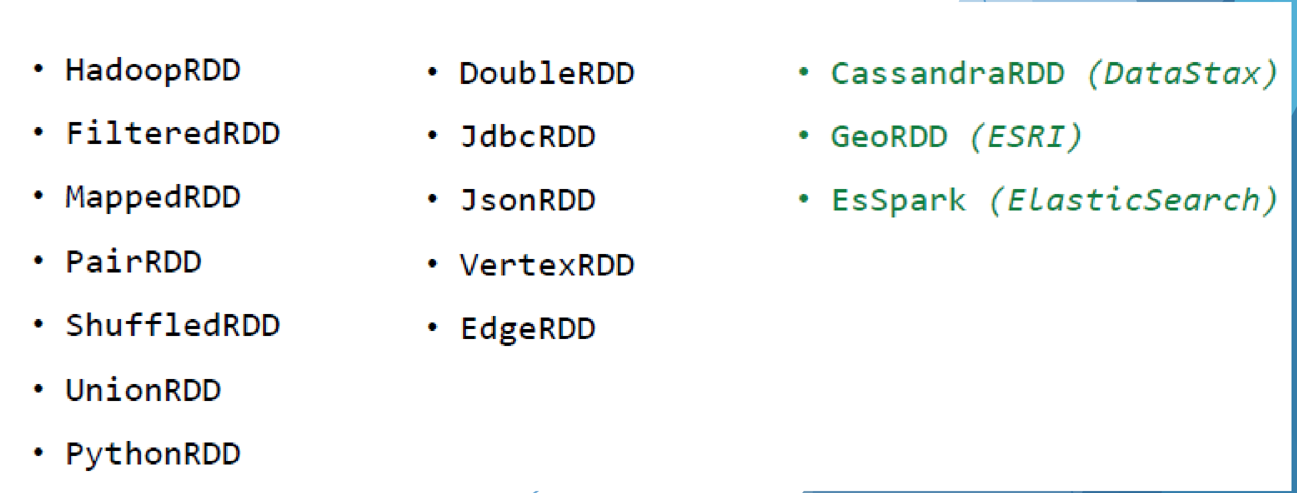
Derived RDD:

- Created by applying transformations on existing RDDs

- Paired RDD

Datasets are organized as key-value pairs

- Each data source that wants to control how data is partitioned typically requires a new type of RDD to be implemented



- HadoopRDD
- FilteredRDD
- MappedRDD
- PairRDD
- ShuffledRDD
- UnionRDD
- PythonRDD

- DoubleRDD
- JdbcRDD
- JsonRDD
- VertexRDD
- EdgeRDD

- CassandraRDD *(DataStax)*
- GeoRDD *(ESRI)*
- EsSpark *(ElasticSearch)*

Application Execution

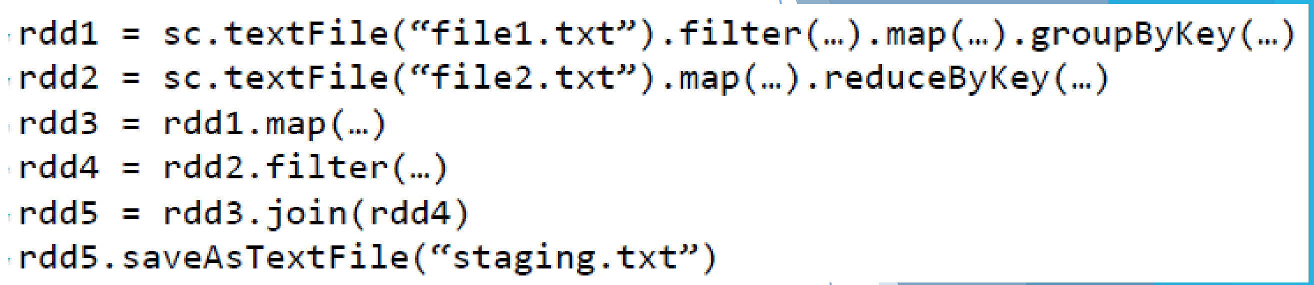
- Transformations build the DAG
- Actions result in execution of the DAG constructed

- But first we need to figure out an optimal execution plan
- Narrow dependencies can be pipelined together
- Wide dependencies mean that all partitions of the parent RDD(s) need to be computed before the execution can proceed

Location Awareness:

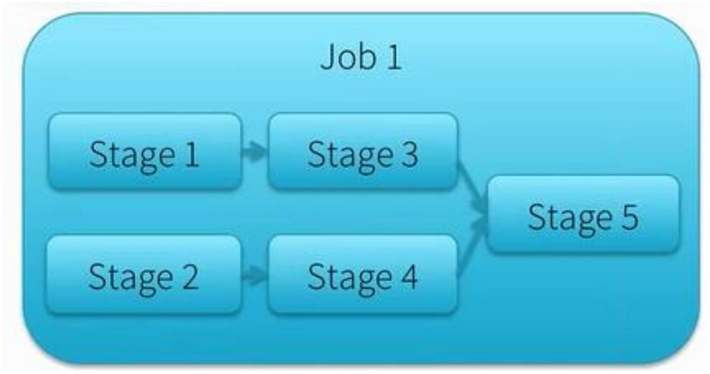
- For Base RDDs, the driver gets location information from data source
- For derived RDDs, the driver keeps track of which executors have which partitions

DAG Scheduling

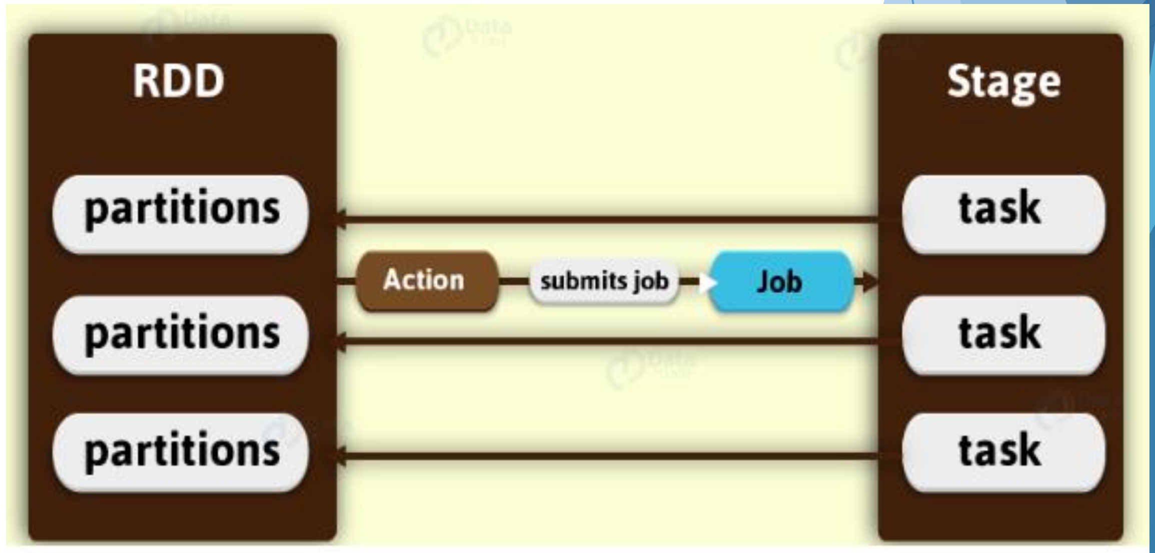


```
rdd1 = sc.textFile("file1.txt").filter(…).map(…).groupByKey(…)
rdd2 = sc.textFile("file2.txt").map(…).reduceByKey(…)
rdd3 = rdd1.map(…)
rdd4 = rdd2.filter(…)
rdd5 = rdd3.join(rdd4)
rdd5.saveAsTextFile("staging.txt")
```

