Ch 13 Advanced RDDs

key–value RDDs - a powerful abstraction for manipulating data

custom partitioning - control exactly how data is laid out on the cluster and manipulate that individual partition accordingly

many methods on RDDs that require you to put your data in a **key–value format**.

- the method will include <some-operation>ByKey
- if **ByKey** is in a method name, it means that you can perform this only on a PairRDD type
- easiest way is to just map over your current RDD to a basic key–value structure.
- two values in each record of your RDD

```
words.map(lambda word:(word.lower(),1))
```

keyBy

- specify function that creates the key from your current value

In this case, you are keying by the first letter in the word. Spark stores the record as the value for the keyed RDD:

```
keyword=words.keyBy(lambda word:word.lower()[0])
```

mapValues

- you have a set of key–value pairs, you can begin manipulating them as such
- If a tuple, Spark will assume that the first element is the key, and the second is the value
- can explicitly choose to map-over the values (and ignore the individual keys)

```
keyword.mapValues(lambda word: word.upper()).collect()
```

flatMap over the rows
- to expand the number of rows that you have to make
- so that each row represents a character

Each character as we converted them into array:

```
keyword.flatMapValues(lambda word: word.upper()).collect()
```

in the key–value pair format, we can also extract the specific keys or values:

```
keyword.keys().collect()
keyword.values().collect()
```

sample an RDD by a set of keys

- via an **approximation** or **exactly**
- both operations can do so with or without replacement, or sampling by a fraction by a given key

- This is done via simple random sampling with one pass over the RDD, which produces a sample of size that's approximately equal to the sum of *math.ceil(numItems *samplingRate)* over all key values
- differs from sampleByKey in that you make additional passes over the RDD toc reate a sample size that's exactly equal to the sum of math.ceil(numItems * samplingRate) over all key values with a 99.99% confidence
- When sampling without replacement, you need **one additional pass over the RDD** to guarantee sample size; when sampling with replacement, you need **two additional passes**

```
import random
distinctChars = words.flatMap(lambda word: list(word.lower())).distinct()\
  .collect()
sampleMap = dict(map(lambda c: (c, random.random()), distinctChars))
words.map(lambda word: (word.lower()[0], word))\
  .sampleByKey(True, sampleMap, 6).collect()
```

reduceByKey

- more stable approach is to perform a flatMap
- then perform a map, to map each letter instance to the number one, and then perform a reduceByKey with a summation function in order to collect back the array
- implementation is much more stable because the reduce happens within each partition and doesn't need to put everything in memory
- no incurred shuffle during this operation; everything happens at each worker individually before performing a final reduce.
- greatly enhances the speed at which you can perform the operation as well as the stability of the operation
- method returns an RDD of a group (the key) and sequence of elements that are not guranteed to have an ordering
- method is completely appropriate when our workload is associative but **inappropriate when the order matters**

```
KVcharacters.reduceByKey(addFunc).collect()
```

aggregate

- function requires a null and start value and then requires you to specify two different functions
- first aggregates within partitions, the second aggregates across partitions
- start value will be used at both aggregation levels
- does have some performance implications because it **performs the final aggregation on the driver**
- If the results from the executors are too large, they can take down the driver with an OutOfMemoryError

```
nums.aggregate(0, maxFunc, addFunc)
```

treeAggregate

treeAggregate

- does the same thing as aggregate (at the user level) but does so in a different way
- basically “pushes down” some of the subaggregations (*creating a tree from executor to executor*) before performing the final aggregation on the driver
- multiple levels can help you to ensure that the driver does not run out of memory in the process of the aggregation
- these tree-based implementations are often to try to improve stability in certain operations

```
depth = 3
nums.treeAggregate(0, maxFunc, addFunc, depth)
```

aggregateByKey

- This function does the same as aggregate but instead of doing it partition by partition, it does it by key

```
KVcharacters.aggregateByKey(0, addFunc, maxFunc).collect()
```

combineByKey

- Instead of specifying an aggregation function, you can specify a **combiner**
- combiner operates on a given key and merges the values according to some function
- then goes to merge the different outputs of the combiners to give us our result
- can specify the number of output partitions as a *custom output partitioner* as well

```
def valToCombiner(value):
  return [value]
def mergeValuesFunc(vals, valToAppend):
  vals.append(valToAppend)
  return vals
def mergeCombinerFunc(vals1, vals2):
  return vals1 + vals2
outputPartitions = 6
KVcharacters\
  .combineByKey(
    valToCombiner,
    mergeValuesFunc,
    mergeCombinerFunc,
    outputPartitions)\
  .collect()
```

foldByKey

- merges the values for each key using an associative function and a neutral “zero value"
- can be added to the result an arbitrary number of times
- must not change the result (e.g., 0 for addition, or 1 for multiplication)

```
KVcharacters.foldByKey(0, addFunc).collect()
```

CoGroups

- give you the ability to group together up to three key–value RDDs together in Scala and two in Python

