# COMP5349 –Cloud Computing

**Week 8:** Programming with Apache Spark and Flink

A/Prof. Uwe Roehm
School of Information Technologies


## Self-Reflection Survey

- Please re-submit in eLearning as we had to re-create the self reflection survey on the weekend
- Please answer **all** questions in a single session
  - please review the instructions
  - if unsure, choose the middle option

- In the lab today:
  - Demos on Azure
  - Exercises with both Spark and Flink

# Review Question 1

**Which system uses Resilient Distributed Datasets (RDD)?**

- **A:** Apache Flink
- **B:** Apache Hadoop
- **C**: Apache Spark

**http://tinyurl.com/mpkjmw**



# Review Question 2

**What is the main strength of Spark?**

- **A:** in-memory processing
- **B:** fail safety
- **C**: pipelined processing

**http://tinyurl.com/knqekz6**

# Review Question 3

**What is the main strength of Flink?**

- **A:** in-memory processing
- **B:** fail safety
- **C**: pipelined processing

**http://tinyurl.com/meqatdd**



# Outline

- **Overview**
  - **Spark & Flink Skeleton Programs**
  - **Wordcount Example**
- **Spark and Flink Details**
  - **RDD and Data Set Operations**
- **Lambda Expression**
- **MR Design Patterns in Spark & Flink**
- **A Complete Example**



[based on slides by Dr Ying Zhou]

# Spark & Flink Program Skeletons

- Regular programs in Java / Scala / Python

1. Initialise the runtime **environment**
2. Load or create **source** data
3. Specify the **data transformations**
   - This might include usage of self-defined UDFs
4. Specify where the **output** should go
5. **Execute**



# Comparison

- In the following, each step with either Spark or Flink
- First Java, but also all examples in Python available

# Step 1: Initialisation

- **Spark**:
- Every Spark application represented as a driver program
  - In Java, the driver program is just a normal java class with a main method
- Every Spark application needs to create a `SparkContext` object, which is used to access a cluster. In Java language The `SparkContext` object is of type `JavaSparkContext`

- **Flink**:
- In Java, a Flink main program is a normal Java class with a main() method
- Every Flink application needs to create an ExecutionEnvironment object, which is used to access the cluster. In Java, this object is of type `ExecutionEnvironment`



# Spark – Driver Initialisation

- A Spark program is just a regular **main** program that creates `SparkContext` object, which is used to access a cluster.
  - In Java, this object is of type `JavaSparkContext`
  - In Python, this object is of type …

- The spark context provides methods for
  - Data input
  - Data output
  - …

- The data processing steps are defined using **RDDs**

# WordCount in Spark (Java)

```
import java.util.Arrays;
import org.apache.spark.SparkConf;
import org.apache.spark.JavaSparkContext;
import org.apache.spark.JavaRDD;
import org.apache.spark.JavaPairRDD;

public class WordCountExample {
   public static void main(String[] args) {
      SparkConf conf = new SparkConf().setAppName("WordcountExample")
                                      .setMaster("…");
      JavaSparkContext sc = new JavaSparkContext(conf);

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

[Cf.: http://spark.apache.org/examples.html]



# WordCount in Spark (Python)

```
from pyspark import SparkConf, SparkContext

myconf = (SparkConf().setMaster("…")
                     .setAppName("WordcountExample")
                     .set("spark.executor.memory", "1g")
                     .pyFiles(['wordcount.py']))
sc = SparkContext(myconf)

text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.lower.split(" ")) \
                  .map(lambda word: (word, 1)) \
                  .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

[Cf.: http://spark.apache.org/examples.html]

# Spark– Linking and Execution

- Need to add additional libraries to the program
- Spark Java:
  - spark-assembly-**1.6.1**-hadoop2.**6.0**.**jar**
  - spark-core_2.**10**-**0.9.0**-incubating.**jar**



# PySpark Shell

- With PySpark, there is also the possibility to use the Python shell to explore data interactively and as a simple way to learn the API:

```
./bin/pyspark
>>> words = sc.textFile("hdfs://…")
>>> words.filter(lambda w: w.startswith("spar")).take(5)
  [u'spar', u'sparable', u'sparada', u'sparadrap', …]
>>> help(pyspark) # Show all pyspark functions
```

- By default, the bin/pyspark shell creates a SparkContext ‘sc’ that runs applications locally on a single core. To connect to a non-local cluster, or use multiple cores, set the MASTER environment variable.

