VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA

AIML427: Big Data

Week 10: Apache Spark

Dr Qi Chen

School of Engineering and Computer Science

Victoria University of Wellington

Qi.Chen@ecs.vuw.ac.nz

Outline

- Apache Mahout and Hadoop Limitations.
- Spark components
- Spark architecture
- Transactions versus Actions
- Directed acyclic graph (DAG)
- Spark running modes
- Programming with Spark
  - SparkContext, SparkConf and SparkSession
  - Resilient distributed dataset (RDD)
  - Function creation
  - Data caching
  - Closure
  - Accumulator
  - Pair RDD
- How to run Spark programs on Hadoop cluster.

Apache Mahout

- Apache Mahout is a library of scalable machine-learning algorithms.
- It is originally implemented on top of

  Apache Hadoop using the MapReduce paradigm.
- Mahout algorithms:
  - primarily focused on the areas of collaborative filtering, clustering and classification.
  - also provides Java libraries for common maths operations (linear algebra and statistics) and primitive Java collections.

Hadoop Limitations

- HDFS can’t handle a large number of small files (< the HDFS block size, default is 128MB).
- Processing Speed – With parallel and distributed algorithm, MapReduce processes requires a lot of time to perform map and reduce tasks thereby increasing latency.
- Support only Batch Processing – not process streamed data.
- Vulnerable by nature – Hadoop is entirely written in Java which is most heavily exploited by cyber-criminal.
- Security- Hadoop is missing encryption at storage and network levels. Hadoop supports Kerberos authentication, which is hard to manage.
- Iterative processing is not efficient in Hadoop. As it does not support cyclic data flow where the input to the next stage is the output from the previous stage.

Mahout Samsara

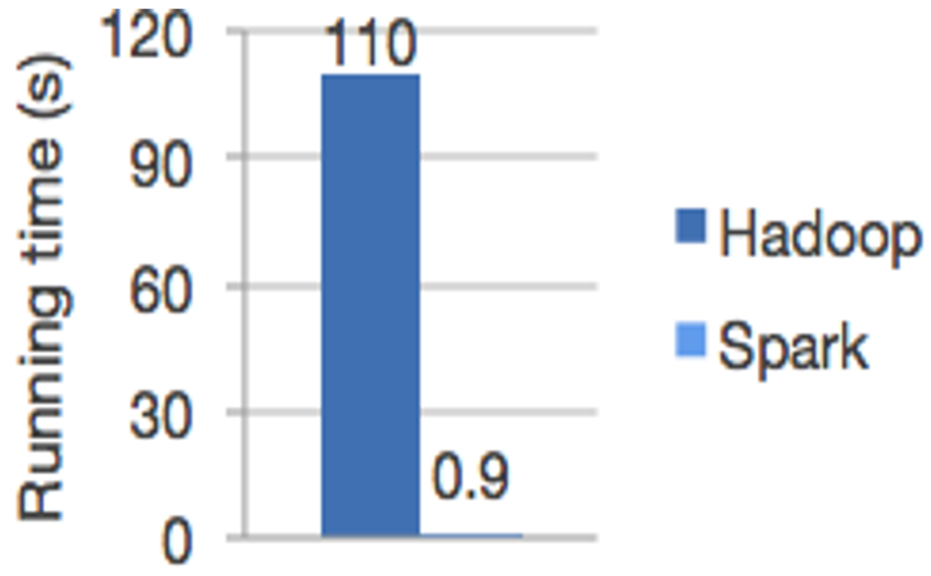
- Starting with the release 0.10.0, Mahout shifts its focus to building backend-independent programming environment, code named "Samsara".
  - http://mahout.apache.org/

Apache Spark

- Apache Spark is a fast, in-memory, big data processing, and general-purpose cluster computing framework
- Provides in-memory, fault tolerant data structure Resilient Distributed Datasets (RDDs).
- Flexible APIs in Scala, Java, Python, SQL and R.
- Supports sophisticated APIs for advanced data analytics.
- Originally developed at the University of California, Berkeley's AMPLab.
- Later donated to the Apache Software Foundation.

Spark's attractions

- Speed of computation
  - Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
- Ease of cluster computing and deployment with different cluster managers
  - Spark can run on Hadoop, Mesos, Kubernetes, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.



Spark’s attractions (cont.)

- Simplicity of data processing and computation
- Scalability and throughput across large-scale datasets
- Sophistication across diverse data types
- Working capabilities and supports with various big data storage and sources
- Multiple options and libraries: Graph, SQL, ML, Streaming
- Diverse APIs are written in widely used and emerging programming languages

