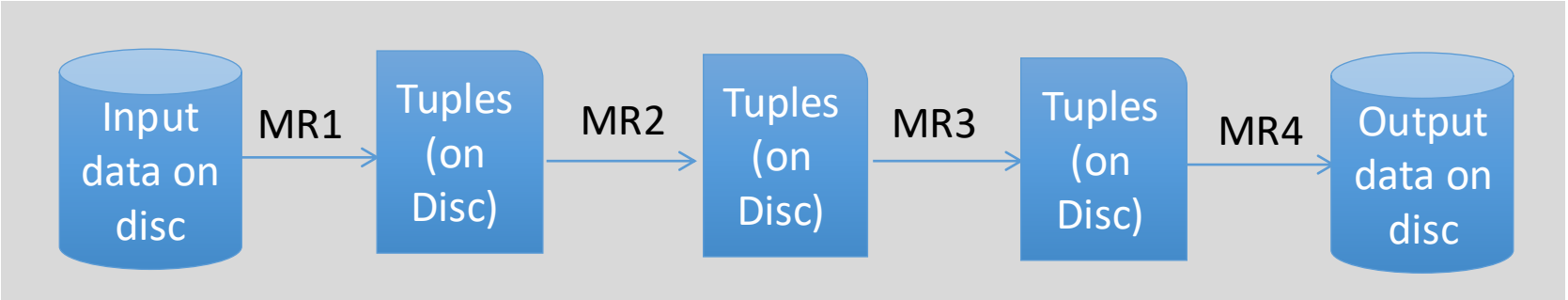
Apache Spark

MapReduce problems

- MapReduce problems:
  - Persistence to disk typically slower than in-memory work
- Alternative: Apache Spark
  - a general-purpose processing engine that can be used instead of MapReduce

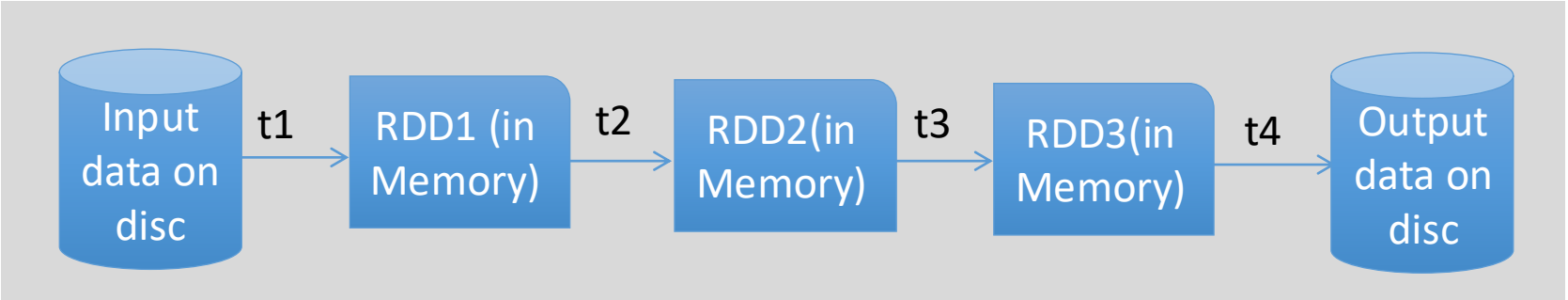
MapReduce

- There is a lot of reading and writing happening to the disk after each MR transformation which makes is too slow and less suitable for iterative processing e.g Machine Learning algos.



Spark

- Results are cached in memory, so no I/O's after every process.
- When the memory is not sufficient enough, data can be either spilled to the drive or is just left to be recreated upon request for the same.

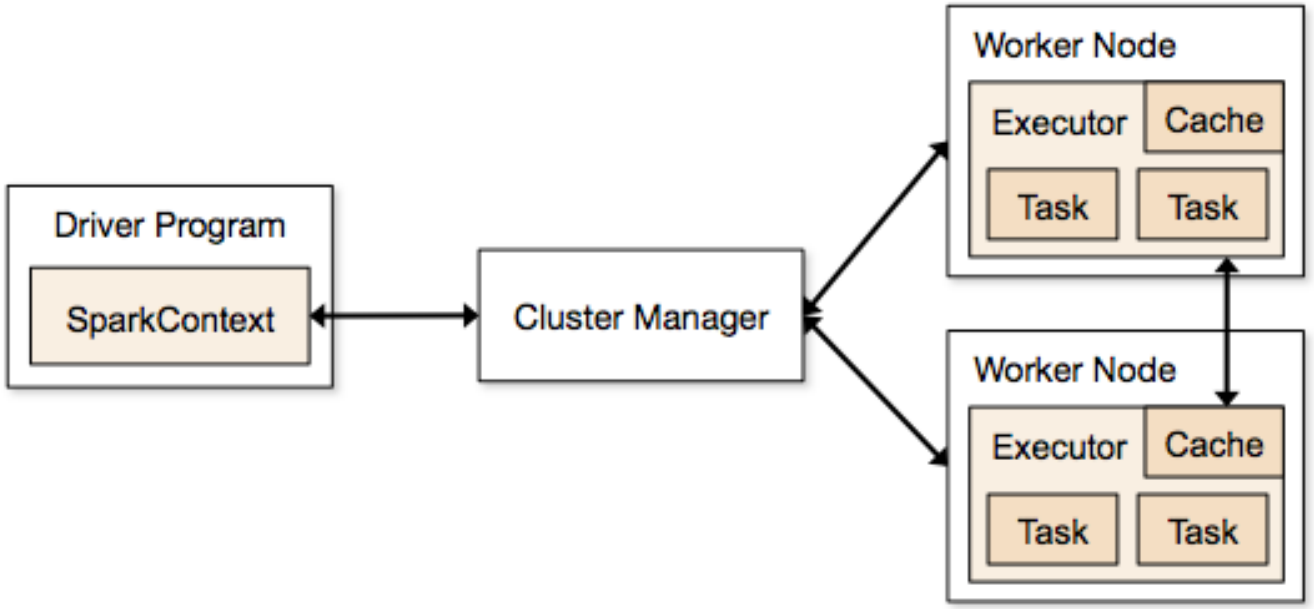


Apache Spark

- Processing engine; instead of just “map” and “reduce”, defines a large set of *operations* (transformations & actions)
  - Operations can be arbitrarily combined in any order
- Open source software
- Supports Java, Scala and Python
- Key construct: Resilient Distributed Dataset (RDD)