- DAG construction and scheduling happens on the driver
- First stage will be created based on each operator in the DAG
- Once action is seen, job will be created and then run all the stages created so far
- Each action creates a spark job



- One stage may have many partitions
- So each stage will be divided into tasks based on no of partitions
- Cores are slots in which we run the tasks



RDD - Partitioning

RDD lineage information includes partitioning information as well

- Partitioner
- Number of Partitions

Partitioner:

- Partitioner determines how elements in RDD are distributed across a no of partitions
- Hash partitioner (Aggregate operation)
- Range partitioner (Sorting operation)
- Custom partitioners can be specified

- Transformations coalesce and repartition let programmers resize RDDs

RDD – Partitioning - Rules

Base RDD Rules:

RDDs created using parallelize:

- If user mentions the no of partitions, that is used
- If user does not mention, then spark.default.parallelism parameter (spark config parameter) will provide the no of partitions

RDDs created using Files:

- Partitioning depends on Data Source
- HDFS files – number of partitions = number of HDFS blocks

If the size is less than 128 MB, then it will create minimum of 2 partitions

- Parquet – number of partitions = number of part files

Derived RDD Rules:

- Partitions from parent RDDs are carried forward
- For transformations which creates shuffle takes argument called numpartitions where user can specify partitions
- If user didn’t set the partition, spark will check if parent is co-partitioned, then use that partition count. Otherwise, it will do a sum of all partitions

Fault Tolerance

- RDDs track lineage information that can be used to efficiently re-compute lost data

Executor failure:

- Driver asks the Cluster Manager to allocate new Executor
- Driver uses the DAG to recreate lost partitions on the new Executor

Driver failure:

- Driver failure can be catastrophic
- Run Driver using –supervise mode and monitor the driver

Cluster Manager failure:

- Failure of Cluster Manager components is handled differently based on cluster manger type
- In general worker node failure are handled by masters and master node failure is handled by stand by master (zoo keeper)

Spark-Submit

- Spark-submit shell script allows you to manage your Spark applications (execution, kill or request status)
- To run spark-submit, you need a compiled Spark application JAR.

./bin/spark-submit –help

- --driver-class-path command-line option sets the extra class path entries (e.g. jars and directories)
- --jars JARS   Comma-separated list of local jars to include on the driver and executor classpaths.
- --queue QUEUE_NAME   The YARN queue to submit to (Default: "default")
- --principal PRINCIPAL   Principal to be used to login to KDC
- --packages External Packages

- To kill the application

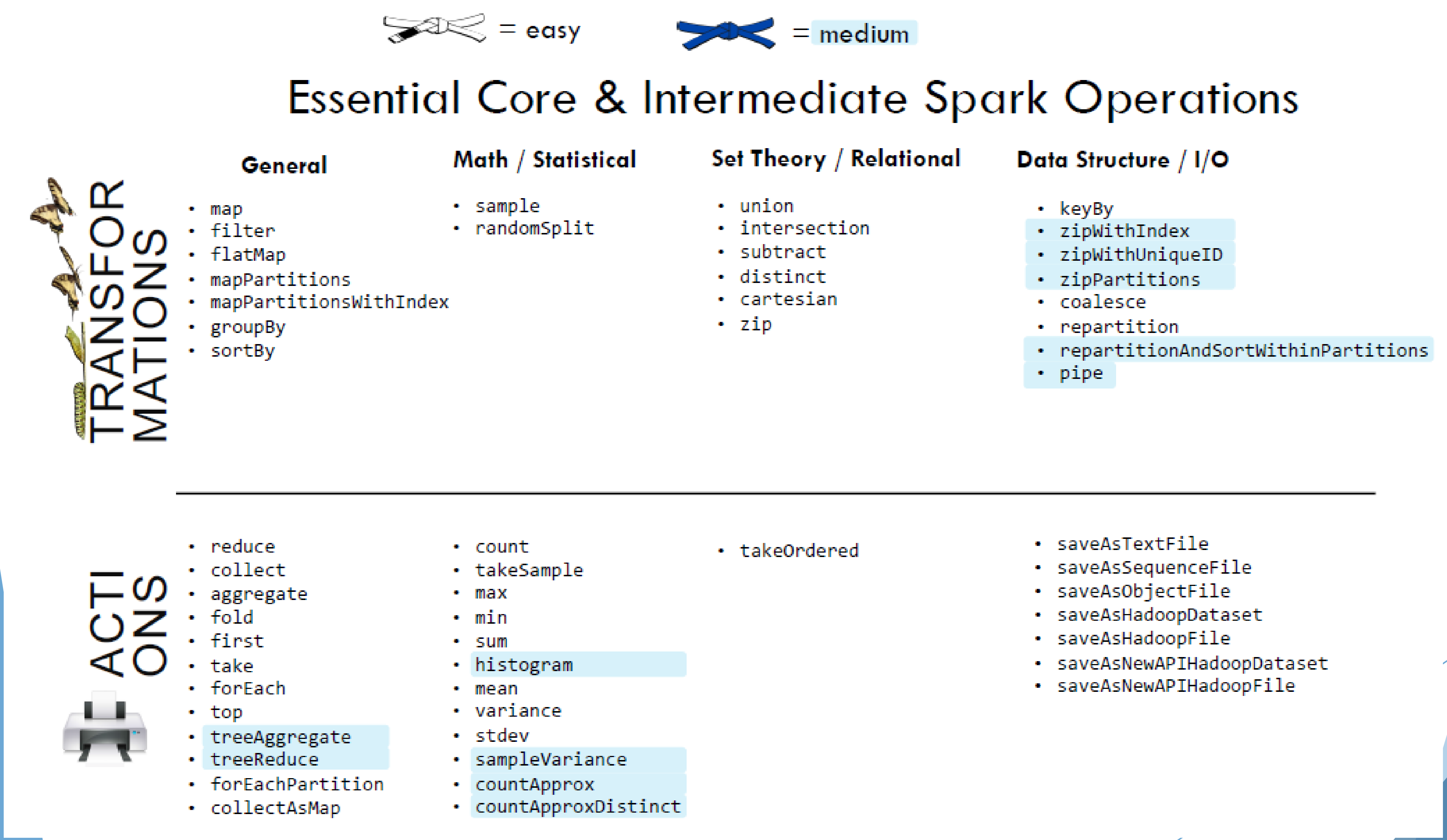
spark-submit --kill SUBMISSION_ID

- Requests the status

spark-submit --status SUBMISSION_ID

YOUR_SPARK_HOME/bin/spark-submit --class SparkWordCount --master local[*] target/scala-2.12/myfirstsparkproject.jar