[https://spark.apache.org/docs/0.9.1/python-programming-guide.html]

# Flink – Program Initialisation

- A Flink program is just a regular **main** program that creates an Flink "execution environment" (an ExecutionEnvironment object), which is used to access the cluster
  - In Java, this object is of type **`ExecutionEnvironment`**
  - In Python, this object is of type **`flink.plan.Environment`**
- The execution environment provides methods for
  - Data input
  - Data output
  - **`execute()`**: actually executes the defined Flink workflow in the cluster
- The processing workflow is defined using **`DataSet`** objects and its API to manipulate data sets



# WordCount in Flink (Java)

```
public class WordCountExample {
   public static void main(String[] args) {
  final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
      DataSet<String> data = env.readTextFile("hdfs://…");

      DataSet<Tuple2<String,Integer>> wordCounts =
         data.flatMap(new Tokenizer())
         .groupBy(0)   // group by tuple field "0"
         .sum(1);      // sum up tuple field "1"
      wordCounts.writeAsCsv("hdfs://…", "\n", "");

      env.execute("Word Count Example");
   }
   public static class Tokenizer
         implements FlatMapFunction<String, Tuple2<String,Integer>> {
      @Override public void flatMap(String line,
                              Collector<Tuple2<String, Integer>> out) {
         for (String word : line.toLowerCase().split(" "))
         { out.collect(new Tuple2<String, Integer>(word, 1));}
      }
   }
}
```

[Cf.: https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/examples.html]

# WordCount in Flink (Python)

```
from flink.plan.Environment import get_environment
from flink.functions.GroupReduceFunction import GroupReduceFunction

class Sum (GroupReduceFunction):
   def reduce(self, iterator, collector):
      word, count = iterator.next();
      count += sum([x[1] for x in iterator])
      collector.collect((word,count))


env = get_environment()
data= env.read_text("hdfs://…");

data.flat_map(lambda x,c: [(word,1) for word in x.lower().split()]) \
    .group_by(0) \
    .reduce_group(Sum(), combinable=True) \
    .write_csv("hdfs://…")

env.execute(local=True)
```

[Cf.: https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/python.html]



# Flink – Linking and Execution

- need to include the corresponding Flink library in our project
- simplest way is via a Maven (build tool) project template
- Flink Java:

```
mvn archetype:generate /
     -DarchetypeGroupId=org.apache.flink/
     -DarchetypeArtifactId=flink-quickstart-java /
     -DarchetypeVersion=0.8.1
```

- Flink Python:
  - Apart from setting up Flink, no additional work is required.
    The flink package, along with the plan and optional packages are automatically distributed among the cluster via HDFS when running a job. Python 2.7 or 3.4 should be used.

# Lessons Learned

- Flink's Python binding is not as mature as its Java API
  - Eg. more manual function definitions needed
- Some operations are pre-defined, but in some we need to introduce and use own functions
  - Cf. in Python example the reduce function before where we need to define an appropriate **sum** aggregator first
  - Or in the Java code, the **tokenizer** is user-defined



# Outline

- Overview
  - Spark & Flink Skeleton Programs
  - Wordcount Example
- **Spark and Flink Details**
  - **RDD and Data Set Operations**
- Lambda Expression
- MR Design Patterns in Spark & Flink
- A Complete Example

# More on Spark

- The following is another example that relates to the MovieLens data set used in the tutorial

```
public class MovieLens {

public static void main(String[] args) {
          SparkConf conf = new SparkConf();
          // we can set lots of information here
          conf.setAppName("Movie Lens Application");
          JavaSparkContext sc = new JavaSparkContext(conf);

          // create some RDDs
          JavaRDD<String> ratingData = sc.textFile(inputDataPath+"u.data"),
                    movieData = sc.textFile(inputDataPath + "u.item");
          // lots and lots of RDD operations
          JavaPairRDD<String, Integer> ratingExtraction =
            ratingData.mapToPair(s ->
              { String[] values = s.split("\t");
                   return new Tuple2<String,Integer>(values[1],Integer.parseInt(values[2]));
                    });
          ...
}
```



# Spark: Resilient Distributed Datasets

- RDDs are created by
  - Parallelizing an existing collection
  - Referencing a dataset in an external storage system, such as HDFS
- RDDs have partitions
  - Based on source file partition (such as blocks of HDFS files)
  - Or created during transformation, repartition