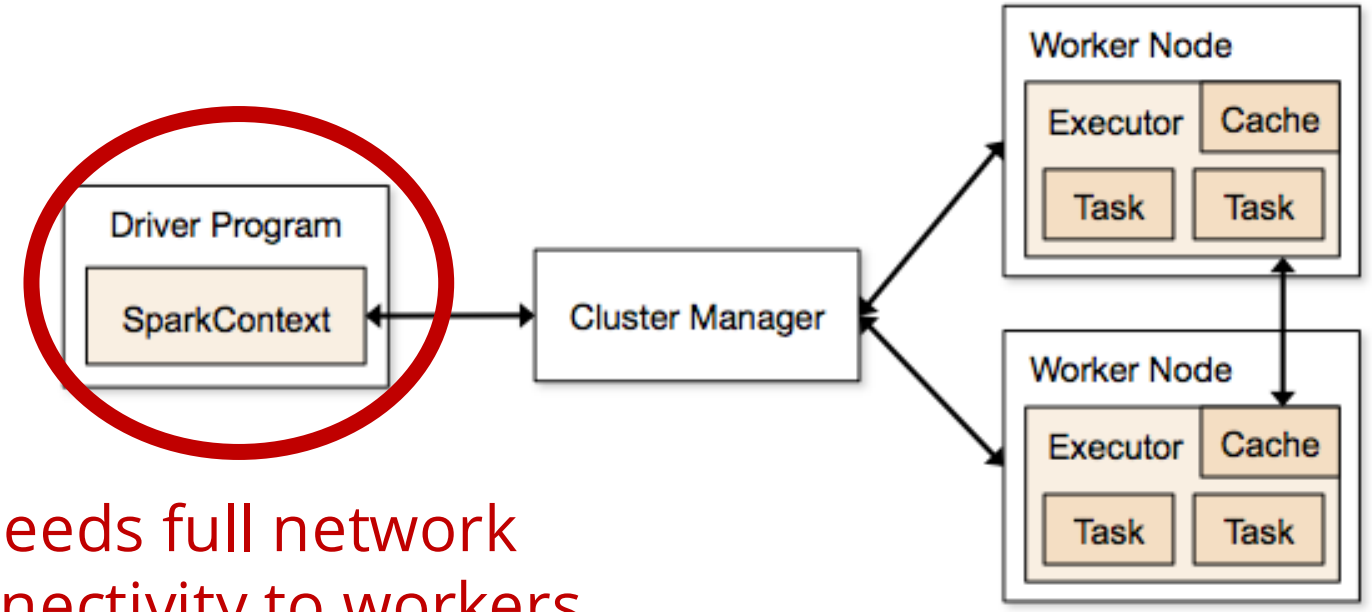
Cluster manager

Cluster manager grants executors to a Spark application



Driver program

Driver program decides when to launch tasks on which executor



Needs full network connectivity to workers

Driver & Executors

- Driver is responsible for converting a user program into units of physical execution called tasks.
- Driver coordinates the individual tasks on executors.
- Driver will look at the current set of executors and try to schedule each task in a appropriate location, based on data placement.

Driver & Executors

- Each executor represents a process capable of running tasks and storing RDD data
- Spark acquires executors on nodes in the cluster with the help of Cluster Manager
- Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads.

Available APIs

- You can write in Java, Scala or Python
- interactive interpreter: Scala & Python only
- standalone applications: any
- performance: Java & Scala are faster thanks to static typing

Apache Spark: Libraries “on top” of core that come with it

- Spark SQL
- Spark Streaming – stream processing of live datastreams
- MLlib - machine learning
- GraphX – graph manipulation
  - extends Spark RDD with Graph abstraction: a directed multigraph with properties attached to each vertex and edge.

On-Disk Sort Record:
Time to sort 100TB

2013 Record: Hadoop

2100 machines

72 minutes

2014 Record: Spark

207 machines

23 minutes

Also sorted 1PB in 4 hours

Source: Daytona GraySort benchmark, sortbenchmark.org

RDD Basics

- Resilient Distributed Datasets
  - Resilient - able to recoil or spring back into shape after bending, stretching, or being compressed.
- Immutable distributed collection of objects
- Split into multiple partitions, which may be computed on different nodes

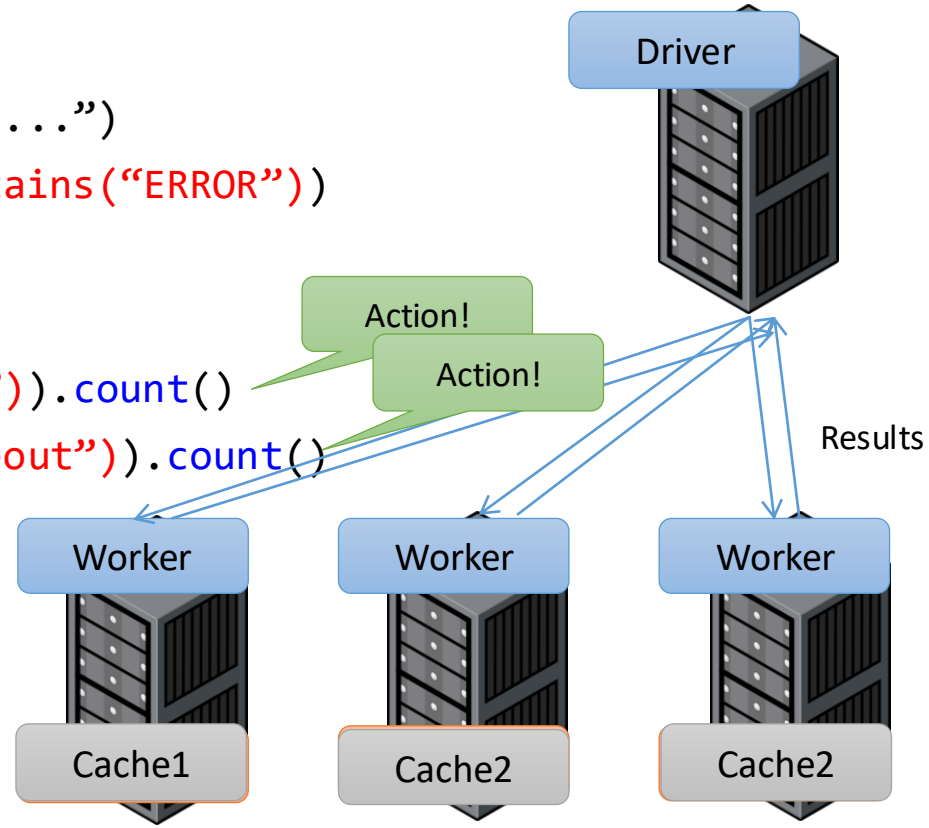
Resilient Distributed Dataset (RDD) – key Spark construct

