Fachhochschule Münster University of Applied Sciences

Wirtschafts-informatik

Masterstudiengang Wirtschaftsinformatik

Big Data Engineering

FH Münster

Master Wirtschaftsinformatik

Wintersemester 2016

Dozent: Lars George

Einheit 3

- Rückblick auf Einheit 2
- Debugging von MapReduce Code
- Weiterführung in MapReduce Konzepte
- Erweiterte MapReduce Beispiele
- **Hauptziel**: MapReduce Kenntnisse vertiefen und abschließen

- **Übung 2**:
  - MapReduce Programme schreiben und ausführen

Einheit 3

- **Rückblick auf Einheit 2**
- Debugging von MapReduce Code
- Vertiefung in MapReduce Konzepte
- Erweiterte MapReduce Beispiele

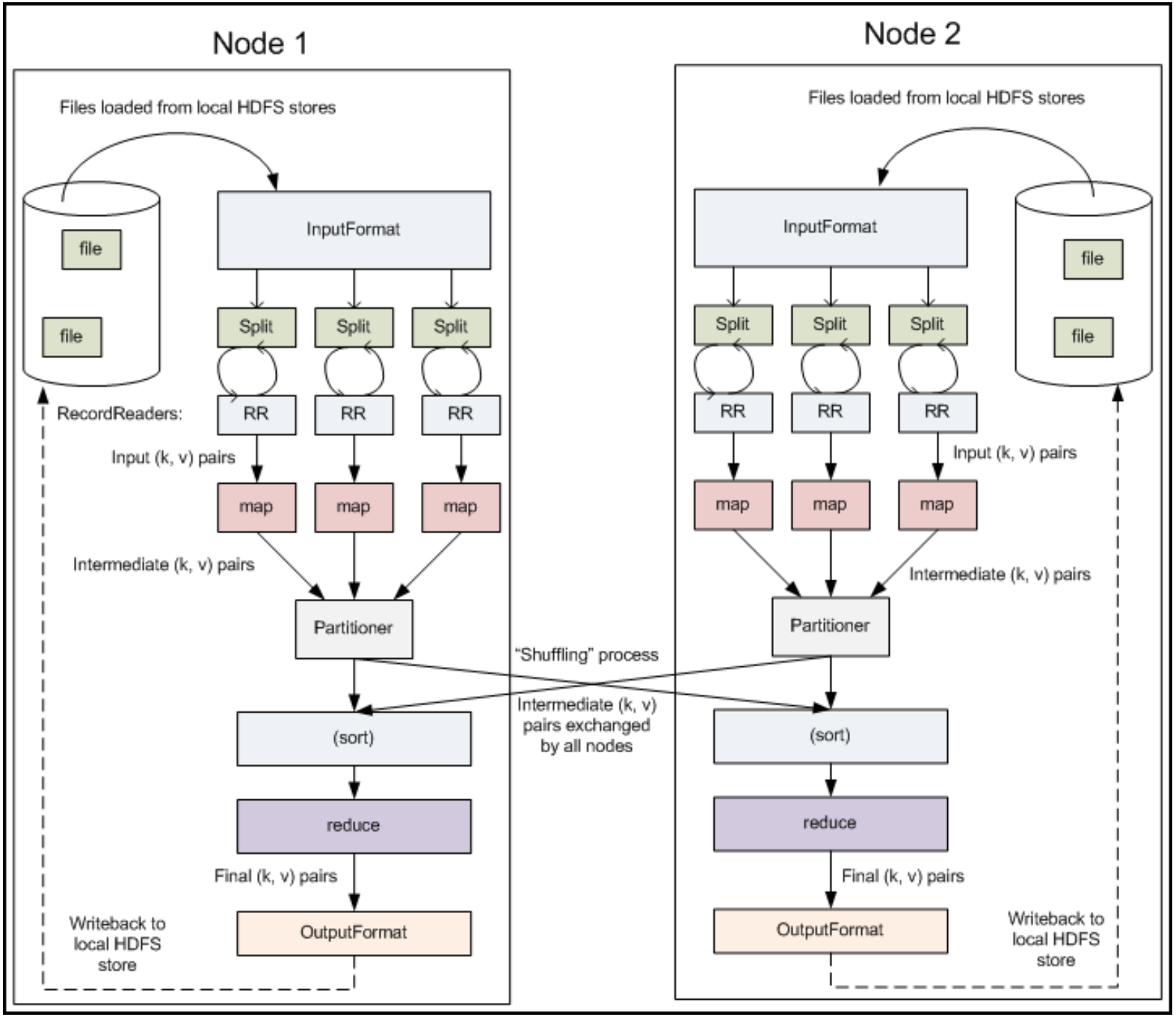
Rückblick auf Einheit 2

- Fragen?
- MapReduce Erlebnisse?
  - Portieren nach YARN?
  - MapReduce vs. MapRed?
  - MapReduce Uis?
  - Scheduler?
- Übung 2: Log Datei mit Auswertung?

Wdhg.: Einführung in MapReduce

Wichtige Konzepte:

- **Parallele** und **verteilte** Verarbeitung
- Liest Daten aus **Dateien** oder **Datenbanken**
- Versucht **Lokalität** der Daten vorteilhaft zu nutzen
- Inspiriert von `map` und `reduce` Funktionen der **funktionalen** Programmierung und gleichzeitig **Namensgeber**
- Google hat **Patent** seit 2010 für MapReduce


Node 1
Files loaded from local HDFS stores
file
file
InputFormat
Split
Split
Split
RecordReaders:
RR
RR
RR
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Partitioner
(sort)
reduce
Final (k, v) pairs
OutputFormat
Writeback to local HDFS store
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
Node 2
Files loaded from local HDFS stores
InputFormat
file
file
Split
Split
Split
RR
RR
RR
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Partitioner
(sort)
reduce
Final (k, v) pairs
OutputFormat
Writeback to local HDFS store

Einheit 3

- Rückblick auf Einheit 2
- **Debugging von MapReduce Code**
- Vertiefung in MapReduce Konzepte
- Erweiterte MapReduce Beispiele

Debugging MapReduce Code

Eine Herausforderung ist es **verteilten** Code zu **debuggen**, wenn um Beispiel das Ergebnis eines Jobs **nicht** das Erwartete ist.

Dazu gibt es **verschiedene** Techniken, welche oft in **Kombination** angewendet werden:

- **MRUnit** für Unit Tests
- Job **Counter** für Statistiken
- **Logging** im Code

Debugging MapReduce Code

Diese Techniken werden dann entweder im **Cluster** oder **lokal** eingesetzt, während der **Entwicklung** oder in der **Produktion**.

Eine weitere Technik ist das **Samplen** der Daten, d. h. das Einschränken der Datenmenge auf eine statistische **relevante**, oder manchmal auch **zufällige** ausgewählte Untermenge.

Frage: Wie kann man in MapReduce die Datenmenge einschränken?

Datensampling

Für das Sampling wird eine **neue** `InputFormat` Klasse benötigt, welche die `getSplits()` Methode **überlädt** und die Splitauswahl **einschränkt**.

Durch den Zugriff auf die **originalen** Splits kann der Code **geschickt** eine Untermenge auswählen, z. B. **basierend** auf Kenntnisse der originalen Daten oder über **Parameter** aus der Job Konfiguration.

Datensampling

```
public List<InputSplit>
getSplits(JobContext job)
throws IOException {
  List<InputSplit> originalSplits =
    super.getSplits(job);
  List<InputSplit> newSplits =
    new ArrayList<InputSplit>
   (originalSplits.size());
  ...
  return newSplits;
}
```

MRUnit

Bevor ein MapReduce Job als **Ganzes** getestet wird, sollte man sicher stellen, dass die eigentlichen Map und Reduce Funktionen **richtig** funktionieren. Dies wird in der **normalen** Programmierung in Java über sogenannte **Unit Tests** (JUnit) erledigt, welche den Code auf der kleinsten Ebene testen. Für MapReduce gibt es dazu **analog** MRUnit.

