Introduction to Apache Spark

pm jat @ daiict

What is Spark - recap

- 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) - recap

- 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) - recap

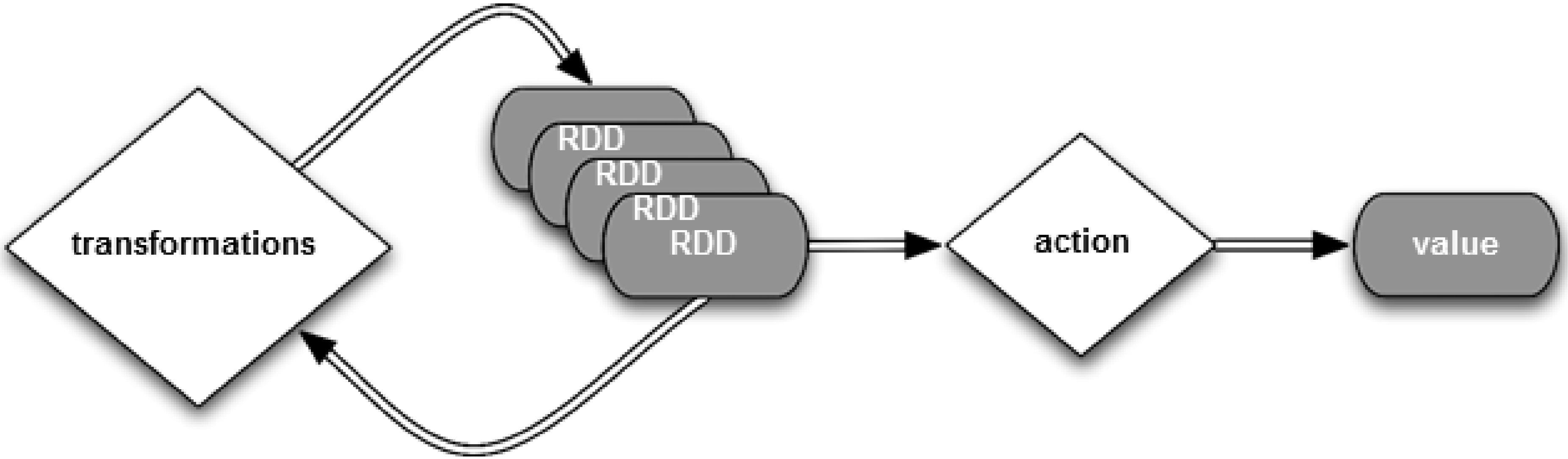
- Characteristics of RDD:
  – a read-only,
  – partitioned collection of records.
  – Fault tolerant
  – Lazily evaluated
  – Can be cached
- RDDs can only be created by:

  (1) by reading from data file,
  (2) distributing (parallelization) a local collection to multiple nodes,
  (3) by applying a “transformations” on existing rdd

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”

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

map transformation

```
map(map-function: T->U): RDD[T] => RDD[U]
```

- When applied on RDD of type T, it produces RDD of type U by applying *map function*. Mapping is 1:1
- RDD is produced on “same partitions”
- Following map function [Java Code] transforms RDD of String to RDD of String Arrays?

```
JavaRDD<String> lines = sc.textFile("data/employee.csv");
JavaRDD<String[]> records = lines.map(line -> line.split(","));
```

```java
JavaRDD<String> lines = sc.textFile("data/employee.csv");
JavaRDD<String[]> records = lines.map(line -> line.split(","));
```

Input: lines [**RDD<String>**]

```
101,John,9-Jan-55,M,30000,102,5
102,Franklin,8-Dec-45,M,40000,105,5
103,Joyce,31-Jul-62,F,25000,102,5
104,Ramesh,15-Sep-52,M,38000,102,5
105,James,10-Nov-27,M,55000,,1
106,Jennifer,20-Jun-31,F,43000,105,4
107,Ahmad,29-Mar-59,M,25000,106,4
108,Alicia,19-Jul-58,F,25000,10
```

map

Output: records [**RDD<String[]>**]

```
<101,John,9-Jan-55,M,30000,102,5>
<102,Franklin,8-Dec-45,M,40000,105,5>
<103,Joyce,31-Jul-62,F,25000,102,5>
<104,Ramesh,15-Sep-52,M,38000,102,5>
<105,James,10-Nov-27,M,55000,,1>
<106,Jennifer,20-Jun-31,F,43000,105,4>
<107,Ahmad,29-Mar-59,M,25000,106,4>
<108,Alicia,19-Jul-58,F,25000,106,4>
```

map (producing Pair RDD)

