Introduction to Apache Spark

pm jat @ daiict

SQL interface over MR

- Database users are used to SQL
- For database operations like SELECT, PROJECT, and JOIN programming in Map Reduce is definitely not a pleasure!
  - too much of programming, and becomes complex too.
- Apache Hive is available as SQL abstractions over Map Reduce!
- Hive, originally created at Facebook, now available as Apache Project!
- We are not going to learn Hive in this course! We shall, however look into **Spark-SQL** which is built on top of Spark, and becoming more popular!

Issues with Map Reduce

Iterative jobs:

- Many common machine learning algorithms require many iteration through a dataset.
- For example: Logistic Regression, k-Means, Page Rank algorithms, etc.
- Map Reduce **reads the data from storage for each iteration**; this turns out to be very inefficient.

Issues with Map Reduce

Interactive analysis:

- Back Reduce runs in batch, an that may take more than required time.
- **Map reduce does not have any kind of caching.**
- We can not take partial results, we can run on sample data
- All this makes map-reduce unsuitable for interactive analysis
- Even if we use higher level interfaces like “Pig” and “Hive”, it is map-reduce that runs under the hood

Issues with Map Reduce

- A survey paper [5] gives a discusses issues with basic map-reduce. Here we enumerate few of them:

(1) We can not process part of a file

- We always need to scan full file, and is inefficient if "**selectivity**" is low.

(2) **Lack of iteration**: If we require iterating through a dataset for multiple times, then every time we read data from disk files. and that happens to be the case with most Analytical and Machine Learning tasks.

Issues with Map Reduce

(3) Redundant and wasteful processing: Multiple MR jobs are processing same data almost at the same time. "Lack of Caching"

(4) The system lacks to "reuse" results of previously executed queries/jobs

(5) Lack of early termination - terminate of a job based on some condition is not possible.

(6) Quick retrieval of approximate results [for example if we want to process only 10% data from the file.

(7) Lack of interactive or real-time processing – Map-Reduce runs in background, and there is no interaction till it finishes the job.