RDD - Operations

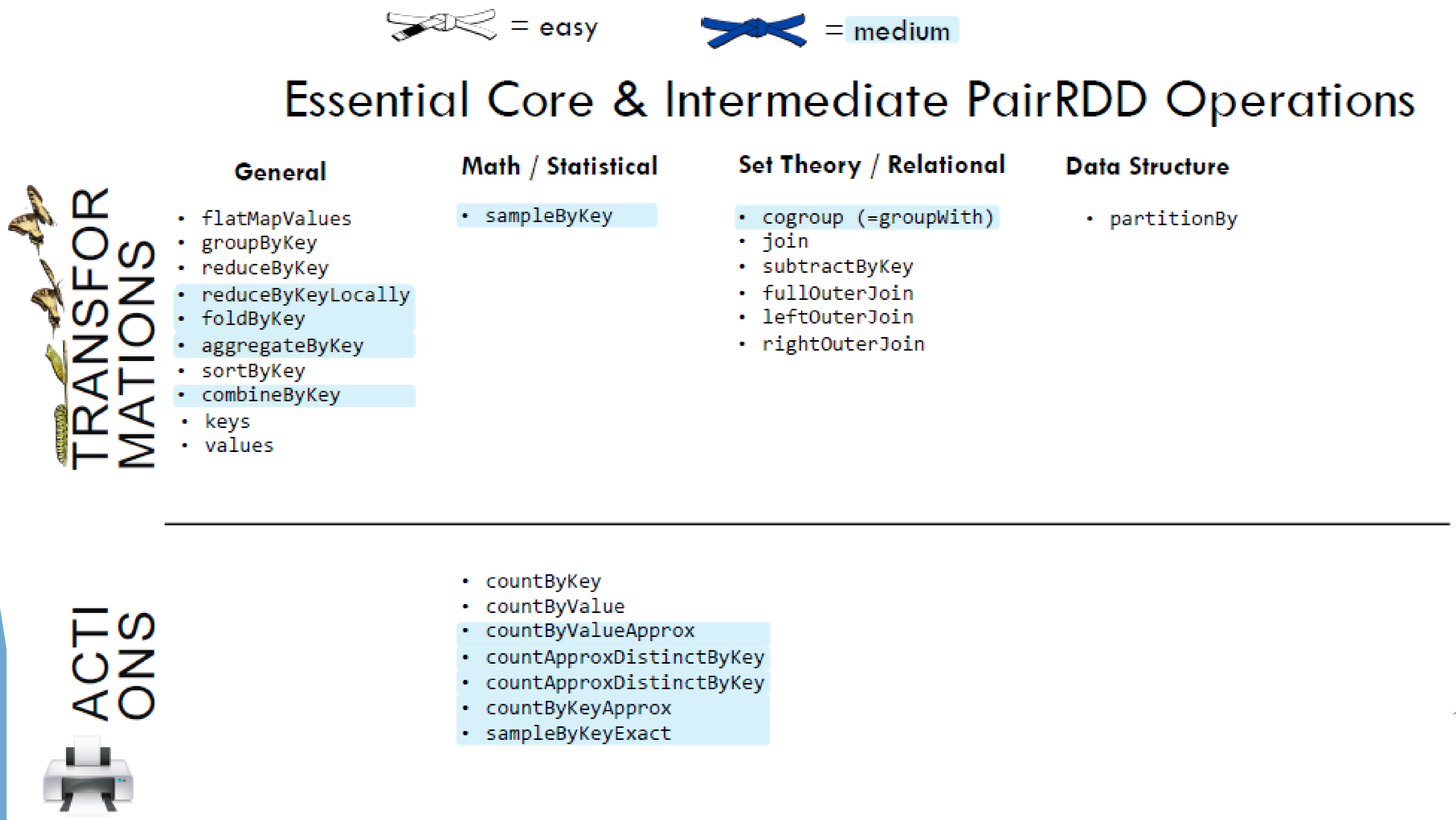


= easy = medium

Essential Core & Intermediate Spark Operations

	General	Math / Statistical	Set Theory / Relational	Data Structure / I/O
TRANSFORMATIONS	• map • filter • flatMap • mapPartitions • mapPartitionsWithIndex • groupBy • sortBy	• sample • randomSplit	• union • intersection • subtract • distinct • cartesian • zip	• keyBy • zipWithIndex • zipWithUniqueID • zipPartitions • coalesce • repartition • repartitionAndSortWithinPartitions • pipe
ACTIONS	• reduce • collect • aggregate • fold • first • take • forEach • top • treeAggregate • treeReduce • forEachPartition • collectAsMap	• count • takeSample • max • min • sum • histogram • mean • variance • stdev • sampleVariance • countApprox • countApproxDistinct	• takeOrdered	• saveAsTextFile • saveAsSequenceFile • saveAsObjectFile • saveAsHadoopDataset • saveAsHadoopFile • saveAsNewAPIHadoopDataset • saveAsNewAPIHadoopFile

Paired RDD - Operations



= easy

= medium

Essential Core & Intermediate PairRDD Operations

TRANSFOR
MATIONS

General

- flatMapValues
- groupByKey
- reduceByKey
- reduceByKeyLocally
- foldByKey
- aggregateByKey
- sortByKey
- combineByKey
- keys
- values

Math / Statistical

- sampleByKey

Set Theory / Relational

- cogroup (=groupWith)
- join
- subtractByKey
- fullOuterJoin
- leftOuterJoin
- rightOuterJoin

Data Structure

- partitionBy

ACTI
ONS

- countByKey
- countByValue
- countByValueApprox
- countApproxDistinctByKey
- countApproxDistinctByKey
- countByKeyApprox
- sampleByKeyExact

Spark REPL and Spark UI

- Bin directory of spark installation location has commands to invoke spark shell
- Spark context is created by the environment and it is made available in the variable called sc and it can be used to create base RDDs

- Python Shell:

pyspark

- Scala Shell:

spark-shell

- Spark R Shell:

sparkR

- To run a job

spark-submit

Ex: spark-submit --master yarn filename.jar

Spark UI has the below information:

