How-to: Translate from MapReduce to Apache Spark (Part 2)

April 28, 2015By Juliet Hougland1 Comment
Categories: How-to MapReduce Spark

The conclusion to this series covers Combiner-like aggregation functionality, counters, partitioning, and serialization.

Apache Spark is rising in popularity as an alternative to MapReduce, in a large part due to its expressive API for complex data processing. A few months ago, my colleague, Sean Owen wrote a post describing how to translate functionality from MapReduce into Spark, and in this post, I'll extend that conversation to cover additional functionality.

To briefly reiterate, MapReduce was originally designed for batch Extract Transform Load (ETL) operations and massive log processing. MapReduce relies on processing key-value pairs in map and reduce phases. Each phase has the following actions:

1. Map: Emits 0, 1, or more key-values pairs as output for every input.
2. Shuffle: Groups key-value pairs with the same keys by shuffling data across the cluster's network.
3. Reduce: Operates on an iterable of values associated with each key, often performing some kind of aggregation.

To perform complex operations, many Map and Reduce phases must be strung together. As MapReduce became more popular, its limitations with respect to complex and iterative operations became clear.

Spark provides a processing API based around Resilient Distributed Datasets (RDDs.) You can create an RDD by reading in a file and then specifying the sequence of operations you want to perform on it, like parsing records, grouping by a key, and averaging an associated value. Spark allows you to specify two different types of operations on RDDs: transformations and actions. Transformations describe how to transform one data collection into another. Examples of transformations include `map`, `flatMap`, and `groupByKey`. Actions require that the computation be performed, like writing output to a file or printing a variable to screen.

Spark uses a lazy computation model, which means that computation does not get triggered until an action is called. Calling an action on an RDD triggers all necessary transformations to be performed. This lazy evaluation allows Spark to smartly combine operations and optimize performance.

As an aid to the successful production deployment of a Spark cluster, in the rest of the blog post, we'll explore how to reproduce functionality with which you may already be familiar from MapReduce in Spark. Specifically, we will cover combiner-like aggregation functionality, partitioning data, counter-like functionality, and the pluggable serialization frameworks involved.

reduceByKey vs Combiner

This simple Mapper featured in Sean's blog post:

```
public class LineLengthMapper
        extends Mapper {
    @Override
    protected void map(LongWritable lineNumber, Text line
            throws IOException, InterruptedException {
        context.write(new IntWritable(line.getLength()), 
    }
}
```

…is part of a job that counts lines of text by their length. It's simple, but inefficient: The Mapper writes a length and count of 1 for every line, which is then written to disk and shuffled across the network, just to be added up on the Reducer. If there are a million empty lines, then a million records representing "length 0, count 1" will be copied across the network, just to be collapsed into "length 0, count 1000000" by a Reducer like the one also presented last time:

```
public class LineLengthReducer
        extends Reducer {
    @Override
    protected void reduce(IntWritable length, Iterable co
            throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable count : counts) {
            sum += count.get();
        }
        context.write(length, new IntWritable(sum));
    }
}
```

For this reason, MapReduce has the notion of a *Combiner*. A Combiner is an optimization that works like a Reducer—in fact, it must be an implementation of Reducer—that can combine multiple records on the Mapper side, before those records are written anywhere. It functions like a miniature Reducer preceding the shuffle. A Combiner must be commutative and associative, which means that its result must be the same no matter the order in which it combines records. In fact, `LineLengthReducer` itself could be applied as the Combiner in this MapReduce job, as well as being the Reducer.

Back to Spark. A terse and literal—but certainly not optimal—translation of `LineLengthMapper` and `LineLengthReducer` is:

```
linesRDD.mapPartitions { lines =>
  lines.map(line => (line.length, 1))
 }.groupByKey().mapValues(_.sum)
```

The Mapper corresponds to `mapPartitions`, the shuffle to `groupByKey`, and the Reducer to the `mapValues` call. A likewise literal translation of the Combiner would inject its logic at the end of the Mapper's analog, `mapPartitions`:

The new code uses Scala's Collections API; these are not Spark operations. As mentioned previously, the new code actually implements exactly the same logic. It's easy to see the resemblance in the expression of the two, since Spark mimics many of Scala's APIs.

