

Masterstudiengang Wirtschaftsinformatik

Big Data Engineering

FH Münster
Master Wirtschaftsinformatik
Wintersemester 2013
Dozent: Lars George



Einheit 2

- Rückblick auf Einheit 1
- Einführung in MapReduce
- MapReduce Beispiele
- YARN Übersicht
- **Hauptziel:** MapReduce Kenntnisse erlangen
- **Übung 2:**
 - MapReduce Program schreiben und ausführen



Einheit 2

- **Rückblick auf Einheit 1**
- Einführung in MapReduce
- MapReduce Beispiele
- YARN Übersicht



Rückblick auf Einheit 1

- Fragen?
- HDFS Erlebnisse?
- Was passiert wenn man dieselbe Datei zweimal mit `-put` nach HDFS kopiert (mit demselben HDFS Pfad)?



Speicherung der Dateien

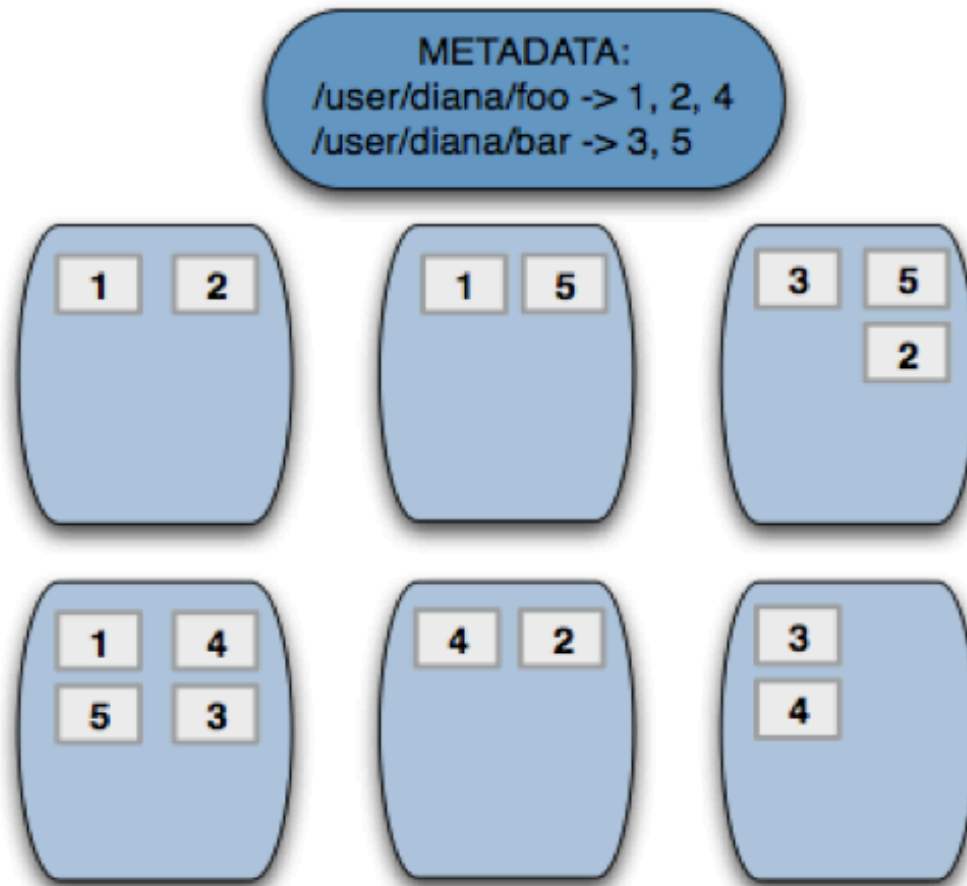
Die Dateien sind in Blöcke **aufgeteilt**.

Die Daten werden verteilt auf viele Rechner während der Speicherung. Verschiedene Blöcke derselben Datei werden auf **verschiedenen** Rechnern abgelegt. Dies **hilft** auch bei dem verteilten Bearbeiten der Daten mit MapReduce (in Einheit 2 besprochen) um dessen Effizienz zu **erhöhen**.



Speicherung der Dateien

NameNode: Speichert nur Metadaten

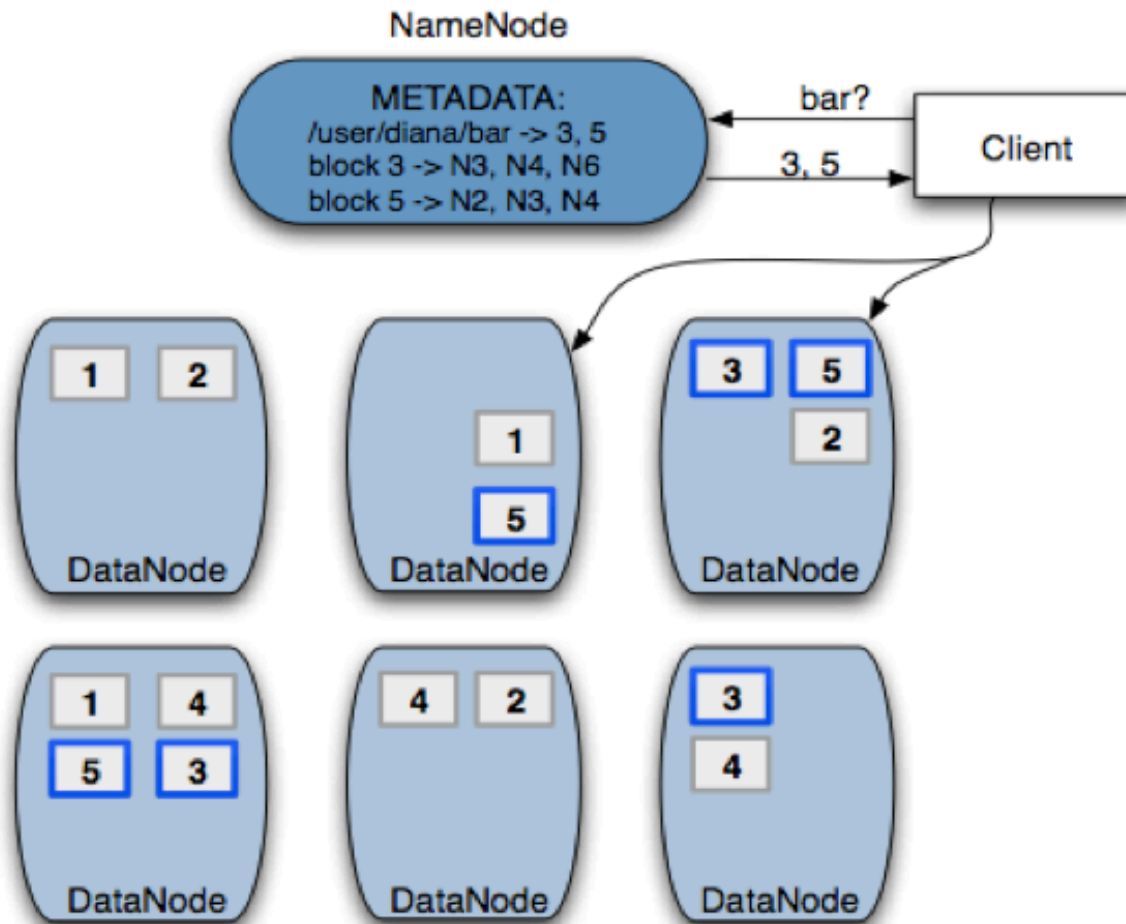


DataNode: Speichert die Daten

- **NameNode** verwaltet die Metadaten für die Datendateien
- **DataNodes** speichern die eigentlichen Daten
 - Jeder Block wird dreimal repliziert abgelegt im Cluster



Lesen der Dateien



Lesezugriff durch Anwendung:

- Die Anwendung kommuniziert mit dem NameNode, um die Datenblöcke der gefragten Datei zu bestimmen und wo diese abgelegt sind.
- Anschließend kommuniziert die Anwendung direkt mit dem DataNodes um die Daten zu lesen.