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

//referencing a data set in HDFS
    JavaRDD<String> ratingData =
        sc.textFile("hdfs://ip-10-171-118-84.ec2.internal:8020/share/ml/u.data");
```

# Handling Key Value Pairs in Spark

- Spark have special RDD types to handle data in key value format
  - In Java, key-value pairs are represented using the **`scala.Tuple2`** class from the **`Scala`** standard library.
  - The RDDs of key-value pair are represented by the **`JavaPairRDD`** type
    - `JavaPairRDD<String, Integer>`
- **`JavaPairRDD`**s are constructed from norm **`JavaRDD`**s



# Spark RDD Operations

- Transformation
  - create a new dataset from an existing one
  - Eg. map(func), flatMap(func), mapToPairs(func), reduceByKey(func)
- Action
  - return a value to the driver program after running a computation on the data set
  - Eg. count(), first(), collect(), saveAsTextFile(path)
- Most RDD operations take one or more functions as parameter
  - Most of them can be viewed as higher order functions
- Spark has strong functional programming flavour!

# Flink DataSet Operations

- Functionality similar to the RDDs in Spark
  - Cf. documentation
  - No in-memory caching though, but inherently pipelined processing
- Most DataSet operations take one or more functions as parameter
  - Most of them can be viewed as higher order functions
- Flink also has a strong functional programming flavour!



# Outline

- Overview
  - Spark & Flink Skeleton Programs
  - Wordcount Example
- Spark and Flink Details
  - RDD and Data Set Operations
- **Lambda Expression**
- MR Design Patterns in Spark & Flink
- A Complete Example

# Anonymous Functions

- Functional argument can be compactly expressed using anonymous functions
  - **E.g: map (a-> 2*a)**
- ***Lambda expressions*** are a way to express anonymous functions in a programming language
- Nearly all functional languages and most script languages support anonymous function
- Java supports anonymous functions beginning with Java 8 in the form of lambda expression
  - **`Function`** interface is used in previous versions



# Lambda Expressions in Java

```
// create an RDD from local file system as a collection of strings
JavaRDD<String> lines = sc.textFile("data.txt");
// create an RDD of Integer using the map transformation
// the transformation take a line as string and returns the number
// of characters
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
// call the reduce action to find total number of characters in the file
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

All transformations in Spark or Flink are *lazy*, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program.

Adding a line `lineLengths.persist();` before reduce action would cause lineLengths to be saved in memory for later use

# Lambda Expression in Java

- A Lambda Expression consists of the following
  - A comma-separated lists of formal parameters enclosed in parentheses
  - The arrow token , ->
  - A body, which consists of a single expression or a statement block

parameter list

```
(a, b) -> a + b
```

single expression

```
(a, b) -> { if (a==b)
              return 0;
            if (a < b)
              return -1;
            return 1;
          }
```

statement block



# RDD Operation Functional Argument

String -> int

```
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());

int totalLength = lineLengths.reduce((a, b) -> a + b);
```

(int, int) -> int

**map**

```
<R> JavaRDD<R> map(Function<T,R> f)
```

Return a new RDD by applying a function to all elements of t

**reduce**

```
T reduce(Function2<T,T,T> f)
```

Reduces the elements of this RDD using the specified commutative and associative binary operator.

```
public interface Function<T1,R>
extends java.io.Serializable
```

**Method Summary**

Methods

| Modifier and Type | Method and Description |
|---|---|
| R | call(T1 v1) |

```
public interface Function2<T1,T2,R>
extends java.io.Serializable
```

A two-argument function that takes arguments of type T1 and T2 and returns an R.

**Method Summary**

Methods

| Modifier and Type | Method and Description |
|---|---|
| R | call(T1 v1, T2 v2) |

# Anonymous Functions with Flink

- The previous examples were using Java and the Spark API
- Same concept (lambda expressions) are also supported natively in Python
- Flink API also relies heavily on lambda expressions
  - Please see documentation and lab for details



# Outline

- Overview
  - Spark & Flink Skeleton Programs
  - Wordcount Example
- Spark and Flink Details
  - RDD and Data Set Operations
- Lambda Expression
- **MR Design Patterns in Spark & Flink**
- A Complete Example

# MR Design Patterns with Spark or Flink

- In Week 5, we covered a series of MR design patterns
- In the following, want to briefly run through how those can be represented in Spark or Flink