Still, it's clunky. The essence of the operation is summing counts, and to know how to sum many counts it's only necessary to know how to sum two counts, and apply that over and over until just one value is left. This is what a true reduce operation does: from a function that makes two values into one, it makes many values into one.

In fact, if just given the reduce function, Spark can intelligently apply it so as to get the effect of the Combiner and Reducer above all at once:

```
linesRDD.mapPartitions { lines =>
  val mapResult = lines.map(line => (line.length, 1))
}.reduceByKey(_ + _)
```

`_ + _` is shorthand for a function of two arguments that returns their sum. This is a far more common way of expressing this operation in Spark, and under the hood, Spark will be able to apply the reduce function before and after a shuffle automatically. In fact, without the need to express the Combiner's counterpart directly in the code, it's also no longer necessary to express how to map an entire partition with mapPartitions, since it's implied by expressing how to map an element at a time:

```
linesRDD.map(line => (line.length, 1)).reduceByKey(_ + _
```

The upshot is that, when using Spark, you're often automatically using the equivalent of a Combiner. For the interested, a few further notes:

- `reduceByKey` is built on a more general operation in Spark, called `combineByKey`, which allows values to be transformed at the same time.
- For those who really are counting values, there is an even more direct Spark method for this: `linesRDD.map(_.length).countByValue()`
- And if speed is more important than accuracy, there is a much faster approximate version that relies on the HyperLogLog algorithm: `linesRDD.map(_.length).countByValueApprox()`

