RDD Actions

Saurav Agarwal

ACTIONS on RDD

Actions are RDD operations that produce non-RDD values

In other words, a RDD operation that returns a value of any type except RDD[T] is an action

They trigger execution of RDD transformations to return values

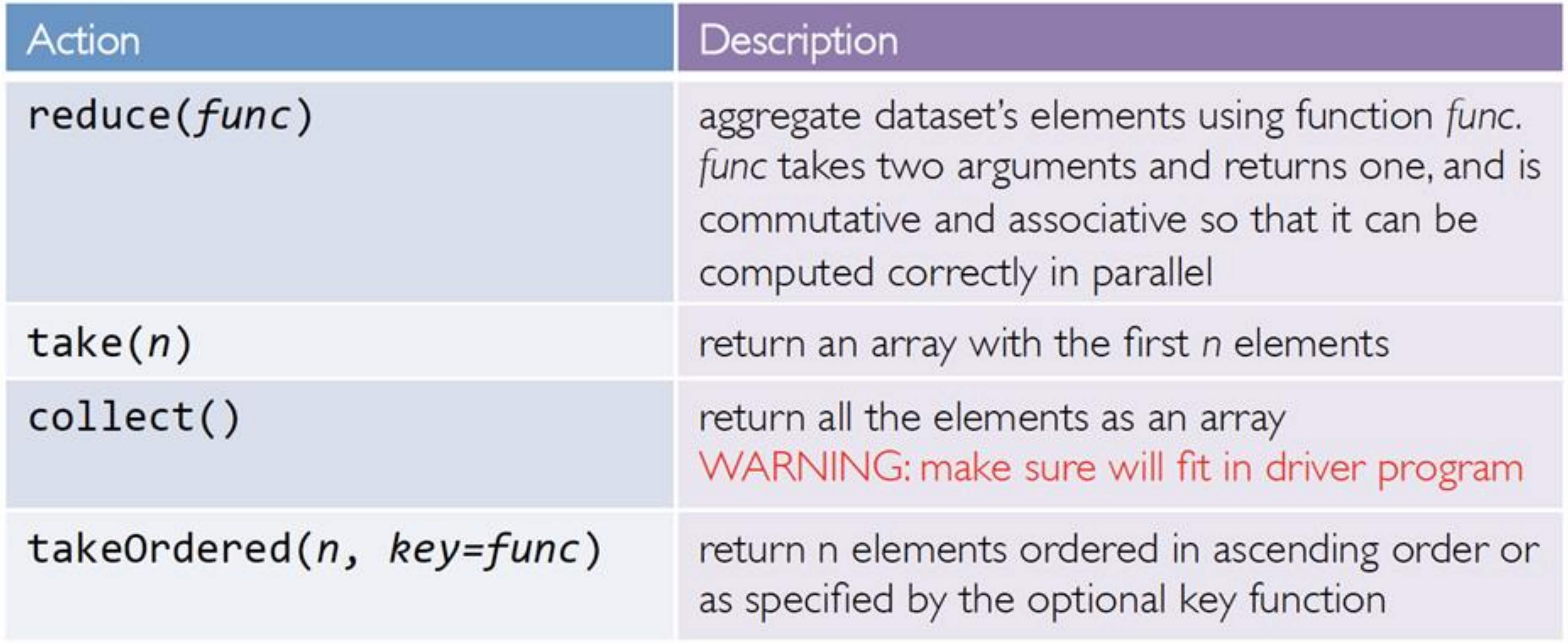
Simply put, an action evaluates the RDD lineage graph

You can think of actions as a valve and until action is fired, the data to be processed is not even in the pipes, transformations

Only actions can materialize the entire processing pipeline with real data
Actions are one of two ways to send data from executors to the driver (the other being accumulators)

Saurav Agarwal

Actions



Action	Description
`reduce(`*`func`*`)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(`*`n`*`)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(`*`n, key=func`*`)`	return n elements ordered in ascending order or as specified by the optional key function

Getting Data Out of RDDs

```
>>> rdd=sc.parallelize([1,2,3])
>>> rdd.reduce(lambda a,b:a*b)

>>> rdd.take(2)
>>> rdd.collect()

>>>rdd =sc.parallelize([5,3,1,2])
>>>rdd.takeOrdered(3,lambda s: -1 * s)
```

```
lines=sc.textFile("...",4)
print lines.count()
```

count() causes Spark to: read data , sum within partitions, combine sums in driver

saveAsTextFile(path) - Save this RDD as a text file, using string representations of elements.

Saurav Agarwal

RDD Transformations

Saurav Agarwal

Transformations of RDD

Transformations are lazy operations on a RDD that create one or many new RDDs, e.g. map, filter, reduceByKey, join, cogroup, randomSplit