- joins the given values by key
- *effectively just a group-based joinon an RDD*
- can also specify a number of output partitions or a custom partitioning function to control exactly how this data is distributed across the cluster

```
import random
distinctChars = words.flatMap(lambda word: word.lower()).distinct()
charRDD = distinctChars.map(lambda c: (c, random.random()))
charRDD2 = distinctChars.map(lambda c: (c, random.random()))
charRDD.cogroup(charRDD2).take(5)
```

joins

- RDDs have much the same joins as we saw in the Structured API
- although RDDs are **much more involved for you**
- follow the same basic format:
  - the two RDDs we would like to join
  - optionally, either -
    - the number of output partitions
    - or the customer partition function to which they should output

inner join:

```
keyedChars = distinctChars.map(lambda c: (c, random.random()))
outputPartitions = 10
KVcharacters.join(keyedChars).count()
KVcharacters.join(keyedChars, outputPartitions).count()
```

Zips

- combine two RDDs, so it's worth labeling it as a join
- zip allows you to "zip" together two RDDs, assuming that they have the same length
- creates a PairRDD
- The two RDDs must have the same number of partitions as well as the same number of elements

```
numRange = sc.parallelize(range(10), 2)
words.zip(numRange).collect()
```

Controlling Partitions

- With RDDs, you have control over how data is exactly physically distributed across the cluster
- Some of these methods are basically the same from what we have in the Structured APIs
- key addition (that does not exist in the Structured APIs) is the *ability to specify a partitioning function* (formally a **custom Partitioner**)

coalesce

coalesce

- effectively collapses partitions on the same worker in order to avoid a shuffle of the data when repartitioning

ex. if our words RDD is currently two partitions, we can *collapse that to one partition* by using coalesce without bringing about a shuffle of the data:

```
words.coalesce(1).getNumPartitions() # 1
```

repartition

- operation allows you to repartition your data up or down but performs a shuffle across nodes in the process
- Increasing the number of partitions can increase the level of parallelism when operating in map- and filter-type operations

```
words.repartition(10)
```

repartitionAndSortWithinPartitions

- operation gives you the ability to repartition as well as specify the ordering of each one of those output partition
- both the partitioning and the key comparisons can be specified by the user

Custom Partitioning

- one of the primary reasons you'd want to use RDDs
- Custom partitioners are not available in the Structured APIs because they don't really have a logical counterpart
- They're a *low-level, implementation detail* that can have a **significant** effect on whether your jobs run successfully
- sole goal of custom partitioning is to even out the distribution of your data across the cluster so that you can work around problems like data skew
- If you're going to use custom partitioners, you should drop down to RDDs from the StructuredAPIs, apply your custom partitioner, and then convert it back to a DataFrame or Dataset
- get the best of both worlds, only dropping down to custom partitioning when you need to
- To perform custom partitioning you need to **implement your own class that extends Partitioner.**
- *only when you have lots of domain knowledge about your problem space*
- if you're just looking to partition on a value or even a set of values (columns), it's worth just doing it in the DataFrame API

```
df = spark.read.option("header", "true").option("inferSchema", "true")\
  .csv("/FileStore/tables/online_retail_dataset.csv")
rdd = df.coalesce(10).rdd

df.printSchema()
```

Spark has two **built-in** Partitioners that you can leverage off in the RDD API

- **HashPartitioner** for discrete values
- **RangePartitioner** - continuous values
- Spark's Structured APIs already use these - but we can use the same thing in RDDs
- hash and range partitioners are useful but rudimentary
- At times, you will need to perform some very low-level partitioning because you're working with *very large data and large key skew*
- **Key skew** simply means that some keys have many, many more values than other keys
- *You want to break these keys as much as possible to improve parallelism* and *prevent OutOfMemoryErrors* during the course of execution
- Might be that you need to partition more keys if and only if the key matches a certain format.

Ex.

- might know that there are two customers in your dataset that always crash your analysis and we need to break them up further than other customer IDs
- these two are so skewed that they need to be operated on alone
- whereas all of the others can be lumped into large groups

```
def partitionFunc(key):
  import random
  if key == 17850 or key == 12583:
    return 0
  else:
    return random.randint(1,2)

keyedRDD = rdd.keyBy(lambda row: row[6])
keyedRDD\
  .partitionBy(3, partitionFunc)\
  .map(lambda x: x[0])\
  .glom()\
  .map(lambda x: len(set(x)))\
  .take(5)
```

Kryo serialization

- Any object that you hope to parallelize (or function) **must be serializable**
- default serialization can be quite slow
- Spark can use the Kryo library (version 2) to serialize objects more quickly
- Kryo is significantly faster and more compact than Java serialization (*often as much as 10x)*
- does not support all serializable types and requires you to register the classes you'll use in the program in advance for best performance
- You can use Kryo by initializing your job with a SparkConf and setting the value of"spark.serializer" to **org.apache.spark.serializer.KryoSerializer**

- setting configures the serializer used for shuffling data between worker nodes and serializing RDDs to disk
- only reason Kryo is not the default is because of the custom registration requirement
- recommend trying it in any network-intensive application
- Since Spark 2.0.0, we internally use Kryo serializer when shuffling RDDs with simple types, arrays of simple types, or string type.
- Spark automatically includes Kryo serializers for the many commonly used core Scala classes covered in the AllScalaRegistrar from the Twitter chill library
- To register your own custom classes with Kryo, use the **registerKryoClasses** method