python

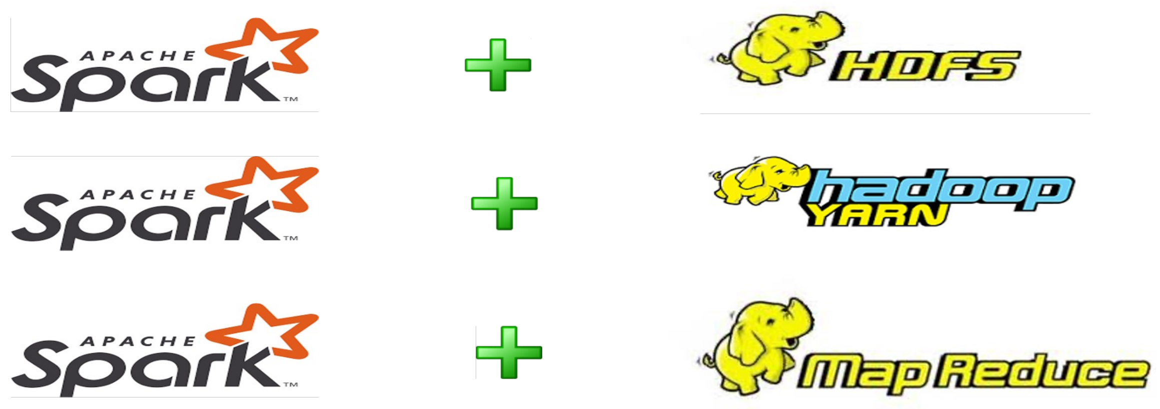
Java

Scala

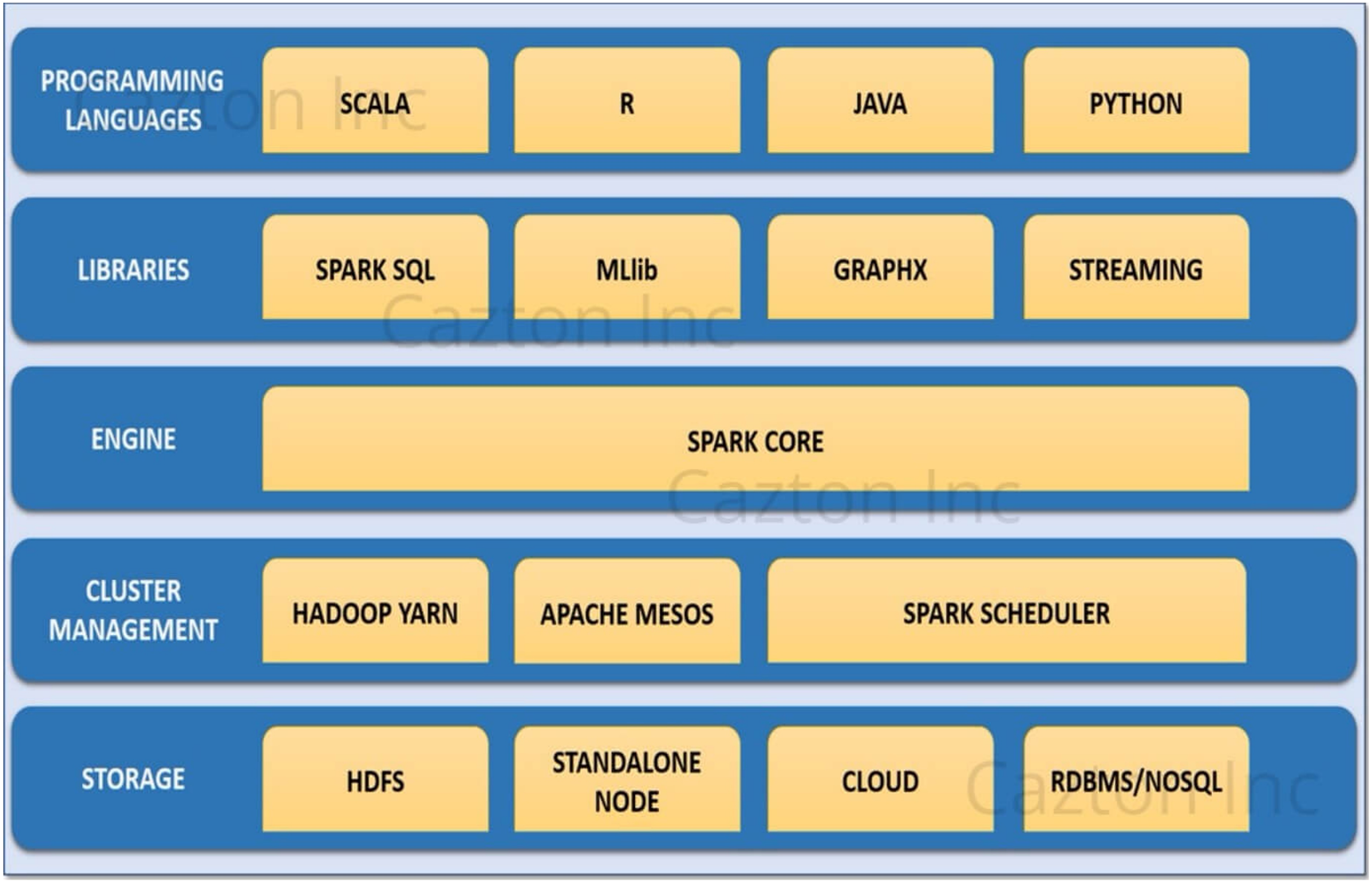
R

Spark and Hadoop

Hadoop	Spark
Stores data in local disk	Stores data in-memory
Slow speed due to a lot of read write from disk	Faster by reducing the number of disk read write operations.
Suitable for batch processing	Suitable for batch and real-time processing through Spark Streaming.
No built-in interactive mode	Has interactive mode (Scala)
MapReduce runs very well on commodity machines with standard amounts of memory.	Spark requires a lot of RAM to run in-memory.
Has a built-in distributed storage system (HDFS)	No built-in distributed storage system. Need to opt for a third party file organizing system.



Spark Ecosystem



https://www.cazton.com/consulting/big-data-development/apache-spark

Spark Core Component

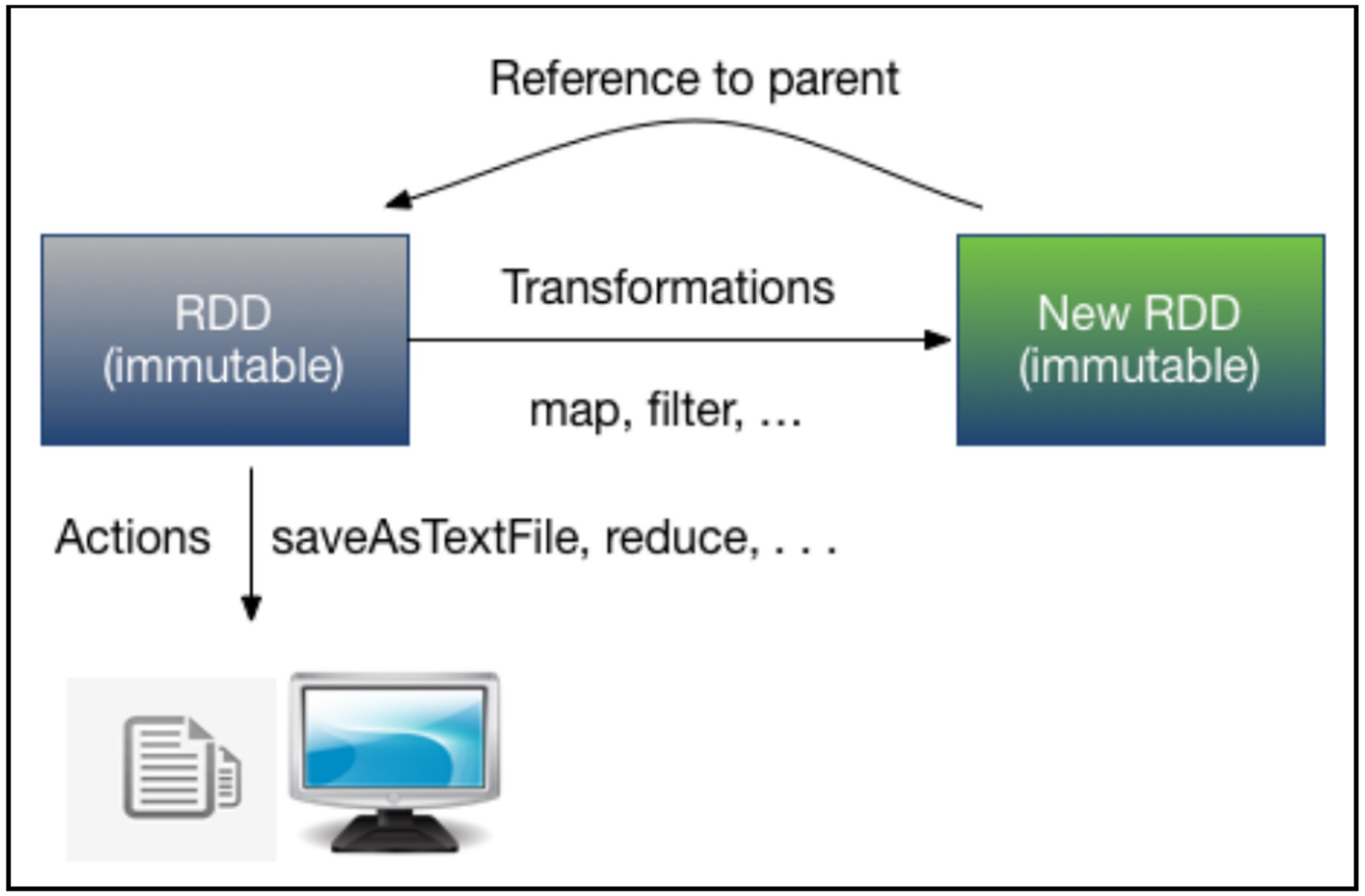
- Responsible for:
  - Memory management and fault recovery
  - Scheduling, distributing, and monitoring jobs
  - Interacting with storage systems.
- RDD: Support in-memory computation => overcomes MapReduce's drawback.
  - **R**esilient: if the data in memory (or on a node) is lost, it can be recreated
  - **D**istributed: data is chunked into partitions and stored in memory across the cluster as a single unit.
  - **D**ataset: initial data come from a file or created programmatically.
- RDDs are read-only and immutable. => allows them to be shared among all different processing systems.

Spark Other Components

- Spark SQL: for querying and processing large-scale structured data.
- SparkR for statistical computing that provides distributed computing using programming language R at scale.
- GraphX: for large-scale graph data processing.
- Spark Streaming: for handling large-scale real-time streaming data to provide a dynamic working environment to static machine learning.
- MLlib: implements machine learning algorithms such as clustering, regression, classification and collaborative filtering. From Spark 2.0, MLlib switches from RDD-based API to DataFrame-based API.

