How-to: Translate from MapReduce to Apache Spark

September 2, 2014By Sean Owen (@sean_r_owen)5 Comments

Categories: How-to MapReduce Spark

The key to getting the most out of Spark is to understand the differences between its RDD API and the original Mapper and Reducer API.

Venerable MapReduce has been Apache Hadoop's work-horse computation paradigm since its inception. It is ideal for the kinds of work for which Hadoop was originally designed: large-scale log processing, and batch-oriented ETL (extract-transform-load) operations.

As Hadoop's usage has broadened, it has become clear that MapReduce is not the best framework for all computations. Hadoop has made room for alternative architectures by extracting resource management into its own first-class component, YARN. And so, projects like Impala have been able to use new, specialized non-MapReduce architectures to add interactive SQL capability to the platform, for example.

Today, Apache Spark is another such alternative, and is said by many to succeed MapReduce as Hadoop's general-purpose computation paradigm. But if MapReduce has been so useful, how can it suddenly be replaced? After all, there is still plenty of ETL-like work to be done on Hadoop, even if the platform now has other real-time capabilities as well.

Thankfully, it's entirely possible to re-implement MapReduce-like computations in Spark. They can be simpler to maintain, and in some cases faster, thanks to Spark's ability to optimize away spilling to disk. For MapReduce, re-implementation on Spark is a homecoming. Spark, after all, mimics Scala's functional programming style and APIs. And the very idea of MapReduce comes from the functional programming language LISP.

Although Spark's primary abstraction, the RDD (Resilient Distributed Dataset), plainly exposes map() and reduce() operations, these are not the direct analog of Hadoop's Mapper or Reducer APIs. This is often a stumbling block for developers looking to move Mapper and Reducer classes to Spark equivalents.

Viewed in comparison with classic functional language implementations of `map()` and `reduce()` in Scala or Spark, the Mapper and Reducer APIs in Hadoop are actually both more flexible and more complex as a result. These differences may not even be apparent to developers accustomed to MapReduce, but, the following behaviors are specific to Hadoop's implementation rather than the idea of MapReduce in the abstract:

- Mappers and Reducers always use key-value pairs as input and output.
- A Reducer reduces values per key only.
- A Mapper or Reducer may emit 0, 1 or more key-value pairs for every input.

h

- Mappers and Reducers may emit any arbitrary keys or values, not just subsets or transformations of those in the input.
- Mapper and Reducer objects have a lifecycle that spans many `map()` and `reduce()` calls. They support a `setup()` and `cleanup()` method, which can be used to take actions before or after a batch of records is processed.

This post will briefly demonstrate how to recreate each of these within Spark — and also show that it's not necessarily desirable to literally translate a Mapper and Reducer!

Key-Value Pairs as Tuples

Let's say we need to compute the length of each line in a large text input, and report the count of lines by line length. In Hadoop MapReduce, this begins with a Mapper that produces key-value pairs in which the line length is the key, and count of 1 is the value:

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

It's worth noting that Mappers and Reducers only operate on key-value pairs. So the input to `LineLengthMapper`, provided by a `TextInputFormat`, is actually a pair containing the line as value, with position within the file thrown in as a key, for fun. (It's rarely used, but, something has to be the key.)

The Spark equivalent is:

```
lines.map(line => (line.length, 1))
```

In Spark, the input is an RDD of Strings only, not of key-value pairs. Spark's representation of a key-value pair is a Scala tuple, created with the (a,b) syntax shown above. The result of the `map()` operation above is an RDD of (Int,Int) tuples. When an RDD contains tuples, it gains more methods, such as `reduceByKey()`, which will be essential to reproducing MapReduce behavior.

Reducer and reduce() versus reduceByKey()

To produce a count of line lengths, it's necessary to sum the counts per length in a Reducer:

```
public class LineLengthReducer
        extends Reducer<IntWritable,IntWritable,IntWrita
```

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

The equivalent of the Mapper and Reducer above together is a one-liner in Spark:

```
val lengthCounts = lines.map(line => (line.length, 1)).r
```

Spark's RDD API has a `reduce()` method, but it will reduce the entire set of key-value pairs to one single value. This is not what Hadoop MapReduce does. Instead, Reducers reduce all values for a key and emit a key along with the reduced value. `reduceByKey()` is the closer analog. But, that is not even the most direct equivalent in Spark; see `groupByKey()` below.