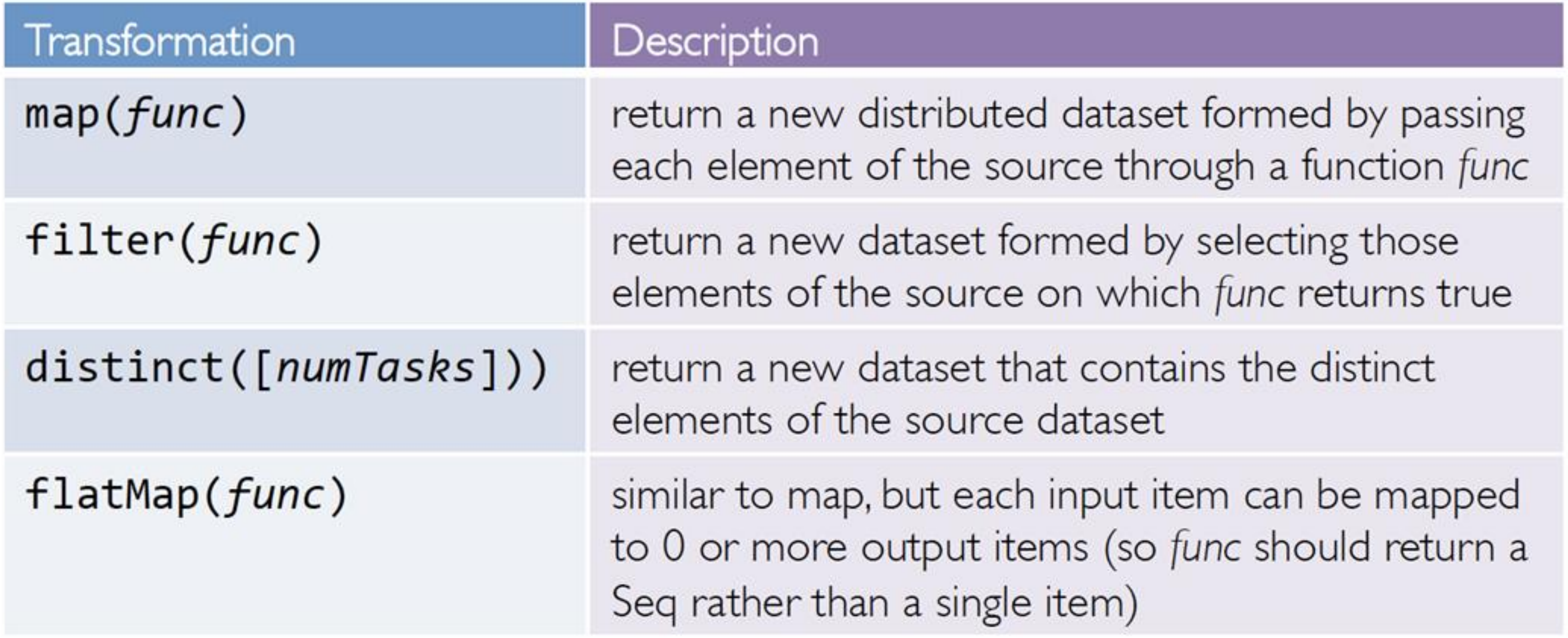
They are functions that take a RDD as the input and produce one or many RDDs as the output They do not change the input RDD (since RDDs are immutable), but always produce one or more new RDDs by applying the computations they represent

Transformations are lazy,they are not executed immediately but only after calling an action are transformations executed

After executing a transformation, the result RDD(s) will always be different from their parents and can be smaller (e.g. filter, count, distinct, sample), bigger (e.g flatMap, union, cartesian) or the same size (e.g. map)

Saurav Agarwal

RDD Spark Transformations



Transformation	Description
`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
`distinct([`*numTasks*`]))`	return a new dataset that contains the distinct elements of the source dataset
`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)

Saurav Agarwal

Transformations

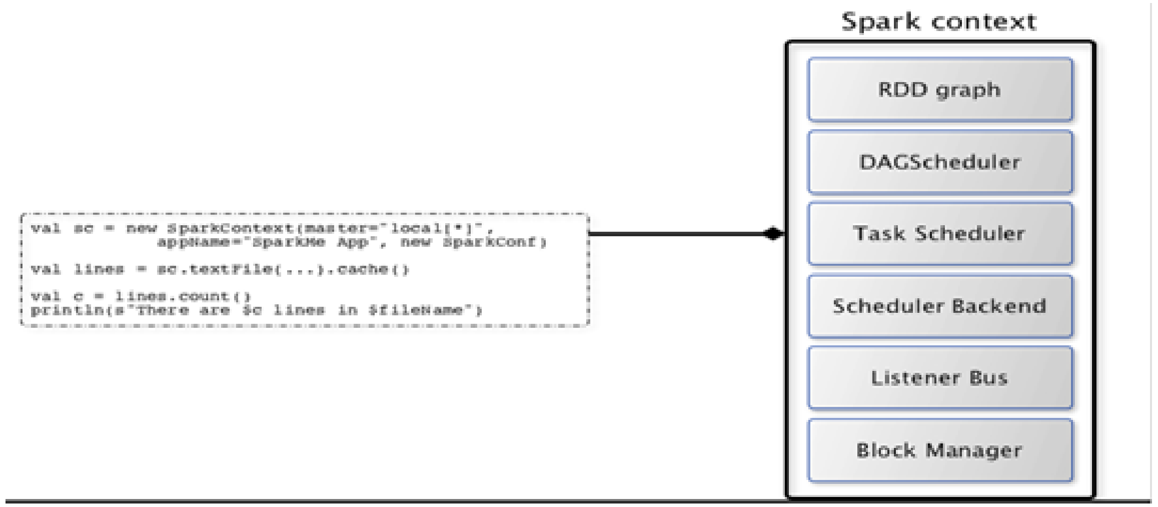
```
>>> rdd=sc.parallelize([1,2,3,4])
>>> rdd.map(lambda x:x*2)

>>> rdd.filter(lambda x:x%2==0)

>>> rdd2=sc.parallelize([1,4,2,2,3])
>>> rdd2.distinct()
```

SparkSession & SparkContext

Saurav Agarwal

SparkContext

- It is the entry point to spark core - heart of the 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)
- A Spark context is essentially a client of Spark's execution environment and acts as the master of your Spark application (don't get confused with the other meaning of Master in Spark, though)



```
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")
```

Saurav Agarwal

SparkContext & SparkSession

Prior to spark 2.0.0

- **sparkContext** was used as a channel to access all spark functionality.
- The spark driver program uses spark context to connect to the cluster through a resource manager (YARN or Mesos..).
- **sparkConf** is required to create the spark context object, which stores configuration parameter like appName (to identify your spark driver), application, number of core and memory size of executor running on worker node
- In order to use APIs of **SQL,HIVE , and Streaming**, separate contexts need to be created.

like

```
val conf=newSparkConf()

val sc = new SparkContext(conf)

val hc = new hiveContext(sc)

val ssc = new streamingContext(sc).
```

SPARK 2.0.0 onwards

- **SparkSession** provides a single point of entry to interact with underlying Spark functionality and allows programming Spark with **Dataframe** and **Dataset** APIs. All the functionality available with **sparkContext** are also available in **sparkSession**.
- In order to use APIs of SQL, HIVE, and Streaming, no need to create separate contexts as sparkSession includes all the APIs.
- Once the SparkSession is instantiated, we can configure Spark's run-time config properties

CREATING SPARKCONTEXT INSTANCE

- You can create a SparkContext instance with or without creating a SparkConf object first

```
>>> spark = SparkSession.builder \
... .master("local") \
... .appName("Word Count") \
... .config("spark.some.config.option", "some-value") \
 ... .getOrCreate()
```

You can create a spark session using this:

```
>>> from pyspark.sql import SparkSession
OR
>>> from pyspark.conf import SparkConf
>>> c = SparkConf()
>>> SparkSession.builder.config(conf=c)
```

From here
http://spark.apache.org/docs/2.0.0/api/python/pyspark.sql.html

Saurav Agarwal

Spark Caching

Caching

- It is one mechanism to speed up applications that access the same RDD multiple times.
- An RDD that is not cached, nor checkpointed, is re-evaluated again each time an action is invoked on that RDD.
- There are two function calls for caching an RDD: cache() and persist(level: StorageLevel).
- The difference among them is that cache() will cache the RDD into memory, whereas persist(level) can cache in memory, on disk, or off-heap memory according to the caching strategy specified by level.
- persist() without an argument is equivalent with cache(). Freeing up space from the Storage memory is performed by unpersist().