- A variation of RDD, that is, RDD of **<K,V> pairs**, called Pair RDD. Most aggregation operations are performed on such RDDs.
- If we want to produce Pair RDD through Map and often we require doing this, a variation of map function called `mapToPair` is available in Java (some languages like Python, Scala do it implicitly).
- Here is a map example that produces pairs of dno as key and a tuple <salary,1> as value.

map (producing Pair RDD)

Pseudo Code

```
records = lines.map(line -> line.split(","));
dnoSalPairs = records.map(
    record -> (record[6], (int) record = [4])
);
```

Input: records [**RDD<String[]>**]

```
<101,John,9-Jan-55,M,30000,102,5>
<102,Franklin,8-Dec-45,M,40000,105,5>
<103,Joyce,31-Jul-62,F,25000,102,5>
<104,Ramesh,15-Sep-52,M,38000,102,5>
<105,James,10-Nov-27,M,55000,,1>
<106,Jennifer,20-Jun-31,F,43000,105,4>
<107,Ahmad,29-Mar-59,M,25000,106,4>
<108,Alicia,19-Jul-58,F,25000,106,4>
```

map

Output: **Pair RDD** dnoSalPairs [String, Tuple**<Int,Int>**]

```
key  value
5, <30000,1>
5, <40000,1>
5, <25000,1>
5, <38000,1>
1, <55000,1>
4, <43000,1>
4, <25000,1>
4, <25000,1>
```

map (producing Pair RDD)

Java Code

```java
JavaPairRDD<String, Tuple2<Integer, Integer>> dnoSalPairs
    = records.mapToPair(rec -> new Tuple2<>(
            rec[6], new Tuple2<>(Integer.parseInt(rec[4]),1))
            ); key                      value
```

Input: records [**RDD<String[]>**]

```
<101,John,9-Jan-55,M,30000,102,5>
<102,Franklin,8-Dec-45,M,40000,105,5>
<103,Joyce,31-Jul-62,F,25000,102,5>
<104,Ramesh,15-Sep-52,M,38000,102,5>
<105,James,10-Nov-27,M,55000,,1>
<106,Jennifer,20-Jun-31,F,43000,105,4>
<107,Ahmad,29-Mar-59,M,25000,106,4>
<108,Alicia,19-Jul-58,F,25000,106,4>
```

map

Output: **Pair RDD** dnoSalPairs [String, Tuple2**<Int,Int>**]

```
key  value
5, <30000,1>
5, <40000,1>
5, <25000,1>
5, <38000,1>
1, <55000,1>
4, <43000,1>
4, <25000,1>
4, <25000,1>
```

map transformation

- Note that instead of two maps for computing <Dno,<Salary,1>> pairs, we can do it one map too.
- Shown on next slide
- While second one sounds like shorter, Spark's strategy of lazy evaluation and optimizer should free us from writing either way.

How do following two solutions differ?

- How do following two solutions for computing <Dno,<Salary,1>> differ?

```
lines = sc.textFile("data/employee.csv");
records = lines.map(line -> line.split(","));
dnoSalPairs = records.map(
    record -> (record[6], (int) record = [4])
);
```

```
lines = sc.textFile("data/employee.csv");
dnoSalPairs = lines.map( {
    record = line.split(",");
    line -> (record[6], (int) record = [4]);
});
```

`filter` transformation

`filter(filter-func:T->bool):RDD[T]->RDD[T]`

- Return a new dataset formed by selecting those elements of the source for which *filter-function* returns true [Java Code]

```
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));
```

- Note the code above also uses two actions "`take`", and "`count`"

flatMap transformation

`flatMap(mapfunc:T->Seq(U)): RDD[T]->RDD[U]`

- Similar to map, but each input item can be mapped to 0 or more output items,
- so *map-function* that we define should return a Sequence rather than a single item
- Following code transforms a RDD of “lines” to RDD of “words”, though both are happens to be of type String. Type wise both are of String though!

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

reduceByKey transformation

```
reduceByKey(reduce-func:(V,V)->V)
           :RDD[(K,V)]->RDD[(K,V)]
```

- When called on a dataset of (K, V) pairs (Pair RDD), returns a dataset of (K, V) pairs. where the values for each key are aggregated using the given *reduce-function*.
- The function may also provide additional parameter for number of reduce tasks.
- Here is an example.

```
sums = dnoSalPairs.reduceByKey((x, y) -> x + y);
```

reduceByKey transformation

```
sums = dnoSalPairs.reduceByKey((x, y) -> x + y);
```

Input: Pair RDD
dnoSalPairs[**<String, Integer>**]

5,	30000
5,	40000
5,	25000
5,	38000
1,	55000
4,	43000
4,	25000
4,	25000

Reduce By Key

output: Pair RDD
sums[**<String, Integer>**]