- Input Transformation
  - In MR: Part of the map() function
  - In Spark: New RDD based on initial RDD;
  - In Flink: New DataSet based on initial Data Set



# MR Design Pattern: Filtering

- Filtering input data
  - In MR: Part of the map() function
  - In Spark: **`RDD.filter(`***`predicate`***`)`** operation
    - Only entries where the *predicate* function returns TRUE are kept
  - In Flink: …
    - Only entries where the *predicate* function returns TRUE are kept

- Example:
  - Spark:

```
JavaRDD<Movie> movies = sc. …
JavaRDD<Movie> result  = movies.filter( m -> 1 <= m.id <= 100);
```

  - Flink:

```
DataSet<String> data = env.readTextFile("hdfs://…");
DataSet<String> result = data.filter( line -> ! line.contains("error") );
```

# MR Design Pattern: Aggregation

- Aggregating values
  - In MR: implemented with the reduce() function
  - In Spark: **reduceByKey**()
  - In Flink: sequence of **groupBy**() followed by **reduce**()
  - both systems offer predefined aggregation functions such as **sum()** or **count()**
- Example:
  - Spark:

```
JavaPairRDD<String, Integer> counts =
      textFile.flatMap( … )
              .mapToPair(word -> new Tuple2<>(word, 1))
              .reduceByKey((a, b) -> a + b);
```

  - Flink:

```
DataSet<Integer> textfile.flatMap( { … return (word,1) } )
                         .groupBy( 0 )
                         .sum(1);
```



# MR Design Pattern: Join

- Joining two input data sets
  - In MR: pair of map() and reduce() function with appropriate keys
  - In Spark: **join()** method of PairRDDs (we need a key per value)
    - performs a hash join across the cluster with resulting <k,(v1, v2)> pairs where (k, v1) was in the first, and (k, v2) in the second RDD
  - In Flink: **join()** method on Data Sets
  - both offer various sub-forms of joins, such as cross-join (Flink), or left or right outer join (Spark)
- Example:
  - Spark

```
JavaPairRDD<> input1, input2;
JavaPairRDD<> result = input1.join(input2)
```

  - Flink:

```
DataSet<> input1, input2;
DataSet<> result = input1.join(input2)
                         .where(0)    // key of first input (tuple field 0)
                         .equalTo(1); // key of 2nd input (tuple field 1)
```

# Combiner, Partitioner, Composite Keys

- Combiner
  - Spark PairedRDDs offer a **combineByKey**() method
- Partitioner
  - Spark RDDs can get their **partitioner** class attribute overridden
  - Spark PairedRDDs have a **partitionBy**() method
  - Flink DataSets support **partitionByHash(), partitionByRange() and partitionCustom()** methods
- Composite Keys
  - In Spark:
  - In Flink: quite natural via Tuple specification

[cf. https://spark.apache.org/docs/0.6.2/api/core/spark/RDD.html]
[cf. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html]



# Combining two data values

- Problem: Both Flink and Spark use **Lazy Evaluation**
  - You cannot use single result values of one computation inside another, because it ahs not been computed yet, but is just a placeholder for a ‘sub-query’
- Two approaches:
  - 1. Cross-Join
  - 2. Broadcast variables (Flink)