Siehe: http://mrunit.apache.org/

MRUnit Beispiel

Mapper Test

```
...
import org.apache.hadoop.mrunit.mapreduce.MapDriver;
import org.junit.*;
public class TokenizingMapperTest {
  @Test
  public void processesValidRecord()
  throws IOException, InterruptedException {
    Text value = new Text("Aber, Ende.");
    new MapDriver<LongWritable, Text, Text, IntWritable>()
      .withMapper(new TokenizingMapper())
      .withInput(new LongWritable(), value)
      .withOutput(new Text("aber"), new IntWritable(1))
      .withOutput(new Text("ende"), new IntWritable(1))
      .runTest();
  }
}
```

MRUnit Beispiel

Reducer Test

```
@Test
public void returnsSumOfValues()
throws IOException, InterruptedException {
  new ReduceDriver<Text, IntWritable, Text, IntWritable>()
    .withReducer(new SummingReducer())
    .withInputKey(new Text("aber"))
    .withInputValues(Arrays.asList(
      new IntWritable(1), new IntWritable(5)))
    .withOutput(new Text("aber"), new IntWritable(6))
    .runTest();
}
```

MRUnit

MRUnit **kann** nicht nur Map und Reduce Funktionen über Eingabe- und Ausgabepaare testen, sondern **auch** über Job **Counter** (Zähler).

Diese erlauben es **Statistiken** über die Daten eines Jobs auszugeben. Während des Testens einer Map oder Reduce Funktion können diese Zähler auch geprüft werden:

```
assertEquals("Expected 1 counter increment",
  1, mapDriver.getCounters().findCounter(
  MyJobCounter.ValidRecords).getValue());
```

Job Counter

Schauen wir uns diese Zähler genauer an. Während ein Job **ausgeführt** wird kann der Benutzercode **eigene** Zähler definieren, entweder als **ENUM**, oder **dynamisch**.

Letzteres ist sehr **nützlich** für wenige, aber nicht **vorausplanbare** Werte, also solche die von den Daten oder dem Parsen **abhängen**.

Die Ergebnisse werden während und am Ende eines Jobs ausgegeben.

Job Counter

```
public class MyMapper extends MapReduceBase implements
    Mapper<Text, Text, Text, Text> {

  static enum RecordCounters { TYPE_A, TYPE_B, TYPE_UNKNOWN };

  // Implementierung hier weggelassen
  public boolean isTypeARecord(Text input) { ... }
  public boolean isTypeBRecord(Text input) { ... }

  public void map(Text key, Text val, OutputCollector<Text, Text> output,
      Reporter reporter) throws IOException {

    if (isTypeARecord(key)) {
      reporter.incrCounter(RecordCounters.TYPE_A, 1);
    } else if (isTypeBRecord(key)) {
      reporter.incrCounter(RecordCounters.TYPE_B, 1);
    } else {
      reporter.incrCounter(RecordCounters.TYPE_UNKNOWN, 1);
    }

    // actually process the record here, call
    // output.collect( .. ), etc.
  }
}
```

Job Counter

Während der Entwicklung und auch später im produktiven Einsatz kann es sinnvoll sein, diese Zähler für das „Debuggen“ eines Jobs zu nutzen.

Beispiel:

```
context.getCounter("Sprachen",
  parser.getRequestLanguage()).increment(1);
try { ... } catch (Exception e) {
  context.getCounter("Fehler",
   e.getClass.getSimpleName()).increment(1);
}
```

Beachte: Neue API!

Job Counter

Bei dynamischen Zählern sollte **sichergestellt** sein, dass deren **Anzahl** im Rahmen bleibt (d. h. weniger als 100 bis 200). Ansonsten kann es sein, dass der Job überhaupt **nicht** ganz läuft.

Die Counter werden am **Ende** durch den JobTracker **zusammengezählt**, sie sind also „thread-safe“ (oder auch Task-safe).

Fehlgeschlagene Tasks werden **verworfen** und **nicht** addiert.

Logging

Man kann auch die **normalen** Logging Pakete in Java benutzen, um **Nachrichten** aus einem MapReduce Job zu **protokollieren**. Selbst `System.{out|err}.println()` ist möglich.

Die Werte werden **pro** Task in einem **separaten** Verzeichnis abgelegt und sind über die JobTracker UI **zugänglich**.

Hadoop benutzt Log4J und den Apache Commons Logging Wrapper.

Logging

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.mapreduce.Mapper;
public class LoggingIdentityMapper ... {
  private static final Log LOG = LogFactory
    .getLog(LoggingIdentityMapper.class);
  @Override
  public void map(KEYIN key, VALUEIN value, Context context)
  throws IOException, InterruptedException {
    // Log to stdout file
    System.out.println("Map key: " + key);
    // Log to syslog file
    LOG.info("Map key: " + key);
    if (LOG.isDebugEnabled()) {
      LOG.debug("Map value: " + value);
    }
    context.write((KEYOUT) key, (VALUEOUT) value);
  }
}
```

Logging

In neueren Versionen (>=0.22) von Hadoop kann man die Log Levels über Parameter anpassen. Dazu muss man `mapred.map.child.log.level` und/oder `mapred.reduce.child.log.level` setzen. Beispiel:

```
$ hadoop jar hadoop-examples.jar
LoggingDriver -D
mapred.map.child.log.level=DEBUG
input/ncdc/sample.txt logging-out
```

Logging

Eine andere Variante für das Setzen des Log Levels pro Job Ausführung ist ein Job Parameter welcher im Benutzercode ausgewertet wird.

Dazu muss der Driver den `-D DEBUG` Parameter abfragen und diesen dem Job in die Konfiguration mit übergeben. In der Methode `configure()` (später dazu mehr) der Map oder Reduce Klasse wird wieder geprüft und dann dort der Log Level programmatisch gesetzt.

Logging & Debugging

Abschließend sei noch erwähnt, dass Logging im MapReduce Code selbst auch **Probleme** bereiten kann, wenn die Ausgabe der Nachrichten **extrem** hoch ist.

Man kann auch mit **Remote** Debugging in Java sich an den Prozess **anschließen** und über eine IDE den Code untersuchen. Dies ist aber sehr **aufwendig** (Tasks haben ein Timeout!) und sollte nur als **letzter** Ausweg gelten.

Lokale Ausführung

Zum **Testen** eines Jobs ist es sinnvoll zuerst den **lokalen** Rechner zu nutzen. Dieser kann meistens (selbsterklärend) nicht die gleichen Datenmengen verarbeiten, führt aber den Job **komplett** aus und ist damit ein **weiterer** Schritt nach dem Unit Test.

Außerdem wird der Job **nicht** im Rahmen des JobTracker‘s ausgeführt, sondern völlig **autonom**. Also gibt es **keine** UI!

Lokale Ausführung

Jeder Hadoop Prozess **liest** dessen Einstellungen aus **lokalen** Konfigurationsdateien aus. Für MapReduce enthalten diese den **Namen** des JobTracker‘s oder „**andere**“ Werte unter demselben Schlüssel, um die **lokale** Ausführung zu ermöglichen.

Es folgt eine **minimale** Beispieldatei und dann ein Liste der **möglichen** Werte.

Lokale Ausführung

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>file:///</value>
  </property>
  <property>
    <name>mapred.job.tracker</name>
    <value>local</value>
  </property>
</configuration>
```

Lokale Ausführung

Wichtig sind hierbei die Werte des JobTracker‘s und des HDFS NameNode‘s. Diese bestimmen, wie und wo ein Job ausgeführt wird.

fs.default.name	mapred.job.tracker	Bedeutung
file:///	local	Lokale Ausführung zum Testen in einem Thread
hdfs://localhost/	localhost:8021	Ausführung lokal aber in JobTracker und TaskTracker
hdfs://namenode/	jobtracker:8021	Echte verteilte Ausführung auf einem Cluster

Lokale Ausführung

In der Praxis werden diese Dateien alle in verschiedenen Verzeichnissen unter `/etc/hadoop/conf-<name>` angelegt und dann über Parameter genutzt.