Zugriff auf Dateien

- Hadoop Kommandozeile:

```
$ hadoop fs -copyFromLocal <local_dir> <hdfs_dir>
```

```
$ hadoop fs -copyToLocal <hdfs_dir> <local_dir>
```

- Hadoop Projekte:

- Flume – Sammelt Daten aus Logquellen (z. B. Webserver, syslog, STDOUT)

- Sqoop – Transportiert Daten aus und/oder nach HDFS aus relationalen Datenbanken

- Business Intelligence Werkzeuge



Zugriff auf Dateien

Wie bereits besprochen kann man HDFS auch über die nativen Java **APIs** ansprechen. Im folgenden ein **Beispiel** welches eine Datei **anlegt**, eine Nachricht **hineinschreibt**, diese wiederum **ausliest** und auf der Konsole **ausgibt**.

Interessant ist das **Prüfen**, ob die Datei bereits besteht und gegebenenfalls **löscht**.

Warum ist
das wichtig?



Programmatischer Zugriff

```
public class HDFSHelloWorld {  
    public static final String theFilename = "hello.txt";  
    public static final String message = "Hello, world!\n";  
  
    public static void main (String [] args)  
        throws IOException {  
        Configuration conf = new Configuration();  
        FileSystem fs = FileSystem.get(conf);  
        Path filenamePath = new Path(theFilename);  
        try {  
            if (fs.exists(filenamePath)) {  
                fs.delete(filenamePath); // remove the file first  
            }  
            FSDataOutputStream out = fs.create(filenamePath);  
            out.writeUTF(message);  
            out.close();  
        }  
    }  
}
```



Programmatischer Zugriff

```
...  
FSDataInputStream in = fs.open(filenamePath);  
String messageIn = in.readUTF();  
System.out.print(messageIn);  
in.close();  
} catch (IOException ioe) {  
    System.err.println("IOException during operation: "  
        + ioe.toString());  
    System.exit(1);  
}  
}  
}
```



Weitere HDFS Details

Im Betrieb sind noch viele **weitere** Funktionen nötig, welche den Cluster **balancieren**, Knoten **hinzu-** und **herausnehmen** können, das Dateisystem auf **Konsistenz** prüfen, die **Zugriffsrechte** bestimmt und setzt, den **Replikationsfaktor** pro Datei bestimmt und vieles mehr.

Bitte schauen Sie sich die Dokumentation an, zum Beispiel unter: <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html>



Einheit 2

- Rückblick auf Einheit 1
- **Einführung in MapReduce**
- MapReduce Beispiele
- YARN Übersicht



Einführung in MapReduce

MapReduce ist ein von Google eingeführtes **Programmiermodel**, welches **nebenläufige** Berechnungen über **große** Datenmengen auf Computerclustern ermöglicht.

Die Verarbeitung wird in **zwei** Phasen aufgeteilt, wodurch sich die Berechnungen **parallelisieren** und über mehrere Rechner **verteilen** lassen.

Die Inspiration für MapReduce kommt aus der **funktionalen** Programmierung.



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



Google MapReduce



Research Publications

[Google Research Home](#)

MapReduce: Simplified Data Processing on Large Clusters

[Jeffrey Dean](#) and [Sanjay Ghemawat](#)

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

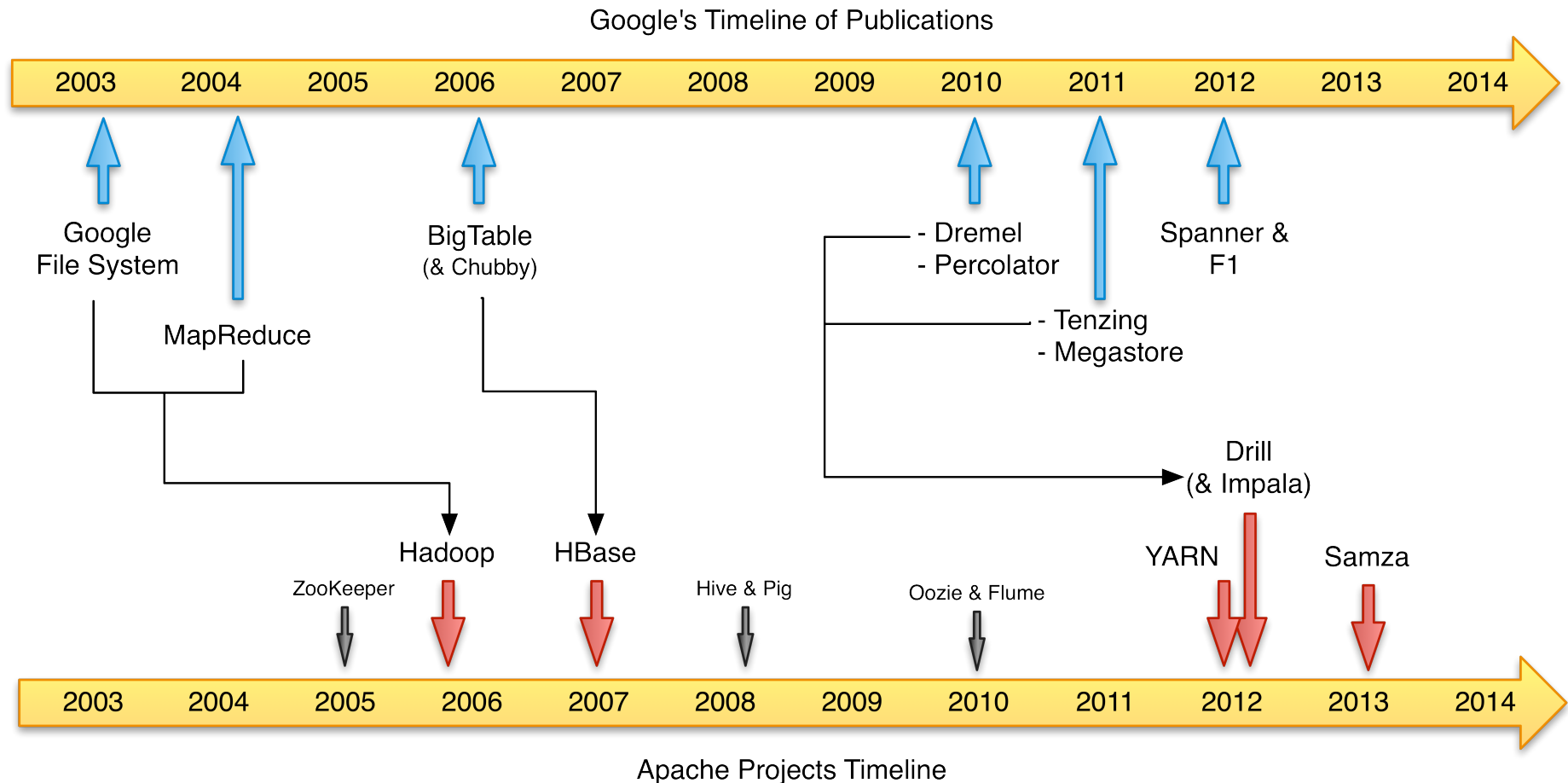
Appeared in:
OSDI'04: Sixth Symposium on Operating System Design and Implementation,
San Francisco, CA, December, 2004.