# Outline

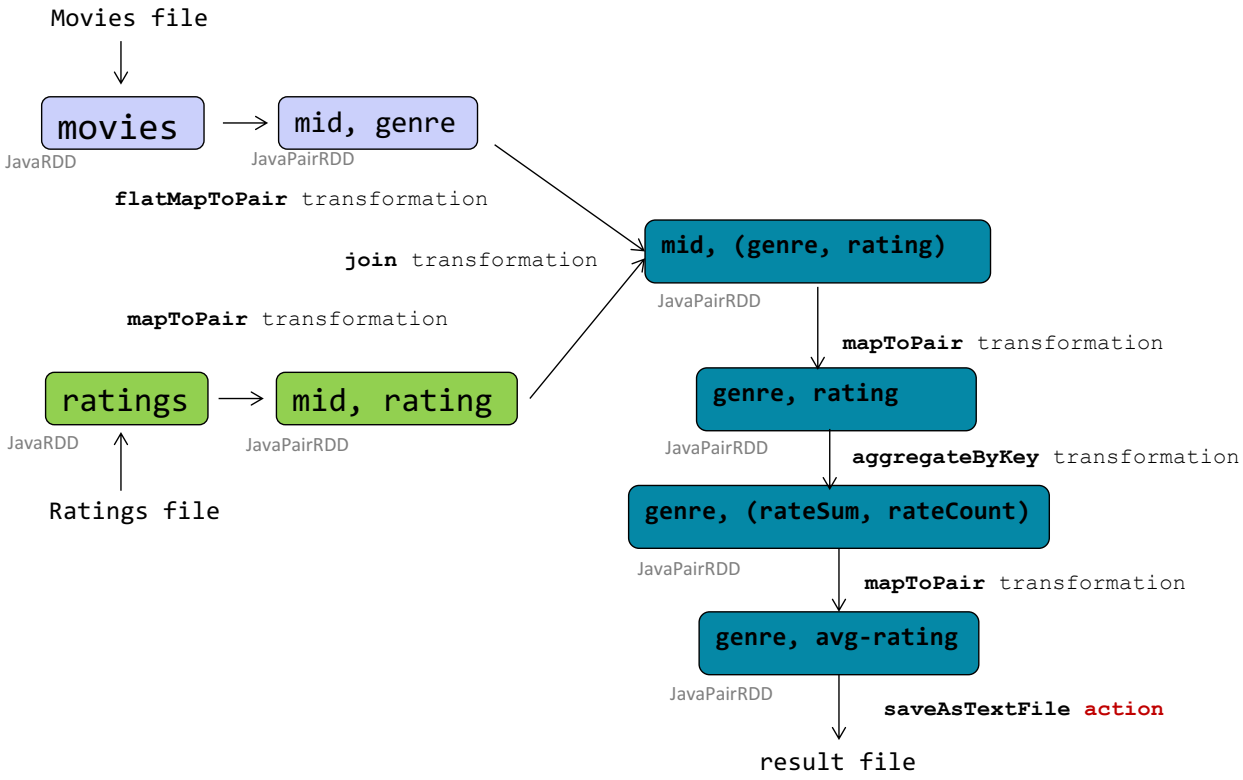
- Overview
  - Spark & Flink Skeleton Programs
  - Wordcount Example
- Spark and Flink Details
  - RDD and Data Set Operations
- Lambda Expression
- MR Design Patterns in Spark & Flink
- **A Complete Example**
  - **Various transformations**



# Sample Program

- Two data sets stored as txt files
- Movies (mid, title, genres)
  - Sample data:
  - `1,Toy Story (1995),Adventure|Animation|Children|Comedy|Fantasy`
- Ratings (uid, mid, rating, timestampe)
  - Sample data:
  - `1,253,3.0,900660748`
- We want to find out the average rate for each genre
  - We would join the two data sets on movie id (**`mid`**) and keep only the genre and rating data, we then group the rating data based on genre and find the average for each genre.

# Spark: RDD Operation Design






# Spark Application Code Snippet: I

```
public class MovieLens {

public static void main(String[] args) {
    String inputDataPath = args[0], outputDataPath = args[1];
    //Prepare Spark Context
    SparkConf conf = new SparkConf();
    conf.setAppName("Movie Lens Application");
    JavaSparkContext sc = new JavaSparkContext(conf);

    //read ratings and movie data as JavaRDD<String>
    JavaRDD<String> ratingData = sc.textFile(inputDataPath+"ratings.csv"),
                    movieData = sc.textFile(inputDataPath + "movies.csv");

    //convert ratingData to a key value pair RDD (mid, rating)
    //the text file format is: userId,movieId,rating,timestamp

    JavaPairRDD<String, Float> ratingExtraction = ratingData.mapToPair(s ->
      {  String[] values = s.split(",");
         return
           new Tuple2<String, Float>(values[1],Float.parseFloat(values[2]));
      }
    );
```

Lambda expression

In Java, a key value pair is wrapped as a type **`Tuple2`**.
The **`mapToPair`** transformation takes a function of type **`PairFunction`** as parameter, this function takes a parameter and returns a **`Tuple2`** object

**mapToPair**

```
<K2,V2> JavaPairRDD<K2,V2> mapToPair(PairFunction<T,K2,V2> f)
```

Return a new RDD by applying a function to all elements of this RDD.