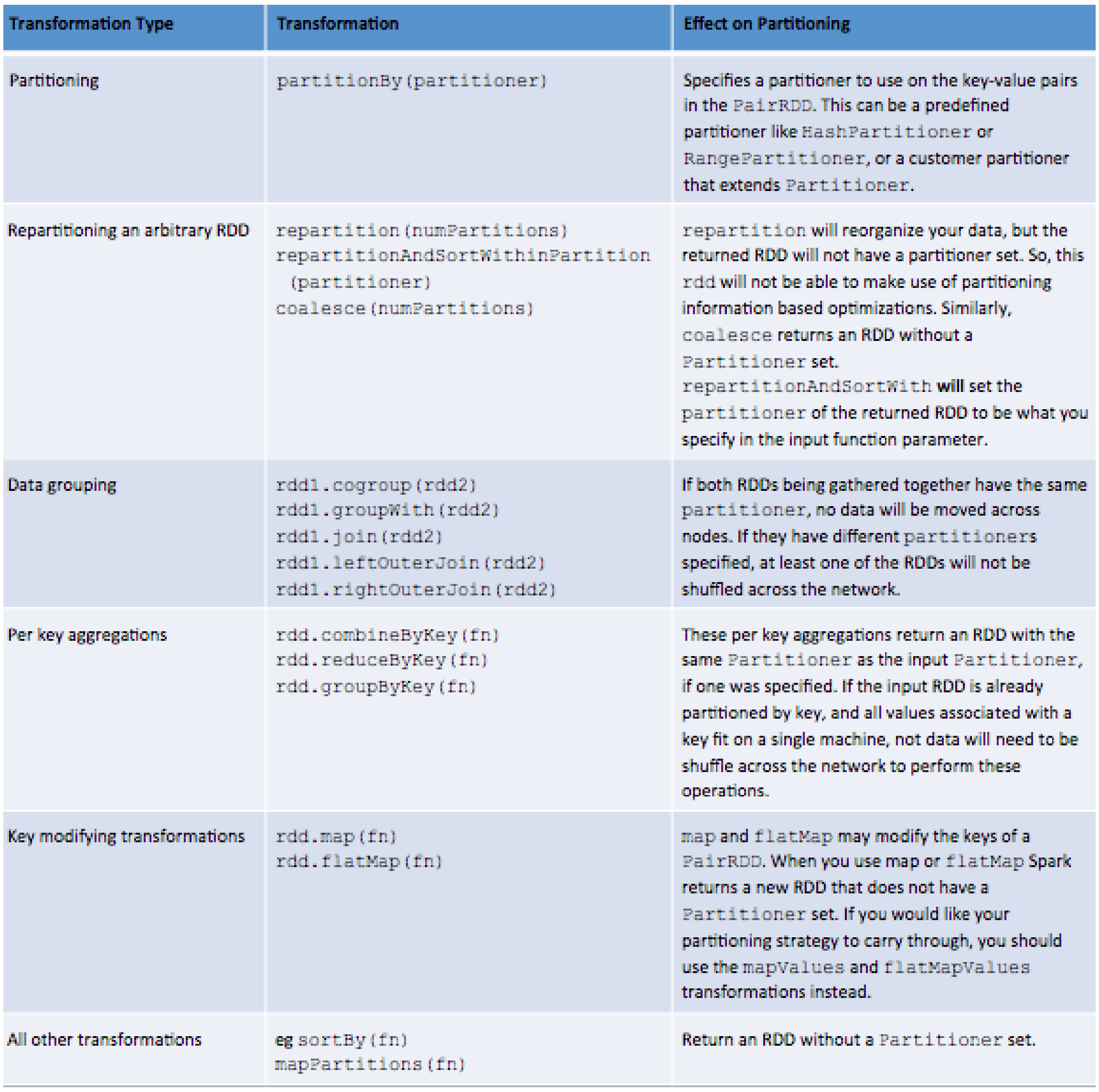
Partitioning and Grouping Data

Both Spark and MapReduce support partitioning of key-value data by key. How data is split into chunks and in turn tasks by the processing framework has a large effect on the performance of common data operations like joining disparate data sets or doing per-key aggregations.

In MapReduce, you can specify a partitioner that determines how key-value pairs are split up and organized amongst the reducers. A well-designed partitioner will approximately evenly distribute the records between the reducers. Both MapReduce and Spark use hash partitioning as their default partitioning strategy, though there are separate implementations for MapReduce and Spark. Hash partitioning works by assigning pairs to partitions based on the hash value of the key. In MapReduce and Spark the partition a key-value pair is assigned to the `hashCode()` method modulo the number of partitions you are creating. The hope is that the hashing function will evenly distribute your keys in the hash-space and you should end up with approximately evenly distributed data between partitions.

A common issue in distributed programs with per-key aggregation is seeing a long tail in the distribution of the number of records assigned to reducers, and having "straggler" reducers that take much more time to complete than the rest. You can often resolve this problem by specifying a different, potentially custom partitioner. To do this in MapReduce, you can define your own customer partitioner by extending Partitioner and specifying your custom Partitioning class in the job configuration. This can be done in the configuration file, or programmatically with `conf.setPartitionerClass(MyPartitioner.class)`.

In Spark, there are operations that benefit from partitioning as well as operations can modify partitioning. The following table explains what types of transformations can affect partitioning and how.