4:	93000
5:	133000
1:	55000

```
reduceByKey(reduce-func:(V,V)->V)
:RDD[(K,V)]->RDD[(K,V)]
```

reduceByKey transformation

```
reduceByKey(reduce-func:(V,V)->V)
:RDD[(K,V)]->RDD[(K,V)]
```

Java Code

```
JavaPairRDD<String, Integer> sums
    = dnoSalPairs.reduceByKey((x, y) -> x + y);
```

- Parameters to the function: value-1 (x) and Value-2 (y) each of type Integer. Can read them as following value-1 being the first one (for a distinct key, where as value-2 varies from second to last)
- Computes: performs simply addition, and returns!
- Note about type parameters: K is String, V is Integer

Compute Department wise Total Salary using Reduce-By-Key

- Compute Pipeline: `map(extract dno, sal)` → `reduceByKey()` → `result(dno, sumsal)`

Pseudo Code

```
lines = sc.textFile("data/employee.csv");

dnoSalPairs = lines.mapToPair( {
    record = line.split(",");
    line -> (record[6], (int) record = [4]);
});

sums = dnoSalPairs.reduceByKey((x, y) -> x + y);

//collect and output on console
output = sums.collect();
output.forEach( t -> print(t._1 + ": " + t._2));
```

Compute Department wise Total Salary using Reduce-By-Key

Java Code

```
JavaRDD<String> lines = sc.textFile("data/employee.csv");
JavaPairRDD<String, Integer> dnoSalPairs
= lines.mapToPair( line ->
    new Tuple2<>(line.split(",")[6], Integer.parseInt(line.split(",")[4]))
);

JavaPairRDD<String, Integer> sums
    = dnoSalPairs.reduceByKey((x, y) -> x + y);
```

MAP: Dno , Sal Pairs

ReduceByKey: Dno , SumSal

Compute Department wise Average Salary using Reduce-By-Key

- Note alone Reduce-By-Key is not enough to compute average; we require additional computation (a map task) after reduce-by-key!

- Here is a pipeline (red-one is additional step w.r.t SUM:
Compute Pipeline:
map(extract dno, <sal,1>)
→ reducebyKey()
→ map(compute average)
→ result(dno,avgsal)

Compute Department wise Average Salary using Reduce-By-Key

```
lines = sc.textFile("data/employee.csv");
dnoSalPairs = lines.map( {
    record = line.split(",");
    line -> (record[6], ((int) record = [4],1))
});

sums = dnoSalPairs.reduceByKey(
    (v1,v2) -> (v1._1 + v2._1, v1._2 + v2._2)
);

aggregate = sums.map( v -> {
    key = v._1;
    avg = 1.0 * v._2._1/v._2._2;
    return (key, avg);
});
```

MAP:
Dno , Sal Pairs

ReduceByKey:
Dno , SumSal

MAP: Dno , AvgSal

Compute Department wise Average Salary using Reduce-By-Key

- Compute Pipeline: map(extract dno, <sal,1>) → reducebyKey() → map(compute average) → result

Java Code

```
JavaRDD<String> lines = sc.textFile("data/employee.csv");
JavaPairRDD<String, Tuple2<Integer, Integer>> dnoSalPairs
    = lines.mapToPair( line ->
        new Tuple2<>(line.split(",")[6],
                new Tuple2<>(Integer.parseInt(line.split(",")[4]),1))
    );

JavaPairRDD<String, Tuple2<Integer, Integer>> sums
    = dnoSalPairs.reduceByKey((v1,v2)
        -> new Tuple2<>(v1._1 + v2._1, v1._2 + v2._2));

JavaPairRDD<String, Double> aggregate = sums.mapToPair( (v)
    -> new Tuple2<String, Double>(v._1, 1.0 * v._2()._1/v._2()._2));
```

MAP:
Dno , Sal Pairs

ReduceByKey:
Dno , SumSal

MAP: Dno , AvgSal

Reduce-By-Key for Average

- For computing Average, function for ReduceByKey is defined such that it produces <salary, sum(count)> pairs for every key (dno).

```
sums = dnoSalPairs.reduceByKey(
       (v1,v2) -> (v1._1 + v2._1, v1._2 + v2._2)
);
```

Input: dnoSalPairs[<String,<Int,Int>>]

```
5, <30000,1>
5, <40000,1>
5, <25000,1>
5, <38000,1>
1, <55000,1>
4, <43000,1>
4, <25000,1>
4, <25000,1>
```

Reduce By Key

output: sums[String,<Int,Int>>]

```
4, <93000,3>
5, <133000,4>
1, <55000,1>
```

```
reduceByKey(reduce-func:(V,V)->V)
:RDD[(K,V)]->RDD[(K,V)]
```