- A list of scheduler stages and tasks
- A summary of RDD sizes and memory usage
- Environmental information.
- Information about the running executors
- http://localhost:4040

Reference Guide:

- http://spark.apache.org/docs/latest/programming-guide.html

Transformations

Transformations	Description	Example
Glom	It constructs array out of every single partition	val x = Array(1, 2, 3,4,5,6,7,8,9,10), 2) x.glom() **o/p: Array(1, 2, 3, 4, 5), Array(6, 7, 8, 9, 10)**
Map	It is used to reshape data and applies function to each element of RDD	val x = Array("b", "a", "c") x.map(z => (z, 1)) **o/p: Array((b,1), (a,1), (c,1))**
Filter	Filters out the element which satisfies the condition	val x = (1 to 20) x.filter(x => x % 2 == 1) **o/p: Array(1, 3, 5, 7, 9, 11, 13, 15, 17, 19)**
Flatmap	Apply function to each element of RDD and then flatten it	val x = Array("This is a single line") x.flatMap(x => x.split(" ")) **o/p: Array(This, is, a, single, line)**
Group By	• A function is passed on each element to create key and then it groups the key. • Creates paired RDD. • It involves a shuffle	val x = Array("John", "Fred", "Anna", "James") x.groupBy(w => w(0)) **o/p: Array((A,(Anna)), (J,(John, James)), (F,Fred)))**

Transformations…

Transformations	Description	Example
GroupByKey	• It requires pair RDD • It does not need any function to create a key • It by default groups by key	val x = Array(('B', 5), ('B', 4), ('A', 3), ('A', 2),('A', 1)) x.groupByKey() **o/p: Array((A,(3, 2, 1)), (B,(5, 4)))**
Union	Returns new RDD containing all items from 2 original RDDs	val x = Array(1, 2, 3) val y = Array(3, 4) x.union(y) **o/p: Array(1, 2, 3, 3, 4)**
Join	It returns a new RDD with all pairs of elements having the same key in the original RDDs	val x = Array(("a", 1), ("b", 2)) val y = Array(("a", 3), ("a", 4), ("b", 5)) x.join(y) **o/p: Array((a,(1,3)), (a,(1,4)), (b,(2,5)))**
Distinct	Returns new RDD containing distinct items from the original RDD	val x = Array(1, 2, 3, 3, 4) x.distinct() **o/p: Array(4, 1, 2, 3)**

Transformations…

Transformations	Description	Example
KeyBy	Creates paired RDD from normal values based on function passed in to it	val x = Array("John", "Fred", "Anna", "James") x.keyBy(s => s(0)) **o/p: Array((J,John), (F,Fred), (A,Anna), (J,James))**
Zip	Creates new RDD where key from first RDD and values from second RDD	val x = Array(1, 2, 3)) val y = x.map(n => n * n) x.zip(y) **o/p: Array((1,1), (2,4), (3,9))**
Partition controlled transformation		
Coalesce	• Return a new RDD which is reduced to smaller no of partitions • To handle data skew	x.coalesce(2) Originally if there are 3 partitions, now it will be reduced to 2 partitions
Repartition	To increase the no of partitions	x.repartition(10) Originally if there are 3 partitions, now it will be reduced to 10 partitions

Actions

Actions	Description	Example
Partitions	Gives the total no of partitions of RDD	val x = sc.parallelize(Array(1, 2, 3), 2) x.partitions.length **o/p: Int = 2**
Reduce	Aggregate all the elements of RDD by applying user function to the elements	val x = Array(1,1,1,1,1) x.reduce { (a, b) => a + b } // or: x.reduce(_ + _) **o/p: Int = 5** a is supplied by system that is accumulator and b is each element in RDD. First a=0 , then a=0, b=1 ; a+b = 1; a=1, b=1; a+b=2…
Aggregate	Aggregate all the elements of RDD by applying user function to combine elements and combine these results in 2nd user function and finally return the result to the driver	

Actions

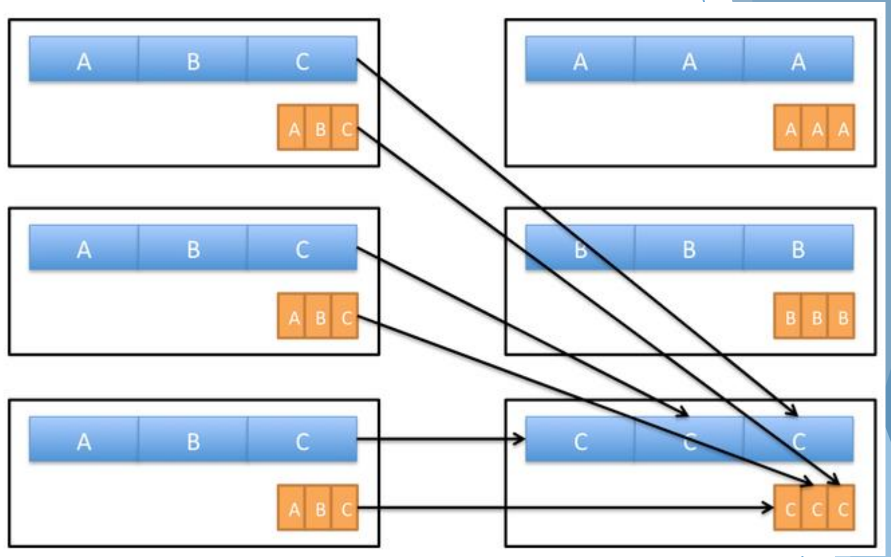
Actions	Description	Example
Count	Returns the number of elements in RDD	val data = spark.read.textFile("spark_test.txt").rdd val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark") println(mapFile.count())
Top	If ordering is present in our RDD, then we can extract top elements from our RDD using top()	val data = spark.read.textFile("spark_test.txt").rdd val mapFile = data.map(line => (line,line.length)) val res = mapFile.top(3) res.foreach(println)
Foreach	When we have a situation where we want to apply operation on each element of RDD, but it should not return value to the driver, foreach is used	val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3) val group = data.groupByKey().collect() group.foreach(println)

Actions…

Actions	Description	Example
Max, Min, Avg, Mean	Compute respective function in RDD	val x = Array(2, 4, 1) x.max() **o/p: Int = 4**
CountByKey	Returns set of keys and count of their occurences	val x = Array(('J', "James"), ('F', "Fred"), ('A', "Anna"), ('J', "John")) x.countByKey() **o/p: Map(A -> 1, J -> 2, F -> 1)**
saveAsTextFile	Save RDD to filesytem in the mentioned path	val x = Array(1,2,3,4,5) x.saveAsTextFile(/tmp/Output) File will be saved in output folder
Collect	Returns all the items in the RDD to the driver in a single list	val x = Array(1,2,3,4,5) x.Collect() **o/p: Array(1, 2, 3, 4, 5)**
Take	Returns an array with first n elements	val x = Array(1,2,3,4,5) x.take(2) **o/p: Array(1, 2)**