Transformation Type	Transformation	Effect on Partitioning
Partitioning	`partitionBy(partitioner)`	Specifies a partitioner to use on the key-value pairs in the `PairRDD`. This can be a predefined partitioner like `HashPartitioner` or `RangePartitioner`, or a customer partitioner that extends `Partitioner`.
Repartitioning an arbitrary RDD	`repartition(numPartitions)` `repartitionAndSortWithinPartition (partitioner)` `coalesce(numPartitions)`	`repartition` will reorganize your data, but the returned RDD will not have a partitioner set. So, this `rdd` will not be able to make use of partitioning information based optimizations. Similarly, `coalesce` returns an RDD without a `Partitioner` set. `repartitionAndSortWith` will set the `partitioner` of the returned RDD to be what you specify in the input function parameter.
Data grouping	`rdd1.cogroup(rdd2)` `rdd1.groupWith(rdd2)` `rdd1.join(rdd2)` `rdd1.leftOuterJoin(rdd2)` `rdd1.rightOuterJoin(rdd2)`	If both RDDs being gathered together have the same `partitioner`, no data will be moved across nodes. If they have different `partitioners` specified, at least one of the RDDs will not be shuffled across the network.
Per key aggregations	`rdd.combineByKey(fn)` `rdd.reduceByKey(fn)` `rdd.groupByKey(fn)`	These per key aggregations return an RDD with the same `Partitioner` as the input `Partitioner`, if one was specified. If the input RDD is already partitioned by key, and all values associated with a key fit on a single machine, not data will need to be shuffle across the network to perform these operations.
Key modifying transformations	`rdd.map(fn)` `rdd.flatMap(fn)`	`map` and `flatMap` may modify the keys of a `PairRDD`. When you use map or `flatMap` Spark returns a new RDD that does not have a `Partitioner` set. If you would like your partitioning strategy to carry through, you should use the `mapValues` and `flatMapValues` transformations instead.
All other transformations	eg `sortBy(fn)` `mapPartitions(fn)`	Return an RDD without a `Partitioner` set.

Counters

MapReduce allows you to count things that happen in your job, and then query that count later. To define customer counters in MapReduce, you first need to define an Enum that describes the counters you will track. Imagine you are using Jackson (`org.codehaus.jackson`) to parse JSON into a Plain Old Java Object (POJO) using a jackson ObjectMapper. In doing so, you may encounter a `JsonParseException` or `JsonMappingException`, and you would like to track how many of each you see. So, you will create an enum that contains an element for both of these possible exceptions:

```
public static enum JsonErr {
            PARSE_ERROR,
            MAPPING_ERROR
}
```

Then, in the map method of your map class you would have

```
public void map(LongWritable key, Text value, Context co
        throws IOException, InterruptedException {
      try {
/* Parse a string into a POJO */
      } catch (JsonParseException parseException) {
context.getCounter(JsonErr.PARSE_ERROR).increment(1L)
```

```
    } catch (JsonMappingException mappingException) {
context.getCounter(JsonErr.MAPPING_ERROR).increment(1L);
    }
}
```

All counters that get incremented during the course of a job will be reported to the JobTracker and displayed in the JobTracker Web UI, along with the default I/O counters. You can also access the counters from the MapReduce driver program, from the Job you create using your Configuration.

Spark exposes Accumulators, which can be used as counters, but more generally support any associative operation. Thus, you can go beyond incrementing and decrementing by integers toward summing arbitrary floating-point numbers—or even better, actually collecting samples of parsing errors you encounter.

If you were to do a literal translation of this parsing-error count to Spark it would look like:

While there does not currently exist a good way to count while performing a transformation, Spark's Accumulators do provide useful functionality for creating samples of parsing errors. This is very notably something that is more difficult to do in MapReduce. An alternative and useful strategy to take is to instead use reservoir sampling to create a sample of error messages associated with parsing errors.

Important Caveat About Accumulators

Now, you should be careful about how and when you use Accumulators. In MapReduce, increment actions on a counter executed during a task that later fails will not be counted toward the final value. MapReduce is careful to count correctly even when tasks fail or speculative execution occurs.

In Spark, the behavior of accumulators requires careful attention. It is strongly recommended that accumulators only be used in an action. Accumulators incremented in an action are guaranteed to only be incremented once. Accumulators incremented in a transformation can have their values incremented multiple times if a task or job stage is ever rerun, which is unexpected behavior for most users.