- RDDs represent data or transformations on data
- RDDs can be created from Hadoop InputFormats (such as HDFS files), “parallelize()” datasets, or by transforming other RDDs (you can stack RDDs)
- Actions can be applied to RDDs; actions force calculations and return values
- Lazy evaluation (transformation) : Nothing computed until an action requires it

Job example

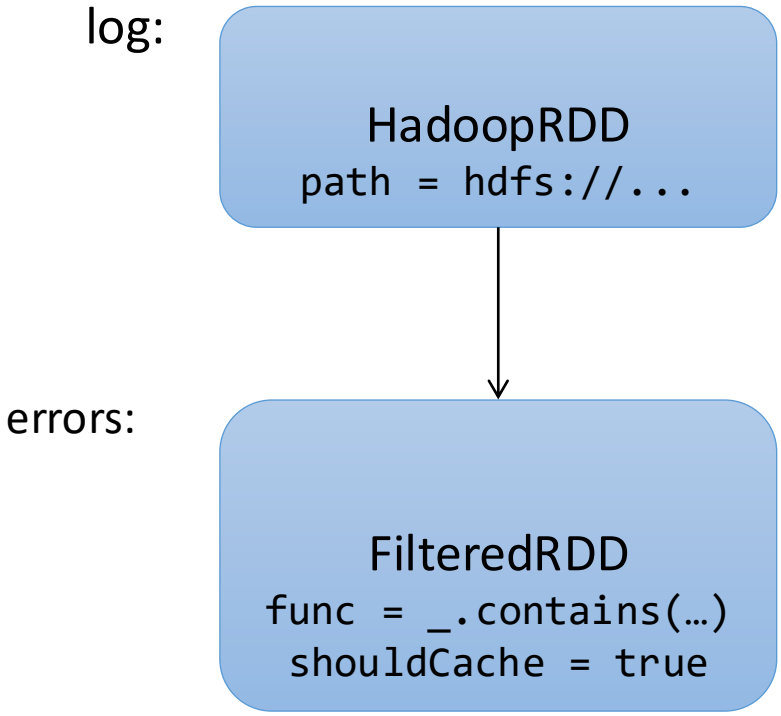
```
val log = sc.textFile("hdfs://...")
val errors = log.filter(_.contains("ERROR"))
errors.cache()

errors.filter(_.contains("I/O")).count()
errors.filter(_.contains("timeout")).count()
```

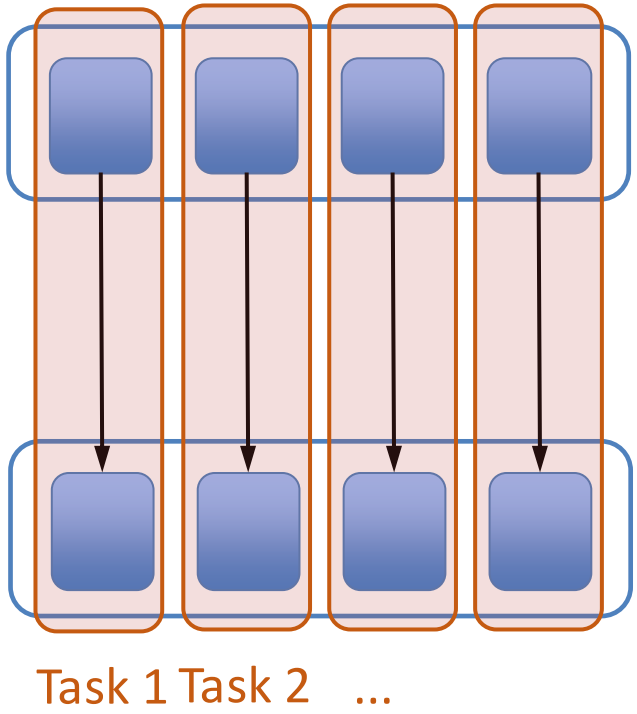


RDD partition-level view

Dataset-level view:



Partition-level view:

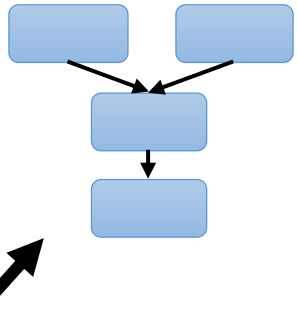


RDD partition

- a single RDD has one or more partitions scattered across multiple node
- a single partition is processed on a single node
- a single node can handle multiple partitions (with optimum 2-4 partitions per CPU)

Job scheduling

RDD Objects

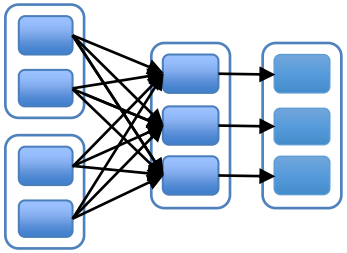


```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
```