Reduce-By-Key for Average

```
reduceByKey(reduce-func:(V,V)->V)
:RDD[(K,V)]->RDD[(K,V)]
```

```
JavaPairRDD<String, Tuple2<Integer, Integer>> sums
    = dnoSalPairs.reduceByKey((v1,v2)
            -> new Tuple2<>(v1._1 + v2._1, v1._2 + v2._2));
```

- Parameters to the function: value-1 and Value-2. each of type <Integer, Integer> tuple.
- Computes: performs pairwise sum, and Returns as tuple!
- Note about type parameters:
  K is String, and V is Tuple<Integer, Integer>

Map function computing average!

Pseudo Code

```
aggregate = sums.map( v -> {
    key = v._1;
    avg = 1.0 * v._2._1/v._2._2;
    return (key, avg);
});
```

input: sums[String,<Int,Int>>]

```
4, <93000,3>
5, <133000,4>
1, <55000,1>
```

map

output: aggregate[**<String,Double**>]

```
4: 31000.0
5: 33250.0
1: 55000.0
```

Java Code

```
JavaPairRDD<String, Double>
    aggregate = sums.mapToPair( (v)
    -> new Tuple2<>(v._1, 1.0 * v._2()._1/v._2()._2));
```

groupByKey transformation

`groupByKey(): RDD[(K,V)]-> RDD[(K,seq(V)]`

- When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
- The function may also provide additional parameter for number of tasks.
- Example:

```
groupedRecords = dnoSalPairs.groupByKey();
```

groupByKey transformation

```
groupedRecords = dnoSalPairs.groupByKey();
```

Input: dnoSalPairs[**RDD<?>**]

```
5, <30000,1>
5, <40000,1>
5, <25000,1>
5, <38000,1>
1, <55000,1>
4, <43000,1>
4, <25000,1>
4, <25000,1>
```

Group By Key

output: dnoSalPairs[**RDD<?>**]

```
4: <43000,1>, <25000,1>, <25000,1>
5: <30000,1>, <40000,1>, <25000,1>, <38000,1>
1: <55000,1>
```

Compute Average using Group-By-Key

```
lines = sc.textFile("data/employee.csv");
dnoSalPairs = lines.map( {
    record = line.split(",");
    line -> (record[6], ((int) record = [4],1));
});

groupedRecords = dnoSalPairs.groupByKey();

aggregate = groupedRecords.map( tuple -> {

    String key = tuple._1;
    values = tuple._2;
    int count = 0;
    double sum = 0;
    for( value : values) {
        sum += value._1;
        count += value._2;
    }
    double avg = sum / count;
    return (key, avg);
});
```

“ImageCounter” implementation

Pseudo Code

```
requests = sc.textFile("data/access_log.txt");

//Define filter function that keeps requests that are for images
imgFilter = request -> {
```

```
//Define Map function
imgMap = request -> {
```

```
//Apply Filter as per the above function
imgRequests = requests.filter( imgFilter );

//Perform MAP as per the defined in line anonymous function
images = imgRequests.mapToPair( imgMap );

//Apply Reduce Function
counts = images.reduceByKey((x, y) -> x + y);

//collect and output on console
output = sums.collect();
output.forEach( t -> print(t._1 + ": " + t._2));
```

"ImageCounter" implementation

Pseudo Code

```
//Define filter function that keeps requests that are for images
imgFilter = request -> {
    String url= request.split(" ")[6];
    String method= request.split(" ")[5];
    int pos = url.lastIndexOf(".");
    if(pos != -1 ) {
        ext=url.substring(pos+1);
        if(isImg.contains(ext) && method.equals("\"GET"))
            return TRUE;
    }
    return FALSE;
};

//Define Map function
imgMap = request -> {
    String url= request.split(" ")[6];
    int pos = url.lastIndexOf(".");
    String ext=url.substring(pos+1);

    if(ext.equals("gif"))   return ("GIF Images",1);
    else if(isJPG.contains(ext)) return ("JPG Images",1);
    else return ("Other Images",1);
};
```

Writing "functions" to be sent as an argument to Transformations

- Java being strongly typed language makes it bit verbose. Agree Java makes it bit complex!