Apache Spark

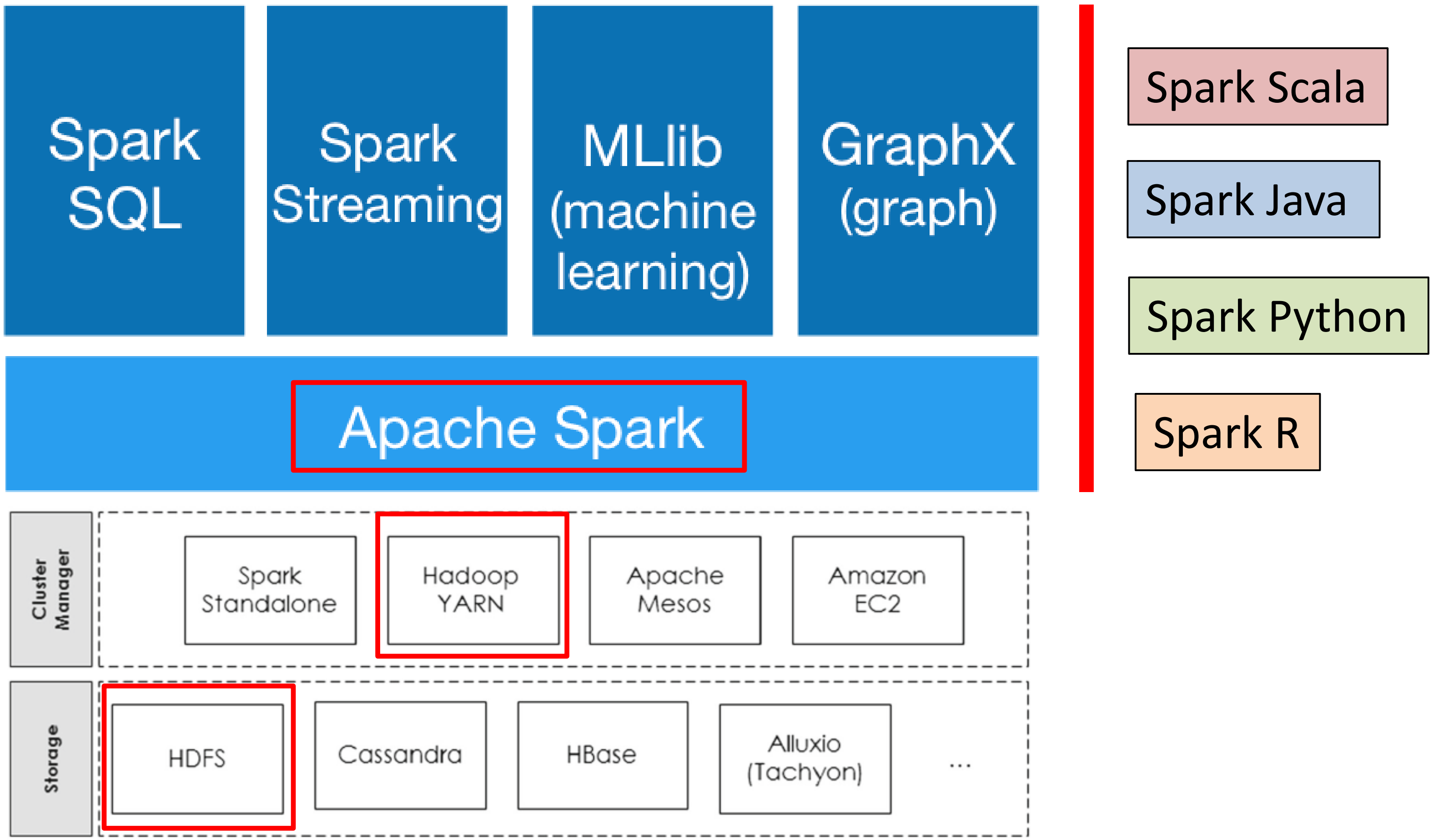
- **Spark** took birth at UC, Berkley, and was introduced by Matei Zaharia, et.al. in 2010 through paper "Spark: Cluster Computing with Working Sets" [1]

- **Spark** primarily has two revolutionary features

  1. Read data can be **kept in primary memory** (distributed on various computers), and **parallel processing** can be performed on them.
  2. Further simplified programming abstraction – **we only require writing drive programs**

- Today Apache Spark is <u>most popular framework for big data analytics</u> with over 1000 contributors

Apache Spark – framework*,[9]



* https://spark.apache.org/
[9] Salloum, Salman, et al. "Big data analytics on Apache Spark." *Internl Journal of Data Science and Analytics* (2016)

Apache Spark – some numbers

Fast and expressive cluster computing system interoperable with Apache Hadoop

Improves efficiency through:

» In-memory computing primitives
» General computation graphs

→ Up to 100× faster (2-10× on disk)

Improves usability through:

» Rich APIs in Scala, Java, Python
» Interactive shell

→ Often 5× less code

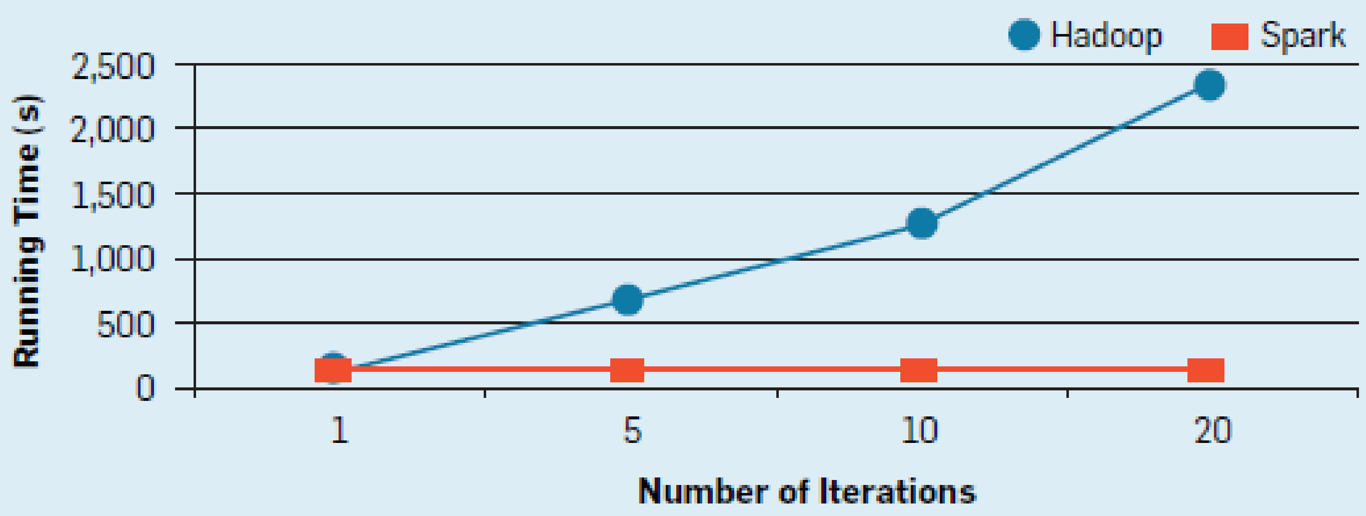
https://rxin.github.io/talks/2017-12-05_cs145-stanford.pdf

Apache Spark – some numbers

- Here is an experimental result for Logistic Regression on Spark in comparison to Map-Reduce.

  (More comparative study with MR is available in [3])

Figure 4. Performance of logistic regression in Hadoop MapReduce vs. Spark for 100GB of data on 50 `m2.4xlarge` EC2 nodes.



Source [2]: Zaharia, Matei, et al. "Apache spark: a unified engine for big data processing." *Communications of the ACM* 59.11 (2016): 56-65..

[3] Shi, Juwei, et al. "Clash of the titans: Mapreduce vs. spark for large scale data analytics." *Proceedings of the VLDB Endowment* 8.13 (2015): 2110-2121.

Apache Spark – some numbers

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

databricks

Source: Daytona GraySort benchmark, sortbenchmark.org

https://rxin.github.io/talks/2017-12-05_cs145-stanford.pdf

Apache Spark – some numbers

Programmability

```java
public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

```scala
val f = sc.textFile(inputPath)
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

WordCount in 50+ lines of Java MR

https://rxin.github.io/talks/2017-12-05_cs145-stanford.pdf

Spark "Word Count" - Java

- Various Type Information, makes it some what clumsy to read and understand
- Spark programs are "driver programs" only, and does not require writing mapper, reducer, combiner etc.

```
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://...");
```

RDDs here:
textFile (say list of String),
counts (list of String, Int) pairs

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

Spark “Word Count” - Scala

- Spark has been created in Scala; and Scala is more native kind of language for Spark
- Scala is “Functional Programming” Language; extensively uses concept of “Lambda expressions”; scala code is quiet compact

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

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

What is Spark - restate

- Spark is further “programming abstraction” on computing cluster, i.e.
  - Works on distributed data
  - Computation is done on several nodes in parallel
  - Has lots of inspiration from map-reduce

What is Spark - restate

- What is new (over map-reduce)
  - Abstraction is further simpler, we only write driver programs
  - There is concept of "Distributed Collection of (in memory) Objects". It is like distributed "array-list"; called as Resilient Distributed Dataset (RDD)
  - Simple abstraction to "manipulate distributed RDDs" through "**Transformations**" and "**actions**"
  - Some kind of "program execution optimization" are becoming available.