# Spark Application Code Snippet: II

```
//convert movieData to a key value pair RDD of fomat (mid, genre)
//the text file format is: movieId,title,genre1|genre2|genre3

JavaPairRDD<String,String> movieGenres = movieData.flatMapToPair(s->{
    String[] values = s.split(",");
    String movieID = values[0];
    int length = values.length;
    ArrayList<Tuple2<String,String>> results = new
            ArrayList<Tuple2<String,String>>();
    if (length >=3 ){ // genre list is present
       String[] genres = values[length -1].split("\\|");
       for (String genre: genres){
              results.add(new Tuple2<String, String>(movieID, genre));
       }
    }
    return results;
}
);
```

Lambda expression

A **`map`** or **`mapToPair`** transformation always expects a one-to-one mapping between the input and output.

If you need to filter the input by some condition and only return the value that satisfy the condition, you should use **`filter`** transformation.

If an input would produce 0 or more output, you should use **`flatMap`** or **`flatMapToPair`** transformation.
The **`flatMapToPair`** transformation takes a function of type **`PairFlatMapFunction`** as parameter. This function takes a parameter and returns an **`Iterable`** collection of **`Tuple2`** objects



# mapToPair and flatMapToPair

### mapToPair

```
<K2,V2> JavaPairRDD<K2,V2> mapToPair(PairFunction<T,K2,V2> f)
```

Return a new RDD by applying a function to all elements of this RDD.

```
public interface PairFunction<T,K,V>
extends java.io.Serializable
```

A function that returns key-value pairs (Tuple2<K, V>), and can be used to construct PairRDDs.

**Method Summary**

Methods

| Modifier and Type | Method and Description |
|---|---|
| scala.Tuple2<K,V> | call(T t) |

### flatMapToPair

```
<K2,V2> JavaPairRDD<K2,V2> flatMapToPair(PairFlatMapFunction<T,K2,V2> f)
```

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

```
public interface PairFlatMapFunction<T,K,V>
extends java.io.Serializable
```

A function that returns zero or more key-value pair records from each input record. The key-value pairs are represented as scala.Tuple2 objects.

**Method Summary**

Methods

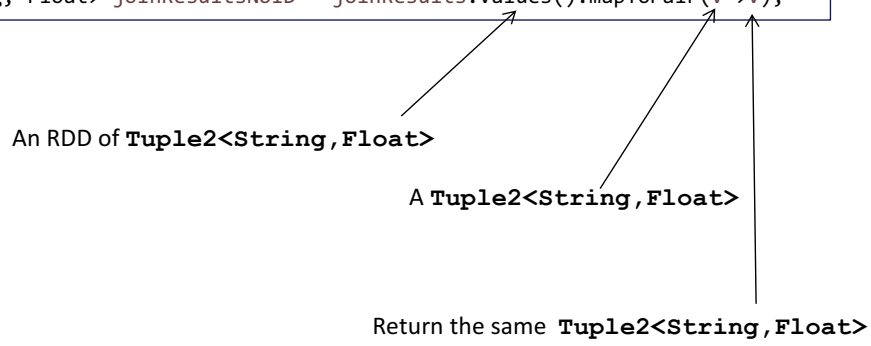
| Modifier and Type | Method and Description |
|---|---|
| Iterable<scala.Tuple2<K,V>> | call(T t) |

# Spark Application Code Snippet: III

```
    //join the two RDDs to find the ratings for each genre
    //join function performs an inner join
    //The result RDD would have the following format
    //(movieID, (genre, rating))

    JavaPairRDD<String, Tuple2<String,Float>> joinResults =
         movieGenres.join(ratingExtraction);

    //Join is based on movieID, which is not useful in our calculation
    //We only want to retain the value which is (genre, rating) and
    //convert it to a PairRDD
    JavaPairRDD<String, Float> joinResultsNoID = joinResults.values().mapToPair(v->v);
```






# Spark Application Code Snippet: IV

```
    JavaPairRDD genreRatingAvg = joinResultsNoID.aggregateByKey(
           new Tuple2<Float, Integer> (0.0f,0),
           1,
           (v1,v2)-> new Tuple2<Float, Integer> (v1._1+ v2, v1._2+1),
           (v3,v4) -> new Tuple2<Float,Integer> (v3._1 + v4._1, v3._2 + v4._2))
           .mapToPair(
             t -> new Tuple2(t._1, (t._2._1 * 1.0 / t._2._2))
           );

    // this is an action
    genreRatingAvg.saveAsTextFile(outputDataPath + "latest.rating.avg.per.genre");
    sc.close();
  }
}
```

There are a few transformations similar to the **reducer** in MapReduc framework

**groupByKey** groups the values for each key. This is like the step of preparing input for reducer in MapReduce framework

**reduceByKey** merge the values for each key using a given reduce function. This will also perform the merging locally on each “mapper” before sending results to a reducer, similarly to a "combiner" in MapReduce. But the type of the merged value should be the same as the type of the input value!