build operator DAG

DAG

DAGScheduler

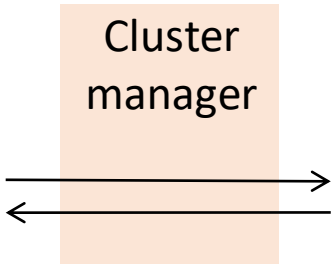


split graph into *stages* of tasks

submit each stage as ready

TaskSet

TaskScheduler

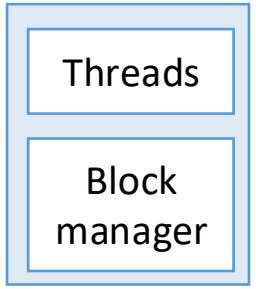


launch tasks via cluster manager

retry failed or straggling tasks

Task

Worker

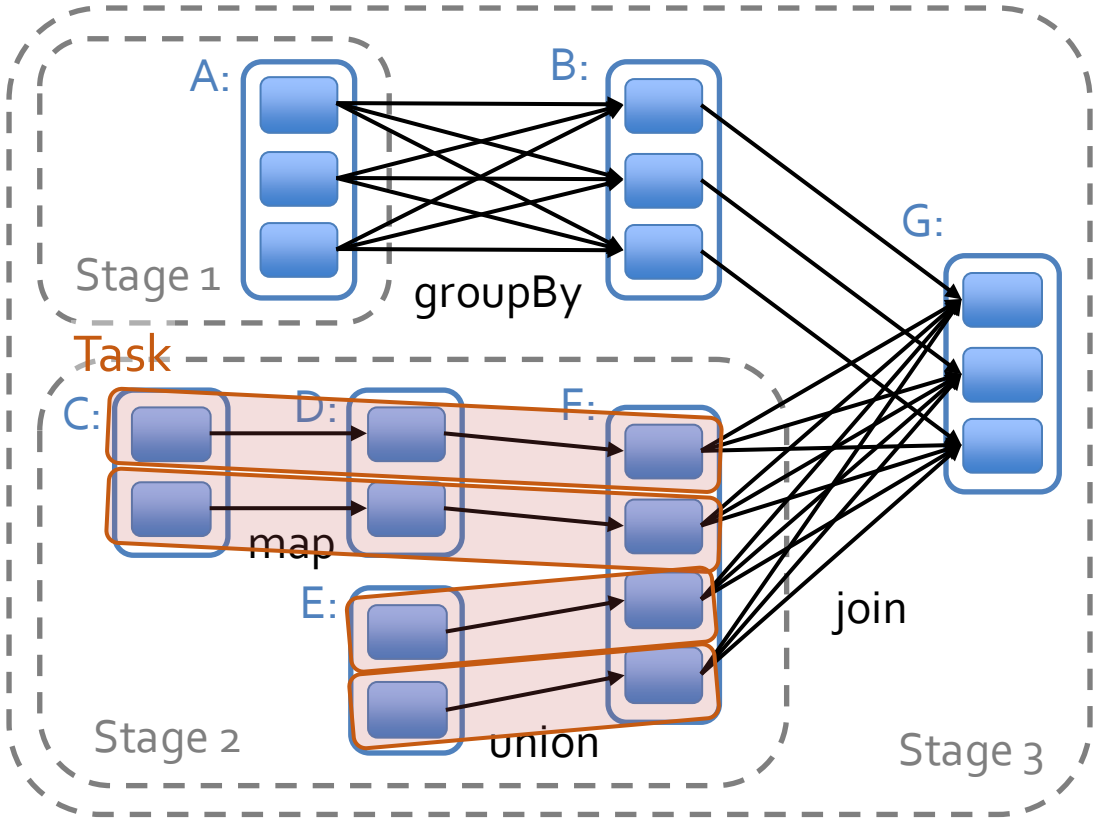


execute tasks

store and serve blocks

Scheduler Optimizations

- Pipelines narrow ops. within a stage
- Picks join algorithms based on partitioning (minimize shuffles)
- Reuses previously cached data



Task Details

- Stage boundaries are only at input RDDs or “shuffle” operations
- So, each task looks like this:

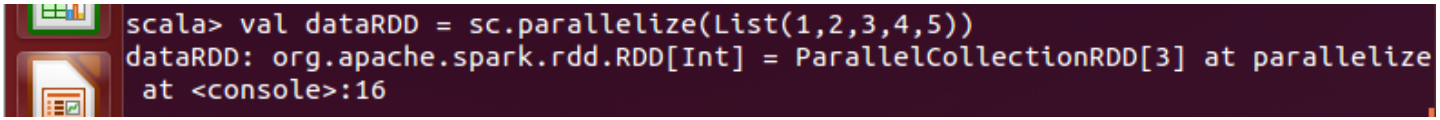
(Note: we write shuffle outputs to RAM/disk to allow retries)

RDD (Resilient Distributed Datasets) ..Cont..

- Resilient distributed dataset (RDD), is a fault-tolerant collection of elements that can be operated on in parallel.
- Restricted form of distributed shared memory – **read-only**
- Partitioned collection of records – can only be built through coarse-grained deterministic transformations
  - Transformations from other RDDs
  - Express computation by – defining RDDs
- RDDs can be created from any data source e.g. Scala collection, local file system, Hadoop, Amazon S3, HBase table etc.
- Spark supports text files, Sequence Files, and any other Hadoop Input Format, and can also take a directory or a glob
- Even though the RDDs are defined**, they don't contain any data**
- The computation to create the data in a RDD is only done when the data is referenced. **Lazy evlolution**

Data Loading in RDD

- RDD is re-computed every time when it is materialized
- So it is a good idea to improve performance by caching a RDD if it is accessed frequently
- One of the easiest way to load data in RDD is to load from a Scala collection
- SparkContext provides parallelize function, which converts the Scala collection into the RDD of the same type



Manipulating RDDs

- Manipulating RDDs is quite similar to Scala collections manipulation
- Manipulating your RDD in Scala is quite simple, especially if you are familiar with Scala's collection library
- Many of the standard functions are available directly on Spark's RDDs with the primary catch being that they are immutable
- Programmer need not to worry about the RDDs being executed on same machine or multiple machines
- The hallmark functions of map and reduce are available by default
- The map and other Spark functions DO NOT transform the existing elements, they always create a new RDD with transformed elements (Immutability in Action!)

```
f(a,b) == f(b,a) and f(a, f(b,c)) == f(f(a,b), c)
```

For example, to sum all the elements,

```
use rdd.reduce(x,y => x+y)
                ( or )
rdd.reduce(new Function2<Integer, Integer, Integer>()
{
public Integer call(Integer x, Integer y)
 { return x+y;  } }
```