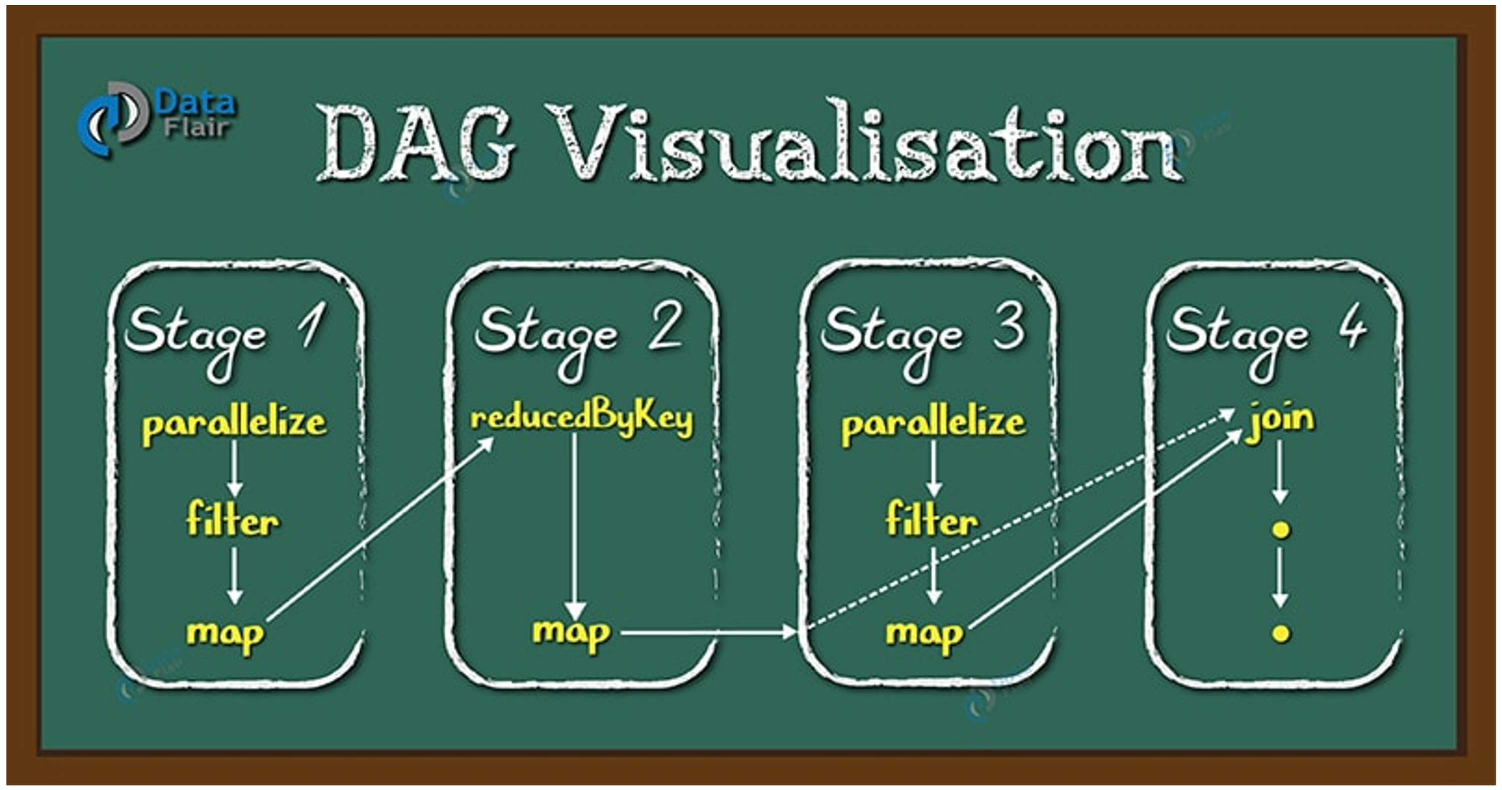
Two Types of Data Operations

- Transformation: return a new dataset from an existing one.
  - Narrow transformation: does not require the shuffling of data across a partition: map(), filter(), sample(), …
  - Wider transformation: groupByKey(), reduceByKey(), union(),…
- Action: return a value from an input dataset.
  - count(), first(), reduce(), saveAsTextFile(), …



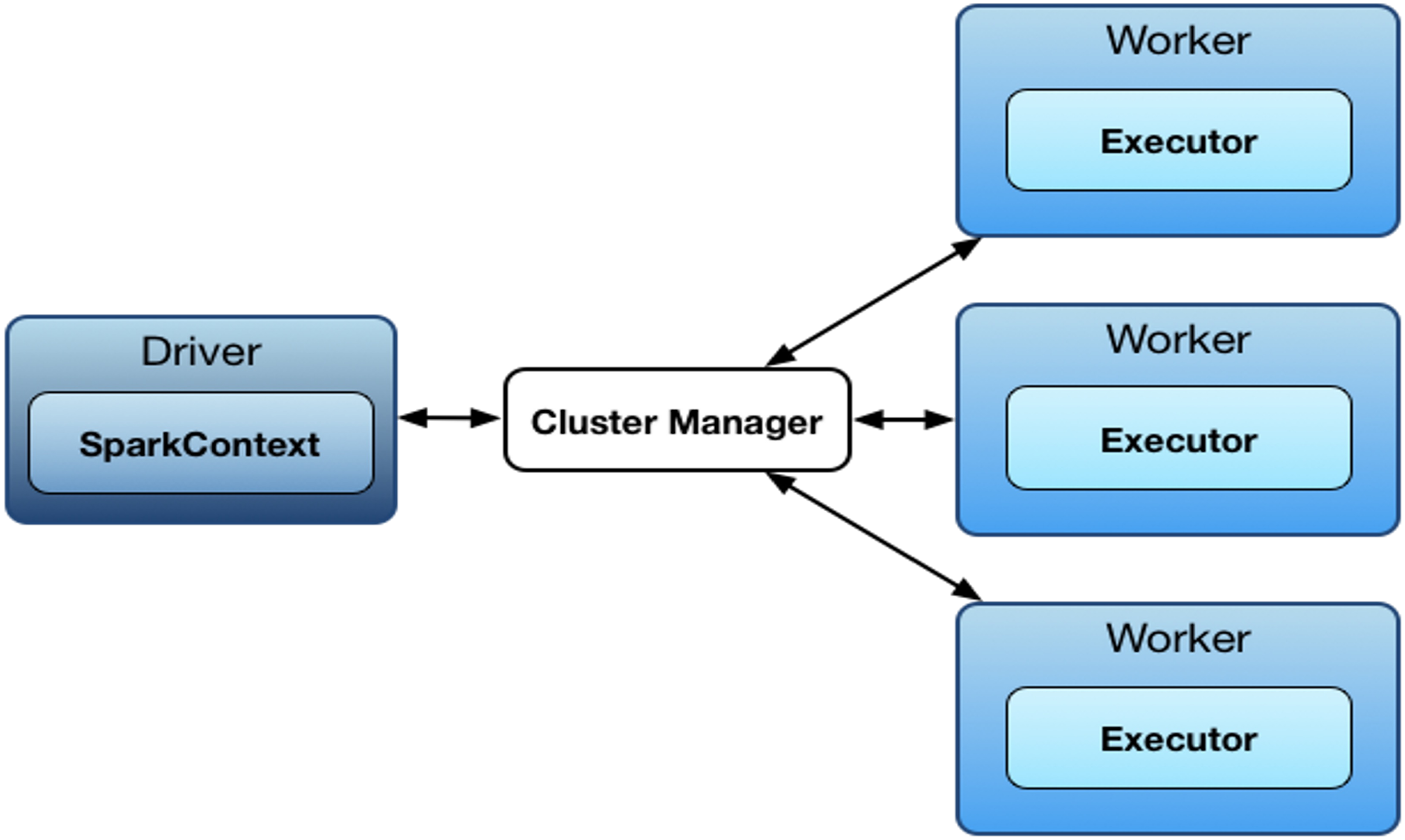
Spark Directed Acyclic Graph (DAG)

- In Spark, a DAG of consecutive computation stages is formed to optimise the execution plan.
- When an Action is called on Spark RDD at a high level, Spark submit the created DAG to DAG Scheduler which further splits the graph into the stages of the task.
- A stage contains tasks based on the partition of the input data. Wide transformation results in stage boundaries.
- The DAG scheduler pipelines operators together. The DAG Optimizer rearrange and combine operators wherever possible.
- The stages are passed on to the Task Scheduler. It launches task through cluster manager.



Spark Architecture

- Apache Spark follows a master/slave architecture with:
  - A single master running the Master/Driver process.
  - Any number of workers running the Slave Processes.
  - A cluster manager.



Spark Architecture (Cont.)

- Spark Driver – Master Node of a Spark Application
  1. runs the main() function of the application and is the place where the Spark Context is created.
  2. creates a logical plan DAG which is converted to physical execution plan with set of stages.
  3. create tasks under each stage to send to executors.
  4. schedules job execution, negotiates with the cluster manager.
  5. defines and stores the metadata about all the RDDs and their partitions distributed on the cluster.
  6. sends tasks to the cluster manager based on data placement after the cluster manager launches executors on the worker nodes on behalf of the driver.
  7. monitors all the executors and tasks.
  8. exposes the information about the running spark application through a Web UI.

Spark Architecture (Cont.)