It is worth pointing out here that a Reducer's `reduce()` method receives a stream of many values, and produces 0, 1 or more results. `reduceByKey()`, in contrast, accepts a function that turns exactly two values into exactly one — here, a simple addition function that maps two numbers to their sum. This associative function can be used to reduce many values to one for the caller. It is a simpler, narrower API for reducing values by key than what a Reducer exposes.

Mapper and map() versus flatMap()

Now, instead consider counting the occurrences of only words beginning with an uppercase character. For each line of text in the input, a Mapper might emit 0, 1 or many key-value pairs:

```
public class CountUppercaseMapper
        extends Mapper<LongWritable,Text,Text,IntWritabl
    @Override
    protected void map(LongWritable lineNumber, Text lin
            throws IOException, InterruptedException {
        for (String word : line.toString().split(" ")) {
            if (Character.isUpperCase(word.charAt(0))) {
                context.write(new Text(word), new IntWri
            }
        }
    }
}
```

htt

The equivalent in Spark is:

```
lines.flatMap(
    _.split(" ").filter(word => Character.isUpperCase(wo
)
```

`map()` will not suffice here, because `map()` must produce exactly one output per input, but unlike before, one line needs to yield potentially many outputs. Again, the `map()` function in Spark is simpler and narrower compared to what the Mapper API supports.

The solution in Spark is to first map each line to an array of output values. The array may be empty, or have many values. Merely `map()`-ing lines to arrays would produce an RDD of arrays as the result, when the result should be the contents of those arrays. The result needs to be "flattened" afterward, and `flatMap()` does exactly this. Here, the array of words in the line is filtered and converted into tuples inside the function. In a case like this, it's `flatMap()` that's required to emulate such a Mapper, not `map()`.

groupByKey()

It's simple to write a Reducer that then adds up the counts for each word, as before. And in Spark, again, `reduceByKey()` could be used to sum counts per word. But what if for some reason the output has to contain the word in all uppercase, along with a count? In MapReduce, that's:

```
public class CountUppercaseReducer
        extends Reducer<Text,IntWritable,Text,IntWritabl
    @Override
    protected void reduce(Text word, Iterable<IntWritabl
            throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable count : counts) {
            sum += count.get();
        }
        context.write(new Text(word.toString().toUpperCa
    }
}
```

But `reduceByKey()` by itself doesn't quite work in Spark, since it preserves the original key. To emulate this in Spark, something even more like the Reducer API is needed. Recall that Reducer's `reduce()` method receives a key and Iterable of values, and then emits some transformation of those. `groupByKey()` and a subsequent `map()` can achieve this:

```
... .groupByKey().map { case (word,ones) => (word.toUppe
```

`groupByKey()` merely collects all values for a key together, and does not apply a reduce function. From there, any transformation can be applied to the key and Iterable of values.

Here, the key is transformed to uppercase, and the values are directly summed.

Be careful! `groupByKey()` works, but also collects all values for a key into memory. If a key is associated to many values, a worker could run out of memory. Although this is the most direct analog of a Reducer, it's not necessarily the best choice in all cases. For example, Spark could have simply transformed the keys after a call to `reduceByKey`:

```
... .reduceByKey(_ + _).map { case (word,total) => (word
```

It's better to let Spark manage the reduction rather than ask it to collect all values just for us to manually sum them.

setup() and cleanup()

In MapReduce, a Mapper and Reducer can declare a `setup()` method, called before any input is processed, to perhaps allocate an expensive resource like a database connection, and a `cleanup()` method to release the resource:

```
public class SetupCleanupMapper extends Mapper<LongWrita
    private Connection dbConnection;
    @Override
    protected void setup(Context context) {
        dbConnection = ...;
    }
    ...
    @Override
    protected void cleanup(Context context) {
        dbConnection.close();
    }
}
```