RDD operations

- *transformations* to build RDDs through deterministic operations on other RDDs
  - transformations like *map*, *filter*, *join*
  - lazy operation
- *actions* to return final value to driver or write data to external storage
  - actions like *count*, *collect*, *save*
  - triggers execution

RDD Operations

Transformations (define a new RDD)	**Parallel operations (Actions)** (return a result to driver)
map filter sample union groupByKey reduceByKey join cache …	reduce collect count save lookupKey …

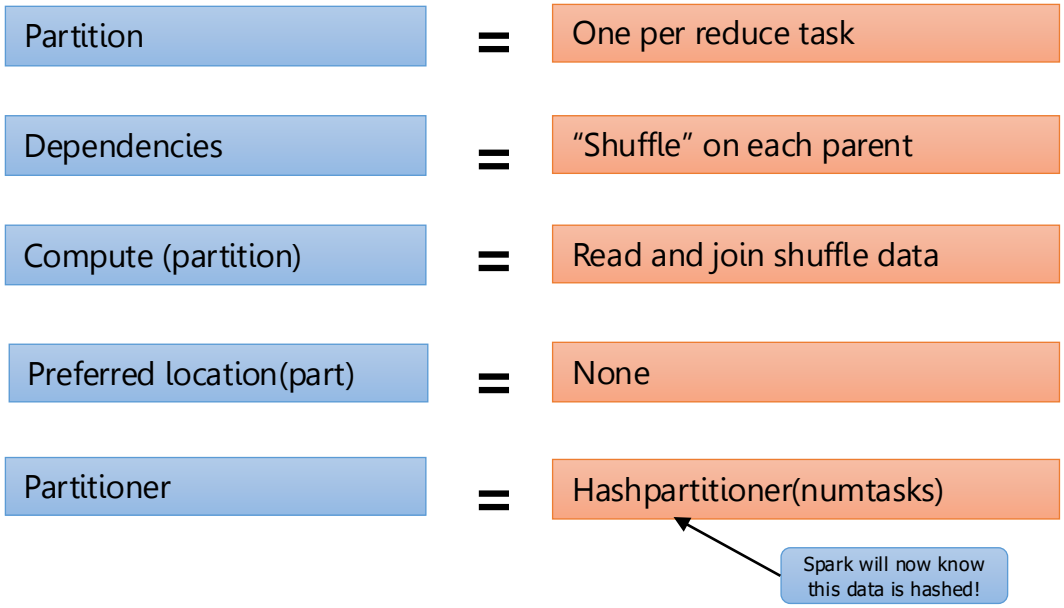
Example: HadoopRDD

Partition	=	One per HDFS block
Dependencies	=	None
Compute (part)	=	Read corresponding block
Preferred location (part)	=	HDFS block location
Partitioner	=	None

Example: FilteredRDD

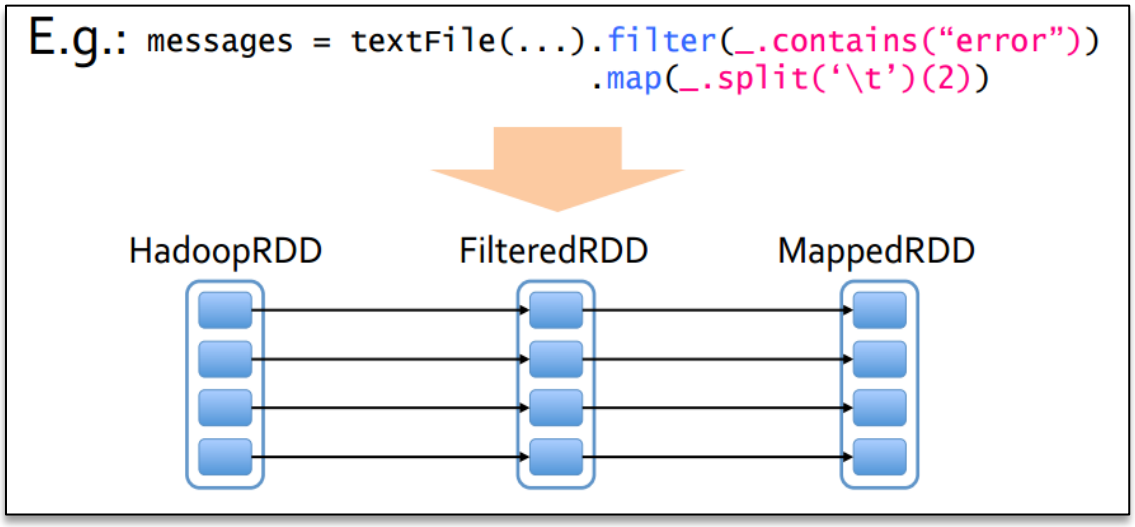
Partition	=	Same as parent RDD
Dependencies	=	"One-to-one" on parent
Compute (part)	=	Compute parent and filter it
Preferred location(part)	=	None (ask parent)
Practitioner	=	None

Example: JoinedRDD



Fault Recovery

Efficient fault recovery using lineage – log one operation to apply to many elements (lineage) – recomputed lost partitions on failure



Lambda

- Lambda also known as anonymous functions

```
Rdd.flatMap(line => line.split(" "))
```

Named Method

```
def addOne(intem: Int) = {
  item + 1
}
```

```
val intList = List(1, 2)
 for (item <- intList) {
   addOne(item)
 } // List(2,3)
```

Lambda Function

```
def addOne(intem: Int) = {
  item + 1
}
```

```
val intList = List(1, 2)
intList.map(x => {
  addOne(x)
}) //List(2,3)
```

Lambda Functions

- Map the method inline

```
val intList = List(1, 2)

intList.map(item => item + 1) //List(2,3)
```

Lambda Functions

```
val intList = List(1,2)

intList.map(item => item + 1) //List(2,3)
```

```
def addOne(item: Int) = {
    item + 1
}

val intList = List(1,2)
for (item <- intList) {
      addOne(item)
} // List(2,3)
```