Download: [PDF Version](#)

Slides: [HTML Slides](#)



Apache Hadoop Zeitleiste





MapReduce Konzepte

Für die **skalierbare** Verarbeitung von Daten auf Computercluster (hunderte bis tausende Rechner) müssen die Daten so **verteilt** werden, dass **keine** weitere Kommunikation erforderlich ist – sonst würde dies eine Einschränkung bedeuten.

Dazu sind alle **Datenelemente** in MapReduce **unveränderbar**, d. h. können **nicht** aktualisiert werden.



MapReduce Konzepte

Fortsetzung:

Wenn also ein Map Task eine Eingabe-Datenpaar (Key, Value) **verändert**, so wird **NICHT** die Ursprungsquelle der Daten verändert.

Kommunikation findet **NUR** über die Ausgabe von **neuen** Datenpaaren statt, welche dann durch die Plattform an die **nächste** Phase weitergeleitet wird.

Vergleiche auch: **Schema-on-Read**.



MapReduce Konzepte

Fortsetzung:

Konzeptionell **transformieren** MapReduce Programme **Listen** von Eingabedatenenelemente in Listen von Ausgabedatenenelementen.

Bei MapReduce passiert dies **zweimal** mit zwei verschiedenen Verarbeitungsschritten: **map** und **reduce**.

Diese Namen stammen aus Listenverarbeitungs-sprachen wie LISP, Scheme oder ML.



Mapping einer Liste

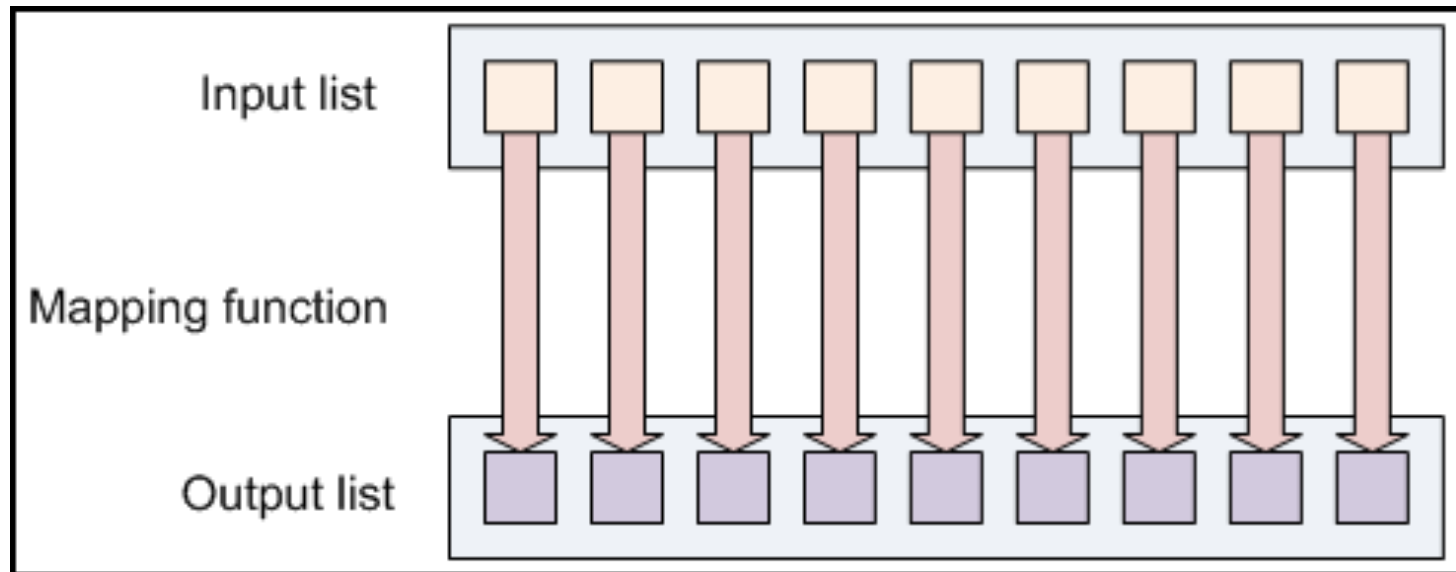
Die **erste** Phase eines MapReduce Programms wird **mapping** genannt. Dabei wird eine **Liste** von Datenelementen, ein Element **pro** Aufruf, an eine **Funktion**, welche als **Mapper** bezeichnet wird, übergeben.

Die Funktion erzeugt eine **neue** Liste welche eine **veränderte** und **unveränderte** Kopie der originalen Elemente enthält. Auch die Anzahl der Elemente **kann** verschieden sein.



Mapping einer Liste

Beispiel: Lese alle Zeichenketten ein und wende eine Funktion `toUpperCase(str)` an. Dabei werden neue Zeichenketten erzeugt und die bestehenden nicht verändert.





