BDT, Praktikumsbericht 4

Gruppe 21: Maximilian Neudert, Kai Pehns

Aufgabe 1

Hadoop

Wir haben uns an dieses Beispiel gehalten. Um die richtigen Umgebungsparameter dauerhaft zu haben bietet es sich an den angegebenen Befehl beim Verbinden einfach auszuführen.

```
echo "source /etc/skel/.bashrc" > ~/.bashrc
```

Ansonsten lässt sich das Hardoop Filesystem mittels

```
hadoop fs -<>
```

mit `<>` gängiger Unix Filesystem Befehl intuitiv bedienen.

Aufgabe 2

Zuerst verbinden wir uns auf den Cluster

```
ssh <istuser>@lannisport.fbi.h-da.de
```

anschließend erstellen wir neue Ordner für das Hadoop Filesystem

```
hadoop fs -mkdir /user/<istuser>
hadoop fs -mkdir /user/<istuser>/praktikum4
```

und schließlich kopieren wir die aufgearbeiteten Daten dort hin

```
cd /mnt/datasets/Grouplens/JSON
hadoop fs -put movies_* /user/<istuser>/praktikum4
```

und überprüfen diese im Hadoop Explorer.

Aufgabe 3

Als Grundlage haben wir das vorgegebene Java Beispiel modifiziert. Für MapReduce arbeiten wir mit Key-Value-Pairs, sprich wir bauen einen Mapper, welcher aus dem JSON Input `(key, value)` Tupel erzeugt, die dann mit dem Reducer aggregiert werden. Für die Performanceanalyse besitzt `org.apache.hadoop.mapreduce.Job` die Methoden `job.getStartTime()` und `job.getFinishTime()` mit denen wir die Execution Time eines Jobs erhalten und vergleichen können.

Als Kurzzusammenfassung der Implementierung:

Wir bauen den Mapper als Erweiterung

```
public static class MoviesMapper extends Mapper
```

Iterieren durch den JSON Input (exemplarisch für MovieRating, Abfrage 4):

```
JSONParser jsonParser = new JSONParser();
JSONObject movieObject = (JSONObject) jsonParser.parse(value.toString());
JSONArray ratingsArrayObject = (JSONArray) movieObject.get("ratings");
Iterator i = ratingsArrayObject.iterator();
```

und speichern die erzeugten Key-Value-Pairs in einen Context

```
context.write(key, value);
```

und analog bauen wir einen Reducer

```
public static class TitlesReducer extends Reducer
```

Welcher als Input den Output vom Mapper erwartet und dieses zu neuen Key-Value-Pairs in einen Context aggregiert.

Anschließend erstellen wir einen Job

```
Job job = Job.getInstance(conf, "MovieRating");
job.setJarByClass(MovieRating.class);
```

Übergeben an diesen den Mapper, Reducer, Input und den Output als Parameter

```
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
job.setOutputFormatClass(TextOutputFormat.class);

job.setMapperClass(MoviesMapper.class);
job.setReducerClass(TitlesReducer.class);
```

führen diesen aus

```
int returnValue = job.waitForCompletion(true) ? 0 : 1;
```

und analysieren schlussendlich den Output

```
if (job.isSuccessful()) {
    System.out.println("Job was successful");
    System.out.println("Time: " + String.valueOf(job.getStartTime() -
job.getFinishTime()));
} else if (!job.isSuccessful()) {
    System.out.println("Job was not successful");
}
return returnValue;
```

Die genauen Quellcodes sind im Anhang. Zur Ausführung der Jobs muss man zuerst den Java Quellcode in eine `jar` kompilieren

```
hadoop com.sun.tools.javac.Main ~/MovieRating.java
jar cf mr.jar MovieRating*.class
```

danach sicherstellen, dass nicht schon ein Outputfolder existiert (sonst Fehler)

```
hadoop fs -rm -r /user/istuser/praktikum4/output
```

und schlussendlich den Hadoop Job starten

```
hadoop jar mr.jar MovieRating \
-libjars /mnt/datasets/libs/json-simple-1.1.1.jar \
/user/istuser/praktikum4/movies_20m.json \
/user/istuser/praktikum4/output
```

Ergebnisse der einzelnen Queries:

1. Abfrage

- 1m: 14575ms
- 10m: 28358ms
- 20m: 39325ms
- Map: Zählt alle Ratings pro Movie.
- Reduce: Addiert alle Resultate von Map.

2. Abfrage

- 1m: 12337ms
- 10m: 13417ms
- 20m: 15461ms
- Map: Schaut, ob ein gegebener Film ein Rating von User 10 hat. Wenn ja, wird der Titel zum Context hinzugefügt.
- Reduce: Formatiert alle übrigen Titel in einen Output-String.