Beispiele:

```
$ hadoop fs -conf conf/conf-localhost.xml \
  -ls .
$ export HADOOP_CONF_DIR=/etc/hadoop/conf-local
$ hadoop fs -put foo.txt .
```

Tool und Driver

Für das Ausführen eines Job im **lokalen** Testmodus ist es **sinnvoll** eine weitere, von Hadoop **mitgelieferte** Funktion auszunutzen, nämlich die `Tool`, `ToolRunner` und `GenericOptionsParser` Klassen.

Diese helfen einen Job **direkt** auszuführen und gleichzeitig **bestimmte** Parameter auf der Kommandozeile zu unterstützen.

Tool und Driver

```
public class WordCountDriver extends Configured
implements Tool {

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.printf("Usage: %s [generic options] \
        <input> <output>\n“,
        getClass().getSimpleName());
      ToolRunner.printGenericCommandUsage(System.err);
      return -1;
    }
    Job job = new Job(getConf(), "Word Count");
    job.setJarByClass(getClass());
    ...
```

Tool und Driver

```
    ...
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.setMapperClass(TokenizingMapper.class);
    job.setCombinerClass(SummingReducer.class);
    job.setReducerClass(SummingReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    return job.waitForCompletion(true) ? 0 : 1;
  }

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

Tool und Driver

Durch das **Parsen** der Kommandozeile mit Hilfe des `GenericOptionsParser` kann man **einzelne** Werte der Konfiguration **überschreiben**.

Beispiele:

```
$ hadoop fs -conf conf/conf-localhost.xml \
  -ls .
$ hadoop jar WordCount -D mapred.job. \
  tracker=local job.jar ...
$ hadoop jar job.jar WordCount -jt local ...
```

Entwicklung & Debugging

Hier **beenden** wir die Diskussion über das Entwickeln und Debuggen von MapReduce Programmen.

Es folgt nun eine **Vertiefung** in MapReduce Konzepte, also **erweiterte** Funktionalitäten der schon **bekannten** Klassen, aber solcher die bis jetzt noch **nicht** besprochen wurden.

Einheit 3

- Rückblick auf Einheit 2
- Debugging von MapReduce Code
- **Vertiefung in MapReduce Konzepte**
- Erweiterte MapReduce Beispiele

MapReduce vs. MapRed

Aus **historischen** Gründen kommt MapReduce in **zwei** verschiedenen Varianten. Wir haben uns soweit MapRed angeschaut. Es folgt eine **kurze** Einführung in MapReduce, welches der **Nachfolger** für die originale API ist.

MapRed war zeitweise markiert als **deprecated**, aber da es bereits zu viel Code für diese API gab, wurde es wieder freigegeben – Gruppenzwang!.

WordCount in „MapReduce“

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import
org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import
org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class WordCount {
...
```

Beachte: Andere Java Pakete!

WordCount in „MapReduce“

```
public static class TokenizingMapper
  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 {
    StringTokenizer itr =
      new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

Extend anstatt Implement

Kombinierte Klasse

WordCount in „MapReduce“

```
public static class IntSumReducer
    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);
  }
}
```

WordCount in „MapReduce“

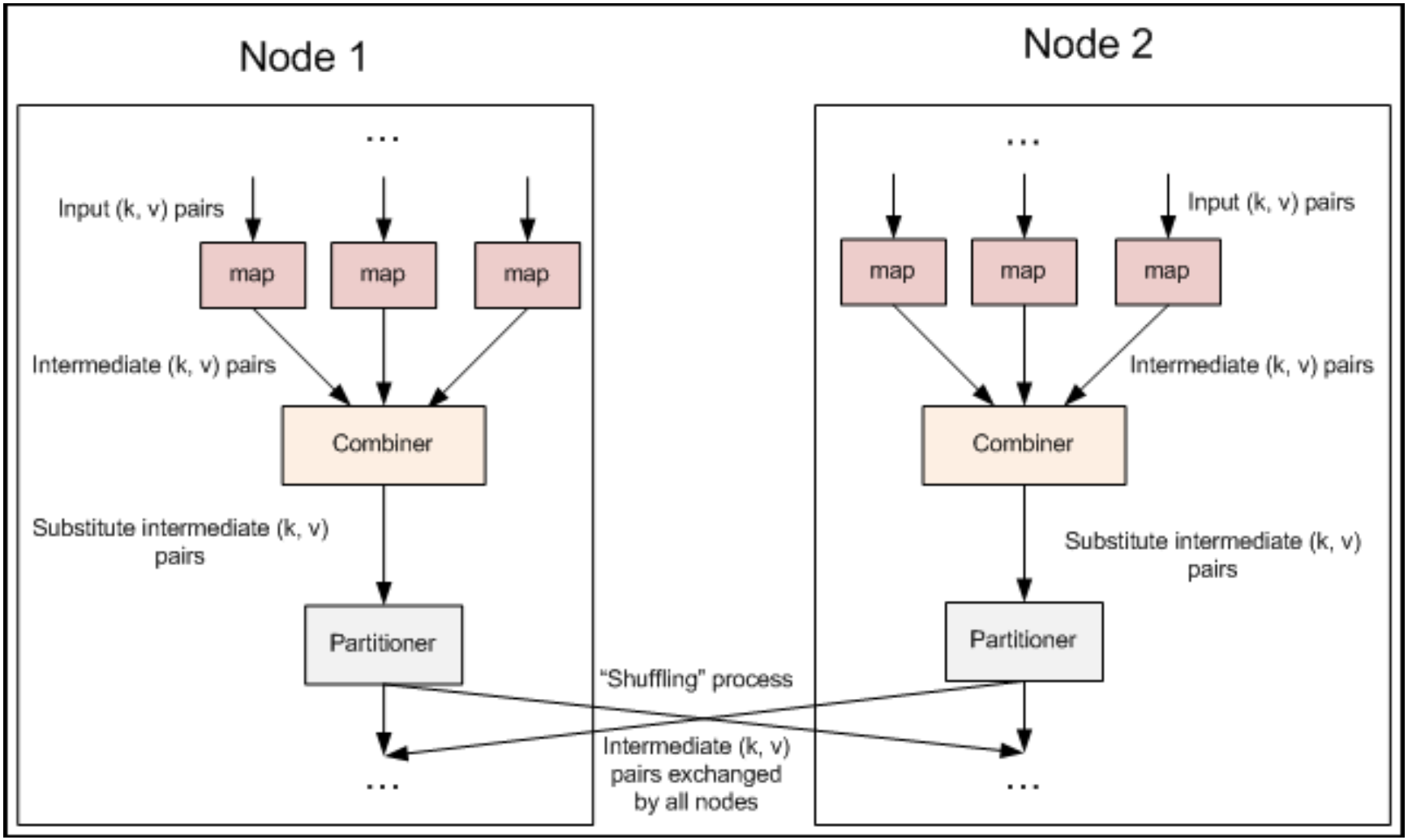
```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  String[] otherArgs = new GenericOptionsParser(conf,
    args).getRemainingArgs();
  if (otherArgs.length != 2) {
    System.err.println("Usage: wordcount <in> <out>");
    System.exit(2);
  }
  Job job = new Job(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizingMapper.class);
  job.setCombinerClass(SummingReducer.class);
  job.setReducerClass(SummingReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
  FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Combiner

Die `Combiner` Klasse erlaubt es einen „**Mini**“ Reduce Schritt am **Ende** des Map Prozesses auszuführen. Das MapReduce Framework gibt **alle** Daten **eines** Map Tasks an **eine** Instanz des Combiners – **bevor** die Daten an den Reduce Task im Shuffle **kopiert** werden.