Reducing einer Liste

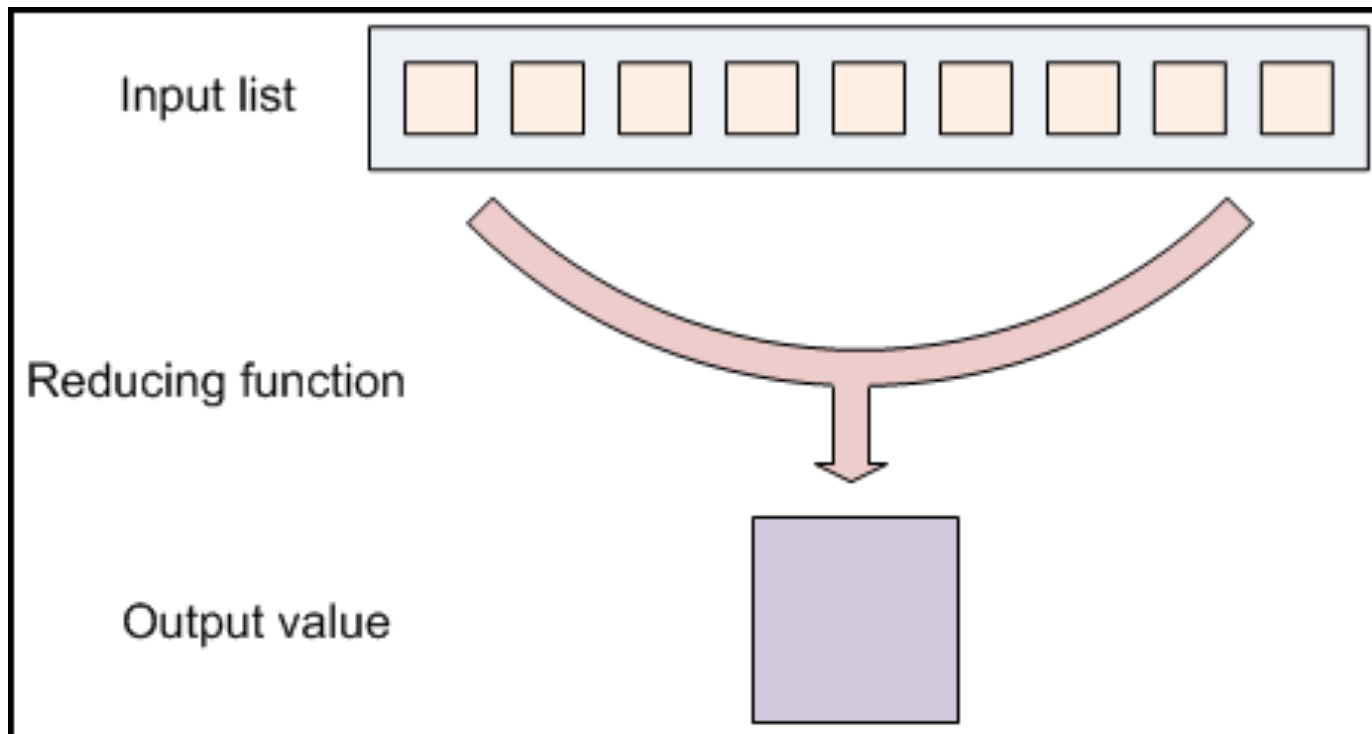
Bei der **zweiten** Phase werden die Datenelemente nun **aggregiert**. Dazu erhält die sogenannte **reducer** Funktion einen **Iterator** von Eingabewerten einer Eingabeliste. Diese werden dann **kombiniert** und als **einzelner** Wert ausgegeben.

Deshalb bietet sich Reducing an Summendaten zu bilden, welche **große** Mengen an Daten in wesentlich **kleinere** Summen derselben verwandelt.



Reducing einer Liste

Beispiel: Lese alle Bestellungen eines Kunden ein
und wende die Funktion
`sumOrderValues (order.value)` an.





Map + Reduce = MapReduce

MapReduce **verbindet** diese beiden Konzepte zur Verarbeitung von sehr großen Datenmengen. Dazu gibt es analog **zwei** Komponenten, wobei einer den **Mapper** und der andere den **Reducer** implementiert.

Es gibt aber einige Unterschiede, jedoch sind die grundlegenden Ideen genau die gleichen.



Schlüssel und Werte

Alle Daten werden in MapReduce als sogenannte Key-Value **Paare** repräsentiert. Dabei identifiziert der Schlüssel (**Key**) den zugeordneten Wert (**Value**). Es kann aber, durch die Natur der verteilten Bearbeitung, ein Schlüssel **mehrere** unabhängige Werte haben:

```
10.0.1.125 → GET index.html
10.0.3.15  → GET index.html
10.0.1.125 → GET contact.html
```



Schlüssel und Werte

Sowohl die Map als auch die Reduce Funktion erhalten Key-Value **Paare**, nicht nur die Werte. Beide geben auch wieder Key-Value Paare aus. Es gibt **Anwendungsfälle** bei denen Teile des Datenpaars **nicht** benötigt werden (siehe WordCount Beispiel später).



Schlüssel und Werte

Obwohl Datenpaare ausgetauscht werden müssen deren **Anzahl** und **Typen** nur bedingt festgelegt werden:

- Sowohl Map als auch Reduce Funktion geben für jedes Eingabedatenpaar **keins**, **eins**, oder **viele** Ausgabedatenpaare aus.
- **Typen** der Schlüssel und Werte werden nur zwischen **direkten** Ein- und Ausgabeverbindungen festgelegt.



Schlüssel und Werte

Typischerweise lesen und schreiben Map Tasks **viele** Datenpaare, während Reduce Tasks diese Datenpaare in **wenige** Ergebnispaare verdichten.

Jede Phase kann dabei **beliebig** viele Datenpaare konsumieren und eine **andere** Anzahl ausgeben.



Schlüssel Wertebereiche

Damit die Reduce Funktion **alle** Werte eines **bestimmten** Schlüssels bearbeiten kann werden diese über eine Partitionsfunktion an den selben Reducer übergeben.

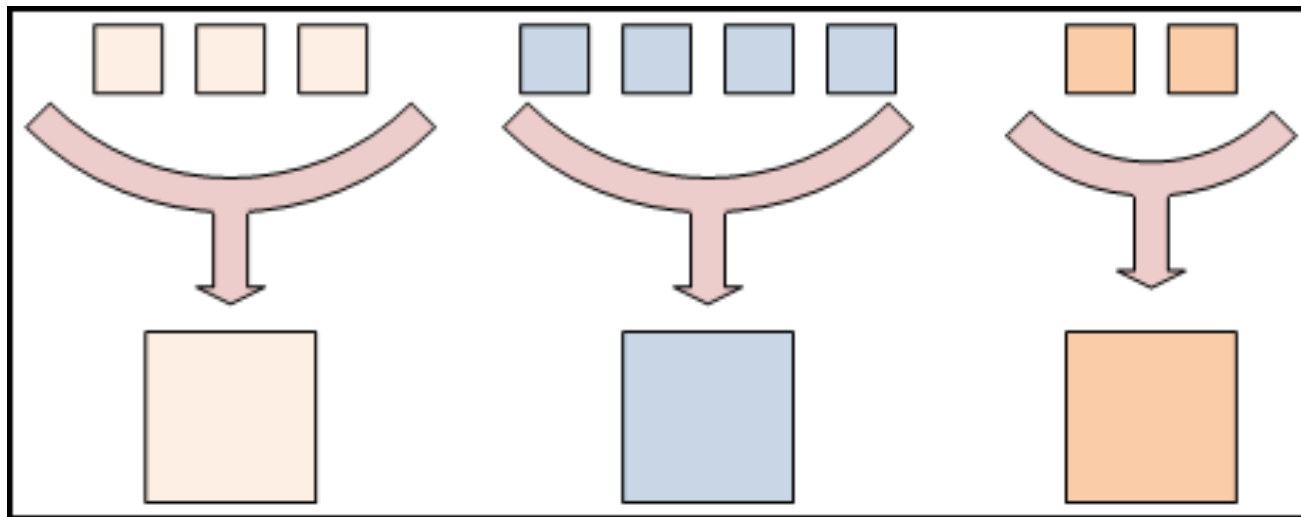
Dies geschieht **unabhängig** von allen anderen Reduce Operationen, welche andere Schlüssel und Werte enthalten.