What is Spark - restate

- Following are two main revolutionary features that make Spark a amazing solution for cluster computing
  - “**In Memory**”, “**Distributed**”, “**Fault Tolerant**” collection of objects (called as RDD)
  - Simple Programming Abstractions;
- Spark offers a revolutionary programming paradigm that makes distributed programming like a desktop programming

Resilient Distributed Dataset (RDD)

- The main abstraction in Spark is **Resilient Distributed Dataset (RDD)**
- RDD are collection of distributed, fault tolerant "object collection" partitioned across a set of machines. (Note that "Object collection" here is "in memory")
- A simple analogy; a "Distributed, Fault Tolerant Array List"
- RDD objects can explicitly be cached in memory, and reused in consequent calls. This "in memory" processing is what makes Spark, amazing fast!
- For Fault Tolerance; RDD objects themselves are not replicated, but maintain information that a partition can be rebuilt if a node fails

Resilient Distributed Dataset (RDD)

- Characteristics of RDD:
  - a read-only,
  - partitioned collection of records.
  - Fault tolerant
  - Lazily evaluated
  - Can be cached
- RDDs can only be created by:

  (1) by reading from data file,

  (2) distributing (parallelization) a local collection to multiple nodes,

  (3) by applying a “transformations” on existing rdd

Construction RDDs

Three ways:

1. Read from file and build an RDD.

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

2. By Parallelization (distribute a collection object and put on multiple machines)

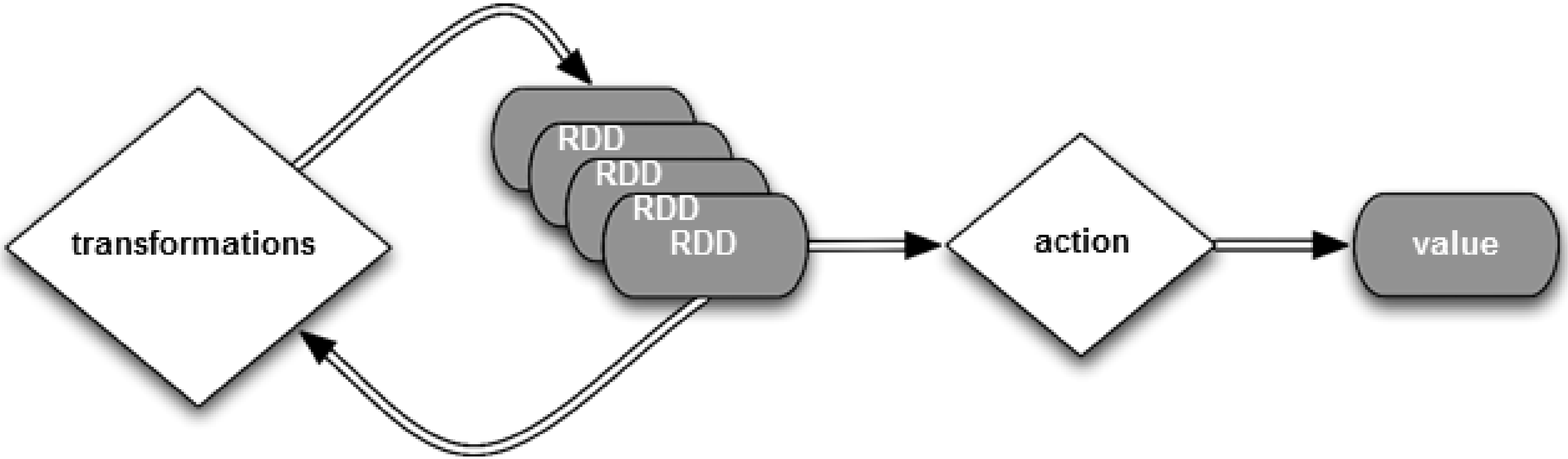
   ```
   List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
   JavaRDD<Integer> distData = sparkcontext.parallelize(data);
   ```

3. By transforming an existing RDD (shall see in a moment)

Operations on RDDs

- Operations
  - Transformations, and
  - Actions
- Recall: RDDs are immutable (read only), that is any operation does not change content of an RDD but generates new RDD or some value.