- Spark Executor - a distributed agent responsible for the execution of tasks.
  - registers themselves with the driver program
  - performs tasks.
  - interacts with the storage systems. Reads from and writes data to external sources.
  - stores the computation results data in-memory, cache or on hard disk drives.

- Cluster Manager - an external service responsible for
  - acquiring resources on the spark cluster.
  - allocating them to a spark job.

Launching Applications In Different Modes

- A Spark application can be run either:
  - locally (on a single JVM)
  - clustered
    - Spark Standalone: Spark's own built-in clustered environment.
    - Spark on Apache Mesos
    - Spark on Hadoop YARN
- Launching application using bin/spark-submit:

```
./bin/spark-submit \
  --class <main-class> \          % Entry point for the application
  --master <master-url> \         % The master URL for the cluster
  --deploy-mode <deploy-mode>
  --conf <key>=<value> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

Launching Applications In Different Modes (Cont.)

- --deploy mode: specify where the driver process runs.
  - "cluster" mode: the framework launches the driver in a worker node of the cluster.
  - "client" mode (default): the submitter launches the driver locally as an external client.
- Examples:

```
# Run application locally on 8 cores
./bin/spark-submit  --class org.apache.spark.examples.SparkPi
 --master local[8]          /path/to/examples.jar 100

# Run on a Spark standalone cluster in client deploy mode
./bin/spark-submit   --class org.apache.spark.examples.SparkPi
--master spark://207.184.161.138:7077   --executor-memory 20G
--total-executor-cores 100   /path/to/examples.jar 1000
```

Launching Applications In Different Modes (Cont.)

```
# Run on a YARN cluster
export HADOOP_CONF_DIR=XXX
./bin/spark-submit
  --class org.apache.spark.examples.SparkPi
  --master yarn
  --deploy-mode cluster  # can be client for client mode
  --executor-memory 20G
  --num-executors 50      %limit amount of cores used by all executors
     /path/to/examples.jar  1000
```

- In the cluster deploying mode, --supervise is used to make sure that the driver is automatically restarted if it fails with a non-zero exit code

```
# Run on a Spark standalone cluster in cluster deploy mode with supervise
./bin/spark-submit  --class org.apache.spark.examples.SparkPi
  --master spark://207.184.161.138:7077
  --deploy-mode cluster  --supervise
  --total-executor-cores 100  %limit amount of cores used by all executors
     /path/to/examples.jar   1000
```

https://spark.apache.org/docs/latest/submitting-applications.html

SparkContext

- The first thing a Spark program must do is
  - to create a SparkContext object, which tells Spark how to access a cluster.
- To create a SparkContext:
  - A SparkConf object should be created to contain information about your application.
  - http://spark.apache.org/docs/latest/configuration.html

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

SparkConf conf = new SparkConf().setMaster("local").setAppName("My App");
JavaSparkContext sc = new JavaSparkContext(conf);
```

  - *local* is a special value that runs Spark on one thread on the local machine, without connecting to a cluster.
  - AppName is to identify your application on the cluster manager's UI if you connect to a cluster.

Spark Session

- SparkContext is a Scala implementation entry point
- JavaSparkContext is a java wrapper of sparkContext.
- Since Spark 2.x.x, SparkSession is the unified entry point of Spark.
- However, we can still convert SparkSession to SparkContext
  - sparkSession.sparkContext()

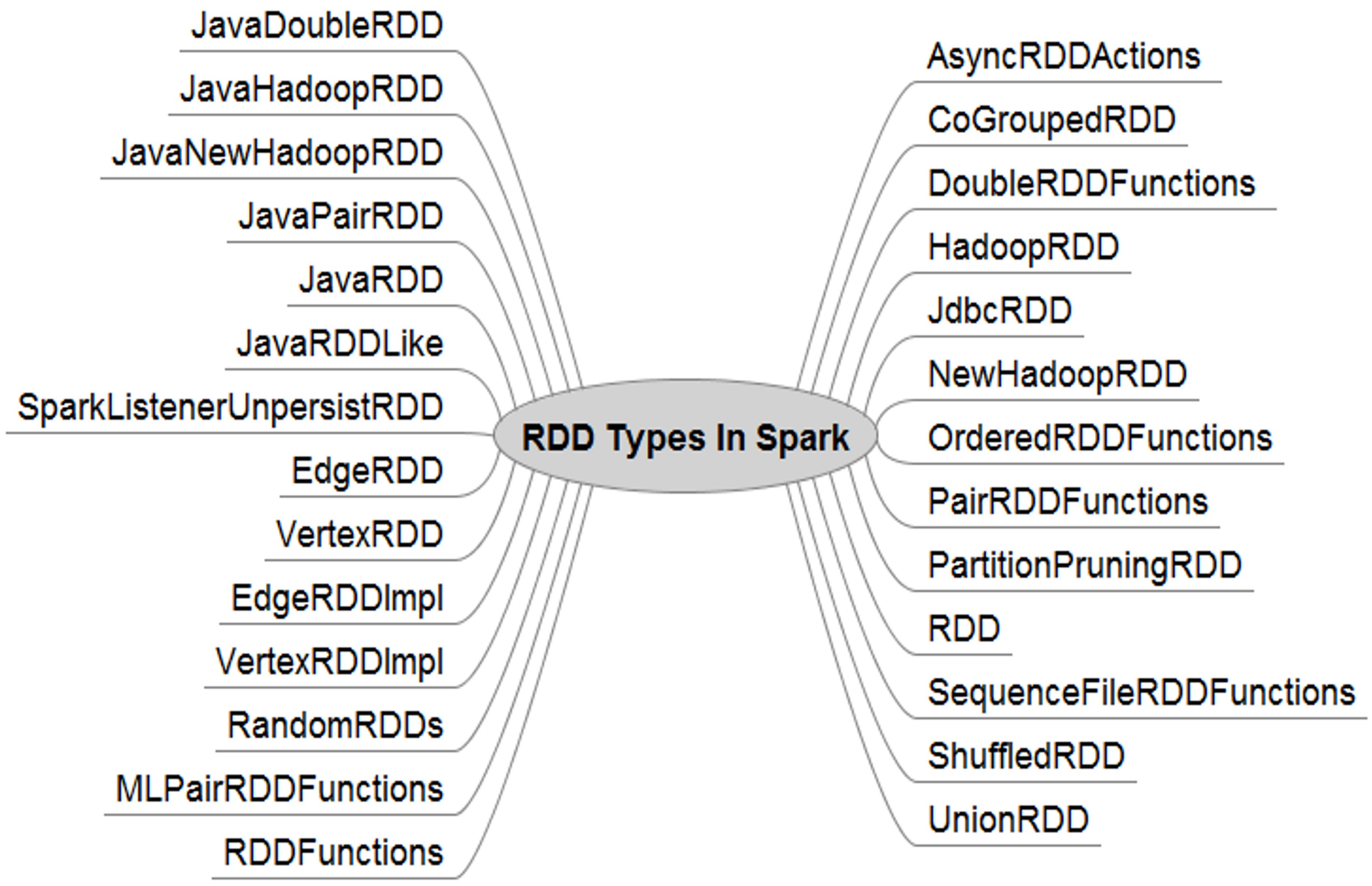
```
import org.apache.spark.sql.SparkSession;

SparkSession spark = SparkSession
        .builder()
        .appName("My App")
        .config("spark.master", "local[2]")
        .getOrCreate();
```

https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/sql/SparkSession.html

Resilient distributed dataset (RDD)

- RDD is the main abstraction Spark core provides.
- RDD is an immutable distributed collection of objects
  - It can not be changed once created.
  - It is split into multiple partitions (similar to splits in Hadoop), which may be computed on different nodes of the cluster.
  - The objects have the same type which can be any type including user-defined classes.
- When data files, blocks, or data structures are converted to RDDs, the data is:
  - broken down into partitions and
  - distributed among the nodes for parallel processing.