Das folgende Diagramm zeigt die bis jetzt nicht angesprochene Combiner Funktion.

Combiner


Node 1
Node 2
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Combiner
Substitute intermediate (k, v) pairs
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Combiner
Substitute intermediate (k, v) pairs
Partitioner

Combiner

Combiner sind **ähnlich** dem Reducer, und implementieren dessen **Signatur**. Manchmal kann man auch **dieselbe** Klasse für beide nehmen.

Fragen:

- Geht das für WordCount?
- Wann geht es nicht?

Combiner

Beispiel: „Suche“ (Search) in Textdateien

Vorgaben:

- Eine Anzahl an Dateien welche Textzeilen enthalten
- Ein Suchmuster welches es zu finden gilt
- Der Mapper Schlüssel ist der Dateiname und Zeilennummer
- Der Mapper Wert ist die Zeile selbst
- Das Suchmuster wird als Parameter übergeben

Combiner

Der **Algorithmus**:

- **Mapper**
  - Gegeben ein Paar (Dateiname, Text) und „Suchmuster“, wenn „Text“ auf das Muster passt, dann gebe (Dateiname, null) aus
- **Reducer**
  - Identitätsfunktion, also Schlüssel/Wert Paare einfach durchreichen

Combiner

Eine **Optimierung**:

Wenn eine Datei einen Treffer enthält, dann brauchen wir sie nur **einmal** zu markieren.

Dazu benutzen wir einen **Combiner**, der schon auf der Map Seite alle Mehrfachnennungen in eine einzige **zusammenfasst**.

Configure() und Close()

Die Map und Reduce Klassen (oder Interfaces) haben **weitere** Hilfsfunktionen eingebaut, welche manchmal sehr **hilfreich** sind. Zu nennen sind die `configure()` (oder `setup()`) und `close()` (oder `cleanup()`) Methoden. Diese werden am **Anfang** und am **Ende** einer Taskausführung aufgerufen und erlauben es, zum Beispiel, Resourcen zu **laden**, oder eine Datenbankverbindung **herzustellen**.

Configure() und Close()

```
static class MultipleOutputsReducer
extends Reducer<Text, Text, NullWritable, Text> {

  private MultipleOutputs<NullWritable, Text>
    multipleOutputs;

  @Override
  protected void setup(Context context)
  throws IOException, InterruptedException {
    multipleOutputs = new MultipleOutputs
      <NullWritable, Text>(context);
  }
```

Configure() und Close()

```
  ...
  @Override
  protected void reduce(Text key,
    Iterable<Text> values, Context context)
  throws IOException, InterruptedException {
    for (Text value : values) {
      multipleOutputs.write(NullWritable.get(),
        value, key.toString());
    }
  }

  @Override
  protected void cleanup(Context context)
  throws IOException, InterruptedException {
    multipleOutputs.close();
  }
}
```

Distributed Cache

Normalerweise sind die **Daten** für einen MapReduce Job als **große** Dateien in HDFS abgelegt. Diese werden beim Speichern in **Blöcke** von 64 oder 128MB **zerlegt** und **verteilt**. **Pro** Block wird **ein** Map Task angestoßen, der die **enthaltenen** Daten liest und verarbeitet.

Manchmal braucht aber eben diese Verarbeitung **kleinere** Hilfsdaten, ebenfalls als Dateien abgelegt. Diese werden aber von **allen** Map oder Reduce Tasks **gelesen**.

Distributed Cache

Ein Beispiel könnte ein **Wörterbuch** als Datei sein, welches **jeder** Mapper benötigt, um die Textdaten **richtig** verarbeiten zu können.

Diese Hilfsdaten sind nur so **groß**, dass sie **ohne** Probleme in den Speicher des Tasks **passen**.
Wenn sie eine **bestimmte** Größe übersteigen, dann sollte vielleicht eine **andere** Lösung gesucht werden.

Distributed Cache

Für den Zweck der Verteilung von Hilfsdaten hat Hadoop einen Mechanismus der „**Distributed Cache**“ heißt, also „verteilter Zwischenspeicher“.

Diese Cache kann eben jene kleineren Dateien enthalten, zum Beispiel Wörterbücher oder auch Bibliotheken (man denke an Bildverarbeitung, oder Machine Learning).

Distributed Cache

Das **folgende** Beispiel zeigt wie eine **lokale** Datei nach HDFS **kopiert** und dann dem Distributed Cache **bekanntgemacht** wird.

Wenn der Task **ausgeführt** wird, dann werden die **angefragten** Dateien in das **lokale** Verzeichnis des Tasks **kopiert** und später automatisch wieder entfernt.

Der Tasks kann die Dateien dann in der `configure()` Methode **direkt** ansprechen.

Distributed Cache - Schreiben

```
public static final String LOCAL_STOPWORD_LIST =
  "file://home/johndoe/stop_words.txt";
public static final String HDFS_STOPWORD_LIST =
  "/data/stop_words.txt";

void cacheStopWordList(JobConf conf)
throws IOException {
  FileSystem fs = FileSystem.get(conf);
  Path hdfsPath = new Path(HDFS_STOPWORD_LIST);
  // upload (create or replace) the file to hdfs
  fs.copyFromLocalFile(false, true,
    new Path(LOCAL_STOPWORD_LIST), hdfsPath);
  DistributedCache.addCacheFile(hdfsPath.toUri(),
    conf);
}
```

Distributed Cache - Lesen

```
void configure(JobConf conf) {
  try {
    String stopwordCacheName =
      new Path(HDFS_STOPWORD_LIST).getName();
    Path[] cacheFiles =
      DistributedCache.getLocalCacheFiles(conf);
    if (null != cacheFiles && cacheFiles.length > 0) {
      for (Path cachePath : cacheFiles) {
        if (cachePath.getName().equals(stopwordCacheName)) {
          loadStopWords(cachePath);
          break;
        }
      }
    }
  } catch (IOException ioe) {
    System.err.println("Error reading from distributed cache");
    System.err.println(ioe.toString());
  }
}
```

Distributed Cache

Die `DistributedCache` Klasse hat **weitere** Funktionen, zum Beispiel die Methode `addArchiveToClassPath()`, welche eine JAR Datei aus dem verteilten Speicher **direkt** in den **Klassenpfad** des Tasks **hinzufügt** und sie damit dem Task zur Verfügung steht.

Hier wiederum der Hinweis auf die Online Dokumente der Java API. Dort stehen alle verfügbaren Optionen und Methoden.

Partitioner

Der Partitioner ist dafür zuständig, dass ein **bestimmter** Schlüssel genau bei einem **bestimmten** Reducer landet, **egal** in welchem Map Task dieser erzeugt wurde. Normalerweise ist dies eine einfache **Hash** Funktion.