- Provides following set of interfaces. An object of class implementing one of these interfaces can be sent as an argument to a transformation function.

```
Function<T1,R> //for processing RDDs
```

Method: `R call(T1 v1)`

```
Function2<T1,T2,R> ////for processing Pair RDDs
```

Method: `R call(T1 v1, T2 v2)`

```
PairFunction<T,K,V> //for returning Pairs
```

Method: `Tuple2<K,V> call(T t)`

Examples: “function” parameters to transformations

```
Function<T1,R>
    Method: R call(T1 v1)
```

Filter Function for ImageFilter

- **`imgFilter`** gets a new object of anonymous class implementing Function2 interface. The class overrides **`Call`** method!

```
         T1       R
Function<String, Boolean> imgFilter = new Function<String, Boolean>() {
    @Override
    public Boolean call(String request) throws Exception {
        String url= request.split(" ")[6];
        String method= request.split(" ")[5];
        int pos = url.lastIndexOf(".");
        if(pos != -1 ) {
            String extension=url.substring(pos+1);
            if(ImageCounter.isImg.contains(extension) && method.equals("\"GET"))
                return Boolean.TRUE;
        }
        return Boolean.FALSE;
```

Type Parameters to the Function: T1 parameter type, i.e. Parent RDD element type. R is return type

Inline Function for Map task [Java]

```
PairFunction<T,K,V>
    Method: Tuple2<K,V> call(T t)
```

```
//Perform MAP as per the defined in line anonymous function
JavaPairRDD<String, Integer> images = imgRequests.mapToPair(
        new PairFunction<String, String, Integer>() {
    @Override
    public Tuple2<String, Integer> call(String request) throws Exception {
        String url= request.split(" ")[6];
        int pos = url.lastIndexOf(".");
        String extension=url.substring(pos+1);

        if(extension.equals("gif"))
            return new Tuple2<String, Integer>("GIF Images",1);
        else if(ImageCounter.isJPG.contains(extension))
            return new Tuple2<String, Integer>("JPG Images",1);
        else
            return new Tuple2<String, Integer>("Other Images",1);
    }
});
```

Type Parameters to the PairFunction: T parameter type, i.e. Parent RDD element type. K,V are Key and Value types of Pair to be returned

Implementation of Average using Group-By-Key

```
lines = sc.textFile("data/employee.csv");
dnoSalPairs = lines.map( {
    record = line.split(",");
    line -> (record[6], ((int) record = [4],1))
});

sums = dnoSalPairs.reduceByKey(
    (v1,v2) -> (v1._1 + v2._1, v1._2 + v2._2)
);

aggregate = sums.map( v -> {
    key = v._1;
    avg = 1.0 * v._2._1/v._2._2;
    return (key, avg);
});
```

Implementation of Average using Group-By-Key

Following are RDD Transformations in Average using Group-By-Key

- Read from file ==> RDD **line**[String]
- Map applied (line) ==> RDD **dnoSalPairs**[<String, <Int,Int>>]
- Group By Key applied (dnoSalPairs)
  ==> RDD **groupedRecords**[<String, Iterable(<Int,Int>)>]
- Map applied to groupedRecords
  ==> RDD **aggregate** [<String, Double>]

This is desired result!

Java Implementation of Average using Group-By-Key

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

JavaPairRDD<String, Tuple2<Integer, Integer>> dnoSalPairs
    = lines.mapToPair( line ->
        new Tuple2<>(line.split(",")[6],
                new Tuple2<>(Integer.parseInt(line.split(",")[4]),1))
    );

JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>>
        groupedRecords = dnoSalPairs.groupByKey();
JavaPairRDD<String, Double> aggregate = groupedRecords.mapToPair(
                { //Implementation of Map function follows }

//collect and output on console
List<Tuple2<String, Double>> output = aggregate.collect();
output.forEach( tuple -> System.out.println(tuple._1 + ": " + tuple._2));
```

Map Function for "Computing Aggregate"

Input: groupedRecords[**<String, Iterable[Int]**>]; i.e. DNO, SalPair List

```
4: <43000,1>, <25000,1>, <25000,1>
5: <30000,1>, <40000,1>, <25000,1>, <38000,1>
1: <55000,1>
```

output: aggregate[**<String,Double**>]; i.e. DNO, Avg

```
4: 31000.0
5: 33250.0
1: 55000.0
```

Map for "Computing Aggregate"

```
PairFunction<T,K,V>
    Method: Tuple2<K,V> call(T t)
```

```
JavaPairRDD<String, Double> aggregate = groupedRecords.mapToPair(
        new PairFunction<Tuple2<String,
            Iterable<Tuple2<Integer, Integer>>>, String, Double>() {
            @Override
            public Tuple2<String, Double> call(Tuple2<String,
                    Iterable<Tuple2<Integer, Integer>>> tuple)
                        throws Exception {
                String key = tuple._1;
                Iterable<Tuple2<Integer, Integer>> values = tuple._2();
                int count = 0;
                double sum = 0;
                for( Tuple2<Integer, Integer> value : values) {
                    sum += value._1;
                    count += value._2;
                }
                double avg = sum / count;
                return new Tuple2<String, Double>(key, avg);
            }
        });
```

map function for Computing Average