RDD Types


RDD Types In Spark
JavaDoubleRDD
JavaHadoopRDD
JavaNewHadoopRDD
JavaPairRDD
JavaRDD
JavaRDDLike
SparkListenerUnpersistRDD
EdgeRDD
VertexRDD
EdgeRDDImpl
VertexRDDImpl
RandomRDDs
MLPairRDDFunctions
RDDFunctions
AsyncRDDActions
CoGroupedRDD
DoubleRDDFunctions
HadoopRDD
JdbcRDD
NewHadoopRDD
OrderedRDDFunctions
PairRDDFunctions
PartitionPruningRDD
RDD
SequenceFileRDDFunctions
ShuffledRDD
UnionRDD

Creating RDDs

- Users create new RDDs in two ways:
  - Using textFile() to load data from text file, sequence file, Hadoop input format, CSV, TSV, TXT, MD, JSON, etc. in any storage source such as local file system, HDFS, etc.

```
JavaRDD<String> distFile = sc.textFile("data.txt");
```

  - From existing collection using parallelize().
    - this way is not widely used since it requires that you have your entire dataset in memory on one machine.

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

Define Functions - Method 1

- Create an anonymous inner class implementing the Function interfaces in org.apache.spark.api.java.function and pass an instance of it to Spark.

```
JavaRDD<String> lines = sc.textFile("data.txt");

JavaRDD<Integer> lineLengths = lines.map(
    new Function<String, Integer>() {
            public Integer call(String s) { return s.length(); }
    } );    //define map() to return the length of the given string.

int totalLength = lineLengths.reduce(
     new Function2<Integer, Integer, Integer>() {
            public Integer call(Integer a, Integer b) { return a + b; }
     } ); //define reduce() to return the sum of two given integers.
```

Standard Java Function Interfaces

Function name	**Method to implement**	**Usage**
Function<T, R>	R call(T)	Take in one input of type T and return one output of type R, for use with operations like map() and filter().
Function2<T1, T2, R>	R call(T1, T2)	Take in two inputs of type T1 and T2, and return one output of type R, for use with operations like aggregate() or fold().
FlatMapFunction <T, R>	Iterable<R> call(T)	Take in one input of type T and return zero or more outputs of type R, for use with operations like flatMap().

Define Functions - Method 2

- Create a named inner class implementing the Function interfaces and pass an instance of it to Spark.

```
class GetLength implements Function<String, Integer> {
  public Integer call(String s) { return s.length(); }
}

class Sum implements Function2<Integer, Integer, Integer> {
  public Integer call(Integer a, Integer b) { return a + b; }
}

JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(new GetLength());
int totalLength = lineLengths.reduce(new Sum());
```

Define Functions - Method 3

- Use lambda expressions to concisely define an implementation
  - supported from Java 1.8 (https://databricks.com/blog/2014/04/14/spark-with-java-8.html)

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

- Can you figure out in the above three lines of code:
  - which is a transformation operator?
  - which is an action operator?

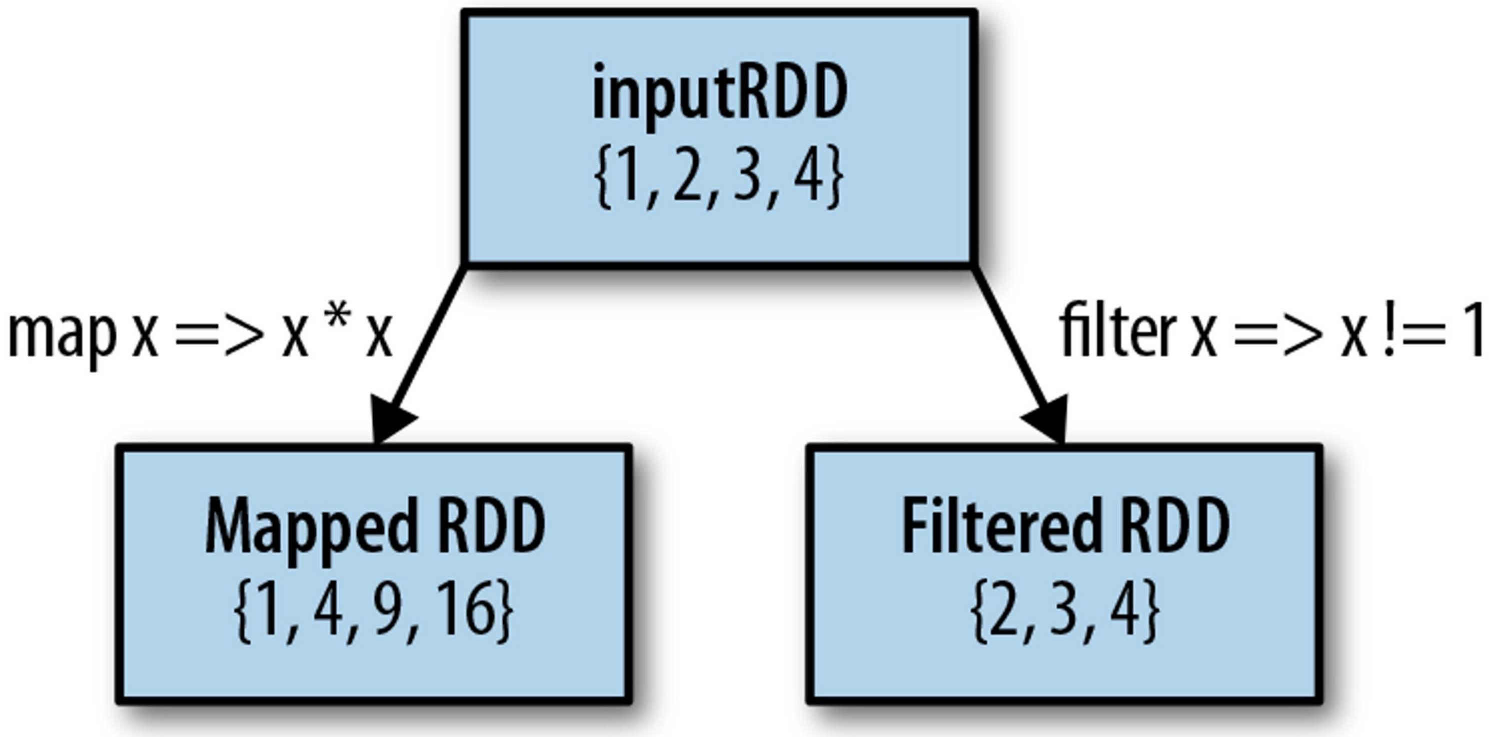
How does it works?

1. `JavaRDD<String> lines = sc.textFile("data.txt");`
   - dataset is not loaded in memory,
   - lines is merely a pointer to the file.
2. `JavaRDD<Integer> lineLengths = lines.map(s -> s.length());`
   - no immediately computation due to lazy evaluation.
3. `int totalLength = lineLengths.reduce((a, b) -> a + b);`
   - Computation is broken into tasks running on separate machines
   - Each machine runs its part of the map and a local reduction,
   - returning only its answer to the driver program.

- Note that after each action, the entire RDD must be computed "from scratch." => What happens if we want to have many actions?

map() and filter()

```
import org.apache.commons.lang.StringUtils;

JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));
JavaRDD<Integer> mapResult = rdd.map(x -> x*x );
JavaRDD<Integer> filterResult = rdd.filter(x -> x != 1 );
System.out.println(StringUtils.join(result.collect(), ","));
```



- Filter lines containing “error” in a log file.

```
JavaRDD<String> inputRDD = sc.textFile("log.txt");
JavaRDD<String> errorsRDD = inputRDD.filter( x -> x.contains("error"));
```