**foldByKey** is similar to **reduceByKey** except that you can supply a natural zero value.

**aggregateByKey** is more general than **reduceByKey**. It allows the merged value to have different type of the input value. It takes at least a natural zero value and two functions as parameter.

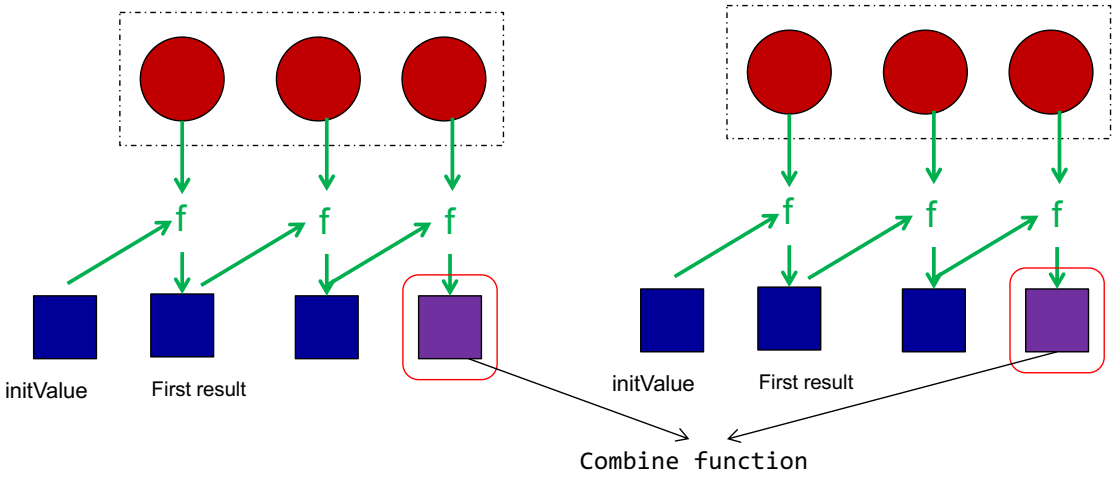
All above transformations can take extra parameter to indicate the number of partition, or a partitioner object. This is like specifying the number of **reducers** in MapReduce, or specifying a customized partitioner.

# aggregateByKey Transformation

**aggregateByKey**

```
public <U> JavaPairRDD<K,U> aggregateByKey(U zeroValue,
                                           int numPartitions,
                                           Function2<U,V,U> seqFunc,
                                           Function2<U,U,U> combFunc)
```

Aggregate the values of each key, using given combine functions and a neutral "zero value". This function can return a different result type, U, than the type of the values in this RDD, V. Thus, we need one operation for merging a V into a U and one operation for merging two U's, as in scala.TraversableOnce. The former operation is used for merging values within a partition, and the latter is used for merging values between partitions. To avoid memory allocation, both of these functions are allowed to modify and return their first argument instead of creating a new U.






# Spark Application Code Snippet: IV (revisit)

```
    JavaPairRDD genreRatingAvg = joinResultsNoID.aggregateByKey(
            new Tuple2<Float, Integer> (0.0f,0),
            1,
            (r,v)-> new Tuple2<Float, Integer> (r._1+ v2, r._2+1),
            (v1,v2) -> new Tuple2<Float,Integer> (v1._1 + v2._1, v1._2 + v2._2))
            .mapToPair(
              t -> new Tuple2(t._1, (t._2._1 * 1.0 / t._2._2))
            );

    // this is an action
    genreRatingAvg.saveAsTextFile(outputDataPath + "latest.rating.avg.per.genre");
    sc.close();
  }
}
```