Die Signatur selbst ist **einfach** gehalten:

```
public interface Partitioner<K, V>
extends JobConfigurable {
  int getPartition(K key, V value,
    int numPartitions);
}
```

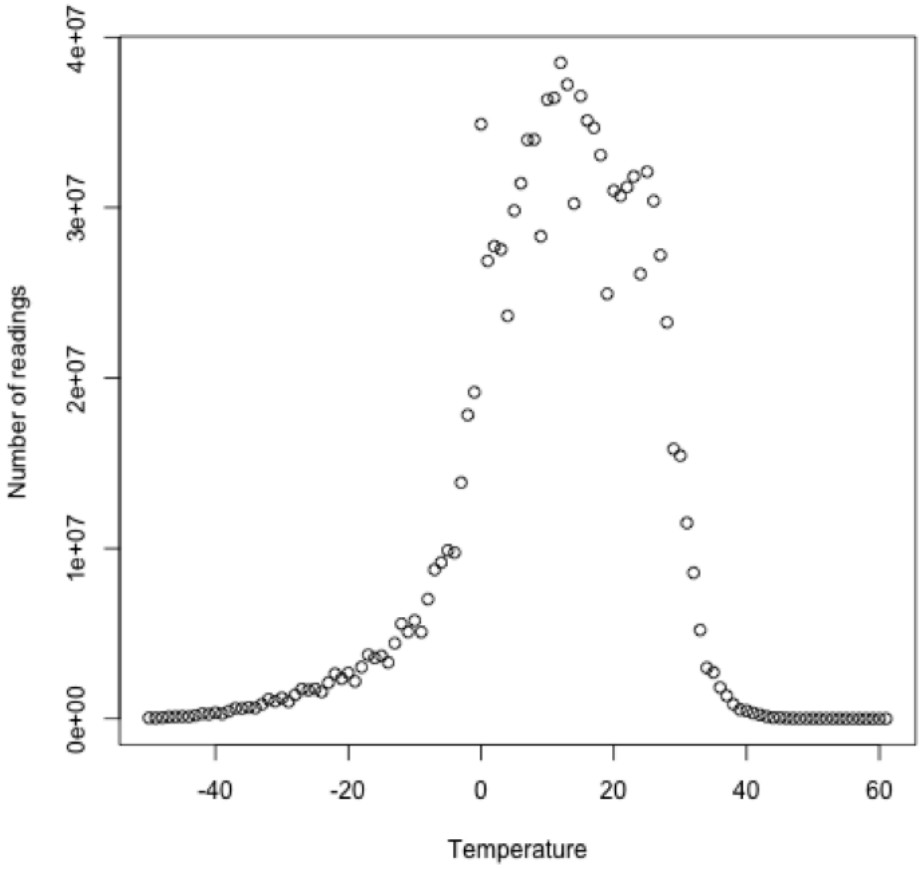
Partitioner

Es gibt **einige** Gründe, warum man einen **speziellen** Partitioner braucht. Zum Beispiel gibt es eine `TotalOrderPartitioner` Klasse, welche es erlaubt die Datenpaare auf **geordnete** Schlüsselpartitionen zu **verteilen**.

Ein weiteres Beispiel sind die sogenannten **Secondary Sorts**, welche eine zusammen-gesetzten Schlüssel benutzen, diesen dann aber nach **Teilwerte** partitionieren.

Partitioner

Ein weiterer, sehr **häufiger** Grund ist ein Verteilung der Daten, die **nicht** gleichmäßig ist. Dann braucht der ganze Job so lange wie der **langsamste** Task, also der mit den **meisten** Daten.



MapReduce JOINs

Ein **Beispiel** für eine Fall, der einen speziellen Partitioner benötigt, sind die **JOIN** Operationen aus SQL aber **abgebildet** in MapReduce. Hier muss aus **Optimierungsgründen** der sogenannte **Secondary Sort** ausgeführt werden.

Zuerst aber eine Übersicht der möglichen JOIN Implementierungen.

MapReduce JOINs

Es gibt zwei Arten der MapReduce basierten JOINs, denn **Map-Side** JOIN und den **Reduce-Side** JOIN.

Wie der Name schon verrät, sind dies die **Orte** an denen der JOIN durchgeführt wird, also entweder im **Mapper** oder **Reducer** Code.

Der Map-Side JOIN ist **effektiver**, da **weniger** Daten bewegt werden, dafür aber **spezieller** und kann **nicht** immer angewandt werden.

MapReduce JOINs

Für den Map-Side JOIN gibt es wiederum **zwei** Varianten, den **Repartition** JOIN und den **Broadcast** JOIN, oder auch In-Memory JOIN genannt.

Der **Repartition** JOIN benötigt **synchronisierte** Quelldaten, welche schon **sortiert** und **partitioniert** sind.

Der Broadcast JOIN lädt ein **kleinere** Quelle in den **Speicher** und macht einen **HashMap** Zugriff.

MapReduce JOINs

Für den **Reduce-Side** JOIN gibt es nur **eine** Variante, die aber **flexibler** ist und **keine** vorbereiteten Daten benötigt. Es müssen **mehrere** Quellen gelesen werden und dann über eine spezielle Verarbeitung die Datensätze mit dem **gleichen** Primärschlüssel zusammen-gebracht werden. Dafür wird ein **besonderer** `Partitioner`, **und sogar** `Comparator`, implementiert.

MapReduce JOINs

Der **Trick** hier ist **zwei** Quellen einzulesen, wobei diese mit einem „Tag“ so versehen werden, dass der Hauptdatensatz an den **Anfang** sortiert wird. Wenn der Reducer nun die Paare einliest bekommt es **erst** den Hauptdatensatz, speichert diesen und **verbindet** ihn mit jedem **weiteren** Datensatz der JOIN Tabelle.

Der `Comparator` wird für das **finale** Gruppieren der Paare pro Hauptschlüssel benötigt.

MapReduce JOINs

Der Partitioner Code:

```
public int getPartition(TextPair key ,
Text value, int numPartitions) {
  return (key.getFirst().hashCode() &
    Integer.MAX_VALUE) % numPartitions ;
}
```