Typischerweise werden die Schlüssel mit einer **Hashfunktion** in Bereiche eingeteilt.



Schlüssel Wertebereiche

Es werden so viele Wertebereiche angelegt wie Reducer Tasks ausgeführt werden.



Grafik: Verschiedene Farben repräsentieren verschiedene Schlüssel. Alle Werte mit dem gleichen Schlüssel werden an einen einzigen Reducer geleitet.



Beispiel: WordCount

Ein **einfaches** MapReduce Programm kann geschrieben werden, welches die **Anzahl** der **verschiedenen** Wörter in einer Menge von Textdateien zählt.

Eingabe:

foo.txt: Super, das ist die Foo Datei
bar.txt: Dies ist die Bar Datei



Beispiel: WordCount

Die Ausgabe sollte wie folgt aussehen:

Wort	Anzahl
super	1
das	1
ist	2
die	2
foo	1
datei	2
dies	1
bar	1



Beispiel: WordCount

Der Code kann so beschrieben werden:

```
mapper (filename, file-contents):  
    for each word in file-contents:  
        emit (word, 1)
```

```
reducer (word, values):  
    sum = 0  
    for each value in values:  
        sum = sum + value  
    emit (word, sum)
```



Beispiel: WordCount

Bemerkungen:

- **Mehrere** Map Tasks laufen **gleichzeitig**
- Jeder Map Task liest **eine** Datei
 - Genauer gesagt, einen HDFS Block
- Der Map Task gibt **pro** Wort **ein** Paar aus:
`(wort, 1)`
- Der Reduce Task nimmt diese Paare entgegen
als **Liste** von Werten für jedes eindeutige
Wort: `(wort, 1, 1, 1, 1, 1)`



Beispiel: WordCount

Bemerkungen (fort.):

- Der (die) Reduce Task(s) **summiert** die Einsen auf und gibt **pro** Wort dessen Kardinalität aus:
(`word`, 5)
- Das Ergebnis wird dann in **eine** Datei **pro** Reducer Task geschrieben

Im folgenden schauen wir uns das Ganze als **echtes** MapReduce Programm an.



Beispiel: WordCount

```
public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }
}
```



Beispiel: WordCount

```
public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter) throws IOException {

        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```



Beispiel: WordCount

Bemerkungen:

- Java hat **keine** `emit()` Funktion, dies wird durch den `OutputCollector` übernommen
- Der Hadoop Textparser übergibt den Text **Zeile** für Zeile und **nicht** komplett
- Der Text einer Zeile wird mit einer `StringTokenizer` Instanz **zerlegt**
 - Damit wird **NICHT** normalisiert, d.h. „das“, „Das“ und „das,“ sind alles verschiedene Wörter!



Beispiel: WordCount

Bemerkungen (fort.):

- Die internen Variablen können **mehrfach** genutzt werden, denn die `collect()` Methode **kopiert** die Werte
 - Dies **spart** Zeit aber auch die Generierung von möglicherweise tausender Klasseninstanzen

Obwohl WordCount sehr **einfach** erscheint ist dieser Ansatz doch **oft** in der Praxis zu finden!



Beispiel: WordCount

Warum ist WordCount interessant?

- **Große** Datenmengen haben ihre **Probleme**
 - Ein einzelner Rechner braucht zu **lange**
 - Verteilte Berechnung benötigt das **Kopieren** der Daten
 - Anzahl eindeutiger Wörter kann den verfügbaren Speicher **überschreiten**
- Grundlagen der Statistik sind oft **einfache** Aggregationsfunktionen
- Viele Aggregationsfunktionen sind **verteilt** von Natur aus, z. B. max, min, sum, count



Die Driver Methode

Es fehlt nur noch eine Komponente für die Ausführung eines MapReduce Programms, welche **Driver** genannt wird. Diese **initialisiert** den Berechnungsauftrag (Job), fordert Hadoop auf das Programm für eine Menge an Eingabedateien **auszuführen** und kontrolliert wohin die Ausgabedateien **geschrieben** werden.



Die Driver Methode

```
public void run(String inputPath, String outputPath)
throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    // the keys are words (strings)
    conf.setOutputKeyClass(Text.class);
    // the values are counts (ints)
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(MapClass.class);
    conf.setReducerClass(Reduce.class);

    FileInputFormat.addInputPath(conf, new Path(inputPath));
    FileOutputFormat.setOutputPath(conf, new Path(outputPath));