Transformations

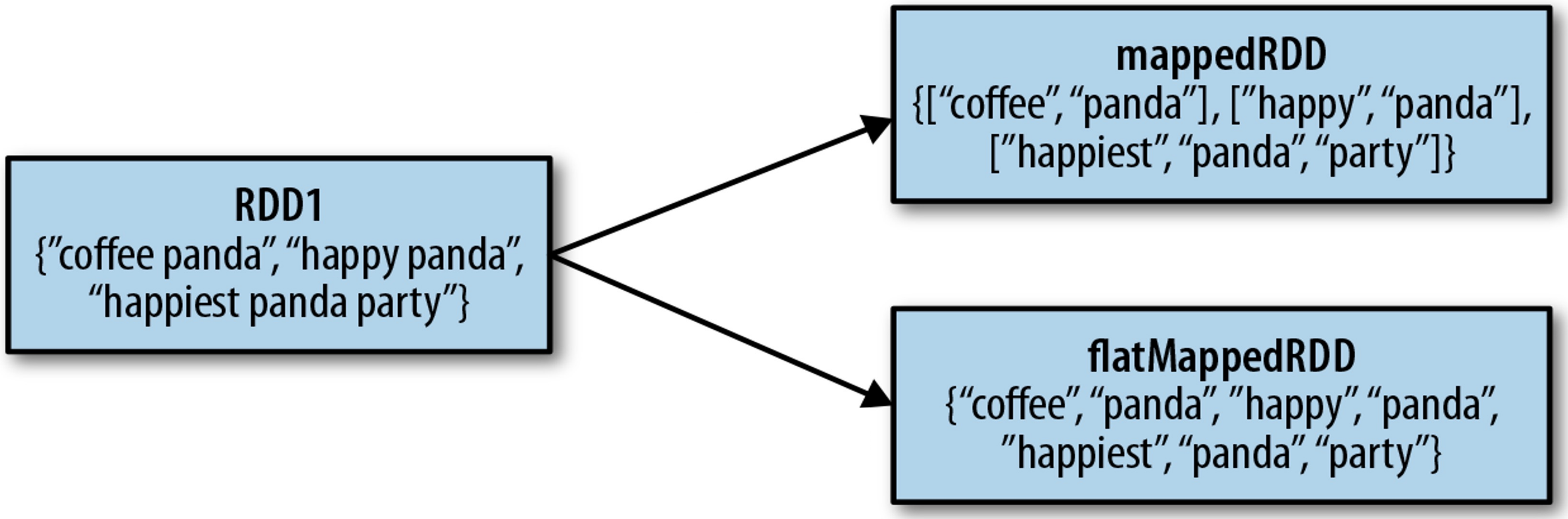
rdd: {1, 2, 3, 3}

Function name	Purpose	Example (Scala)	Result
map(func)	Apply a function to each element in the RDD and return an RDD of the result.	rdd.map (x => x + 1)	{2, 3, 4, 4}
flatMap(func)	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	rdd.flatMap(x => x.to(3))	{1, 2, 3, 2, 3, 3, 3}
filter(func)	Return an RDD consisting of only elements that pass the condition passed to filter().	rdd.filter (x => x != 1)	{2, 3, 3}
distinct()	Remove duplicates.	rdd.distinct()	{1, 2, 3}
sample(withReplacement, fraction, [seed])	Sample an RDD, with or without replacement.	rdd.sample (false, 0.5, 123)	

map() and flatMap()

- flatMap(): Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned.
  - "flattening" the iterators returned to it
  - outputs an RDD of the elements in the lists instead of an RDD of lists as in map().

```
    JavaRDD<String[]> mappedRDD = RDD1.map( s -> s.split(" "));
JavaRDD<String> flatMappedRDD = RDD1.flatMap( s -> Arrays.asList
                 (s.split(" ")).iterator() );
```


RDD1
{"coffee panda", "happy panda", "happiest panda party"}
mappedRDD
{["coffee", "panda"], ["happy", "panda"], ["happiest", "panda", "party"]}
flatMappedRDD
{"coffee", "panda", "happy", "panda", "happiest", "panda", "party"}

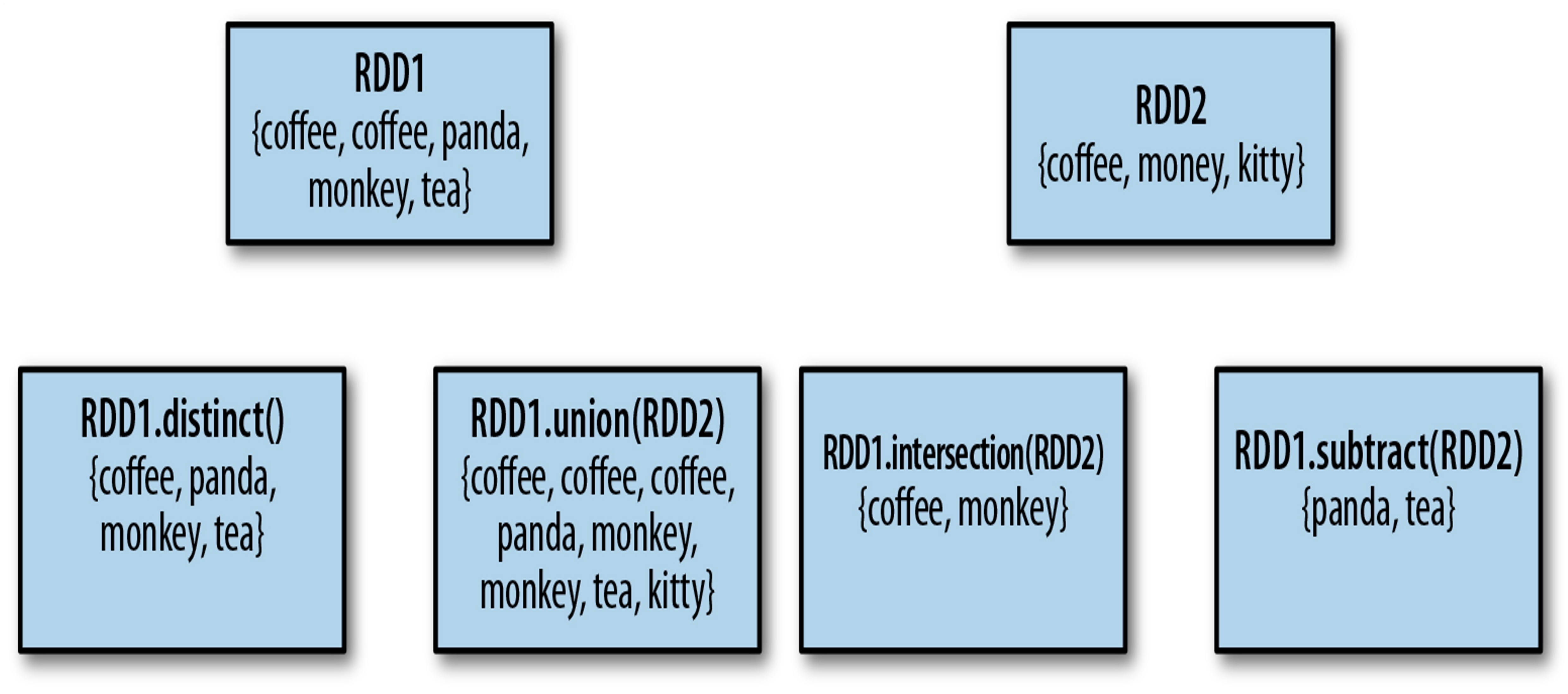
Transformations (Cont.)

rdd: {1, 2, 3, 3}

other: {3, 4, 5}

Function name	Purpose	Example (Scala)	Result
union(RDD)	Produce an RDD containing elements from both RDDs.	rdd.union(other)	{1, 2, 3, 3, 4, 5}
intersection (RDD)	RDD containing only elements found in both RDDs.	rdd.intersection(other)	{3}
subtract (RDD)	Remove the contents of one RDD (e.g., remove training data).	rdd.subtract(other)	{1, 2}
cartesian (RDD)	Cartesian product with the other RDD.	rdd.cartesian(other)	{(1, 3), (1, 4), … }

Pseudo Set Operations



Actions

rdd: {1, 2, 3, 3}

Function name	Purpose	Example (Scala)	Result
collect()	Return all elements from the RDD.	rdd.collect()	{1, 2, 3, 3}
count()	Number of elements in the RDD.	rdd.count()	4
countByValue()	Number of times each element occurs in the RDD.	rdd.countByValue()	{(1, 1), (2, 1), (3, 2)}
take(num)	Return num elements from the RDD.	rdd.take(2)	{1, 2}
top(num)	Return the top num elements the RDD.	rdd.top(2)	{3, 3}
reduce(func)	Combine the elements of the RDD together in parallel (e.g., sum).	rdd.reduce((x, y) => x + y)	9

Actions (Cont.)

rdd: {1, 2, 3, 3}

Function name	Purpose	Example (Scala)	Result
aggregate(zeroValue)(seqOp, combOp)	Similar to reduce() but used to return a different type.	`rdd.aggregate((0, 0)) ( (x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2) )`	(9, 4)
takeOrdered(num)(ordering)	Return num elements based on provided ordering.	rdd.takeOrdered(2)(Ordering [Int].reverse)	{3, 3}
takeSample(withReplacement, num, [seed])	Return num elements at random.	rdd.takeSample(false, 1, 1234)	{3, 1}
foreach(func)	Apply the provided function to each element of the RDD.	rdd.foreach(print)	Print whole rdd on screen) (may be in different order)