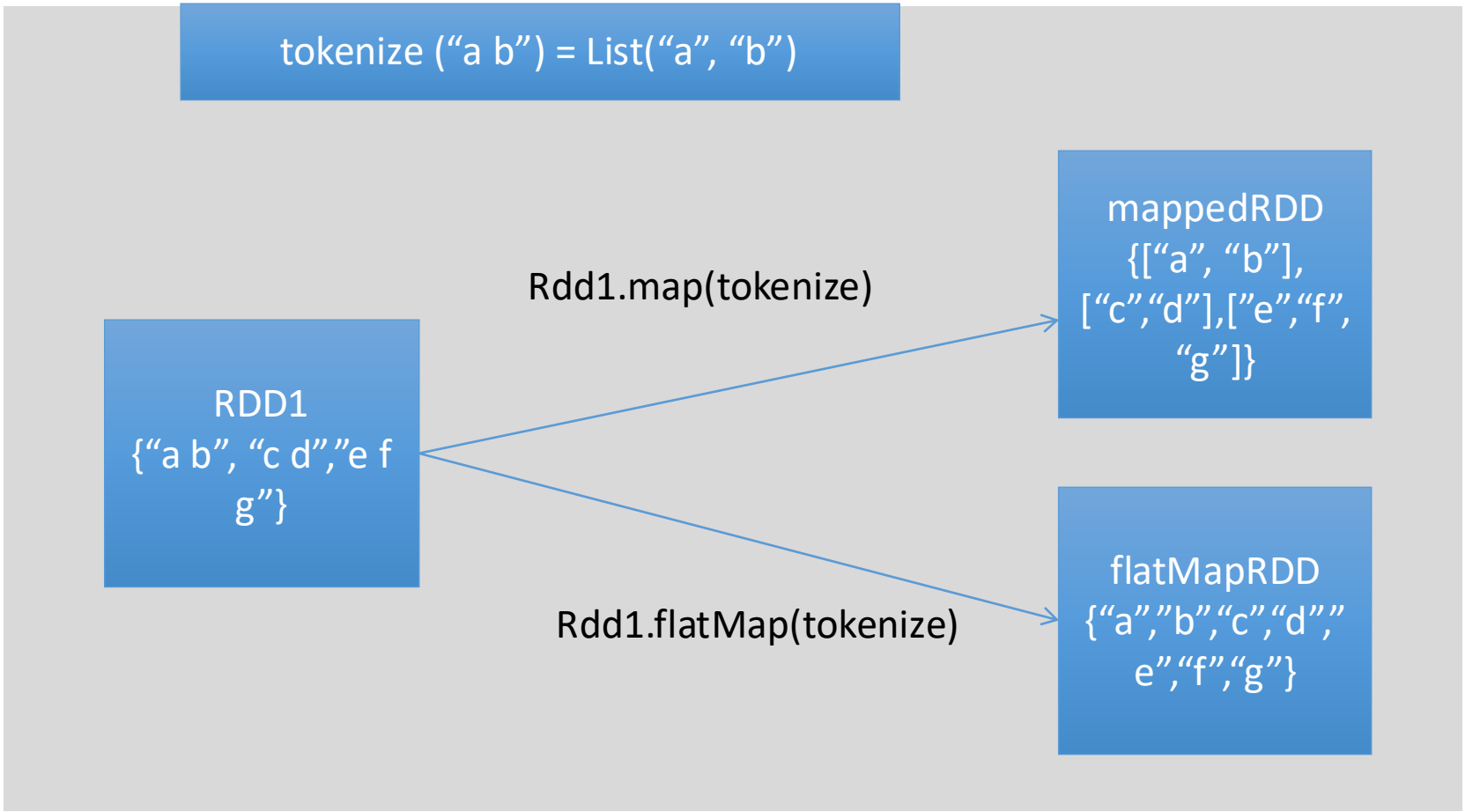
Spark example #1 (Scala)

```
// "sc" is a "Spark context" – this transforms the file into an RDD
val textFile = sc.textFile("README.md")
// Return number of items (lines) in this RDD; count() is an action
textFile.count()
// Demo filtering.  Filter is a tranform.  Lazy operation
val linesWithSpark = textFile.filter(line => line.contains("Spark"))
// Chaining – how many lines contain "Spark"?  count() is an action.
textFile.filter(line => line.contains("Spark")).count()
// Length of line with most words.  Reduce is an action.
textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
// Word count – traditional map-reduce.  collect() is an action
val wordCounts = textFile.flatMap(line => line.split(" ")).map(word => (word,
1)).reduceByKey((a, b) => a + b)
wordCounts.collect()
```