When to use caching

It is recommended to use caching in the following situations:

- RDD re-use in iterative machine learning applications
- RDD re-use in standalone Spark applications
- When RDD computation is expensive, caching can help in reducing the cost of recovery in the case one executor fails

Caching RDDs

```
lines=sc.textFile("...",4)
lines.cache()
#save,don't recompute!
comments=lines.filter(isComment)
print lines.count(),comments.count()
```

Storage Level defines how a RDD is persisted.

rdd.persist(StorageLevel. MEMORY_ONLY) is same as rdd.cache()

- NONE (default)
- DISK_ONLY
- MEMORY_ONLY (default for cache operation for RDD’s)
- MEMORY_AND_DISK
- MEMORY_AND_DISK_SER

Don’t spill to disk unless the functions that computed your datasets are expensive, or they filter a large amount of the data. Otherwise, recomputing a partition may be as fast as reading it from disk.

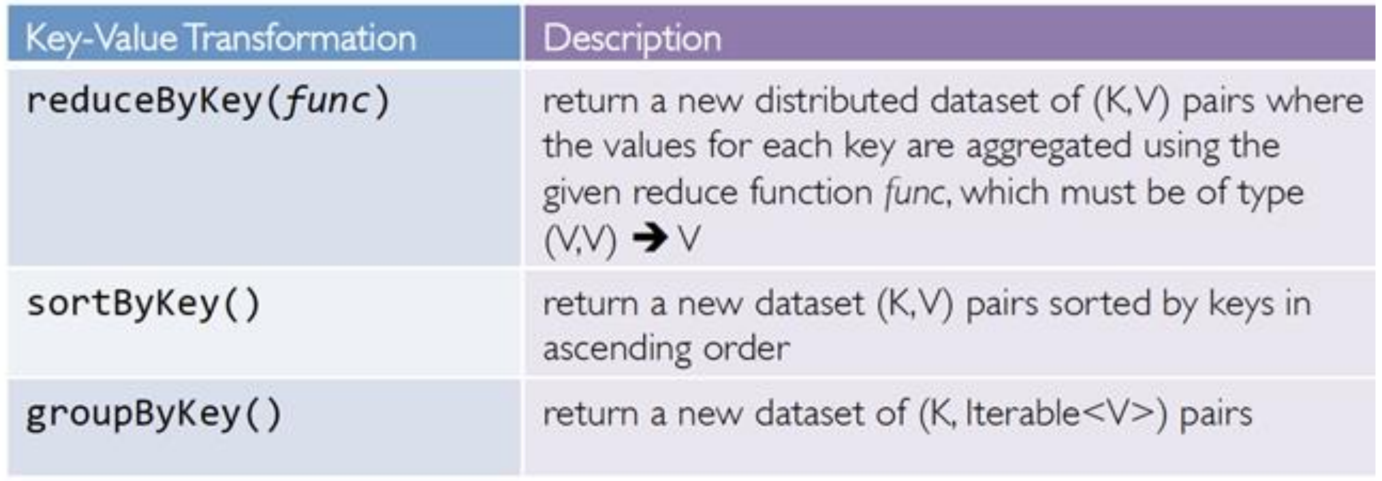
Spark Pair RDD

Saurav Agarwal

Spark Key-Value RDDs

Similar to Map Reduce, Spark supports Key-Value pairs
Each element of a Pair RDD is a pair tuple

```
>>>rdd=sc.parallelize([(1,2),(3,4)])
```



Key-Value Transformation	Description
`reduceByKey(func)`	return a new distributed 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
`sortByKey()`	return a new dataset (K,V) pairs sorted by keys in ascending order
`groupByKey()`	return a new dataset of (K, Iterable<V>) pairs

Spark Key-Value RDDs

```
>>> rdd = sc.parallelize([(1,2),(3,4),(3,6)])
>>> rdd.reduceByKey(lambda  a,b:a+b)
>>>rdd2=sc.parallelize([(1,'a'),(2,'c'),(1,'b')])
>>>rdd2.sortByKey()
>>> rdd2=sc.parallelize([(1,'a'),(2,'c'),(1,'b')])
>>> rdd2.groupByKey()
```

Be careful using groupByKey() as it can cause a lot of data movement across the network and create large Iterables at workers

Pair RDD Joins

X.join(Y)

» Return RDD of all pairs of elements with matching keys in X and Y

» Each pair is (k, (v1, v2)) tuple, where (k, v1) is in X and (k, v2) is in Y

```
>>> x=sc.parallelize([("a",1),("b",4)])
>>> y=sc.parallelize([("a",2),("a",3)])
>>> sorted(x.join(y).collect())
```

X.fullOuterJoin(Y)

» For each element (k, v) in X, resulting RDD will either contain

All pairs (k, (v, w)) for w in Y, or (k, (v, None)) if no elements in Y have k

» For each element (k, w) in Y, resulting RDD will either contain

All pairs (k, (v, w)) for v in X, or (k, (None, w)) if no elements in X have k

```
>>> x=sc.parallelize([("a",1),("b",4)])
>>> y=sc.parallelize([("a",2),("c",8)])
>>> sorted(x.fullOuterJoin(y).collect())
```

Spark Shared Variables

Saurav Agarwal

pySpark Shared Variables

Broadcast Variables

» Efficiently send large, read-only value to all workers

» Saved at workers for use in one or more Spark operations

» Like sending a large, read-only lookup table to all the nodes

At the driver	>>>broadcastVar=sc.broadcast([1,2,3])
At a worker (in code passed via a closure)	>>>broadcastVar.value