Alternativ hätte man hier auch alle Filmtitel zum Context hinzufügen können mit Filmname als Key und True/False als Value und im Reducer anschließend alle Filme mit True in den Output geschrieben. Da wir aber die Möglichkeit haben gleich im ersten Schritt die zu verarbeitenden Daten zu reduzieren, haben wir genau dies getan.

3. Abfrage

- 1m: 17452ms
- 10m: 31570ms
- 20m: 42186ms
- Map: Bildet Key-Value Paare mit `userId` als Key und `1` als Value per Iteration durch alle Ratings jedes Films.
- Reduce: Aggregation mittels Summe aller Values pro Key.

4. Abfrage

- 1m: 12234ms
- 10m: 16836ms
- 20m: 17637ms
- Map: Zählt und addiert alle Ratings pro Film. Wenn das durchschnittliche Rating größer gleich 4 ist, wird der Titel zum Context hinzufügt.
- Reduce: Formatiert alle übrigen Titel in einen Output-String.

Ähnlich wie bei 2. hätte man hier auch Alle Filme mit jeweils der Durchschnittsbewertung zum Context hinzufügen und dann im Reducer mit Bedingung aggregieren können.

Anhang

1. Abfrage

```
public class RatingCounter extends Configured implements Tool {
    public static class MoviesMapper extends Mapper<Object, Text, Text,
IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
            try {
                JSONParser jsonParser = new JSONParser();
                JSONObject movieObject = (JSONObject)
jsonParser.parse(value.toString());
                JSONArray ratingsArrayObject = (JSONArray)
movieObject.get("ratings");
                Iterator i = ratingsArrayObject.iterator();
                while (i.hasNext()) {
                    JSONObject ratingsObject = (JSONObject) i.next();
                    word.set("SumRatings");
                    context.write(word, one);
                }
            } catch (ParseException ex) {
                Logger.getLogger(RatingCounter.class.getName()).log(Level.SEVERE,
null, ex);
            }
        }
    }

    public static class RatingsReducer extends Reducer<Text, IntWritable, Text,
IntWritable> {
        private IntWritable result = new IntWritable();
        public void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new RatingCounter(), args);
        System.exit(exitCode);
    }

    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.printf("Usage: %s needs two arguments, input and output
files\n", getClass().getSimpleName());
            return -1;
        }

    // when implementing tool
        Configuration conf = this.getConf();

    // create job
    Job job = Job.getInstance(conf, "RatingCounter");
        job.setJarByClass(RatingCounter.class);

    // set the input and output path
    FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

    // set the key and value class
    job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setOutputFormatClass(TextOutputFormat.class);

    // set the mapper and reducer class
        job.setMapperClass(MoviesMapper.class);
        job.setReducerClass(RatingsReducer.class);


    // wait for the job to finish
        int returnValue = job.waitForCompletion(true) ? 0 : 1;

      // monitor & output execution time
        if (job.isSuccessful()) {
            System.out.println("Job was successful");
            System.out.println("Time: " + String.valueOf(job.getStartTime() -
job.getFinishTime()));
        } else if (!job.isSuccessful()) {
            System.out.println("Job was not successful");
        }
        return returnValue;
    }


}
```

2. Abfrage

```
public class TenTitles extends Configured implements Tool {
    public static class MoviesMapper extends Mapper<Object, Text, Text, Text> {
        private Text word = new Text();
    private Text word2 = new Text();
        public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
```

```
            try {
                JSONParser jsonParser = new JSONParser();
                JSONObject movieObject = (JSONObject)
jsonParser.parse(value.toString());
                JSONArray ratingsArrayObject = (JSONArray)
movieObject.get("ratings");
                Iterator i = ratingsArrayObject.iterator();
        boolean ten = false;
                while (i.hasNext()) {
                    JSONObject ratingsObject = (JSONObject) i.next();
                    if ((Long) ratingsObject.get("userId") == 10) {
            ten = true;
          }

                }

        if (ten) {
          word.set("Title");
          word2.set((String) movieObject.get("title"));
          context.write(word, word2);
        }

            } catch (ParseException ex) {
                Logger.getLogger(TenTitles.class.getName()).log(Level.SEVERE,
null, ex);
            }
        }
    }

    public static class TitlesReducer extends Reducer<Text, Text, Text, Text> {
    private Text word = new Text();
        public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException, InterruptedException {
      String titles = "";
            for (Text val : values) {
                titles += val + ", ";
            }
            word.set(titles);

            context.write(key, word);
        }
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new TenTitles(), args);
        System.exit(exitCode);
    }

    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.printf("Usage: %s needs two arguments, input and output
files\n", getClass().getSimpleName());
            return -1;
        }
    // when implementing tool
        Configuration conf = this.getConf();

    // create job
    Job job = Job.getInstance(conf, "TenTitles");
        job.setJarByClass(TenTitles.class);

    // set the input and output path
    FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

    // set the key and value class
    job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.setOutputFormatClass(TextOutputFormat.class);

    // set the mapper and reducer class
        job.setMapperClass(MoviesMapper.class);
        job.setReducerClass(TitlesReducer.class);


    // wait for the job to finish
        int returnValue = job.waitForCompletion(true) ? 0 : 1;

      // monitor & output execution time
        if (job.isSuccessful()) {
            System.out.println("Job was successful");
            System.out.println("Time: " + String.valueOf(job.getStartTime() -
job.getFinishTime()));
        } else if (!job.isSuccessful()) {
            System.out.println("Job was not successful");
        }
        return returnValue;
    }


}
```