Printing an RDD

- The collect() method brings the RDD to the driver node.
  - `System.out.println(StringUtils.join(result.collect(), ","));`
  -
    ```
    for(String line : lines.collect() ){
            System.out.println("* "+line);
        }
    ```
- However, be careful, collect() can cause the driver to run out of memory, because collect() fetches the entire RDD to a single machine;
- If you only need to print a few elements of the RDD, a safer approach is to use the take():
  -
    ```
    for(String line : lines.take(100) ){
            System.out.println("* "+line);
        }
    ```

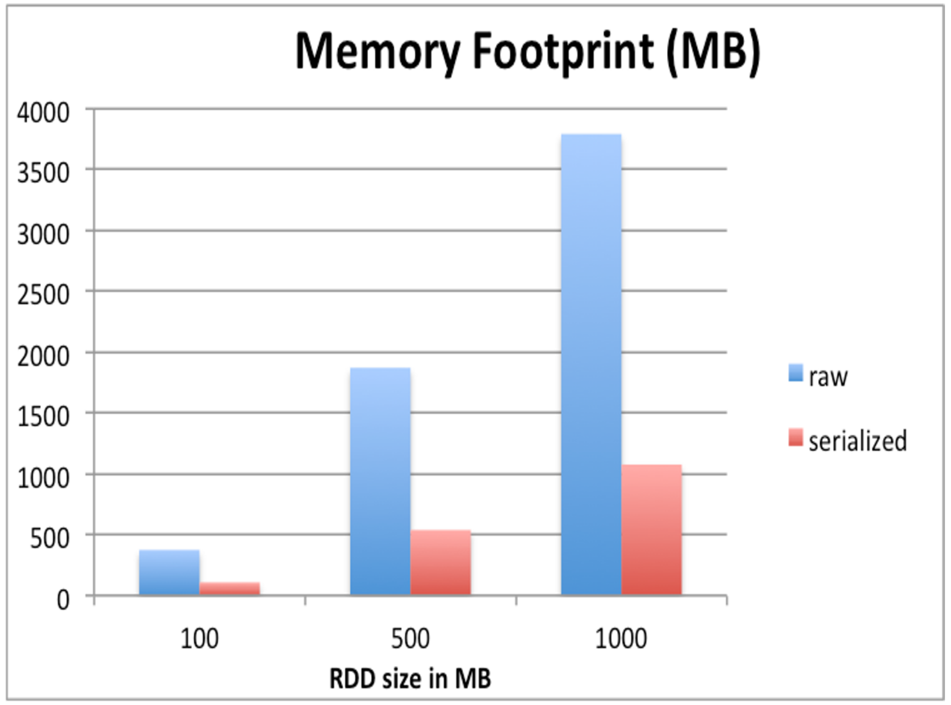
Data Caching (Persistence)

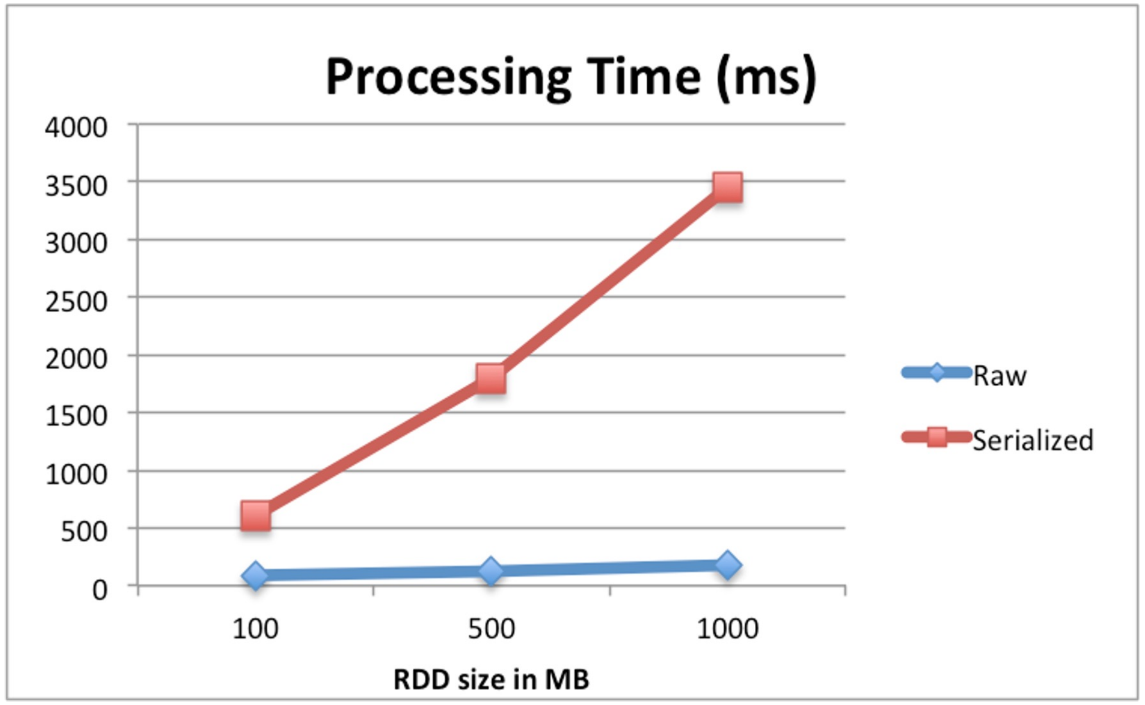
- Data caching is used to avoid recompute the RDD and all of its dependencies each time we call an action on the RDD.
- There are two options for caching in Spark:
  - Raw storage
  - Serialized: only for Java and Scala.

Raw caching	**Serialized Caching**
Pretty fast to process	Slower processing than raw caching
Can take up 2x-4x more space. E,g.: 100MB data cached could consume 350MB memory	Overhead is minimal
can put pressure in JVM and JVM garbage collection	less pressure
usage: rdd.persist(StorageLevel.MEMORY_ONLY) or rdd.cache()	usage: rdd.persist(StorageLevel.MEMORY_ONLY_SER)

Data Caching (Cont.)

- Raw caching is good for:
  - small data sets (few hundred MB) although consume more memory but not much pressure on Java garbage collection.
  - iterative work loads because the processing is very fast.
- Serialized is good for:
  - medium / large data sets (10s of Gigs or 100s of Gigs) because of its small memory requirement and garbage collecting GBs of memory can be costly.





Storage Levels For RDD Persistence

Level	**Space used**	**CPU time**	**In memory**	**On disk**
MEMORY_ONLY (default)	High	Low	Y	N
MEMORY_ONLY_SER (*)	Low	High	Y	N
MEMORY_AND_DISK	High	Medium	Some	Some
MEMORY_AND_DISK_SER (*)	Low	High	Some	Some
DISK_ONLY	Low	High	N	Y

(*): only for Java and Scala.

- Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion.
- Manually remove an RDD: rdd.`unpersist()`

Closure

- The closure is those 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.
- foreach() and counter are closures in this example:

```
int counter = 0;
JavaRDD<Integer> rdd = sc.parallelize(data);
rdd.foreach(x -> counter += x);
println("Counter value: " + counter);
```

- Since the executors only see the copy from the serialized closure, the behavior of the above code is undefined.
- In general, closures - constructs like loops or locally defined methods, should not be used to mutate some global state.
- To ensure well-defined behavior in these sorts of scenarios one should use an Accumulator.

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.

```
LongAccumulator accum = jsc.sc().longAccumulator();
    sc.parallelize(Arrays.asList(1, 2, 3, 4)).foreach(x ->
accum.add(x));
    accum.value();
```

- Note that Accumulators do not change the lazy evaluation model of Spark.
- Accumulators are used only in actions to ensure each task update to each accumulator only once.

Pair RDDs

- Pair RDDs are RDDs containing key/value pairs.
- useful building block in many programs, as they expose operations that allow you to act on each key in parallel or regroup data across the network.
- In Java, key-value pairs are represented by the JavaPairRDD class using the scala.Tuple2 class from the Scala standard library.
- To create a tupe: new Tuple2(a, b)
- To access its fields use ._1() and ._2() methods.