Accumulators

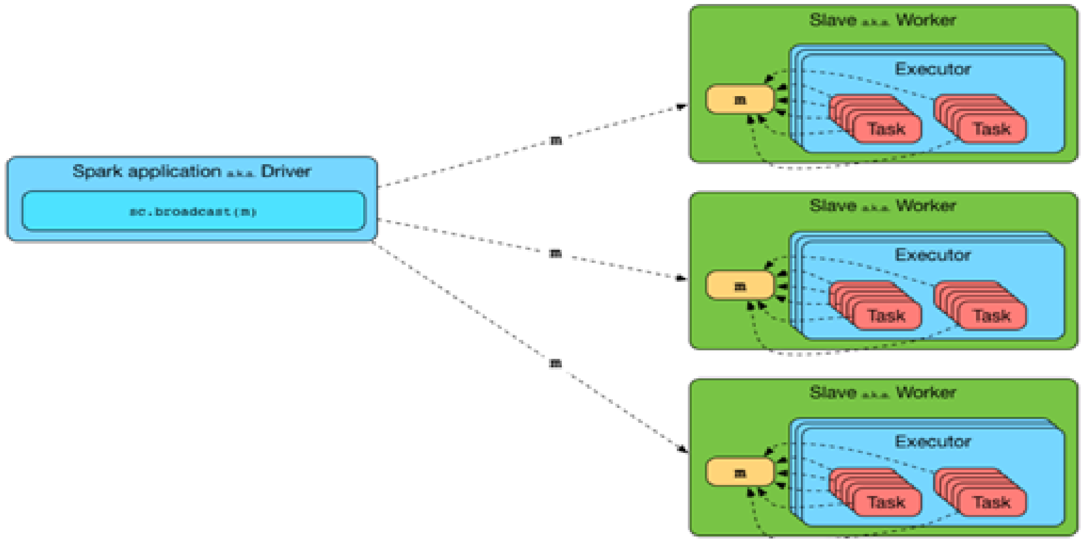
» Aggregate values from workers back to driver

» Only driver can access value of accumulator

» For tasks, accumulators are write-only

» Use to count errors seen in RDD across workers

```
>>>accum=sc.accumulator(0)
>>> rdd=sc.parallelize([1,2,3,4])
>>> def f(x):
>>>   global accum
>>>   accum+=x
>>>rdd.foreach(f)
>>>accum.value
```



Accumulators Example

Tasks at workers cannot access accumulator's values

Tasks see accumulators as write-only variables

Accumulators can be used in actions or transformations:

» Actions: each task's update to accumulator is applied only once

» Transformations: no guarantees (use only for debugging)