Hash Partitioning

- Hash Partitioning attempts to spread the data evenly across various partitions based on the key.
- Uses Java's Object.hashCodemethod to determine the partition as partition = key.hashCode() % numPartitions.
- When we are using Hash-partition the data will be shuffled and all same key data will shuffle at same worker

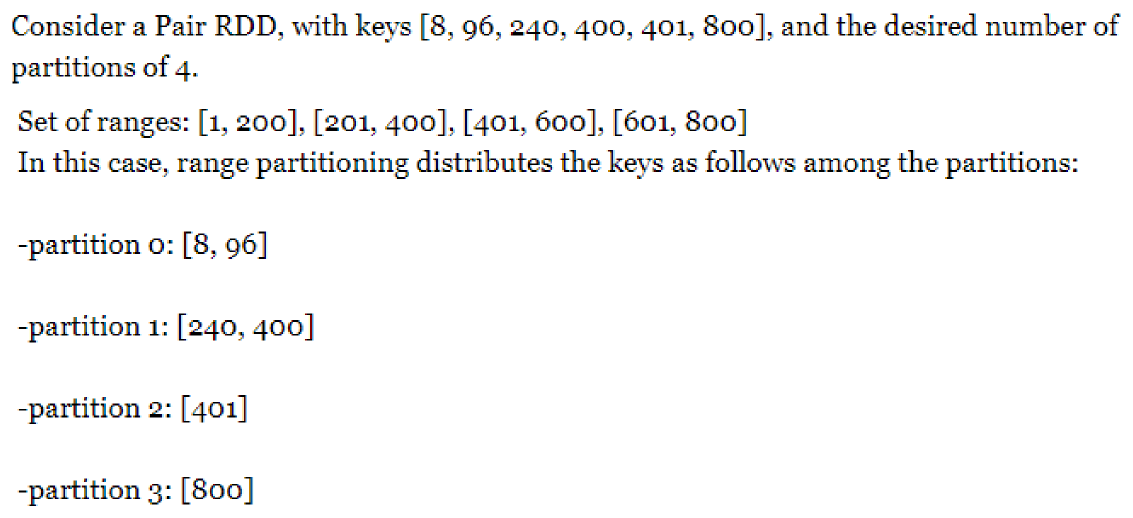


```
val data = sc.parallelize(List((2, 3),(5,6),(3, 6), (4,3),(2, 4),(3, 7),(3,8),(4,1)),4)
val hashpartdata = data.partitionBy(new HashPartitioner(2))
```

Range Partitioning

- RangePartitioner will sort the records based on the key and then it will divide the records into a number of partitions based on the given value.
- Uses a range to distribute to the respective partitions the keys that fall within a range.
- This method is suitable where there’s a natural ordering in the keys

```
val data = sc.parallelize(List((2, 3),(5,6),(3, 6), (4,3),(2, 4),(3, 7),(3,8),(4,1)),4)
val rangepartdata = data.partitionBy(new RangePartitioner(2,data))
```



Consider a Pair RDD, with keys [8, 96, 240, 400, 401, 800], and the desired number of partitions of 4.

Set of ranges: [1, 200], [201, 400], [401, 600], [601, 800]
In this case, range partitioning distributes the keys as follows among the partitions:

-partition 0: [8, 96]

-partition 1: [240, 400]

-partition 2: [401]

-partition 3: [800]

Hash Partition vs Range Partition

```
val data = sc.parallelize(List((2, 3),(5,6),(3, 6), (4,3),(2, 4),(3, 7),(3,8),(4,1)),4)
data.glom().collect
```

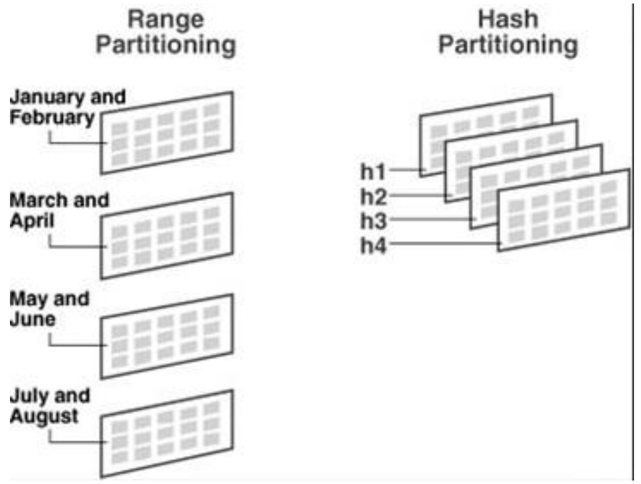
```
Array[Array[(Int, Int)]] = Array(Array((2,3), (5,6)), Array((3,6), (4,3)), Array((2,4), (3,7)), Array((3,8), (4,1)))
```

```
val hashpartdata = data.partitionBy(new HashPartitioner(2))
hashpartdata.glom().collect
```

```
Array[Array[(Int, Int)]] = Array(Array((2,3), (4,3), (2,4), (4,1)), Array((5,6), (3,6), (3,7), (3,8)))
```

```
val rangepartdata = data.partitionBy(new RangePartitioner(2,data))
rangepartdata.glom().collect
```

```
Array[Array[(Int, Int)]] = Array(Array((2,3), (3,6), (2,4), (3,7), (3,8)), Array((5,6), (4,3), (4,1)))
```



Coalesce vs Repartition

Repartition():

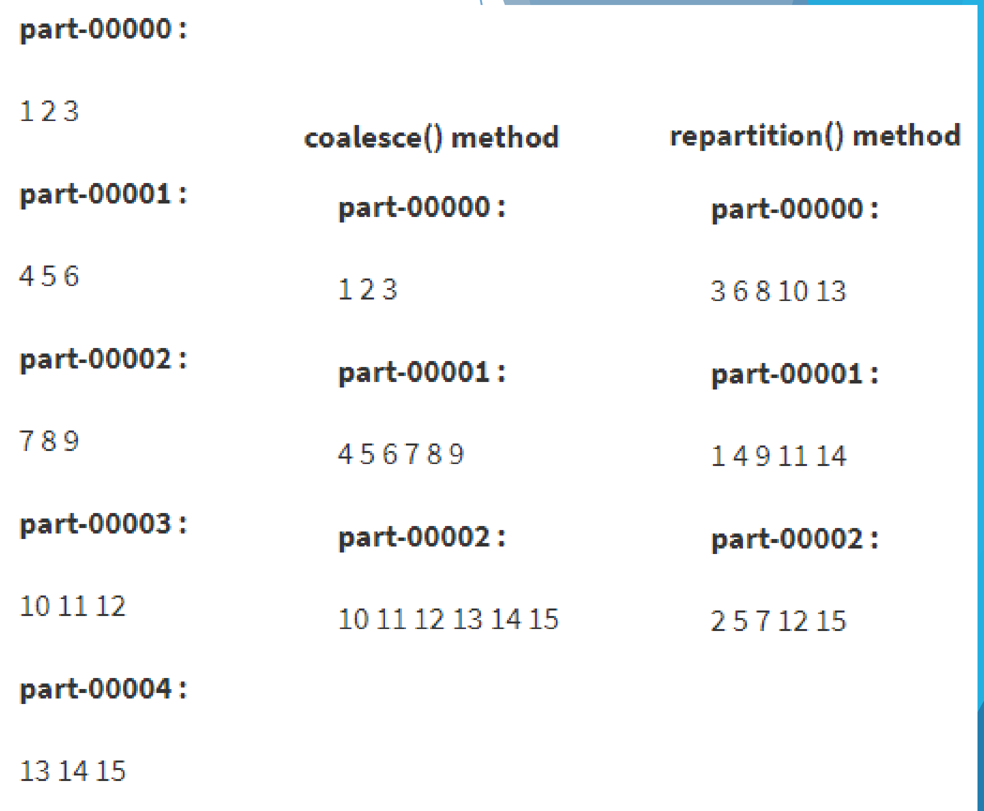
- The number of partitions can be increased/decreased
- repartition creates new partitions and does a full shuffle