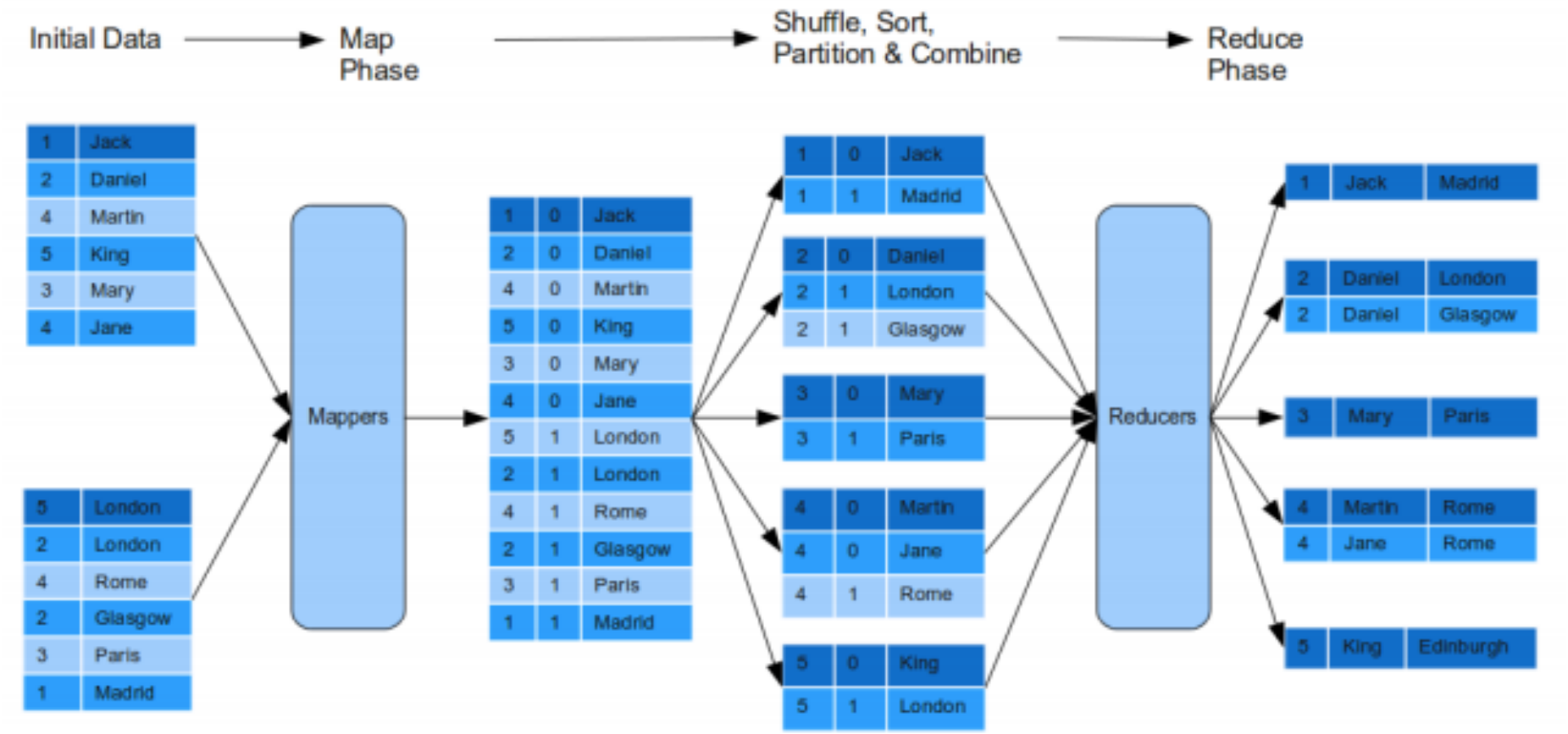
MapReduce JOINs

Der Driver Code:

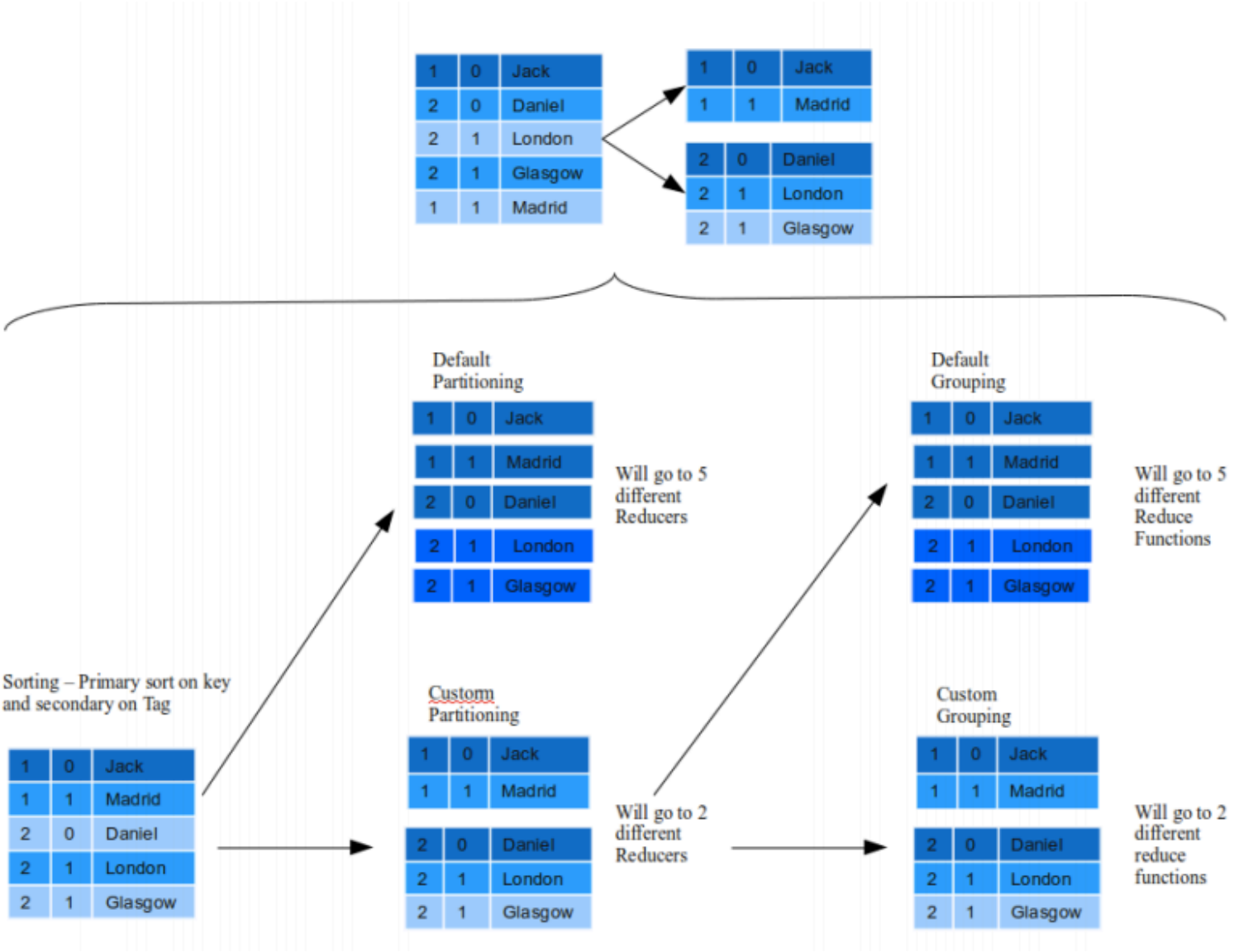
```
MultipleInputs.addInputPath(job, inputPath1,
  TextInputFormat.class,
  JoinPrimaryMapper.class);
MultipleInputs.addInputPath(job, inputPath2,
  TextInputFormat.class, JoinSecondaryMapper.class);
FileOutputFormat.setOutputPath(job, outputPath);
job.setPartitionerClass(KeyPartitioner.class);
job.setGroupingComparatorClass(
  TextPair.FirstComparator.class);
job.setMapOutputKeyClass(TextPair.class);
job.setReducerClass(JoinReducer.class);
job.setOutputKeyClass(Text.class);
```

MapReduce JOINs

Beispiel:



MapReduce JOINs



MapReduce Web Interface

Die MapReduce Prozesse haben, wie HDFS auch, eine **eingebaute** Weboberfläche, welche die **Details** des Systems darstellt. Unter anderem kann man dort die **einzelnen** Jobs genauer betrachten. Man kann auch sehen welcher **Scheduler** aktiv ist und dessen aktueller Status.

Im Rahmen der Übung werden wir uns diese Oberfläche genauer anschauen.

Scheduler

Hadoop hat **verschiedene** Job Scheduler schon eingebaut, diese sind:

- FIFO Scheduler
  - Dies ist der originale Scheduler, aber auch der einfachste
- Capacity Scheduler
  - Bietet feste Queues für Benutzergruppen
- Fair Scheduler
  - Echte Mandantenfähigkeit