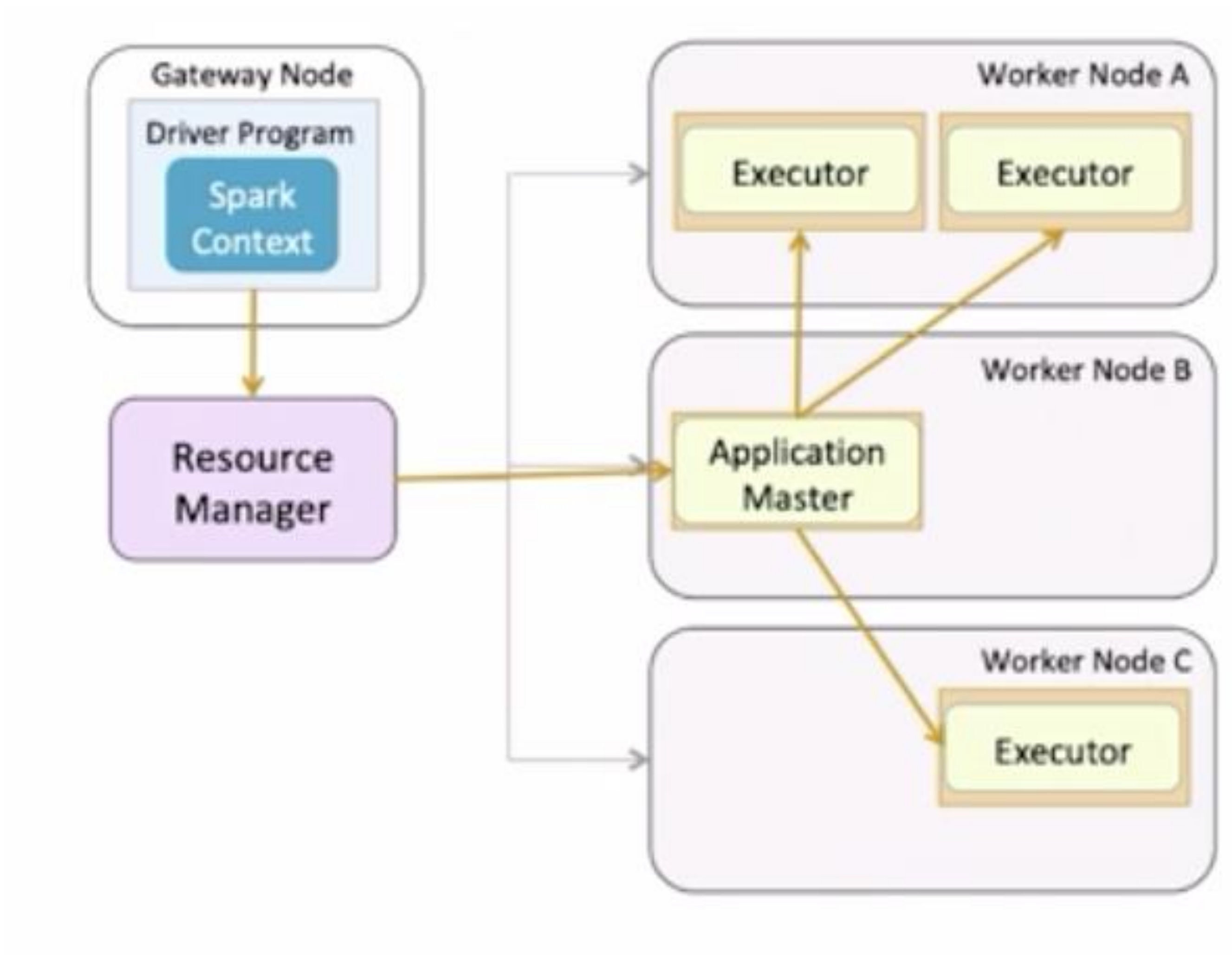
Types: integers, double, long, float

Counting Empty Lines

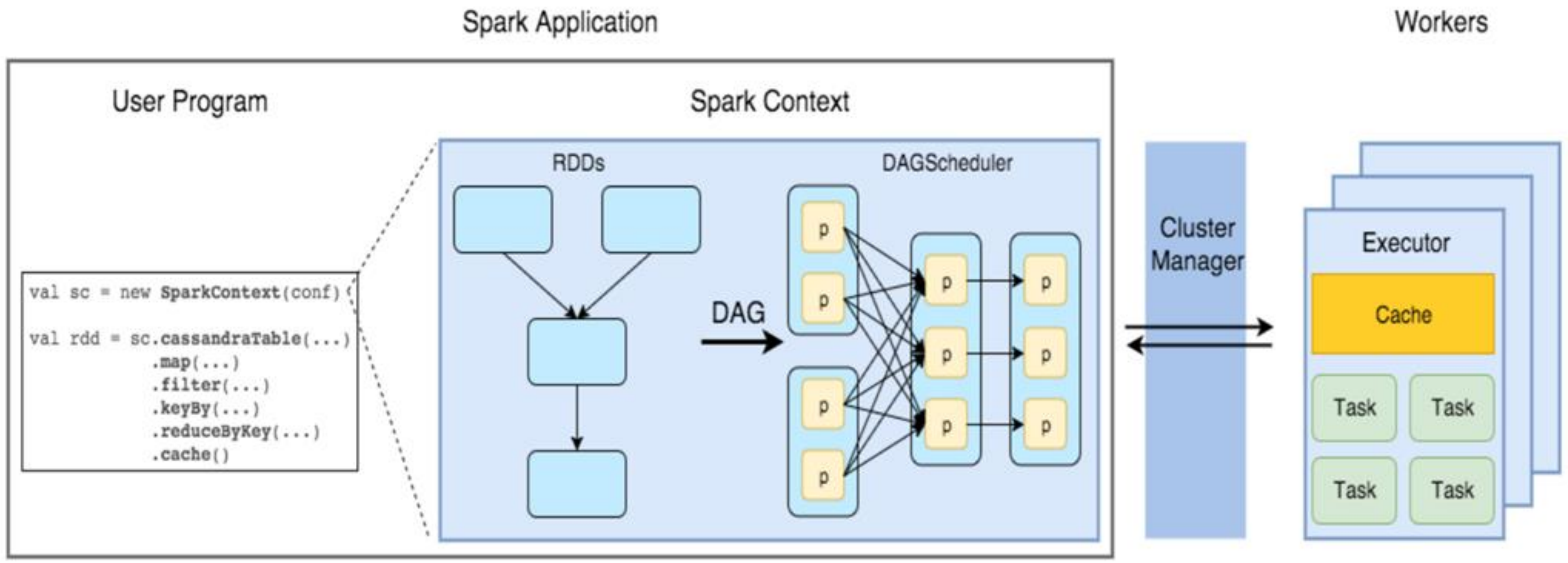
```
file=sc.textFile(inputFile)
#Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)
def extractCallSigns(line):
    global blankLines
    #Make the global variable accessible
    if(line==""):
      blankLines+=1
    return line.split("")
callSigns = file.flatMap(extractCallSigns)
print "Blank lines:%d" % blankLines.value
```

Spark Core Concepts , Internals & Architecture

Spark Execution Workflow



Spark Execution Workflow

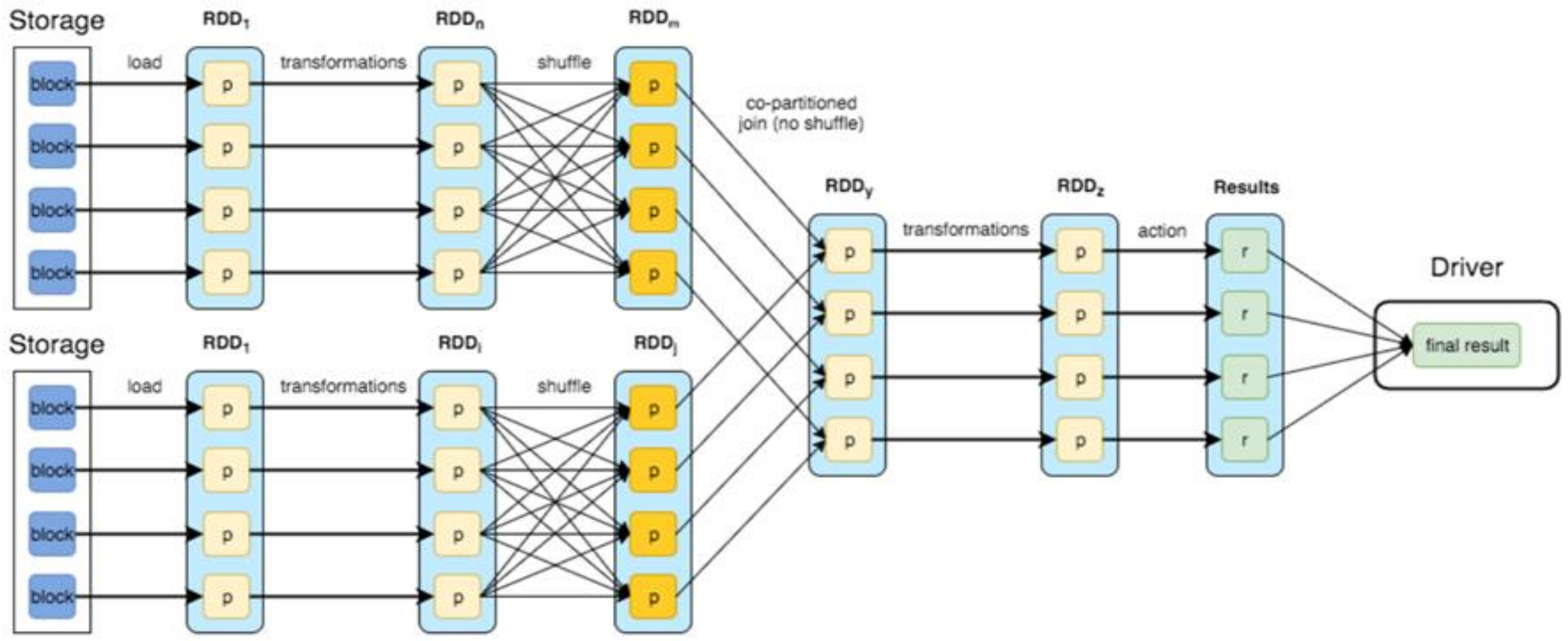


```
val sc = new SparkContext(conf)

val rdd = sc.cassandraTable(...)
            .map(...)
            .filter(...)
            .keyBy(...)
            .reduceByKey(...)
            .cache()
```

Spark Execution Workflow

- User code containing RDD transformations forms Direct Acyclic Graph
- DAG then split into stages of tasks by DAGScheduler.
- Stages combine tasks which don't require shuffling/repartitioning.
- Tasks run on workers and results then return to client.

