Introduction to MapReduce and Workflows

Module Objectives

In this module, you will learn about:

- MapReduce and Workflows
- Configuring MapReduce jobs
- Mapping and reducing

MapReduce and Workflows

MapReduce is used to process data in batch

- MapReduce jobs can be written as in a conventional Hadoop system
- Additionally, Reactor **DataSets** can be accessed from MapReduce jobs as both input and output

Workflows are used to execute a series of MapReduce jobs

MapReduce in an Application

To process data using MapReduce, specify `withMapReduce()` in your Application specification:

```
public ApplicationSpecification configure() {
return ApplicationSpecification.Builder.with()
   ...
   .withMapReduce()
     .add(new WordCountJob())
   ...
```

You must implement the `MapReduce` interface, which requires the implementation of three methods:

- `configure()`
- `beforeSubmit()`
- `onFinish()`

---

configure() method

```
public class WordCountJob implements MapReduce {
  @Override
  public MapReduceSpecification configure() {
    return MapReduceSpecification.Builder.with()
      .setName("WordCountJob")
      .setDescription("Calculates word frequency")
      .useInputDataSet("messages")
      .useOutputDataSet("wordFrequency")
      .build();
  }
```

- Configure method similar to the one found in Flow and Application
- Defines the name and description of the MapReduce job
- Can also specify DataSets to be used as input or output for the job

---

beforeSubmit() method

- Invoked at runtime, before the MapReduce job is executed
- Passed an instance of the `MapReduceContext`
- Provides access to the actual Hadoop job configuration, as though you were running the MapReduce job directly on Hadoop

For example, you can specify the Mapper and Reducer classes as well as the intermediate data format:

```
@Override
public void beforeSubmit(MapReduceContext context) throws Exception {
  Job job = context.getHadoopJob();
  job.setMapperClass(TokenizerMapper.class);
  job.setReducerClass(IntSumReducer.class);
  job.setMapOutputKeyClass(Text.class);
  job.setMapOutputValueClass(IntWritable.class);
}
```

---

onFinish() method

- Invoked after the MapReduce job has finished
- Can perform cleanup or send a notification of job completion, if required:

```
private static final Logger LOG = LoggerFactory.getLogger(MyMapReduce.class);

@Override
public void onFinish(boolean succeeded, MapReduceContext context) throws Exception {
  LOG.info("Action taken on MapReduce job " +
                       (succeeded ? "" : "un") + "successful completion");
}
```

- Because many MapReduce jobs do not need this method, the `AbstractMapReduce` class provides a default implementation that does nothing:

```
@Override
public void onFinish(boolean succeeded, MapReduceContext context) throws Exception {
  // do nothing
}
```

Mapping and Reducing (1 of 2)

Continuuity Reactor `Mapper` and `Reducer` implement the standard Hadoop APIs:

```
public static class TokenizerMapper
    extends Mapper<byte[], byte[], Text, IntWritable> {

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void map(byte[] key, byte[] value, Context context)
      throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(Bytes.toString(value));
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

Mapping and Reducing (2 of 2)

```
public static class IntSumReducer
    extends Reducer<Text, IntWritable, byte[], byte[]> {

  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    context.write(key.copyBytes(), Bytes.toBytes(sum));
  }
}
```

Module Summary

You should be able describe:

- Describe the difference between MapReduce and Workflows
- Configure a MapReduce job
- Implement mapping and reducing

Module Completed