The Spark `map()` and `flatMap()` methods only operate on one input at a time though, and provide no means to execute code before or after transforming a batch of values. It looks possible to simply put the setup and cleanup code before and after a call to `map()` in Spark:

```
val dbConnection = ...
lines.map(... dbConnection.createStatement(...) ...)
dbConnection.close() // Wrong!
```

However, this fails for several reasons:

- It puts the object `dbConnection` into the map function's closure, which requires that it be serializable (for example, by implementing `java.io.Serializable`). An object like a database connection is generally not serializable.

- `map()` is a transformation, rather than an operation, and is lazily evaluated. The connection can't be closed immediately here.
- Even so, it would only close the connection on the driver, not necessarily freeing resources allocated by serialized copies.

In fact, neither `map()` nor `flatMap()` is the closest counterpart to a Mapper in Spark — it's the important `mapPartitions()` method. This method does not map just one value to one other value, but rather maps an Iterator of values to an Iterator of other values. It's like a "bulk map" method. This means that the `mapPartitions()` function can allocate resources locally at its start, and release them when done mapping many values.

Adding setup code is simple; adding cleanup code is harder because it remains difficult to detect when the transformed iterator has been fully evaluated. For example, this does not work:

```
lines.mapPartitions { valueIterator =>
    val dbConnection = ... // OK
    val transformedIterator = valueIterator.map(... dbCo
    dbConnection.close() // Still wrong! May not have ev
    transformedIterator
}
```

A more complete formulation (HT Tobias Pfeiffer) is roughly:

```
lines.mapPartitions { valueIterator =>
    if (valueIterator.isEmpty) {
        Iterator[...]()
    } else {
        val dbConnection = ...
        valueIterator.map { item =>
            val transformedItem = ...
            if (!valueIterator.hasNext) {
                dbConnection.close()
            }
            transformedItem
        }
    }
}
```

Although decidedly less elegant than previous translations, it can be done.

There is no `flatMapPartitions()` method. However, the same effect can be achieved by calling `mapPartitions()`, followed by a call to `flatMap(a => a)` to flatten.

The equivalent of a Reducer with `setup()` and `cleanup()` is just a `groupByKey()` followed by a `mapPartitions()` call like the one above. Take note of the caveat about

using `groupByKey()` above, though.

But Wait, There's More

MapReduce developers will point out that there is yet more to the API that hasn't been mentioned yet:

- MapReduce supports a special type of Reducer, called a Combiner, that can reduce shuffled data size from a Mapper.
- It also supports custom partitioning via a Partitioner, and custom grouping for purposes of the Reducer via grouping Comparator.
- The `Context` objects give access to a Counter API for accumulating statistics.
- A Reducer always sees keys in sorted order within its lifecycle.
- MapReduce has its own Writable serialization scheme.
- Mappers and Reducers can emit multiple outputs at once.
- MapReduce alone has tens of tuning parameters.

There are ways to implement or port these concepts into Spark, using APIs like the Accumulator, methods like `groupBy()` and the partitioner argument in various of these methods, Java or Kryo serialization, caching, and more. To keep this post brief, the remainder will be left to a follow-up post.

The concepts in MapReduce haven't stopped being useful. It just now has a different and potentially more powerful implementation on Hadoop, and in a functional language that better matches its functional roots. Understanding the differences between Spark's RDD API, and the original Mapper and Reducer APIs, helps developers better understand how all of them truly work and how to use Spark's counterparts to best advantage.

In Part 2 of this post, Juliet Hougland covers aggregation functionality, counters, partitioning, and serialization.

Sean Owen is Director of Data Science at Cloudera, an Apache Mahout committer/PMC member, and a Spark committer.

apache | apache hadoop | apache Mahout | cloudera | data | Data Science | developers | Hadoop | hadoop | mapreduce | impala | java | log | mahout | MapReduce | platform | release | resource management | sql | statistics | Support