Scheduler

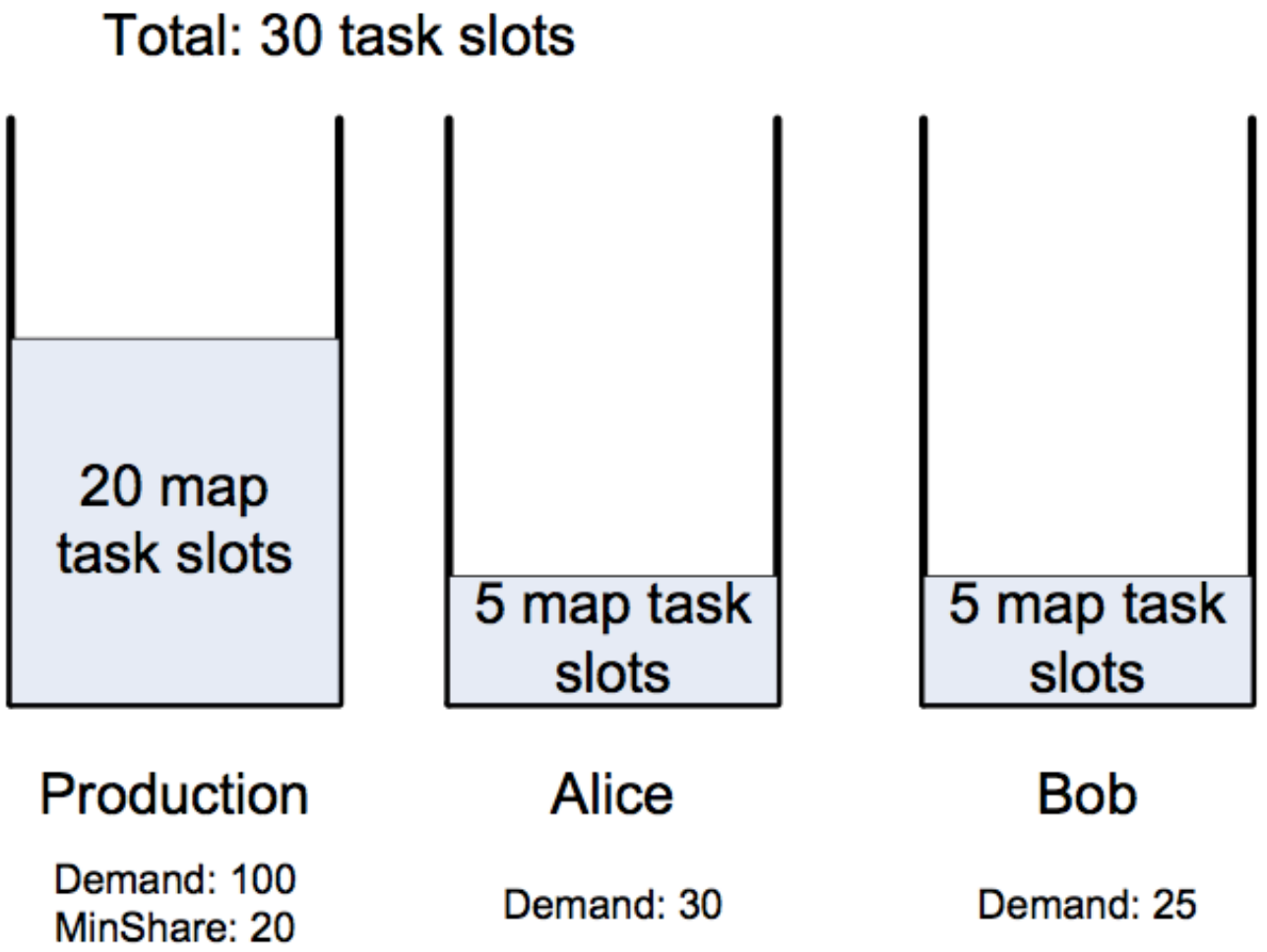
In der Praxis wird fast **immer** der Fair Scheduler eingesetzt, denn er ist der **einzige** der brauchbare Mandantenfähigkeiten hat.

Der Fair Scheduler kann **Pools** für Benutzer und/oder Gruppen von Benutzer **anlegen**, und diese **dynamisch** verwalten.

Generell: Scheduling ist sehr interessantes, aber auch schwieriges Thema im Bereich der verteilten Verarbeitung.

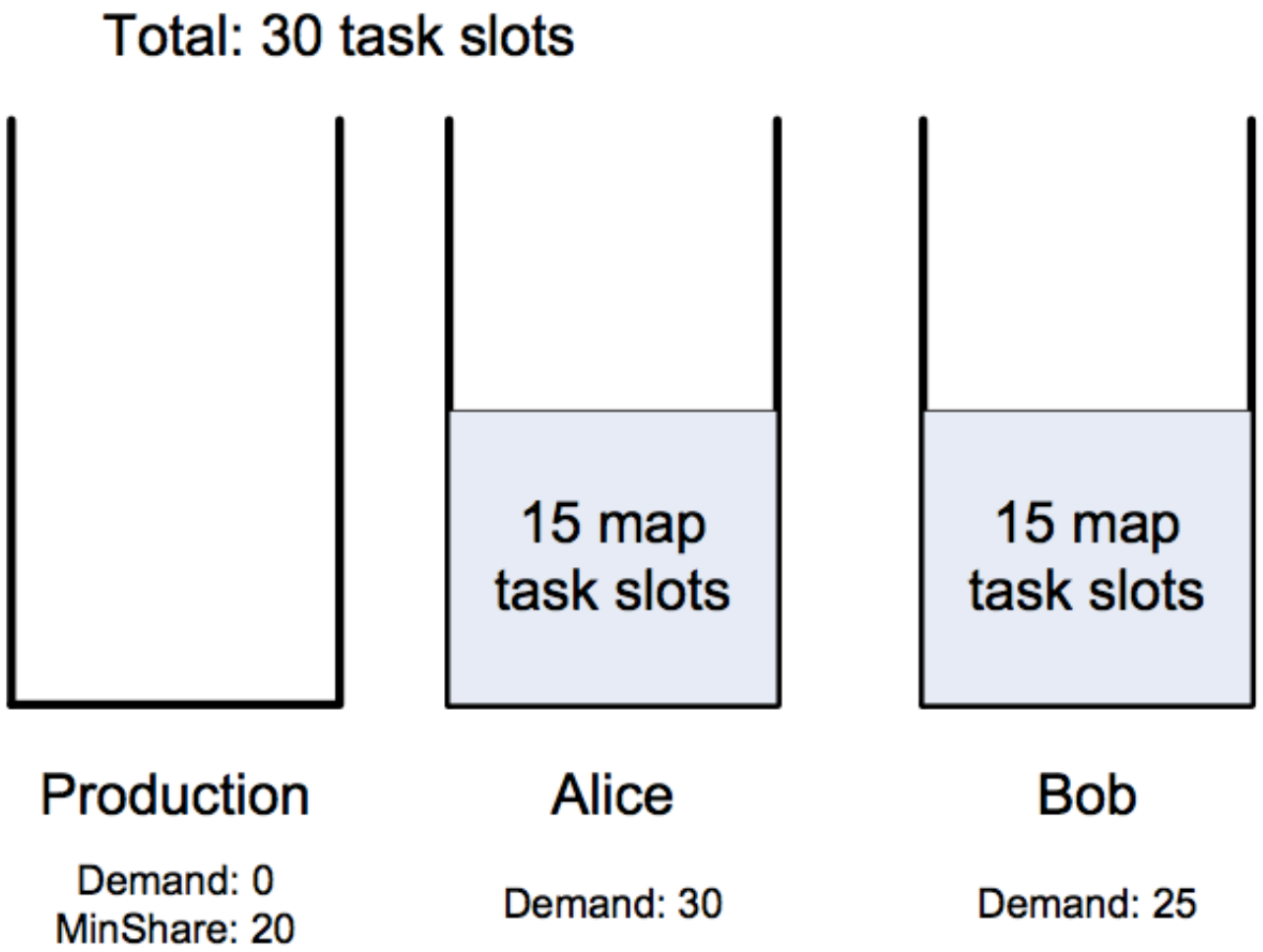
Fair Scheduler

Beispiel:



Fair Scheduler

Beispiel:



Fair Scheduler

Der Fair Scheduler hat viele **weitere** Interessante Eigenschaften:

- Dynamische Pools
  - Regeln **definieren** wie viele Pools es gibt und wie sich diese **verhalten**
  - Pro Pool kann man Gewichte und Queue Verhalten definieren (FIFO, Fair usw.)
- „Preemption“ Regeln
  - Definieren aggressivere Verhaltensweisen

Custom Writable‘s

Alle Schlüssel und Werte in Hadoop werden als **abgeleitete** Klasse von `Writable` abgebildet. Diese **definiert** die Signatur für das Serialisieren der **rohen** Daten während der Job Ausführung, zwischen den **einzelnen** Phasen.