    JobClient.runJob(conf);
}
```



Die Driver Methode

Bemerkungen:

- **Initialisiert** den Auftrag/Job
 - Überschreibt **nur** Parameter welche von den Vorgabewerten **abweichen**
- Liest **alle** Dateien aus dem Verzeichnis spezifiziert mit `inputPath`
- Die Ausgabe wird in das Verzeichnis angegeben mit `outputPath` geschrieben



Die Driver Methode

Bemerkungen (fort.):

- Die `JobConf` Instanz beinhaltet alle Konfigurationsinformationen
- Die Map und Reduce Funktionen werden zugewiesen mit `setMapperClass()` und `setReducerClass()`
- Die Datentypen werden mit `setOutputKeyClass()` und `setOutputValueClass()` festgelegt
 - Legt implizit auch die Map Ausgabetypen fest



Die Driver Methode

Bemerkungen (fort.):

- Bei Bedarf können die weiteren Typen mit `setMapOutputKeyClass()` und `setMapOutputValueClass()` bestimmt werden
- Die Eingabetypen für den Mapper werden durch das benutzte `InputFormat` festgelegt (später mehr dazu)



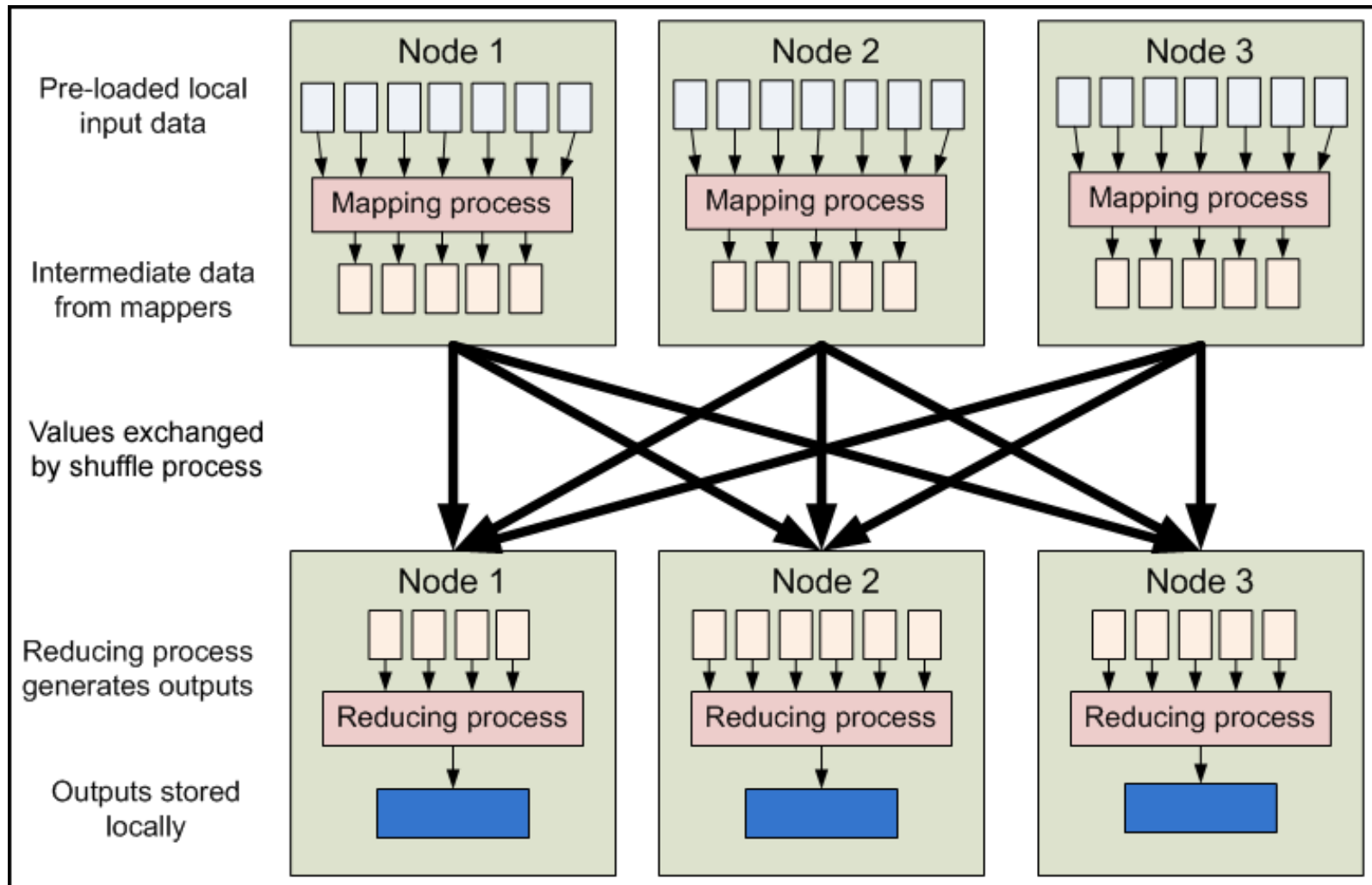
Die Driver Methode

Bemerkungen (fort.):

- Default ist `TextInputFormat` welches `(LongWritable, Text)` Paare erzeugt
 - Der Schlüssel ist der Byte Offset der Zeile in der Textdatei
 - Der Wert ist die Zeile selbst als String
- Der Aufruf von `JobClient.runJob(conf)` führt den Auftrag aus und wartet bis zum Ende
 - `submitJob()` kehrt sofort zurück



MapReduce Datenfluss





MapReduce Datenfluss

Die Daten kommen aus HDFS und sind dort als **Blöcke** gleichmäßig verteilt abgelegt. Das MapReduce Programm wird auf **vielen** oder **allen** Knoten im Cluster ausgeführt.

Dabei sind alle Map Tasks **gleichwertig**, d. h. irgendein Mapper kann irgendeine Eingabedatei bearbeiten.

Jeder Mapper liest die **rechnerlokalen** Daten (Blöcke) und verarbeitet diese.



MapReduce Datenfluss

Bemerkungen:

- Am **Ende** der Map Phase werden alle Werte mit den **gleichen** Schlüsseln an denselben Reducer geschickt
- Die Reducer laufen auf **denselben** Knoten wie auch die Mapper laufen
- Dies ist der **einzige** Kommunikationsschritt in MapReduce!



MapReduce Datenfluss

Bemerkungen:

- Einzelne Map Tasks tauschen **keine** Informationen aus, sie sind sich sogar völlig **unbewusst**, dass diese existieren
- Das gleiche gilt für Reduce Tasks
- Der ganze Datenfluss passiert **ohne** Eingriff des Anwenders
- Wenn ein Knoten **ausfällt**, dann kann deswegen Tasks **neu** ausgeführt werden



MapReduce Datenfluss

Bemerkungen (fort.):

- Sollte ein Task **Seiteneffekte** verursachen, z. B. mit anderen Prozessen kommunizieren, dann muss der Status selbst behandelt werden
- Durch die **Eliminierung** von Kommunikation und Seiteneffekten können Task Neustarts wesentlich **eleganter** abgehandelt werden

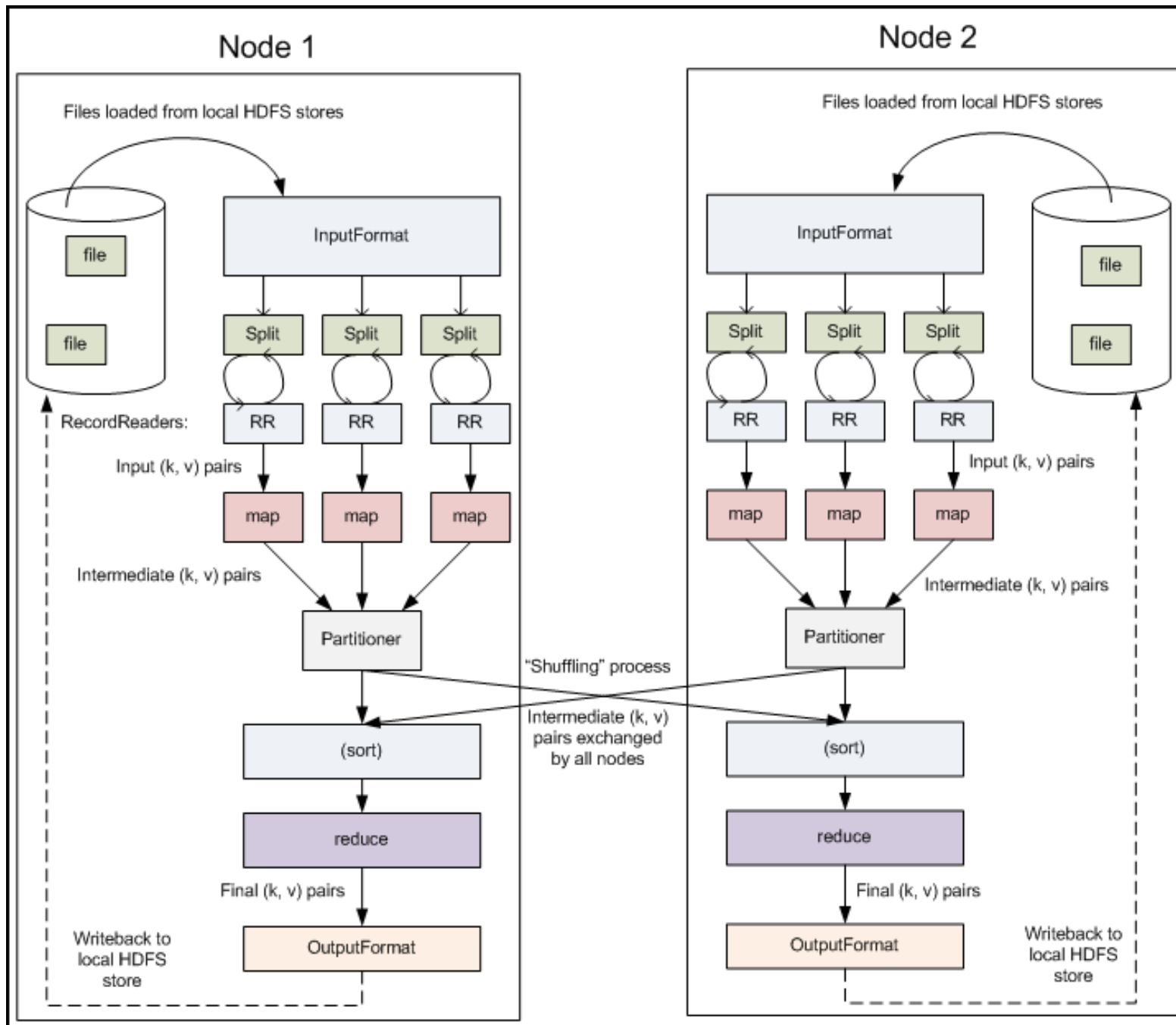
Es folgt eine detailliertere Darstellung des Prozesses.



MapReduce Datenfluss #2

Im folgenden Diagramm wird noch einmal **genauer** erklärt, welche Komponenten alle im Spiel sind **während** ein MapReduce Auftrag abläuft. Hier kann man sehen, wo die **selbst-geschriebenen** Teile des Codes ausgeführt werden und welche **anderen** Klassen noch involviert sind. Diese sind fast alle über den die **Driver** Methode konfigurierbar.

Wir besprechen nun die einzelnen Teile.





MapReduce Datenfluss #2

Input files

Dies sind Rohdaten, welche typischerweise in HDFS abgelegt sind. Es können aber auch andere Speichersysteme als Quellen dienen.

Das Format der Dateien spielt keine Rolle, denn ein Parser im InputFormat übernimmt das zerlegen.

Eingabedateien sind oft sehr groß!



MapReduce Datenfluss #2

InputFormat

Wie erwähnt, übernimmt das `InputFormat` das Verteilen und Lesen der Rohdateien. Es besitzt dazu folgende Funktionen:

- Wählt die Dateien oder andere Objekte aus welche gelesen werden sollen
- Definiert die `InputSplits`, welche die Dateien verteilt auf Mapper
- Stellt den `RecordReader` zur Verfügung



MapReduce Datenfluss #2

InputFormat (fort.)

Es werden standardmäßig einige InputFormat Klassen mitgeliefert. Einige basieren auf FileInputFormat (siehe setInputFormat() der JobConf Klasse), welche grundlegende Dienste zur Verfügung stellt. Ein Beispiel ist das Aufteilen in einen oder mehrere InputSplits.

Abgeleitete Klassen beschäftigen sich dann mit der eigentlichen Dateistruktur.



MapReduce Datenfluss #2

InputFormat (fort.)

Folgenden Klassen existieren:

InputFormat	Beschreibung	Schlüssel	Wert
TextInputFormat	Standardformat, liest Textzeilen aus Dateien	Der Byte Offset der Zeile	Der Inhalt der Zeile
KeyValueInputFormat	Zerlegt Zeilen in Key/Value Paare	Alles bis zum ersten TAB Zeichen	Der Rest der Zeile
SequenceFileInputFormat	Spezifisches, binäres Hadoop Dateiformat	Benutzerdefiniert	Benutzerdefiniert