Let Us Analyze a DAG

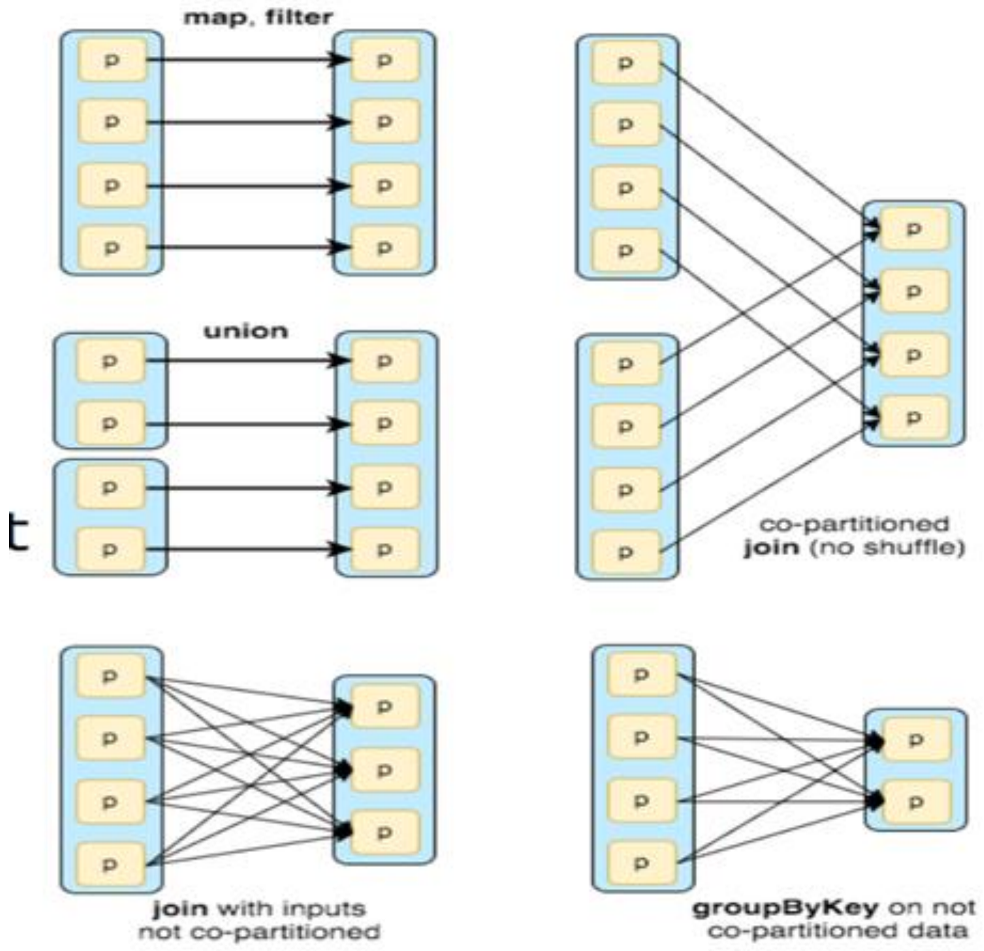


Let Us Analyze a DAG

- Any data processing workflow could be defined as reading the data source.
- Applying set of transformations and materializing the result in different ways.
- Transformations create dependencies between RDDs.
- The dependencies are usually classified as "narrow" and "wide".