val repartitionRDD = rdd.repartition(3)

Coalesce()

- The number of partitions can only be decreased.
- It avoids a full shuffle
- coalesce uses existing partitions to minimize the amount of data that's shuffled

val coalesceRDD = rdd.coalesce(3)



part-00000 :

1 2 3

part-00001 :

4 5 6

part-00002 :

7 8 9

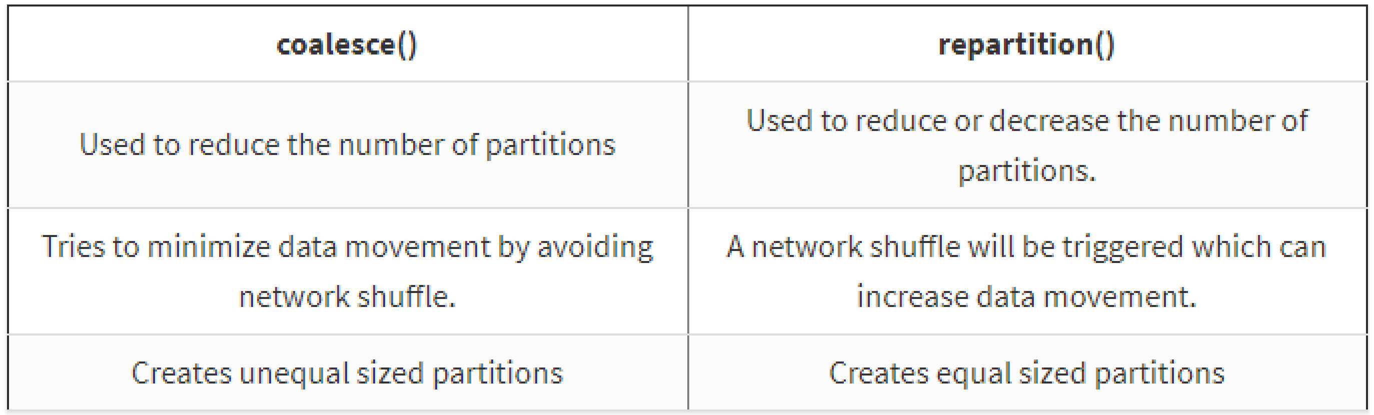
part-00003 :

10 11 12

part-00004 :

13 14 15

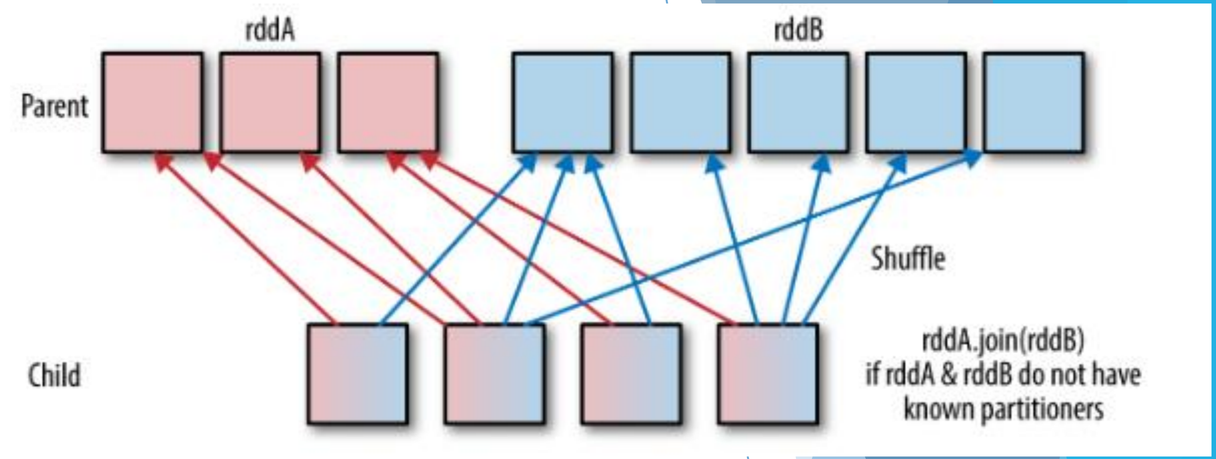
coalesce() method	**repartition() method**
part-00000 :	**part-00000 :**
1 2 3	3 6 8 10 13
part-00001 :	**part-00001 :**
4 5 6 7 8 9	1 4 9 11 14
part-00002 :	**part-00002 :**
10 11 12 13 14 15	2 5 7 12 15