MapReduce Datenfluss #2

InputSplits

Beschreibt die **Teilaufgabe** für einen einzelnen Map Task. Eine **große** Datei wird in **viele** Splits aufgeteilt und an Map Tasks übergeben. Ein Split entspricht oft der Größe eines HDFS Blockes, also **64** oder **128MB**. Dies ermöglicht das **parallele** Abarbeiten einer Datei und **gleichzeitig** eine günstige Verteilung der Teilaufgaben an **viele** Knoten im Cluster.



MapReduce Datenfluss #2

Das InputFormat **definiert** die Liste der Teilaufgaben, **pro** Split eine. Die Teilaufgaben werden dann an die Knoten **verteilt**, mit Beachtung der **Datenlokalität**, d. h. jeder Split kann theoretisch auf einem von drei (dem Replikationsfaktor) Rechnern ausgeführt werden. Jeder Rechner kann wiederum **viele** Tasks **parallel** ausführen. Dies kann über die Job Konfiguration **angepasst** werden.

Siehe
Übung 2!



MapReduce Datenfluss #2

RecordReader

Der Split definiert die Grenzen einer Teilaufgabe, aber nicht wie auf die Daten zugegriffen werden kann. Das übernimmt der `RecordReader` und erzeugt beliebige Datenpaare, welche dann einer nach dem anderen an den Mapper übergeben werden. Dies geschieht bis die Daten der Teilaufgabe komplett gelesen sind. Siehe auch: `LineRecordReader`.



MapReduce Datenfluss #2

Mapper

Der Mapper ist der Anwendercode und bekommt zwei Parameter:

- `OutputCollector`
 - Hat eine Methode namens `collect()`, welche die Ausgabepaare des Mappers entgegen nimmt
- `Reporter`
 - Kann über `getInputSplit()` die Split Informationen abgeben
 - Hat `setStatus()` und `incrCounter()` für Laufzeit-informationen des Jobs – mehr dazu in Einheit 3



MapReduce Datenfluss #2

Partition & Shuffle

Hier werden die Datenpaare über eine **Partitionierungsfunktion**, angewendet auf den Schlüssel eines gegebenen Paares, an **zugeordnete** Reducer übergeben. Ein Reducer **erhält** typischerweise Datenpaare von **vielen** verschiedenen Map Tasks. Das **Kopieren** der Daten zwischen **Map** und **Reduce** Phase wird Shuffle genannt.



MapReduce Datenfluss #2

Sort

Bevor die Werte eines **eindeutigen** Schlüssels an die Reduce Funktion übergeben werden kann muss Hadoop zuerst die Teilergebnisse **sortieren** und zuordnen. Damit bleiben **pro** Schlüssel **ein** oder **mehrere** Werte übrig.

Der Sort Schritt **sortiert** dazu alle Schlüssel und ruft dann für **jeden** in Reihenfolge den Reducer Code des Anwenders auf.



MapReduce Datenfluss #2

Reduce

Für **jeden** Reduce Task wird eine `Reducer` Instanz erzeugt. Für jeden **eindeutigen** Schlüssel bekommt der `Reducer` einen Aufruf der `reduce()` Funktion, welche neben dem Schlüssel einen `Iterator` über alle zugeordneten Werte übergeben bekommt.

Dabei sind die Werte nicht geordnet.

Weiterhin bekommt der Aufruf von `reduce()` auch einen `OutputCollector` und `Reporter`, genau wie die `map()` Funktion.



MapReduce Datenfluss #2

OutputFormat

Diese Klasse **gibt** die Datenpaare **aus**. Dies ist analog zum `InputFormat` (siehe oben). Es gibt **vorgefertigte** Klassen in Hadoop welche die Daten in HDFS oder dem lokalen Dateisystem ablegen.

Jeder `Reducer` schreibt in das **selbe** Ausgabeverzeichnis, deswegen ist die Reduce Task ID Teil des Dateinames: `<pfad>/part-nnnnnn`.

`setOutputFormat()` und `setOutputPath()` legen die Klasse und den Pfad für den Job fest.



MapReduce Datenfluss #2

OutputFormat (fort.)

Folgenden Klassen existieren:

OutputFormat	Beschreibung
TextOutputFormat	Standardformat, schreibt Textzeilen im Format „key \t value“
SequenceFile-OutputFormat	Schreibt binäre, Hadoop spezifische Dateien, welche für weitere Jobs benutzt werden können
NullOutputFormat	Verwirft die Datenpaare



MapReduce Datenfluss #2

RecordWriter

Wiederum analog zu dem RecordReader des InputFormat. Stellt eine **spezifische** Implementierung basierend auf dem **gewählten** Ausgabeformat bereit.

Output Files

Die **finalen** Dateien, nachdem der oder die Reducer ihre Arbeit abgeschlossen haben.



Einführung in MapReduce

Damit sind wir für diese Einheit am Ende angekommen, in Bezug auf MapReduce. Einheit 3 wird einige weiterführende Konzepte und Merkmale von MapReduce vorstellen. Dazu gehören Partitioner, Combiner, Job Informationen während der Ausführung, weitere Methoden der Map und Reduce Klassen und so weiter.



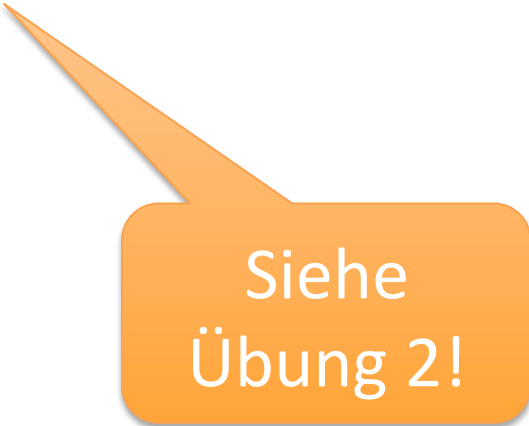
Einheit 2

- Rückblick auf Einheit 1
- Einführung in MapReduce
- **MapReduce Beispiele**
- YARN Übersicht



MapReduce Beispiel #1

Das WordCount Beispiel von vorhin. Dazu werden beliebige Textdateien an das MapReduce Programm übergeben und die Ergebnisse ausgegeben.



Siehe
Übung 2!



Log Format

Eine kurze Übersicht:

LogFormat "%h %l %u %t \"%r\" %>s %b" common