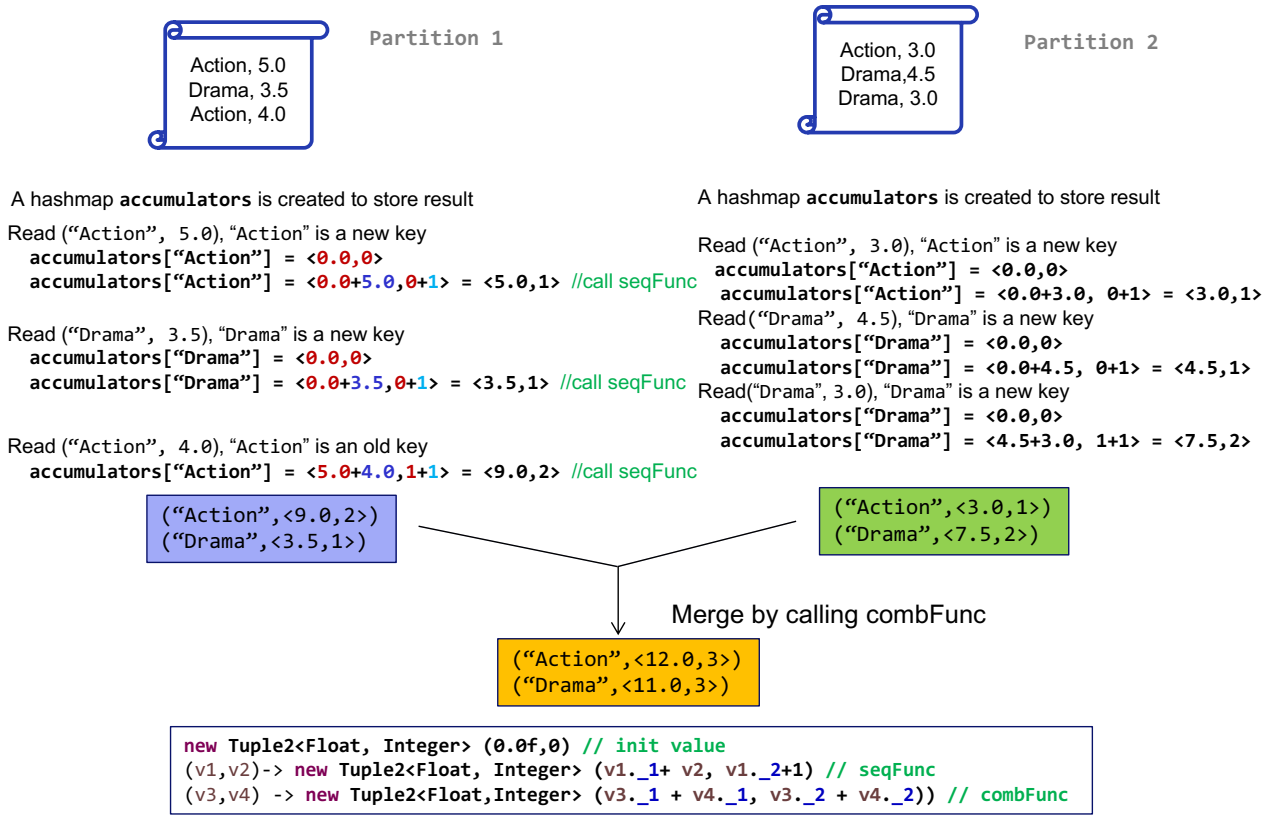
we want to use **aggregateByKey** to find out the rating sum and rating count per genre. the sum and count can be stored in a **Tuple2** object of float and integer, initialized to 0.0 and 0

For each rating value, we increment the sum and the count respectively

To merge two intermediate results, the partial sums are added up and the partial count added up

A **mapToPair** transformation is applied to calculate the average using the final sum and count

# aggregateByKey Running Example






# References


- 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.
  - https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf
- Spark Document, Spark Programming Guide
  - https://spark.apache.org/docs/1.2.0/programming-guide.html#parallelized-collections
  - Submitting Applications in Spark: https://spark.apache.org/docs/1.2.1/submitting-applications.html
- Flink 1.2 Documentation, FlinkProgramming Guide
  - https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html
- Sandy Ryza, How-to: Tune Your Apache Spark Jobs (part 1) published on March 09, 2015
  - http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/
- Sandy Ryza, How-to: Tune Your Apache Spark Jobs (part 2) published on March 30, 2015
  - http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/
- AWS Webinar about Spark on AWS: https://aws.amazon.com/webinars/anz-webinar-series/