RDD Transformations and Actions



Operations on RDDs

- Transformations
  - map, filter, reduce, group-by, and join
- Actions
  - count (which returns the number of elements in the dataset),
  - collect (which returns the elements themselves), and
  - save (which saves on storage)
- Other operations
  - Persistence/Caching, and
  - “Partitioning”

A simple complete program!

```
public class SparkDemo {

    public static void main( String[] args ) {

        System.out.println( "Hello World!" );

        //setup Spark Context
        SparkConf sparkConf = new SparkConf()
                .setAppName("SparkWordCount")
                .setMaster("local"); //Local mode, alternatively CLuster mode
        JavaSparkContext sc = new JavaSparkContext(sparkConf);

        String path = "mm.csv";
        JavaRDD<String> lines = sc.textFile(path);

        System.out.println("Lines count: " + lines.count());

        sc.stop();
        sc.close();
```

Initialize Spark Context

input

count: action

Example #1 (word count in spark)

```
//setup Spark Context
SparkConf sparkConf = new SparkConf()
        .setAppName("SparkWordCount")
        .setMaster("local"); //Local mode, alternatively CLuster mode
JavaSparkContext sc = new JavaSparkContext(sparkConf);

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

Input

```
//following work is done in map funtion of MR job
JavaRDD<String> words = lines.flatMap(
        line -> Arrays.asList(line.split(" ")).iterator());
JavaPairRDD<String, Integer> word_maps
        = words.mapToPair(w -> new Tuple2<>(w, 1));
```

Map functionality

```
//following work is done in reduce function of M
JavaPairRDD<String, Integer> counts
        = word_maps.reduceByKey((x, y) -> x + y);

counts.saveAsTextFile("output/wordcount.txt");
```

Reduce functionality

output

Example #1 (word count in spark)

Output can be dumped on the file system or on console

```
//output on file system
counts.saveAsTextFile("output/wordcount.txt");

//outputs on console
List<Tuple2<String, Integer>> output = counts.collect();
for (Tuple2<?, ?> tuple : output)
    System.out.println(tuple._1() + ": " + tuple._2());
```

Spark "Word Count" - Scala

- Spark has been created in Scala; and Scala is more native kind of language for Spark
- Scala is "Functional Programming" Language; extensively uses concept of "Lambda expressions"; scala code is quiet compact

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

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

Example #2: grouped sum

```
SELECT dno, sum(salary) FROM employee group by dno;
```

```
public static void main(String[] args) throws Exception {

    SparkConf conf = new SparkConf().setAppName("firstSparkProject")
            .setMaster("local[*]");
    JavaSparkContext sc = new JavaSparkContext(conf);

    JavaRDD<String> lines = sc.textFile("data/employee.csv");

    JavaRDD<String[]> records = lines.map(line -> line.split(","));
    JavaPairRDD<String, Integer> salrecs
        = records.mapToPair(rec -> new Tuple2<>(rec[6], Integer.parseInt(rec[4])));
    JavaPairRDD<String, Integer> sums
        = salrecs.reduceByKey((x, y) -> x + y);

    sums.foreach( pair -> System.out.println(pair));
```

Map

Reduce

Output

Filter Operation

- Filter operation applies a filter condition on every element of operand rdd, and produces result as new rdd.

```
JavaRDD<String> lines = sc.textFile("data/log.txt");
JavaRDD<String> debugLines = lines.filter( line -> line.contains("DEBUG") );
debugLines.foreach( line -> System.out.println(line));
debugLines.saveAsTextFile("output/log_debug.txt");
```

“map” operation

- map(T, f) => U
- Specified function is applied on every element of input RDD, T, and produces another RDD U.
- Here T and U are types of elements of RDD.
- Cardinality of U remains same as T.
- An Example

```
JavaRDD<Integer> rdd = sc.parallelize( Arrays.asList(1, 2, 3, 4));
JavaRDD<Integer> result = rdd.map( x -> x*x );
```

map operation

- flatMap operations is similar to map, except that each element of operant element may add multiple elements in result rdd.

- Example:

```
JavaRDD<String> words = lines.flatMap(
        line -> Arrays.asList(line.split(" ")).iterator());
```

flatMap operation

- flatMap operations is similar to map, except that each element of operant element may add multiple elements in result rdd.

- Example:

```
JavaRDD<String> words = lines.flatMap(
        line -> Arrays.asList(line.split(" ")).iterator());
```

More operations

- Suppose we have an rdd: `rddX={(1, 2),(3, 4),(3,6)}`

`reduceByKey(func)`

:Combine values with the same key, as per the specified function.

- Example: `rddX.reduceByKey((x, y) => x + y)`
  Result: `{(1,2), (3,10)}`

`groupByKey()`

:Groups values with the same key.

- Example: `rddX.groupByKey()`
  Result: `{(1,[2]),(3, [4,6])}`

RDD Transformations[4]

- Transformations create a new RDD, and are "lazy operations"; i.e. computed only when required!
- Some of the may require shuffling of data: like reducebyKey, partition, sort, join etc.

$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
$sample(fraction : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
$union()$	:	$\mathrm{(RDD[T], RDD[T])} \Rightarrow \mathrm{RDD[T]}$
$join()$	:	$\mathrm{(RDD[(K, V)], RDD[(K, W)])} \Rightarrow \mathrm{RDD[(K, (V, W))]}$
$cogroup()$	:	$\mathrm{(RDD[(K, V)], RDD[(K, W)])} \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
$crossProduct()$	:	$\mathrm{(RDD[T], RDD[U])} \Rightarrow \mathrm{RDD[(T, U)]}$
$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$

You can get more concrete coverage at: http://spark.apache.org/docs/latest/rdd-programming-guide.html

RDD Actions[4]

- Actions launch a computation to return a value to the driver program or write data to external storage.

$count()$	:	$\text{RDD}[\text{T}] \Rightarrow \text{Long}$
$collect()$	:	$\text{RDD}[\text{T}] \Rightarrow \text{Seq}[\text{T}]$
$reduce(f : (\text{T},\text{T}) \Rightarrow \text{T})$	:	$\text{RDD}[\text{T}] \Rightarrow \text{T}$
$lookup(k : \text{K})$	:	$\text{RDD}[(\text{K}, \text{V})] \Rightarrow \text{Seq}[\text{V}]$ (On hash/range partitioned RDDs)
$save(path : \text{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

- More Actions:
  - take(n) //if collect can return huge list
  - takeOrdered(n, ordering_function)

You can get more concrete coverage at: http://spark.apache.org/docs/latest/rdd-programming-guide.html

Persistence and caching of RDD

- We can make RDDs to live only in primary memory, or on disk
- RDD API provides two methods for this persist, and cache.
- Cache tells RDD to be living only in primary memory, where as
- With persist, we can specify various methods of persistence. Typical values: MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY, MEMORY_ONLY_SER, and so.

Action Examples

- Code below shows “take”, “count” actions. Should be self explanatory.

```
JavaRDD<String> lines = sc.textFile("data/log.txt");
JavaRDD<String> debugLines = lines.filter( line -> line.contains("DEBUG") );

System.out.println( "Count - Debug Line: " + debugLines.count() );

List<String> top3 = debugLines.take(3);

top3.forEach( line -> System.out.println(line));
```

Example

- Let us say a tab separated data file called “**SalesProduct.txt**”, where attributes Name, and Weight at 2nd and 8th position respectively.

- Following Spark program (in python) lists (Name, Weight) of top 15 products in the descending order of their weight