Entwickler können **eigene** Klassen implementieren und im Cluster als JAR Datei **installieren**. Damit können dann **spezifische** Anwendungsfälle abgedeckt werden.

Custom Input/OutputFormat‘s

Wie auch die vielen anderen, bereits besprochenen Komponenten kann man die Ein- und Ausgabeformate über **eigene** Klassen erweitern. Dies ist zuweilen **nötig**, um Rohdaten einzulesen, welche **nicht** von den von Hadoop bereitgestellten Klassen gelesen werden können.

Job Tuning

Wenn die Last auf dem Hadoop Cluster größer und größer wird, dann stellt sich oft die Frage, ob die bestehenden MapReduce Jobs effizient genug arbeiten. Dazu helfen die Job Statistiken wie Zähler und andere Informationen.

Es folgen einige typische Fragen, welche sich ein MapReduce Entwickler/Administrator stellen sollte.

Job Tuning

- Anzahl der Mapper
  - Viele Mapper welche nur kurz laufen? Ggf. kleine Dateien zu großen zusammenfassen.
- Anzahl der Reducer
  - Sollte kleiner als verfügbare Slots sein, damit der Job in einer „Welle“ fertig wird.
- Combiners
  - Kann ein Combiner eingesetzt werden?
- Komprimierung der Zwischendaten
  - Sollte fast immer eingeschaltet sein

Job Tuning

- Größe der Spill Daten
  - Einsehbar in Job Counter „Spill Records“, welche für Map und Reduce kombiniert gelten
  - Sollte nicht mehr als einmal Zwischenspeichern
  - Tuning möglich wenn Größe der Map Ausgabedaten bekannt ist (`io.sort.*`)

Vertiefung in MapReduce

Dies **schließt** die Vertiefung in MapReduce Konzepte ab. Es gibt aber noch viele **weitere** Einstellungen und Merkmale, die hier **nicht** alle besprochen werden können. Es empfiehlt sich **weiterführende** Literatur zu studieren.

Im folgenden schauen wir uns nun **weitere** MapReduce Beispiele an, welche **interessante** Lösungen implementieren.

Einheit 3

- Rückblick auf Einheit 2
- Debugging von MapReduce Code
- Vertiefung in MapReduce Konzepte
- **Erweiterte MapReduce Beispiele**

Machine Learning

Ein sehr **heißes** Thema in der Big Data Welt ist das sogenannte **Machine Learning**, also das Anwenden **mathematischer** Funktion auf große Datenmengen mit dem Ziel **Zahlenmodelle** zu bilden. Diese werden dann **eingesetzt**, um Ad-hoc Entscheidungen eines Systems zur Laufzeit zu **beeinflussen**.

Entweder werden die Algorithmen **direkt** in MapReduce **implementiert** oder über **verfügbare** Bibliotheken bereitgestellt.

TF-IDF

Ein Beispiel für eine „Data Mining“ Funktion ist das Bilden eines TF-IDF Indexes. Erläuterung:

- Term Frequency – Inverse Document Frequency
  - Relevant in der Verarbeitung von Textdaten
  - Sehr verbreitet in der Analyse von Web Inhalten

TF-IDF

Der Algorithmus ist wie folgt definiert:

$$tf_i = \frac{n_i}{\sum_k n_k}$$

$$idf_i = log \frac{|D|}{|\{d: t_i \in d\}|}$$

$$tfidf = tf \cdot idf$$

- $|D|$ ist die Anzahl aller Dokumente im Datenbestand
- $|\{d: t_i \in d\}|$ ist die Anzahl der Dokumente die den Term t enthalten

TF-IDF

Folgende Informationen sind nötig:

- Anzahl des Auftretens des Terms X in einem gegebenen Dokument
- Anzahl des Terme für jedes Dokument
- Anzahl der Dokumente in welcher X enthalten ist
- Gesamte Anzahl von Dokumenten

TF-IDF

Job 1: Wortfrequenz in Dokument

- Mapper
  - Eingabe: (dokname, inhalt)
  - Ausgabe: ((wort, dokname), 1)
- Reducer
  - Summiert Anzahl des Wortes im Dokument
  - Ausgabe: ((wort, dokname), n)
- Combiner ist der gleiche wie der Reducer

TF-IDF

Job 2: Wortanzahl pro Dokument

- Mapper
  - Eingabe: ((wort, dokname), n)
  - Ausgabe: (dokname, (wort, n))
- Reducer
  - Summiert Frequenz individueller Terme im gleichen Dokument
  - Gibt Originaldaten wieder aus
  - Ausgabe: ((wort, dokname), (n, N))

TF-IDF

Job 3: Wortfrequenz im Datenbestand

- Mapper
  - Eingabe: ((wort, dokname), (n, N))
  - Ausgabe: (wort, (dokname, n, N, 1))
- Reducer
  - Summiert Anzahl des Wortes im Datenbestand
  - Ausgabe: ((wort, dokname), (n, N, m))

TF-IDF

Job 4: Berechne TF-IDF

- Mapper
  - Eingabe: ((wort, dokname), (n, N, m))
  - Setzt voraus das D bekannt ist (ansonsten über einfachen MapReduce Job herausfinden)
  - Ausgabe: ((wort, dokname), TF*IDF)
- Reducer
  - Die reine Identitätsfunktion

TF-IDF

Optimierungen für große Datenmengen:

- Zwischenspeichern der (dok, n, N) Werte während der Summierung der 1en in m passt möglicherweise nicht in den Speicher
  - In wie vielen Dokumenten taucht das Wort „der“ auf?
- Mögliche Lösungen
  - Hochfrequente Wörter ignorieren
  - Zwischendaten in Datei abspeichern
  - Noch ein MapReduce Durchgang

TF-IDF

Zusammenfassung:

- Mehrere kleinere Jobs fügen sich zum gesamten Algorithmus zusammen
- Viele Teile des Codes kann wiederverwendet werden
  - Es gibt schon einige mitgelieferte Klassen für Aggregation und Identität
- Job 3 und 4 könnten zusammengefasst werden in einem Reducer Durchlauf

Amazon Web Services

Nur kurz soll hier Amazon als Beispiel für Cloud basierte Cluster Installationen genannt werden.

AWS sind die ganzen Dienste welche angeboten werden, dazu gehört

- Elastic Compute Cloud (EC2)
  - Virtuelle Server in Stufen verfügbar
- Simple Storage Service (S3)
  - Hadoop hat Treiber für S3 eingebaut
- Elastic MapReduce (EMR)

Einheit 3

An dieser Stelle endet die dritte Einheit mit einer Weiterführung in MapReduce. In der nächsten Einheit werden wir uns die verschiedenen Datei- und Serialisierungsformate, sowie Abfrage-schnittstellen anschauen.

Bis bald!

Übung 3

Ziele:

- Log Datei aus Übung 2 fertigstellen
- MapReduce WordCount #2
  - In MapReduce Stil schreiben und ausführen
  - Tokenizer verbessern (Normalisierung)
  - Unit Test implementieren
  - Job Counter einbauen und auswerten
- TF-IDF Implementieren
- Modell bilden?

Übung 3

Code:

https://github.com/larsgeorge/fh-muenster-bde-lesson-3

Quellen

- JOINs
  - http://www.inf.ed.ac.uk/publications/thesis/online/IM100859.pdf
- MapReduce Konzepte
  - http://developer.yahoo.com/hadoop/tutorial/module4.html
  - http://developer.yahoo.com/hadoop/tutorial/module5.html
  - „Hadoop - The Definitive Guide“ von Tom White http://shop.oreilly.com/product/0636920021773.do