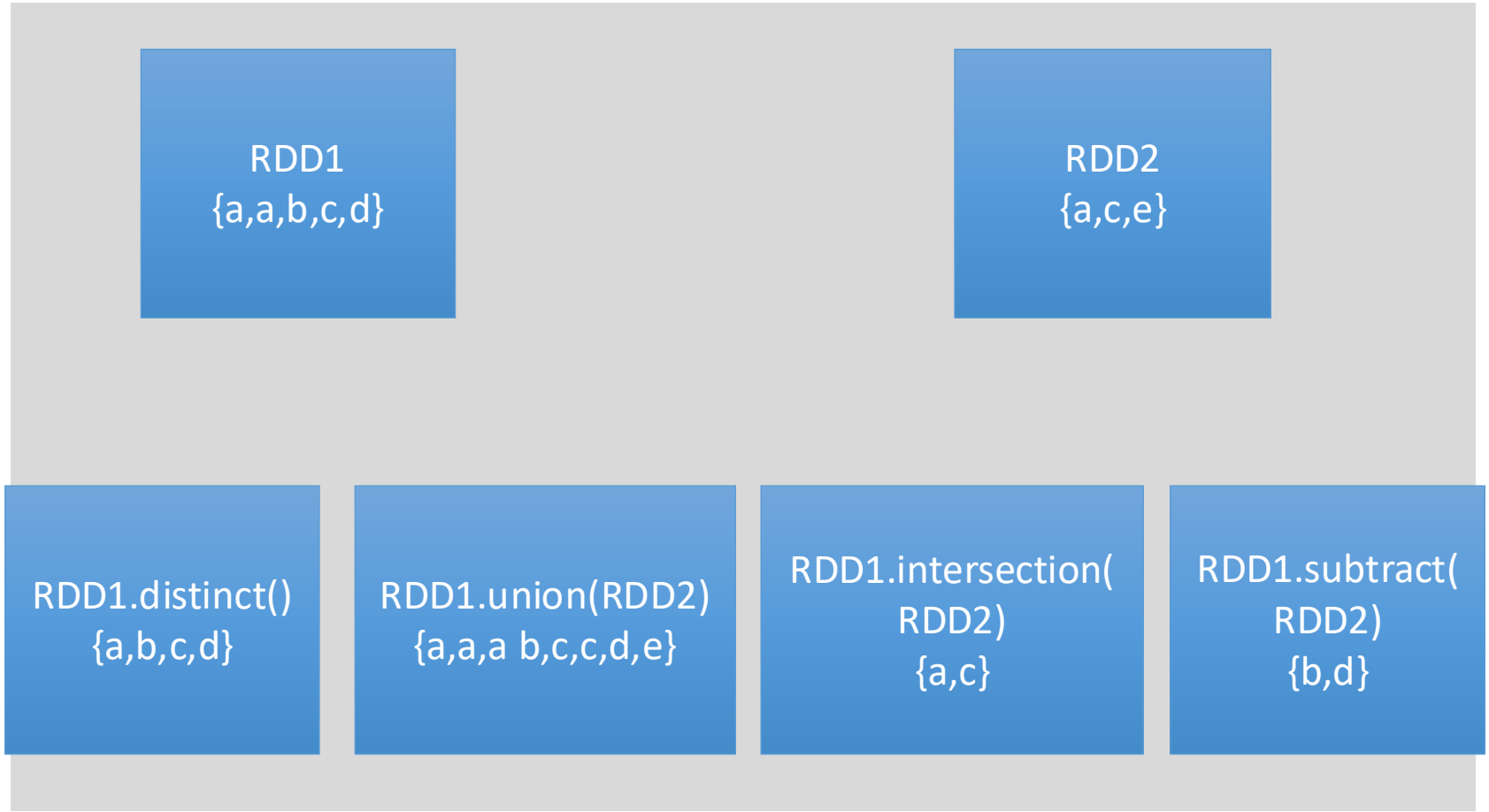
Sample Spark transformations

- **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
- **union(otherDataset):** Return a new dataset that contains the union of the elements in the source dataset and the argument.
- **intersection(otherDataset)**: Return a new RDD that contains the intersection of elements in the source dataset and the argument.
- **distinct():** Return a new dataset that contains the distinct elements of the source dataset
- **join(otherDataset)**: 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.

flatMap vs map



Operations


RDD1
{a,a,b,c,d}
RDD2
{a,c,e}
RDD1.distinct()
{a,b,c,d}
RDD1.union(RDD2)
{a,a,a b,c,c,d,e}
RDD1.intersection(
RDD2)
{a,c}
RDD1.subtract(
RDD2)
{b,d}

Transformations on Pair RDD's

Function	Parameter options	Explanation	Output
mapValues	{(1,2),(3,4),(3,6)} rdd.mapValues(x => x+1)	Apply the function to each value of a pair RDD without changing the key	{(1,3),(3,5),(3,7)}
reduceByKey	{(1,2),(3,4),(3,6)} rdd.reduceByKey ((x,y) => x+y)	Combine values with same key	{(1,2),(3,10)}
groupByKey	{(1,2),(3,4),(3,6)} rdd.groubByKey()	Group values with the same key	{(1,[2]),(3,[4,6])}

Transformations (Cont'd)

Transformation	Meaning
map(func)	Return a new distributed dataset formed by passing each element of the source through a function func
filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true
intersection(otherDataset)	Return a new RDD that contains the intersection of elements in the source dataset and the argument
groupByKey([numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs
reduceByKey(func, [numTasks])	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

Transformations (Cont'd)

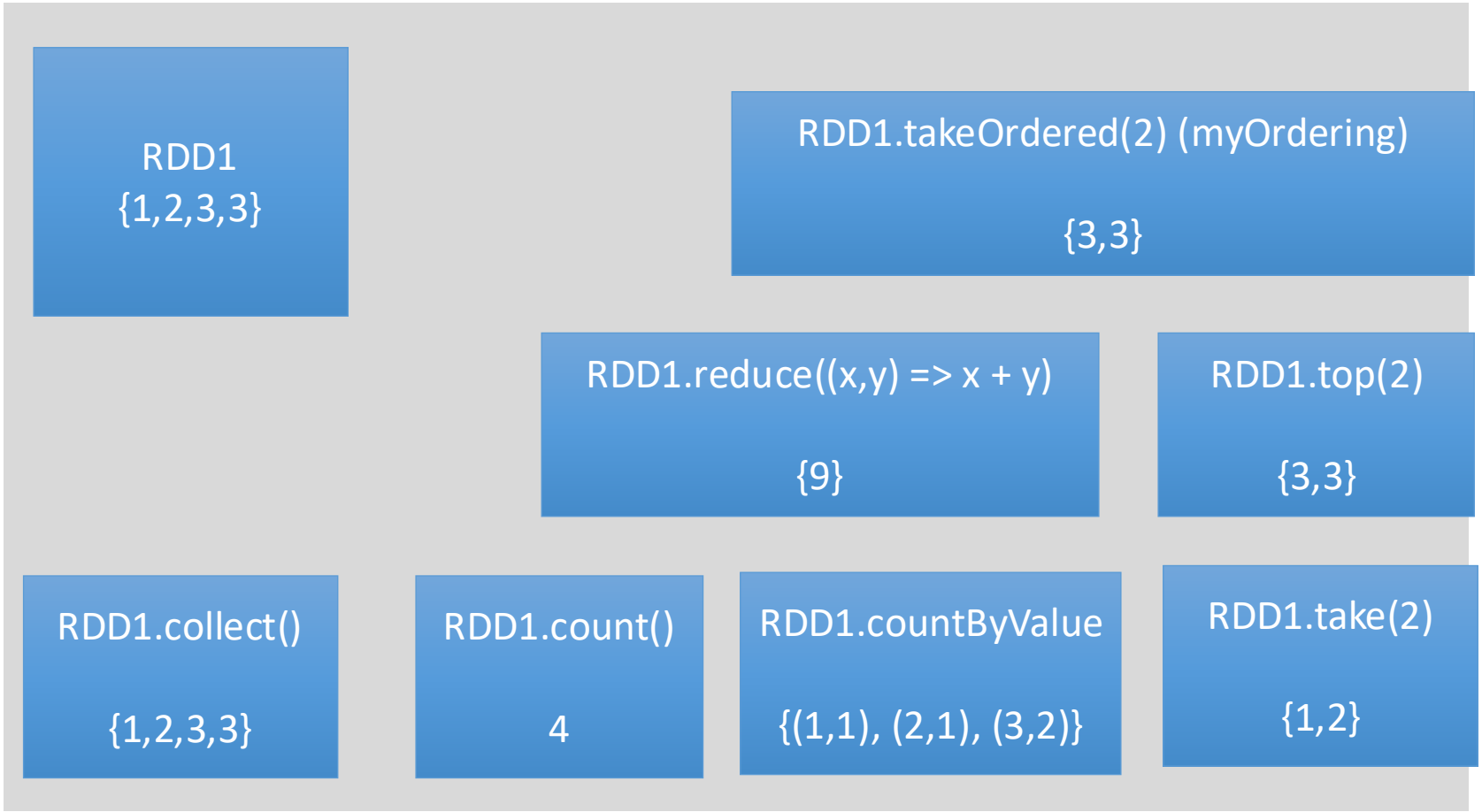
Transformation	Meaning
join(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key
cartesian(otherDataset)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)
intersection(otherDataset)	Return a new RDD that contains the intersection of elements in the source dataset and the argument
coalesce(numPartitions)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset
cogroup(otherDataset, [numTasks])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Iterable<V>, Iterable<W>) tuples