```
dataFile = spark.read.text("SalesLTProduct.txt")
header = dataFile.first()
products = dataFile.filter(lambda line: line != header)
products.filter( lambda line: line.split("\t")[7] != "NULL")
    .map(lambda line: (line.split("\t")[1], float(line.split("\t")[7])))
                .takeOrdered(15, lambda x : -x[1])
```

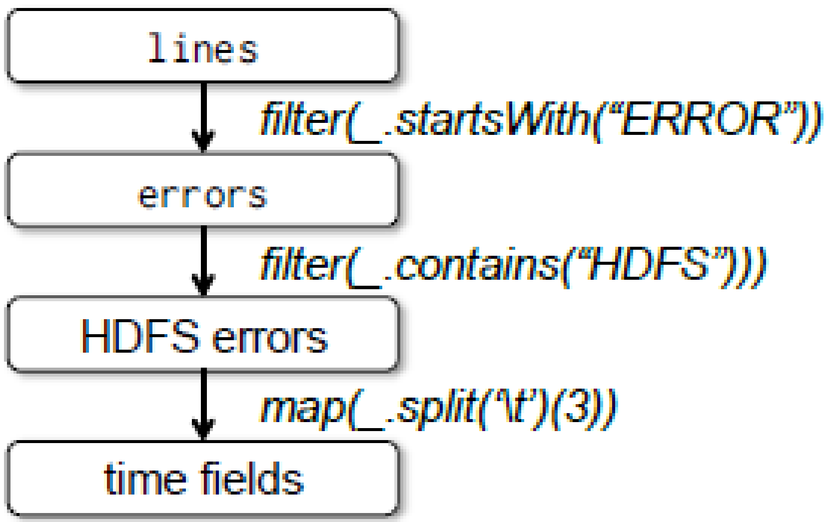
Source: https://datascience-enthusiast.com/Python/DataFramesVsRDDsVsSQLSpark-Part1.html

RDD – lazy transformations

- RDD transformations are "lazy operations", i.e. they are evaluated on request of some "Action"
- This helps in optimizing the execution
  - Multiple operations can be performed in a single scan
  - Amount of Data to be shuffled can be minimized by performing local aggregations, etc

RDDs – fault tolerance[4]

- RDD evaluation engine maintains lineage graph like this for all transformations (This is lineage graph for a work flow on next slide)



- Suppose one of the node containing errors RDD fails.
- That partition of errors can be computed from other replica of data chunk on other available node

Spark Example [4]

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()


errors.count()

// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

Higher abstractions

- RDD are still low level to perform analytical tasks!
- Spark provides higher abstractions
  - Dataframe API
  - Spark-SQL
- These abstractions makes “cluster programming” amazingly simple, and
- Primarily the reason Spark is becoming popular for big data processing.

Shared variables

- Shared variables is a mechanism of capturing notion of "Global Variables" – global across computing nodes
- Are of two types
  - Broadcast variables
  - Accumulators

Broadcast variables:

- Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.
- If a large read-only piece of data (e.g., a lookup table) is used in multiple parallel operations, it is preferable to distribute it to the workers, only once.
- Spark uses sophisticated broadcast algorithms to reduce the communication cost!

Example in Java

Broadcast

```
Broadcast<int[]> broadcastVar = sc.broadcast(new int[] {1, 2, 3});
```

Accessing in some local function like map or so

```
broadcastVar.value();
// returns [1, 2, 3]
```

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

Accumulators

- Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel.
- Accumulators typically allows, mappers to put data in parallel!
- Therefore, a mechanism of building some aggregations, like sums and counters (as in Map Reduce) .
- Spark natively supports accumulators of numeric types, and programmers can add support for new types.

Accumulators

- Java example

```
LongAccumulator accum = jsc.sc().longAccumulator();
```

Updaing in some local function like map, or forEach

```
sc.parallelize(Arrays.asList(1, 2, 3, 4)).foreach(x -> accum.add(x));
// ...
// 10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s
```

Collecting in Driver code

```
accum.value();
// returns 10
```


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

Accumulators

- Typically, it works as following (local accumulation and then aggregation on request)

Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	E
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

Further Reading

- Chapter 3 and Chapter 4 of book
  “Learning spark: lightning-fast big data analysis“,
  O'Reilly Media, Inc.", 2015.
  - The book discussed Spark programming in three languages: Scala, Python, Java!

- RDD Programming Guide
  http://spark.apache.org/docs/latest/rdd-programming-guide.html

References

[1] Zaharia, Matei, et al. "Spark: Cluster computing with working sets." *HotCloud* 10.10-10 (2010): 95.

[2] Zaharia, Matei, et al. "Apache spark: a unified engine for big data processing." *Communications of the ACM* 59.11 (2016): 56-65.

[3] Shi, Juwei, et al. "Clash of the titans: Mapreduce vs. spark for large scale data analytics." *Proceedings of the VLDB Endowment* 8.13 (2015): 2110-2121.

[4] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

[5] Doulkeridis, Christos, and Kjetil NØrvåg. "A survey of large-scale analytical query processing in MapReduce." *The VLDB Journal—The International Journal on Very Large Data Bases* 23.3 (2014): 355-380.