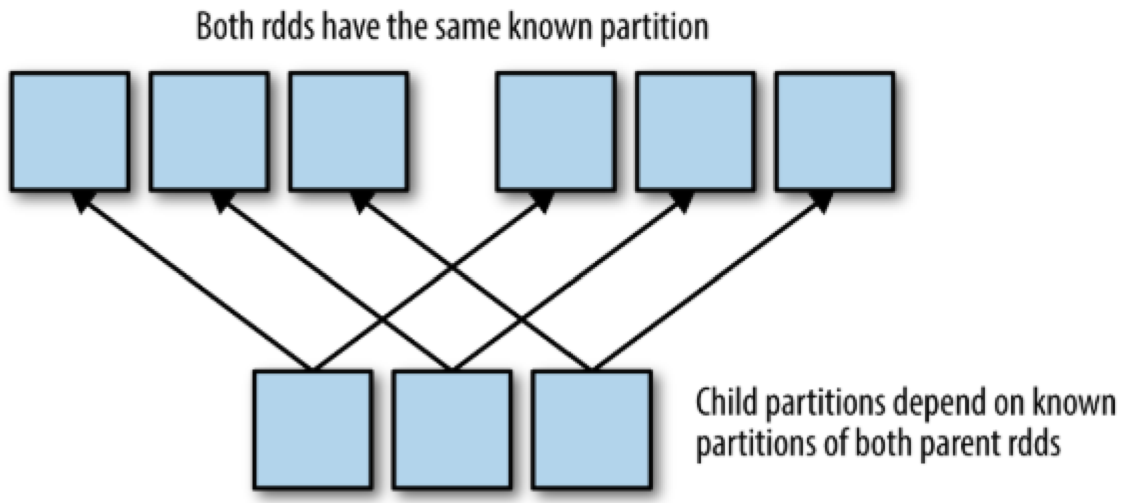


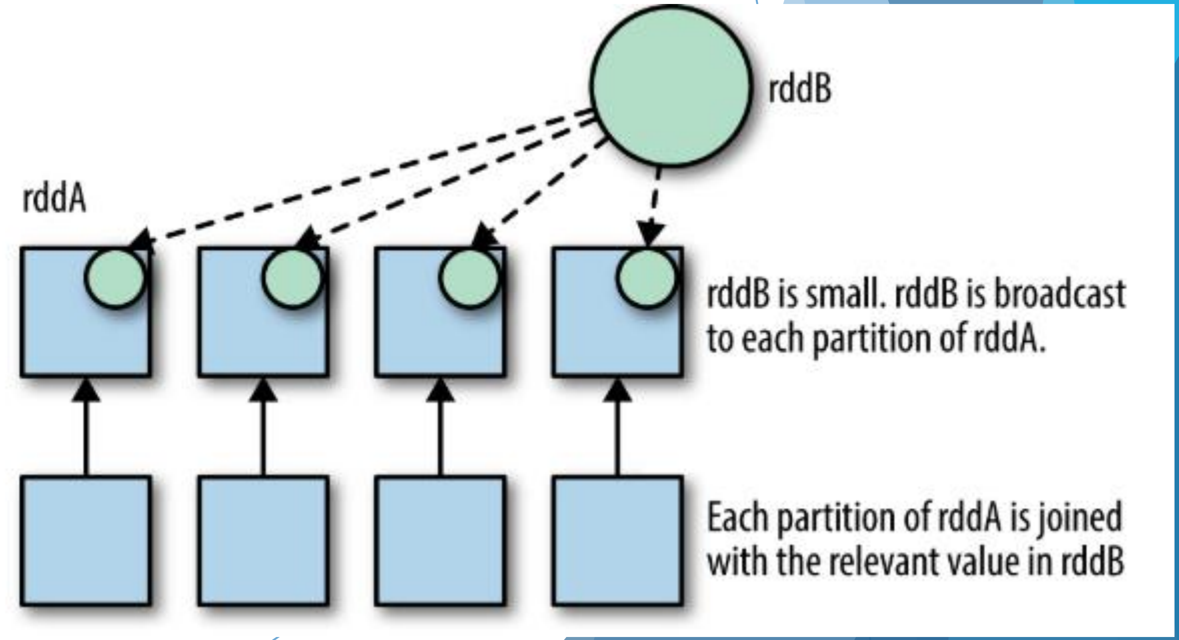
coalesce()	**repartition()**
Used to reduce the number of partitions	Used to reduce or decrease the number of partitions.
Tries to minimize data movement by avoiding network shuffle.	A network shuffle will be triggered which can increase data movement.
Creates unequal sized partitions	Creates equal sized partitions

Join, Co-Partition & Broadcast-Join

- Joins in general are expensive since they require that corresponding keys from each RDD are located at the same partition so that they can be combined locally
- Co-partitioned RDD's uses same partitioner and thus have their data distributed across partitions in same manner.
- A broadcast hash join pushes one of the RDDs (the smaller one) to each of the worker nodes. Then it does a map-side combine with each partition of the larger RDD.



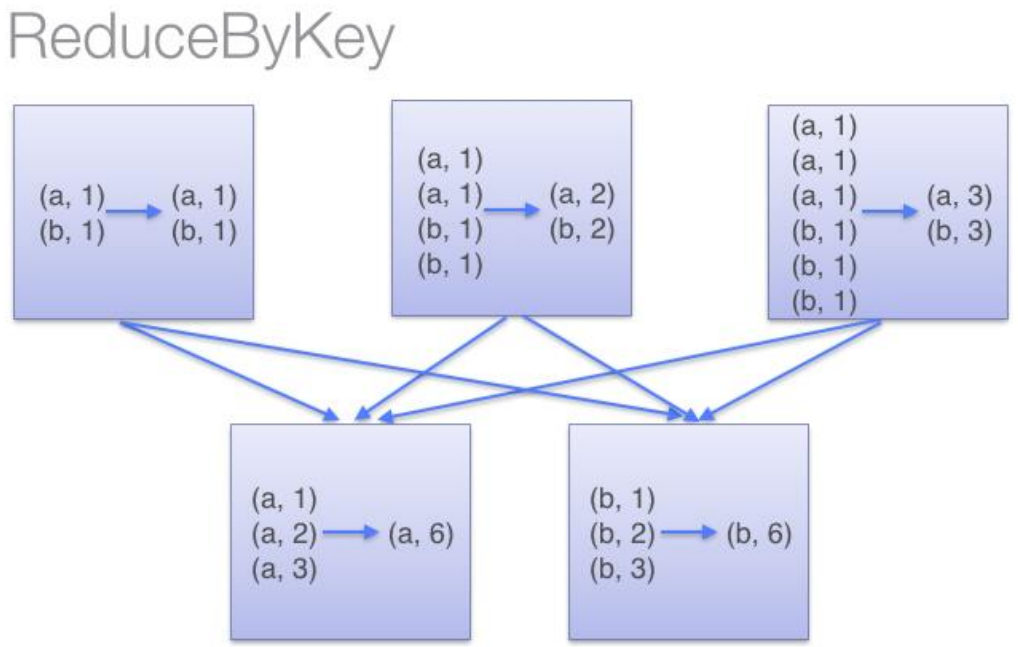

Both rdds have the same known partition
Child partitions depend on known
partitions of both parent rdds