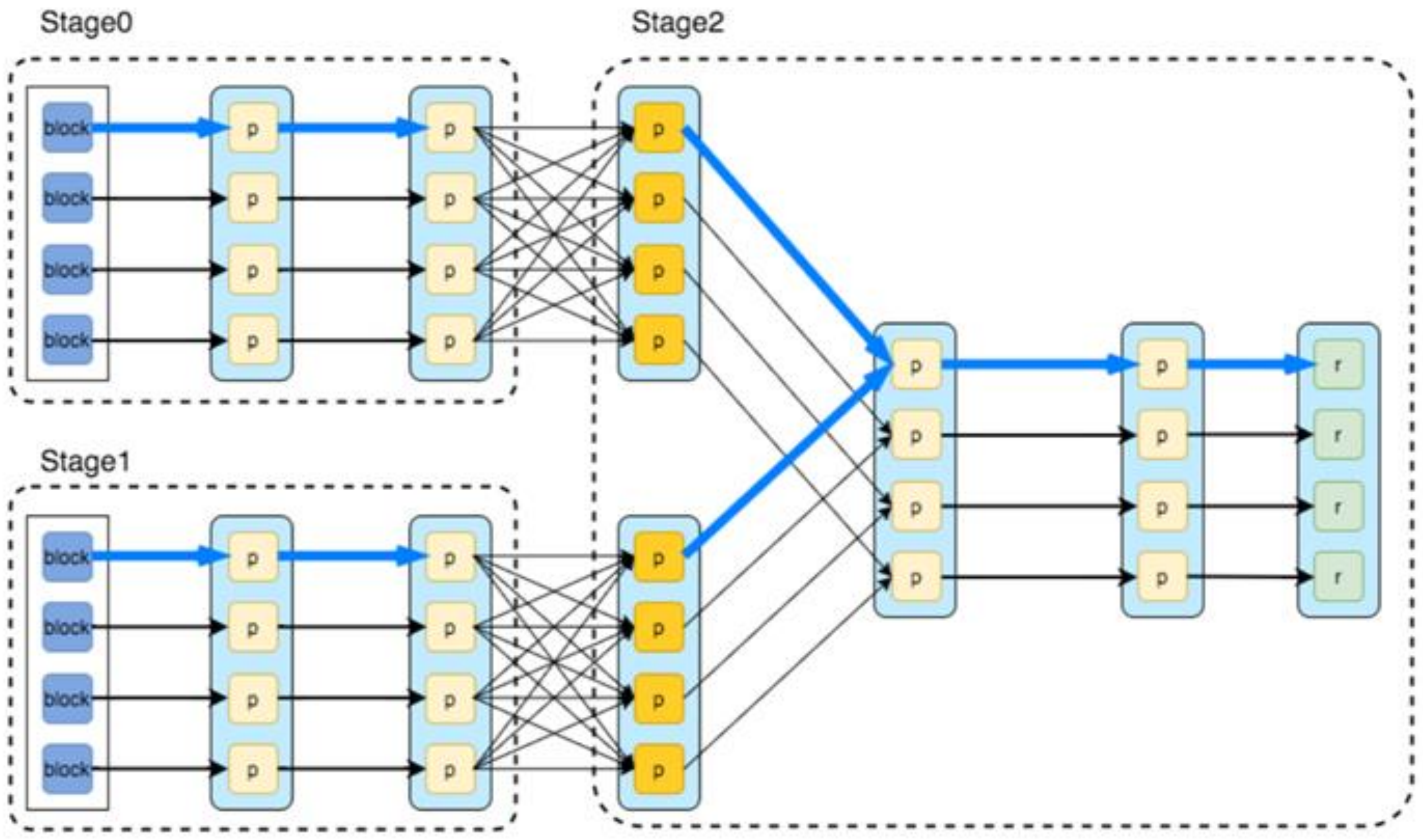
Let Us Analyze a DAG

- Narrow (pipelineable)
  - each partition of the parent RDD is used by at most one partition of the child RDD
  - allow for pipelined execution on one cluster node
  - failure recovery is more efficient as only lost parent partitions need to be recomputed
- Wide (shuffle)
  - multiple child partitions may depend on one parent partition
  - require data from all parent partitions to be available and to be shuffled across the nodes
  - if some partition is lost from all the ancestors a complete recomputation is needed


map, filter
union
co-partitioned join (no shuffle)
join with inputs not co-partitioned
groupByKey on not co-partitioned data
p

Splitting DAG Into Stages

- Spark stages are created by breaking the RDD graph at shuffle boundaries



LIST OF NARROW VS WIDE TRANSFORMS

Transformations with *(usually)* Narrow dependencies:

- map
- mapValues
- flatMap
- filter
- mapPartitions
- mapPartitionsWithIndex

Transformations with (usually) Wide dependencies: (might cause a shuffle)

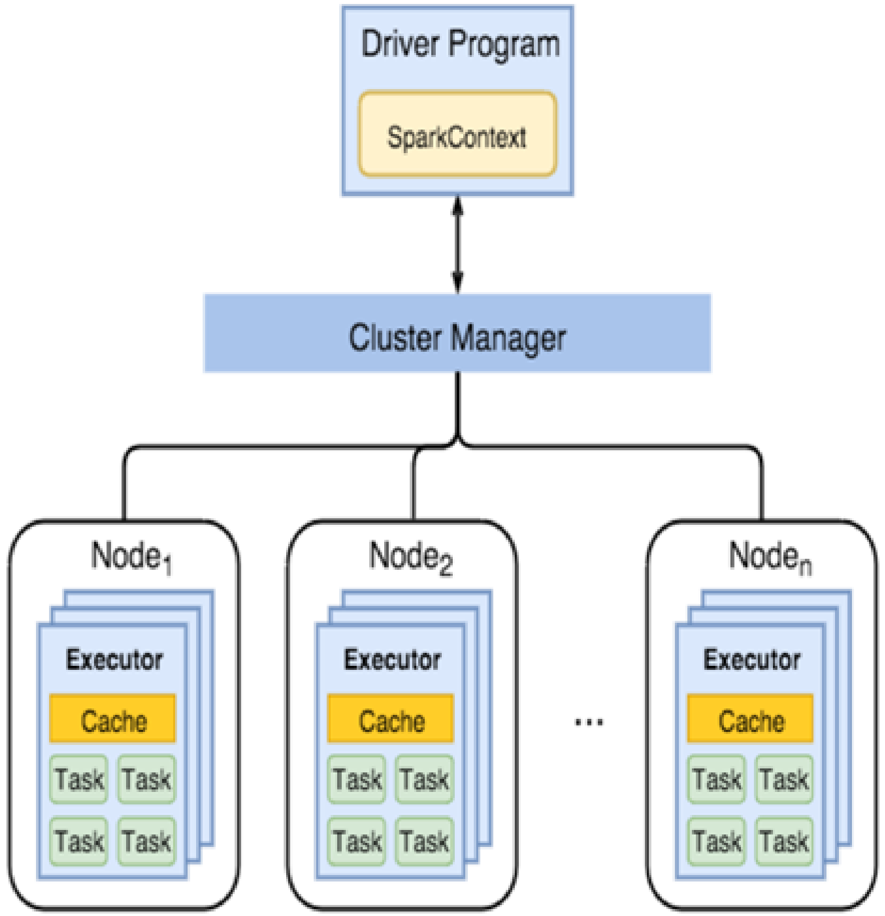
- cogroup
- groupWith
- join
- leftOuterJoin
- rightOuterJoin
- groupByKey
- reduceByKey
- combineByKey
- distinct
- intersection
- repartition
- coalesce

Splitting DAG Into Stages

- RDD operations with "narrow" dependencies, like map() and filter(), are pipelined together into one set of tasks in each stage operations with shuffle dependencies require multiple stages (one to write a set of map output files, and another to read those files after a barrier).
- In the end, every stage will have only shuffle dependencies on other stages, and may compute multiple operations inside it. The actual pipelining of these operations happens in the RDD.compute() functions of various RDDs

Spark Components

- From high level there are three major components.
- Spark driver
  - separate process to execute user applications
  - creates SparkContext to schedule jobs execution and negotiate with cluster manager