In the example below, an RDD is created and then mapped over while an accumulator is incremented. Since Spark uses a lazy evaluation model, these RDDs are only computed once an action is invoked and a value is required to be returned. Calling another action on `myRdd2` requires that the preceding steps in the workflow are recomputed, incrementing the accumulator again.

```
scala> val myrdd = sc.parallelize(List(1,2,3))
myrdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectio

scala> val myaccum = sc.accumulator(0, "accum")
myaccum: org.apache.spark.Accumulator[Int] = 0

scala> val myRdd2 = myrdd.map(x => myaccum += 1)
myRdd2: org.apache.spark.rdd.RDD[Unit] = MappedRDD[1] at

scala> myRdd2.collect
res0: Array[Unit] = Array((), (), ())

scala> myaccum
res1: org.apache.spark.Accumulator[Int] = 3

scala> myRdd2.collect
res2: Array[Unit] = Array((), (), ())

scala> myaccum
res3: org.apache.spark.Accumulator[Int] = 6
```

Beyond situations where Spark's lazy evaluation model causes transformations to be reapplied and accumulators incremented, it is possible that tasks getting rerun because of a partial failure will cause accumulators to be incremented again. The semantics of accumulators are distinctly not a once-and-only-once (aka counting) model.

The problem with the example of counting parsing errors above is that it is possible for the job to complete successfully with no explicit errors, but the numeric results may not be valid —and it would be difficult to know either way. One of the most common uses of counters in MapReduce is parsing records and simultaneously counting errors, and unfortunately there is no way to reliably count using accumulators in Spark.

Serialization Frameworks

MapReduce and Spark both need to be able to take objects in the JVM and serialize them into a binary representation to be sent across the network when shuffling data. MapReduce uses a pluggable serialization framework, which allows users to specify their own implementation(s) of `org.apache.hadoop.io.serializer.Serialization` by setting `io.serialization` in the Hadoop configuration if they wish to use a custom serializer. `HadoopWritable` and Avro specific and reflection-based serializers are configured as the default supported serializations.

Similarly, Spark has a pluggable serialization system that can be configured by setting the spark.serializer variable in the Spark configuration to a class that extends `org.apache.spark.serializer.Serialize`r. By default, Spark uses Java Serialization that works out of the box but is not as fast as other serialization methods. Spark can be configured to use the much faster Kryo Serialization protocol by setting `spark.serializer` to `org.apache.spark.serializer.KryoSerializer` and setting `spark.kryo.registrator` to the class of your own custom registrator, if you have one. In order to get the best performance out of Kryo, you should register the classes with a `KryoRegistrator` ahead of time, and configure Spark to use your particular Kryo registrator.

If you wanted to use Kryo for serialization and register a `User` class for speed, you would define your registrator like this.

```
import com.mycompany.model.User
import org.apache.spark.serializer.KryoRegistrator

class MyKryoRegistrator extends KryoRegistrator {
   override def registerClasses(kryo: Kryo) {
       kryo.register(classOf[User])
   }
}
```

You would then set spark.serializer to `spark.KryoSerializer` and `spark.kryo.registrator` to `com.mycompany.myproject.MyKryoRegistrator`. It is worth noting that if you are working with Avro objects, you will also need to specify the `AvroSerializer` class to serialize and deserialize. You would modify our `Registrator` code like so:

```
import com.mycompany.model.UserAvroRecord
import org.apache.spark.serializer.KryoRegistrator

class MyKryoRegistrator extends KryoRegistrator {
   override def registerClasses(kryo: Kryo) {
       kryo.register(classOf[UserAvroRecord], new AvroSe
   }
```

Note: while the data sent across the network using Spark will be serialized with the serializer you specify in the configuration, the closures of tasks will be serialized with Java serialization. This means anything in the closures of your tasks must be serializable, or you will get a `TaskNotSerializableException`.

For Spark to operate on the data in your RDD it must be able to serialize the function you specify in `map`, `flatMap`, `combineByKey` on the driver node, ship that serialized function to the worker nodes, deserialize it on the worker nodes, and execute it on the data. This is always done with Java Serialization, which means you can't easily have Avro objects in the closure of function in Spark because Avro objects have not been Java serializable up until version 1.8.

Conclusion

As you hopefully observed, there are similarities but also important differences between MapReduce and Spark with respect to combiner-like aggregation functionality, partitioning, counters, and pluggable serialization frameworks. Understanding these nuances can help ensure that your Spark deployment is a long-term success.

Juliet Hougland is a Data Scientist at Cloudera.

apache Avro cloudera configuration data Hadoop java log MapReduce REST Support ui