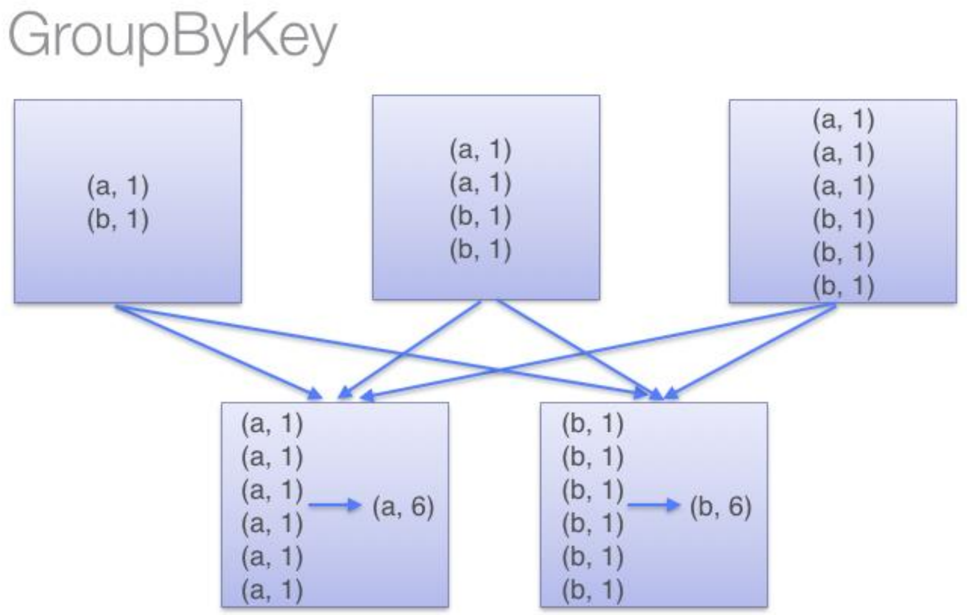

rddB
rddA
rddB is small. rddB is broadcast
to each partition of rddA.
Each partition of rddA is joined
with the relevant value in rddB

GroupByKey vs ReduceByKey

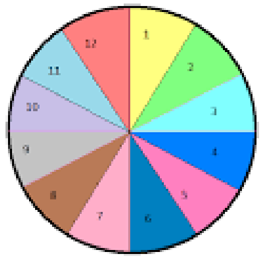
- While both of these functions will produce the correct answer, the reduceByKey example works much better on a large dataset.
- In reduceByKey, Spark combine the output with common key on each partition before shuffling the data.
- In groupByKey , all the key-value pairs are shuffled around.

```
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))

val wordCountsWithReduce = wordPairsRDD
  .reduceByKey(_ + _)
  .collect()

val wordCountsWithGroup = wordPairsRDD
  .groupByKey()
  .map(t => (t._1, t._2.sum))
  .collect()
```


ReduceByKey
(a, 1) (b, 1) → (a, 1) (b, 1)
(a, 1) (a, 1) (b, 1) (b, 1) → (a, 2) (b, 2)
(a, 1) (a, 1) (a, 1) (b, 1) (b, 1) (b, 1) → (a, 3) (b, 3)
(a, 1) (a, 2) (a, 3) → (a, 6)
(b, 1) (b, 2) (b, 3) → (b, 6)


GroupByKey
(a, 1) (b, 1)
(a, 1) (a, 1) (b, 1) (b, 1)
(a, 1) (a, 1) (a, 1) (b, 1) (b, 1) (b, 1)
(a, 1) (a, 1) (a, 1) (a, 1) (a, 1) (a, 1) → (a, 6)
(b, 1) (b, 1) (b, 1) (b, 1) (b, 1) (b, 1) → (b, 6)

Performance Tuning Tips



1. Set partition numbers properly

- The default level of parallelism (2 by default).
- Can be set it in the code or use the spark.default.parallelism parameter.

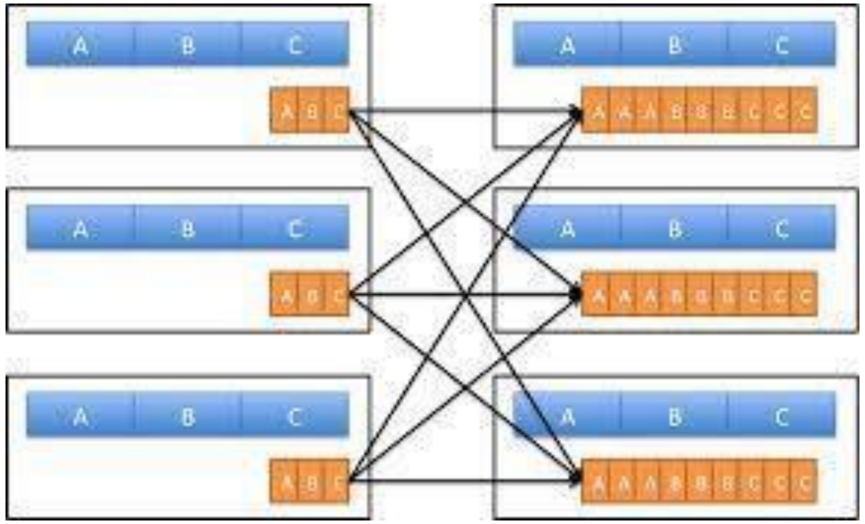
When specifying partitions,

- Bigger is not always better. More partitions mean more tasks processing data in parallel.
- Each task has its own overhead, including those relating to communication and data. Submitting too many tasks causes multiple TaskStarted and TaskEnded messages to be sent and handled in the messaging channel, and multiple events to be written to the event log
- Both factors slow down driver performance. Task metrics also consume memory when the Spark UI is rebuilt and might sometimes crash the driver program if a Java OutOfMemory exception occurs.

Performance Tuning Tips…

2. Be aware of data shuffling

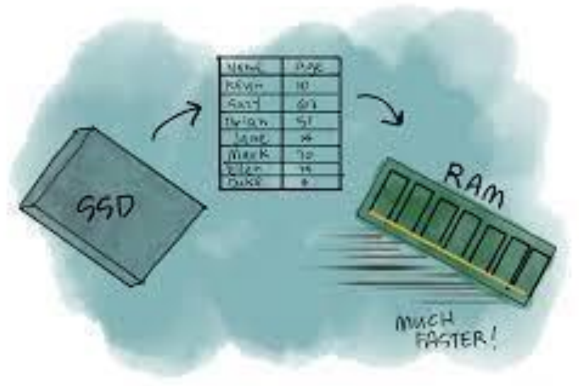
- For most repartition jobs, such as groupByKey and join, a ShuffleDependency is created, which causes data to be shuffled.
- However, if the partitioner of all the RDDs involved is the same, the Spark scheduler uses a OneToOneDependency instead. In this case,tasks can be pipelined and data is not shuffled.
- Explicitly setting a partitioner for your RDDs might help reduce data shuffles.



Performance Tuning Tips…

3. Make the best of data caching

- Chained Programming Conventions can be used to make the code shorter and simpler.
- However, if some parts of the chain are to be reused and its computation cost is high, this part of data must be cached by invoking the "cache()"or "persist(storageLevel)" actions.
- By calling "cache()", you actually call "persist(MEMORY_ONLY)" and data is cached to memory.
- Other storage levels also we discussed earlier.
- Need to know when/what to cache.



Hands On

Thank You

Keerthiga Barathan