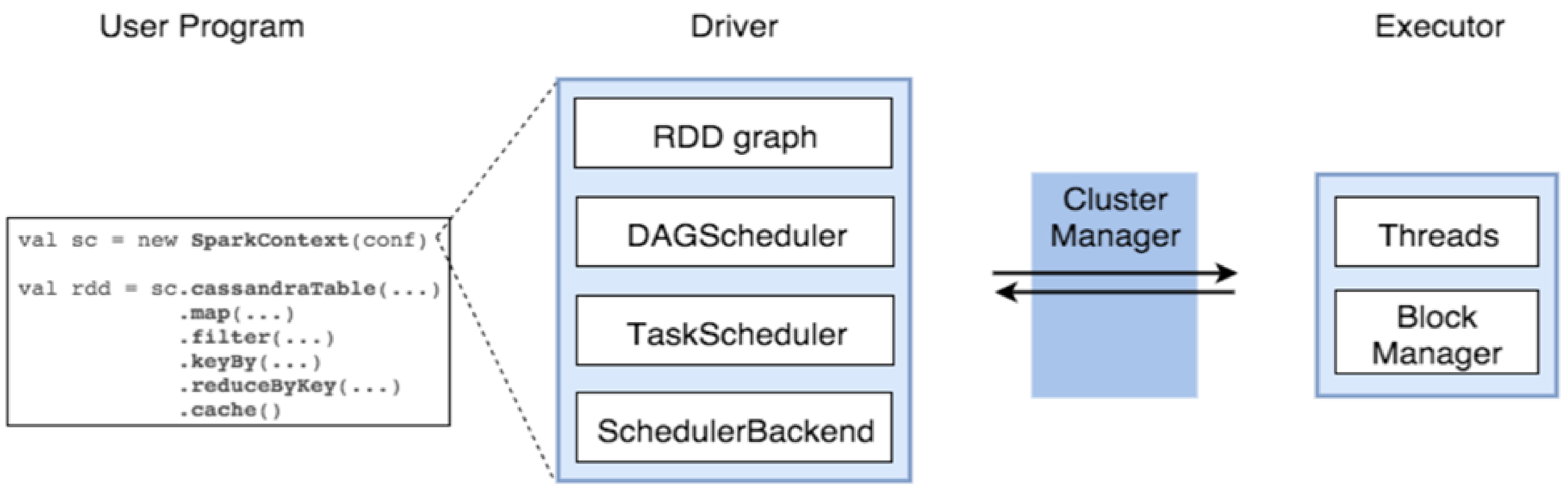


Spark Components

- Executors
  - Run tasks scheduled by driver
  - store computation results in memory, on disk or off-heap
  - interact with storage systems
- Cluster Manager
  - Mesos
  - YARN
  - Spark Standalone

Spark Components

- Spark Driver contains more components responsible for translation of user code into actual jobs executed on cluster:



```
val sc = new SparkContext(conf)

val rdd = sc.cassandraTable(...)
            .map(...)
            .filter(...)
            .keyBy(...)
            .reduceByKey(...)
            .cache()
```

Spark Components

- SparkContext
  - represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster
- DAGScheduler
  - computes a DAG of stages for each job and submits them to TaskScheduler
  - determines preferred locations for tasks (based on cache status or shuffle files locations) and finds minimum schedule to run the jobs
- TaskScheduler
  - responsible for sending tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers(slowness)

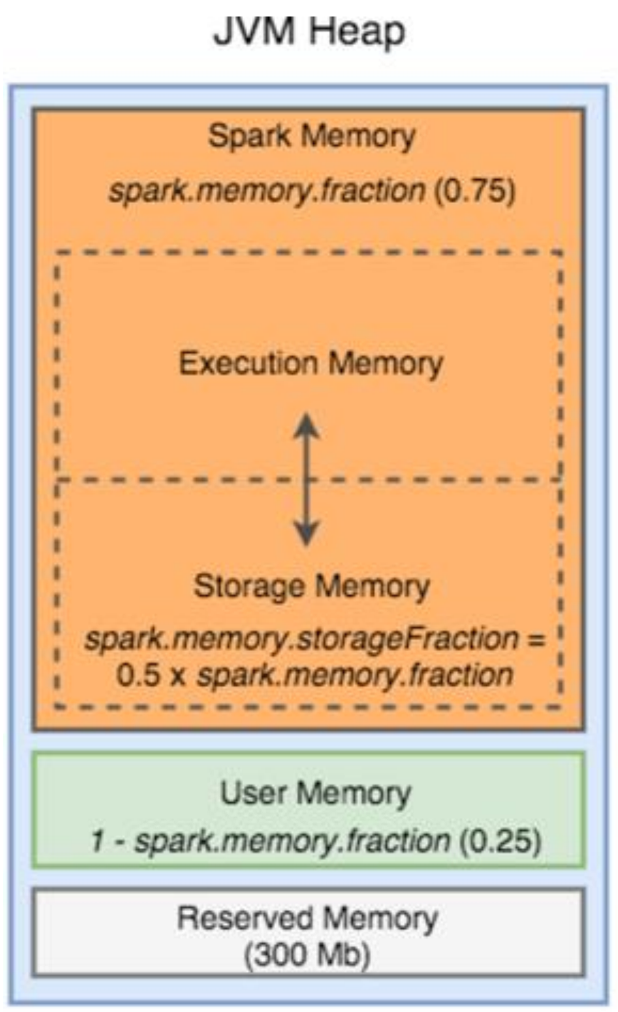
Spark Components

- SchedulerBackend – Resource Manager
  - backend interface for scheduling systems that allows plugging in different implementations(Mesos, YARN, Standalone, local)

- BlockManager
  - provides interfaces for putting and retrieving blocks both local and external

Memory Management

- Execution Memory
  - storage for data needed during tasks execution
  - shuffle-related data
- Storage Memory
  - storage of cached RDDs and broadcast variables
  - possible to borrow from execution memory (spill otherwise)
  - safeguard value is 50% of Spark Memory when cached blocks are immune to eviction
- User Memory
  - user data structures and internal metadata in Spark
  - safeguarding against OOM
- Reserved Memory
  - memory needed for running executor itself and not strictly related to Spark


JVM Heap
Spark Memory
spark.memory.fraction (0.75)
Execution Memory
Storage Memory
spark.memory.storageFraction = 0.5 x spark.memory.fraction
User Memory
1 - spark.memory.fraction (0.25)
Reserved Memory (300 Mb)