```
PairFunction<T,K,V>
   Method: Tuple2<K,V> call(T t)
```

- Since we require returning Pair, we use **PairFunction**.
- Parameters to **call** function: Tuple of <DNO, List<Salary, 1>>, i.e. Tuple2<String, Iterable[Tuple2<Integer, Integer>]>
- Computes: Aggregates Value list, and computes Average=Σsalary/Σcount, and
- Return <DNO, Average>
- Type Parameters:
  T=Tuple2<String, Iterable[Tuple2<Integer, Integer>]>,
  K=String, V=Double

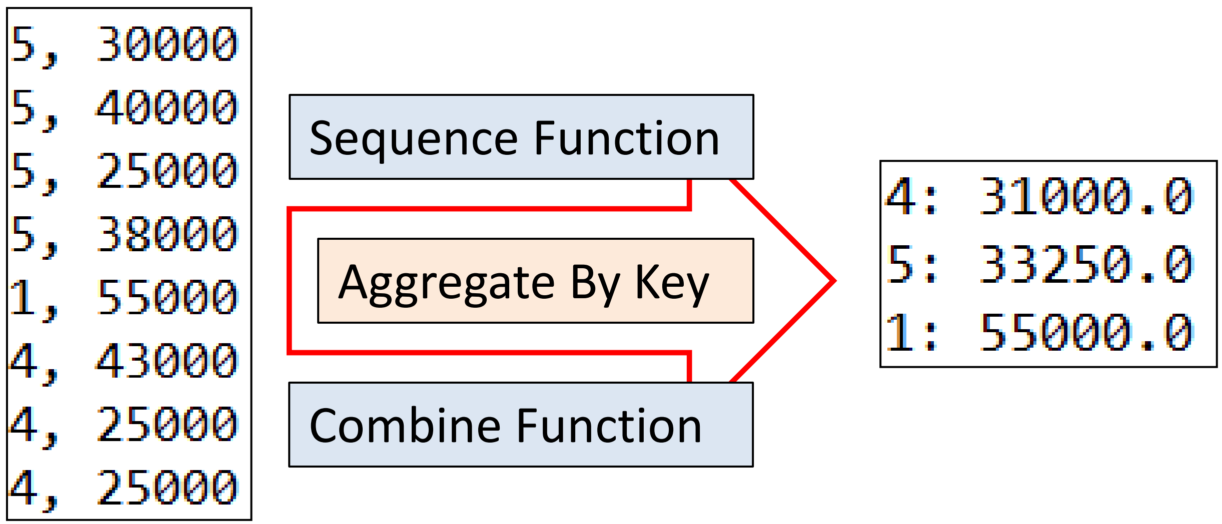
aggregateByKey transformation

```
aggregateByKey(zeroValue, [numPartitions],
seqFunction, combFunction): <K,V> => <K, U>
```

- When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a "zero" value.
- Four Parameters: (1) Zero value (used for initialization purpose while aggregation, (2) optional: number of tasks (3) sequence operation, that runs on every value of value list (sequence), typically accumulates into an accumulator. Corresponds to "**map**" function in a MR job.
  (4) combine operations: Corresponds to "**combine**" function in a MR job.
- Allows defining an "aggregated value" type that is different than the input value type. Object of this type is what returned for every key.

Average using Aggregate By Key

- We can use Aggregate By Key for computing



Average using **Aggregate By Key**

```
lines = sc.textFile("data/employee.csv");
dnoSalPairs = lines.map( {
    record = line.split(",");
    line -> (record[6], ((int) record = [4],1));
});
//define sequence function
addAndCount = ( aggregator, value) -> {
        aggregator.sum += value;
        aggregator.count += 1;
        return aggregator;
    }
};
//define combine function
combine = (aggr1, aggr2) -> {
        aggr1.sum += aggr2.sum;
        aggr1.count += aggr2.count;
        return aggr1;
    }
};
initialAggr = (0,0);
aggregate = dnoSalPairs
    .aggregateByKey(initialAggr, addAndCount, combine);
```

Average using Aggregate By Key

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

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

Function2<AvgCounter, Integer, AvgCounter> addAndCount = { … }

Function2<AvgCounter, AvgCounter, AvgCounter> combine = { … }

AvgCounter initial = new AvgCounter(0,0);
JavaPairRDD<String, AvgCounter> aggregate
    = dnoSalPairs.aggregateByKey(initial, addAndCount, combine);
```