Sample Spark Actions

- **reduce(func):** Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- **take(n)** retrieve small number of elements in the RDD at the driver
- **collect():** Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
- **count():** Return the number of elements in the dataset.
- **saveAsTextFile()** or **saveAsSequenceFile()**

Remember: *Actions* cause calculations to be performed; *transformations* just set things up (lazy evaluation)

Actions


RDD1
{1,2,3,3}
RDD1.takeOrdered(2) (myOrdering)
{3,3}
RDD1.reduce((x,y) => x + y)
{9}
RDD1.top(2)
{3,3}
RDD1.collect()
{1,2,3,3}
RDD1.count()
4
RDD1.countByValue
{(1,1), (2,1), (3,2)}
RDD1.take(2)
{1,2}

Supported RDD Operations – Actions

Spark forces the calculations for execution only when actions are invoked on the RDDs

Action	Meaning
reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel
first()	Return the first element of the dataset (similar to take(1))
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator
count()	Return the number of elements in the dataset

Actions (Cont'd)

Action	Meaning
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
saveAsSequenceFile(path)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local file system, HDFS or any other Hadoop-supported file system
foreach(func)	Run a function func on each element of the dataset. This is usually done for side effects such as updating an accumulator variable
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key

Aggregate

- The aggregate function allows the user to apply two different reduce functions to the RDD.
- The first reduce function is applied within each partition to reduce the data within each partition into a single result.
- The second reduce function is used to combine the different reduced results of all partitions together to arrive at one final result.

Aggregate cont..

```
val result = input.aggregrate((0, 0))((acc, value) => (acc._1 + value, acc._2 + 1),
(acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))

val avg = result._1 / result._2.toDouble
```

Spark – RDD Persistence

- You can persist (cache) an RDD
- When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it)
- Allows future actions to be much faster (often >10x).
- Mark RDD to be persisted using the persist() or cache() methods on it. The first time it is computed in an action, it will be kept in memory on the nodes.
- Cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it
- Can choose storage level (MEMORY_ONLY, DISK_ONLY, MEMORY_AND_DISK, etc.)
- Can manually call unpersist()

Spark Data cache

Function	Argument	Details
cache	()	Caches an RDD reused without re-computing. The cache() method is a shorthand for using the default storage level, which is StorageLevel.MEMORY_ONLY .
persist	() ,(newLevel: StorageLevel)	Different StorageLevels can be seen in StorageLevel: • **MEMORY_ONLY** - Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level. • **MEMORY_AND_DISK**: Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed. Min recomputing. • **MEMORY_ONLY_SER**: Store RDD as serialized Java objects • **MEMORY_AND_DISK_SER:** Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed. • **DISK_ONLY:** Store the RDD partitions only on disk.
unpersist	()	Remove the persistent blocks of the RDD from the memory/disk

Advantages of the RDD Model

- Efficient fault recovery – fine-grained and low-overhead using lineage
- Immutable nature can mitigate stragglers – backup tasks to mitigate stragglers
- Graceful degradation when RAM is not enough

Supported RDD Operations – Transformations

- Transformations create a new dataset from an existing one
- All transformations in Spark are lazy: they do not compute their results right away
- Instead they remember the transformations applied to some base dataset
- This helps in:
  - Optimizing the required calculations
  - Recover from lost data partitions

Spark vs. Hadoop MapReduce

- Performance: Spark normally faster but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms MapReduce, but it often needs lots of memory to do well; if there are other resource-demanding services or can't fit in memory, Spark degrades
  - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
  - MapReduce is designed for data that does not fits in memory
- Ease of use: Spark is easier to program. However, Hadoop MapReduce is written in Java and is infamous for being very difficult to program. On the other hand, technologies like PIG, HIVE make it easier

Spark vs. Hadoop MapReduce

- Maturity: Spark maturing, Hadoop MapReduce mature
- MapReduce doesn't have an interactive mode like Spark.
- Staffing - Hadoop has been around since 2005, there is still a shortage of MapReduce experts out there on the market. What does this mean for Spark, which has only been around since 2010.
- Apache Spark can run as standalone or on top of Hadoop YARN or Mesos on-premise or on the cloud.
- Spark integration with BI tools via JDBC and ODBC is still a big challenge.
- Spark can do real-time processing as well as batch processing.

Spark vs. Hadoop MapReduce

- Apache Spark can do more than plain data processing: it can process graphs and use the existing machine-learning libraries.
- Failure Tolerance - MapReduce relies on hard drives, if a process crashes in the middle of execution, it could continue where it left off, whereas Spark will have to start processing from the beginning if cached data is lost.
- Security - Spark is a bit bare at the moment when it comes to security

Summary

- concept not limited to single pass map-reduce
- avoid soring intermediate results on disk or HDFS
- speedup computations when reusing datasets