3. Abfrage

```
public class UserRatingCounter extends Configured implements Tool {
    public static class MoviesMapper extends Mapper<Object, Text, Text,
IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
            try {
                JSONParser jsonParser = new JSONParser();
                JSONObject movieObject = (JSONObject)
jsonParser.parse(value.toString());
                JSONArray ratingsArrayObject = (JSONArray)
movieObject.get("ratings");
                Iterator i = ratingsArrayObject.iterator();
                while (i.hasNext()) {
                    JSONObject ratingsObject = (JSONObject) i.next();
                    word.set(Long.toString((Long)ratingsObject.get("userId")));
                    context.write(word, one);
                }
            } catch (ParseException ex) {

Logger.getLogger(UserRatingCounter.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }

    public static class RatingsReducer extends Reducer<Text, IntWritable, Text,
IntWritable> {
        private IntWritable result = new IntWritable();
        public void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new UserRatingCounter(), args);
        System.exit(exitCode);
    }

    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.printf("Usage: %s needs two arguments, input and output
files\n", getClass().getSimpleName());
            return -1;
        }

    // when implementing tool
        Configuration conf = this.getConf();

    // create job
    Job job = Job.getInstance(conf, "UserRatingCounter");
        job.setJarByClass(UserRatingCounter.class);

    // set the input and output path
    FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

    // set the key and value class
    job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setOutputFormatClass(TextOutputFormat.class);

    // set the mapper and reducer class
        job.setMapperClass(MoviesMapper.class);
        job.setReducerClass(RatingsReducer.class);


    // wait for the job to finish
        int returnValue = job.waitForCompletion(true) ? 0 : 1;

      // monitor & output execution time
        if (job.isSuccessful()) {
            System.out.println("Job was successful");
            System.out.println("Time: " + String.valueOf(job.getStartTime() -
job.getFinishTime()));
        } else if (!job.isSuccessful()) {
            System.out.println("Job was not successful");
        }
        return returnValue;
    }


}
```

4. Abfrage

```
public class MovieRating extends Configured implements Tool {
    public static class MoviesMapper extends Mapper<Object, Text, Text, Text> {
        private Text word = new Text();
    private Text word2 = new Text();
        public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
            try {
                JSONParser jsonParser = new JSONParser();
                JSONObject movieObject = (JSONObject)
jsonParser.parse(value.toString());
                JSONArray ratingsArrayObject = (JSONArray)
movieObject.get("ratings");
                Iterator i = ratingsArrayObject.iterator();
        Long ratings = 0L;
        Long sum = 0L;
                while (i.hasNext()) {
                    JSONObject ratingsObject = (JSONObject) i.next();
          ratings++;
          sum += (Long)ratingsObject.get("rating");
                }
        if (ratings == 0)
          return;
        if (sum / ratings >=4) {
          word.set("title");
          word2.set((String) movieObject.get("title"));
          context.write(word, word2);
        }
            } catch (ParseException ex) {
                Logger.getLogger(MovieRating.class.getName()).log(Level.SEVERE,
null, ex);
            }
        }
    }

    public static class TitlesReducer extends Reducer<Text, Text, Text, Text> {
    private Text word = new Text();
        public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException, InterruptedException {
      String titles = "";
            for (Text val : values) {
                titles += val + ", ";
            }
            word.set(titles);

            context.write(key, word);
        }
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new MovieRating(), args);
        System.exit(exitCode);
    }

    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.printf("Usage: %s needs two arguments, input and output
files\n", getClass().getSimpleName());
            return -1;
        }

    // when implementing tool
        Configuration conf = this.getConf();

    // create job
    Job job = Job.getInstance(conf, "MovieRating");
        job.setJarByClass(MovieRating.class);

    // set the input and output path
    FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

    // set the key and value class
    job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.setOutputFormatClass(TextOutputFormat.class);

    // set the mapper and reducer class
```

```
        job.setMapperClass(MoviesMapper.class);
        job.setReducerClass(TitlesReducer.class);


    // wait for the job to finish
        int returnValue = job.waitForCompletion(true) ? 0 : 1;

      // monitor & output execution time
        if (job.isSuccessful()) {
            System.out.println("Job was successful");
            System.out.println("Time: " + String.valueOf(job.getStartTime() -
job.getFinishTime()));
        } else if (!job.isSuccessful()) {
            System.out.println("Job was not successful");
        }
        return returnValue;
    }


}
```