Create PairRDD Examples

- Creating a pair RDD using the first word as the key

# count the number of occurences per key (key is first word)

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaPairRDD<String, Integer> pairs = lines.mapToPair
                                  (s -> new Tuple2(s.split(" ")[0], 1);
JavaPairRDD<String, Integer> counts = pairs.reduceByKey((a, b) -> a + b );
```

- Creating a pair RDD using the whole line as the key

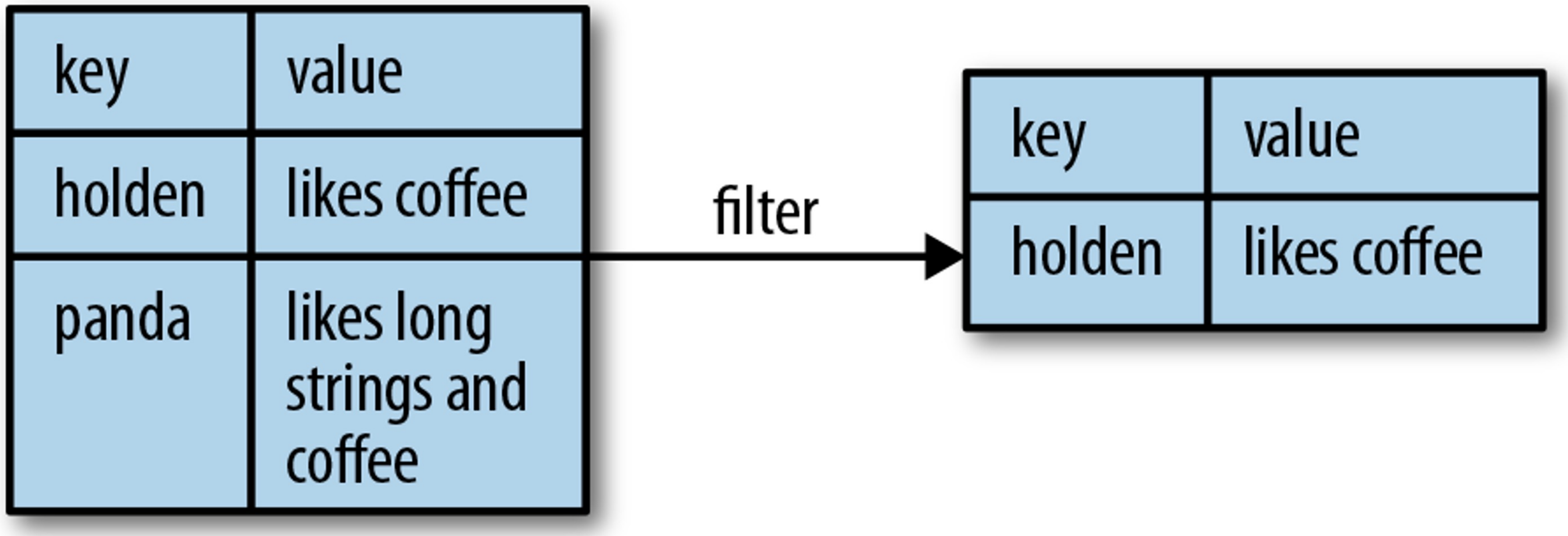
# count the number of same lines in a file

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaPairRDD<String, Integer> pairs = lines.mapToPair(s -> new Tuple2(s, 1) );
JavaPairRDD<String, Integer> counts = pairs.reduceByKey((a, b) -> a + b );
```

Example 1: A simple filter

- Filter out pairs with long values.

```
Function<Tuple2<String, String>, Boolean> longWordFilter =
  new Function<Tuple2<String, String>, Boolean>() {
    public Boolean call(Tuple2<String, String> keyValue) {
      return (keyValue._2().length() < 20);
    }
  };
JavaPairRDD<String, String> result = pairs.filter(longWordFilter);
```



key	value
holden	likes coffee
panda	likes long strings and coffee

key	value
holden	likes coffee

Example 2: Word Count

- Count the occurences of each word

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);
counts.saveAsTextFile("hdfs://...");
```

- Count the number of words in a file.
- Find the distinct words in a file.
- Count the distinct words in a file.

Java Interfaces For Type-Specific Functions

Function name	Method to implement	Usage
PairFlatMapFunction<T, K, V>	Iterator <Tuple2<K,V>> call(T t)	Return zero or more key-value pair records of type <K,V> from each input record of type T. (Eg. flatMapToPair)
PairFunction <T, K, V>	Tuple2<K,V> call(T t)	Returns key-value pairs of type Tuple2<K, V> from each input record. (Eg. mapToPair)
DoubleFlatMapFunction<T>	Iterator<Double> call(T t)	Returns zero or more records of type Double from each input record. (Eg. flatMapToDouble)
DoubleFunction<T>	double call(T t)	A function that returns Doubles from each input record. (Eg. mapToDouble)

Transformations on one pair RDD

- Example pair RDD: {(1, 2), (3, 4), (3, 6)}

Function name	Purpose	Example	Result
reduceByKey (func)	Combine values with the same key.	rdd.reduceByKey ((x, y) => x + y)	{(1, 2), (3, 10)}
groupByKey()	Group values with the same key.	rdd.groupByKey()	{ (1, [2]), (3, [4, 6]) }
mapValues (func)	Apply function to each value of RDD without changing the key.	rdd.mapValues (x => x+1))	{(1, 3), (3, 5), (3, 7)}
flatMapValues (func)	Apply function that returns an iterator to each value of RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	rdd.flatMapValues (x => (x to 5))	{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}
keys	Return an RDD of just the keys.	rdd.keys	{1, 3, 3}
values	Return an RDD of just the values.	rdd.values	{2, 4, 6}
sortByKey()	Return an RDD sorted by the key.	rdd.sortByKey()	{(1, 2), (3, 4), (3, 6)}

Transformations on Two Pair RDDs

rdd: {(1, 2), (3, 4), (3, 6)}

other: {(3, 9)}

Function name	Purpose	Example	Result
subtractByKey	Remove elements with a key present in the other RDD.	rdd.subtractByKey(other)	{(1, 2)}
join	Perform an inner join between two RDDs.	rdd.join(other)	{ (3, (4, 9)), (3, (6, 9))}
rightOuterJoin	Perform a join between two RDDs where the key must be present in the first RDD.	rdd.rightOuterJoin(other)	{(3,(Some(4),9)), (3,(Some(6),9))}
leftOuterJoin	Perform a join between two RDDs where the key must be present in the other RDD.	rdd.leftOuterJoin(other)	{(1,(2,None)), (3,(4,Some(9))), (3,(6,Some(9)))}
cogroup	Group data from both RDDs sharing the same key.	rdd.cogroup(other)	{(1,([2],[])), (3,([4, 6],[9]))}

Run Spark on Hadoop

- Unzip the package into /local/scratch
  - Tar -zxvf <zipfile.tgz>
  - ls /local/scratch/spark-2.3.0-bin-hadoop2.7/
- Set environment variables:
  - setenv HADOOP_VERSION 2.8.0
  - setenv HADOOP_PREFIX /local/Hadoop/hadoop-$HADOOP_VERSION
  - setenv SPARK_HOME /local/scratch/spark-2.3.0-bin-hadoop2.7/
  - setenv PATH ${PATH}:$HADOOP_PREFIX/bin:$SPARK_HOME/bin
  - setenv HADOOP_CONF_DIR $HADOOP_PREFIX/etc/hadoop
  - setenv YARN_CONF_DIR $HADOOP_PREFIX/etc/hadoop
  - setenv LD_LIBRARY_PATH $HADOOP_PREFIX/lib/native:$JAVA_HOME/jre/lib/amd64/server
  - need java
  - source ~/setup_hadoop_classpath.csh

Run Spark on Hadoop (Cont.)

- Create a folder to save the java program file and a lib folder for all the jar files needed to compile the program.
- Go to the new folder
- Create a folder for the classes
  - mkdir sparkwordcount_classes
- When compile your program, specify where the jar files.
  - javac -cp "lib/*" -d sparkwordcount_classes JavaWordCount.java
  - jar cvf JavaWordCount.jar -C sparkwordcount_classes/ .
- Submit the job
  - spark-submit --class "mySpark.sparkWordCount.JavaWordCount" --master yarn --deploy-mode cluster JavaWordCount.jar /user/tranbinh/File1.txt /user/tranbinh/File1out

References

- Learning Spark (Book 2015) https://www.safaribooksonline.com/library/view/learning-spark/9781449359034/