Sequence Function for Average

- Created of type Function2, which is basically a function with two parameters. Try understanding the code below
- Type parameters(U,V,U): U is aggregate type and V is source rdd value type
- Function parameters: 1: aggregate object, 2: value of source rdd
- Function returns: update aggregate object; source value is updated to aggregate object

```
Function2<AvgCounter, Integer, AvgCounter> addAndCount
        = new Function2<AvgCounter, Integer, AvgCounter>() {
    @Override
    public AvgCounter call(AvgCounter a, Integer x) {
        a.sum += x;
        a.count += 1;
        return a;
    }
};
```

Combine Function for Average

- Again of type Function2. Try understanding the code below
- Type parameters(U,U,U): U is aggregate type
- Function parameters: 1: aggregate object a, 2: aggregate object b
- Function returns: updated aggregated object a; b is updated into a.

```
Function2<AvgCounter, AvgCounter, AvgCounter> combine
        = new Function2<AvgCounter, AvgCounter, AvgCounter>() {
    @Override
    public AvgCounter call(AvgCounter a, AvgCounter b) {
        a.sum += b.sum;
        a.count += b.count;
        return a;
    }
};
```

RDD Operations - recap

- Computations in Spark are performed by following two types of operations on RDD
  - Transformations, and
  - Actions
- In all cases RDD is operand. Transformations derive another RDD by applying specified transformation.
- Some transformations are unary while others are binary
- Most Transformations require specifying transformation function
- Actions are used by driver program to collect “some result”!

RDD Operations - summarize

`map(map-function: T->U): RDD[T] => RDD[U]`

- When applied on RDD of type T, it produces RDD of type U by applying *map function*. Mapping is 1:1

`filter(filter-func: T->bool):RDD[T]->RDD[T]`

- Return a new dataset formed by selecting those elements of the source for which *filter-function* returns true [Java Code]

`flatMap(mapfunc:T->Seq(U)): RDD[T]->RDD[U]`

- Similar to map, but each input item can be mapped to 0 or more output items

You can get more concrete coverage at:
http://spark.apache.org/docs/latest/rdd-programming-guide.html
https://training.databricks.com/visualapi.pdf

RDD Operations - summarize

```
reduceByKey(reduce-func:(V,V)->V)
            :RDD[(K,V)]->RDD[(K,V)]
```

- When called on a dataset of (K, V) pairs (Pair RDD), returns a dataset of (K, V) pairs. where the values for each key are aggregated using the given *reduce-function*.

```
groupByKey(): RDD[(K,V)]-> RDD[(K,seq(V)]
```

- When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.

```
aggregateByKey(zeroValue, seqFunction,
combFunction):<K,V> => <K, U>
```

- When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions.

RDD Operations - summarize

`union``(otherRDD):(RDD[T],RDD[T])=>RDD[T]`

- Return a new dataset that contains the union of the elements in the source dataset and the argument.

`join``(otherRDD)`
`:(RDD[(K,V)],RDD[(K,W)])=>RDD[(K,(V,W))]`

- 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.

`mapValues``(map-func:V=>W):RDD[K,V]=>RDD[K,W]`

- Key remains unchanged, only value is transformed. It becomes efficient, partition does not change.

RDD Operations - summarize

`COALESCE(numPartitions, shuffle_flag)`

- Return a new RDD which is reduced to a smaller number of partitions
- Shuffle_flag tells, if shuffling is to be done, or merely merging of some of partitions!

You can get more concrete coverage at:
http://spark.apache.org/docs/latest/rdd-programming-guide.html
https://training.databricks.com/visualapi.pdf

RDD Transformations[4]

- Transformations create a new RDD, and are "lazy operations"; i.e. computed only when required!

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

Cogroup is also called groupWith

Partition is basically Repartition

You can get more concrete coverage at:
http://spark.apache.org/docs/latest/rdd-programming-guide.html
https://training.databricks.com/visualapi.pdf

RDD Actions[4]

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

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

- Reduce action returns a single value of type T. The reduce function should be commutative and associative so that it can be computed correctly in parallel.

RDD Actions[5]

- More Actions:
  - take(n) //if collect can return huge list
  - takeSample(num, seed)
  - takeOrdered(n, ordering_function)
  - countByKey
  - foreach

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));
```

Persistence and caching of RDD

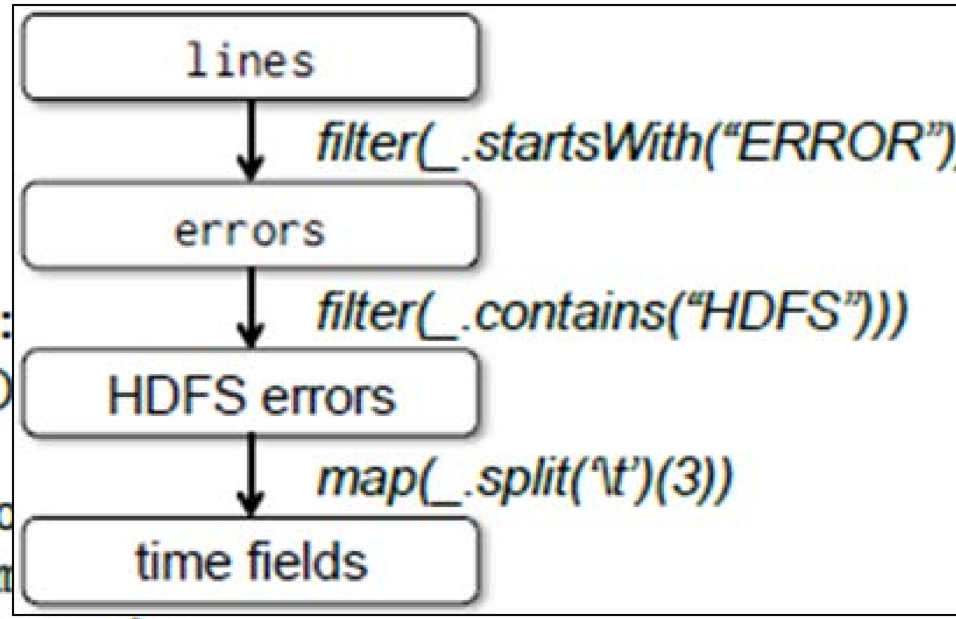
- We can make RDDs to live only in primary memory, or on disk
- RDD API provides two methods for this persist, and cache.
- Cache tells RDD to be living only in primary memory, where as
- With persist, we can specify various methods of persistence. Typical values: MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY,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

RDD – lazy transformations

- RDD transformations are "lazy operations", i.e. they are evaluated on request of some "Action"
- RDD execution engine maintains "lineage graph"; all operations specified on RDD are added to lineage graph.
- Lineage graph is evaluated on demand.
- This arrangement 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

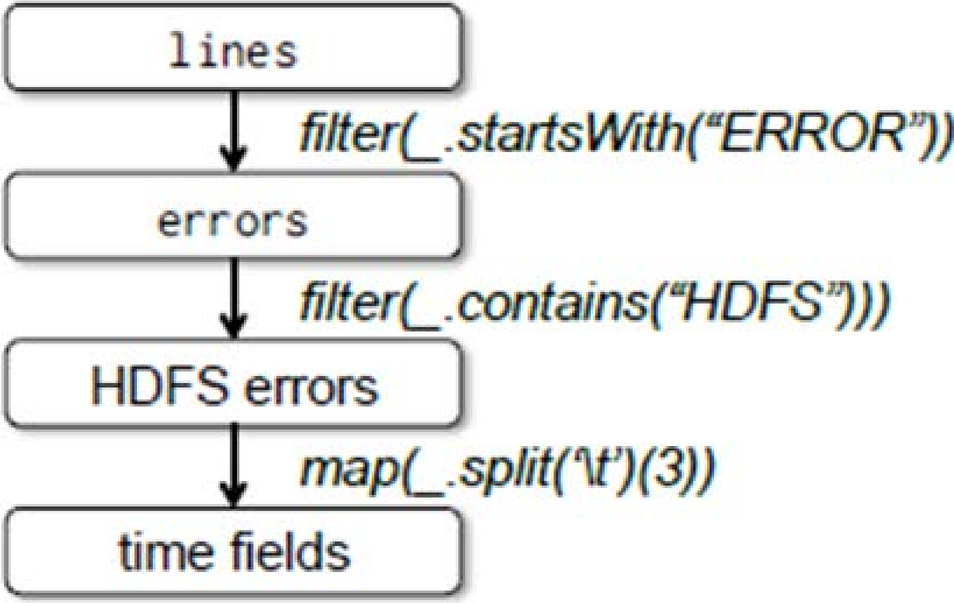
Example: Lineage Graph[4]

```
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 erro
// HDFS as an array (assuming tim
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```



RDDs – fault tolerance[4]

- Replication is mechanism of fault tolerance.
- RDD themselves are not replicated but they are reconstructed on another node if a computing node holding part of an RDD fails.
- 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

Shared variables

- Shared variables is a mechanism of capturing notion of "Global Variables" – global across computing nodes
- Are of two types
  - **Broadcast variables**: arrangement of making some data always data available on (data) partitions. Broadcast variables are in Read Only Mode.
  - **Accumulators**: a global short of data that tasks can accumulate some counters or aggregators.

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” 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 discusses Spark programming in three languages: Scala, Python, Java!

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

Further 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.

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