```
127.0.0.1 - frank [10/Oct/2000:13:55:36  
-0700] "GET /apache_pb.gif HTTP/1.0" 200  
2326
```

**LogFormat "%h %l %u %t \"%r\" %>s %b \"%
{Referer}i\" \"%{User-agent}i\"" combined**

```
127.0.0.1 - frank [10/Oct/2000:13:55:36  
-0700] "GET /apache_pb.gif HTTP/1.0" 200  
2326 "http://www.example.com/start.html"  
"Mozilla/4.08 [en] (Win98; I ;Nav)"
```




Log Format

Abschnitt	Erläuterung
127.0.0.1 (%h)	Die IP Adresse des Aufrufers
- (%l)	Identität („-“ bedeutet Wert nicht verfügbar)
frank (%u)	Benutzer gemäß Authentifizierung
[10/Oct/2000:13:55:36 -0700] (%t)	Zeitpunkt des Übertragungsende
"GET /apache_pb.gif HTTP/1.0" (\ "%r\"")	Die URI der Abfrage, zeigt auf die angeforderte Resource
200 (%>s)	Status Code des Servers für den Aufruf
2326 (%b)	Größe der zurückgesendeten Daten (ohne Kopfdaten)
"http://www.example.com/ start.html" (\ "% {Referer}i\"")	Combined: Woher der Aufruf kommt, z. B. welche Seite eine aufgerufene Grafik enthält
"Mozilla/4.08 [en] (Win98; I ;Nav)" (\ "% {User-agent}i\"")	Combined: Informationen welche der Client Browser über sich selbst berichtet

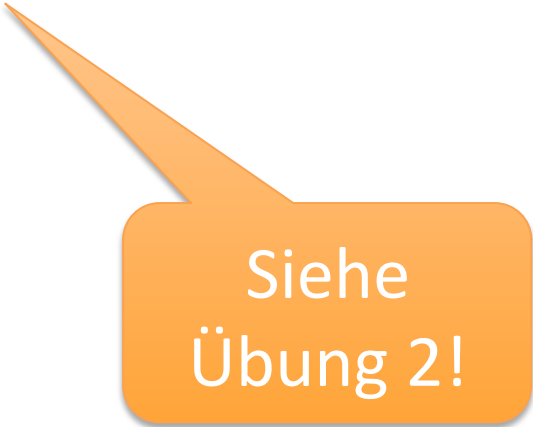
Quelle: <http://httpd.apache.org/docs/1.3/logs.html>



MapReduce Beispiel #2

Hier ist eine typische Abfrage, welche Daten aus einem Weblog aggregiert:

```
SELECT weblog.url,  
       sum(weblog.bytes)  
FROM weblog  
GROUP BY weblog.url;
```



Siehe
Übung 2!



MapReduce Beispiel #3 bis ...

Das MapReduce Packet hat eine eigene JAR Datei mit vielen Beispielen schon enthalten.

```
$ hadoop jar /opt/cloudera/parcels/CDH-4.4.0-1.cdh4.4.0.p0.39/lib/hadoop-0.20-mapreduce/hadoop-examples-2.0.0-mr1-cdh4.4.0.jar
```

An example program must be given as the first argument.

Valid program names are:

- aggregatewordcount: An Aggregate based map/reduce program that counts the words in the input files.
- aggregatewordhist: An Aggregate based map/reduce program that computes the histogram of the words in the input files.
- dbcount: An example job that count the pageview counts from a database.
- grep: A map/reduce program that counts the matches of a regex in the input.
- join: A job that effects a join over sorted, equally partitioned datasets
- multifilewc: A job that counts words from several files.
- pentomino: A map/reduce tile laying program to find solutions to pentomino problems.
- pi: A map/reduce program that estimates Pi using monte-carlo method.
- randomtextwriter: A map/reduce program that writes 10GB of random textual data per node.
- randomwriter: A map/reduce program that writes 10GB of random data per node.
- secondarysort: An example defining a secondary sort to the reduce.
- sleep: A job that sleeps at each map and reduce task.
- sort: A map/reduce program that sorts the data written by the random writer.
- sudoku: A sudoku solver.
- teragen: Generate data for the terasort
- terasort: Run the terasort
- teravalidate: Checking results of terasort
- wordcount: A map/reduce program that counts the words in the input files.



Einheit 2

- Rückblick auf Einheit 1
- Einführung in MapReduce
- MapReduce Beispiele
- **YARN Übersicht**



Was sagt das Internet?

Then Google evolved. Can Hadoop catch up?

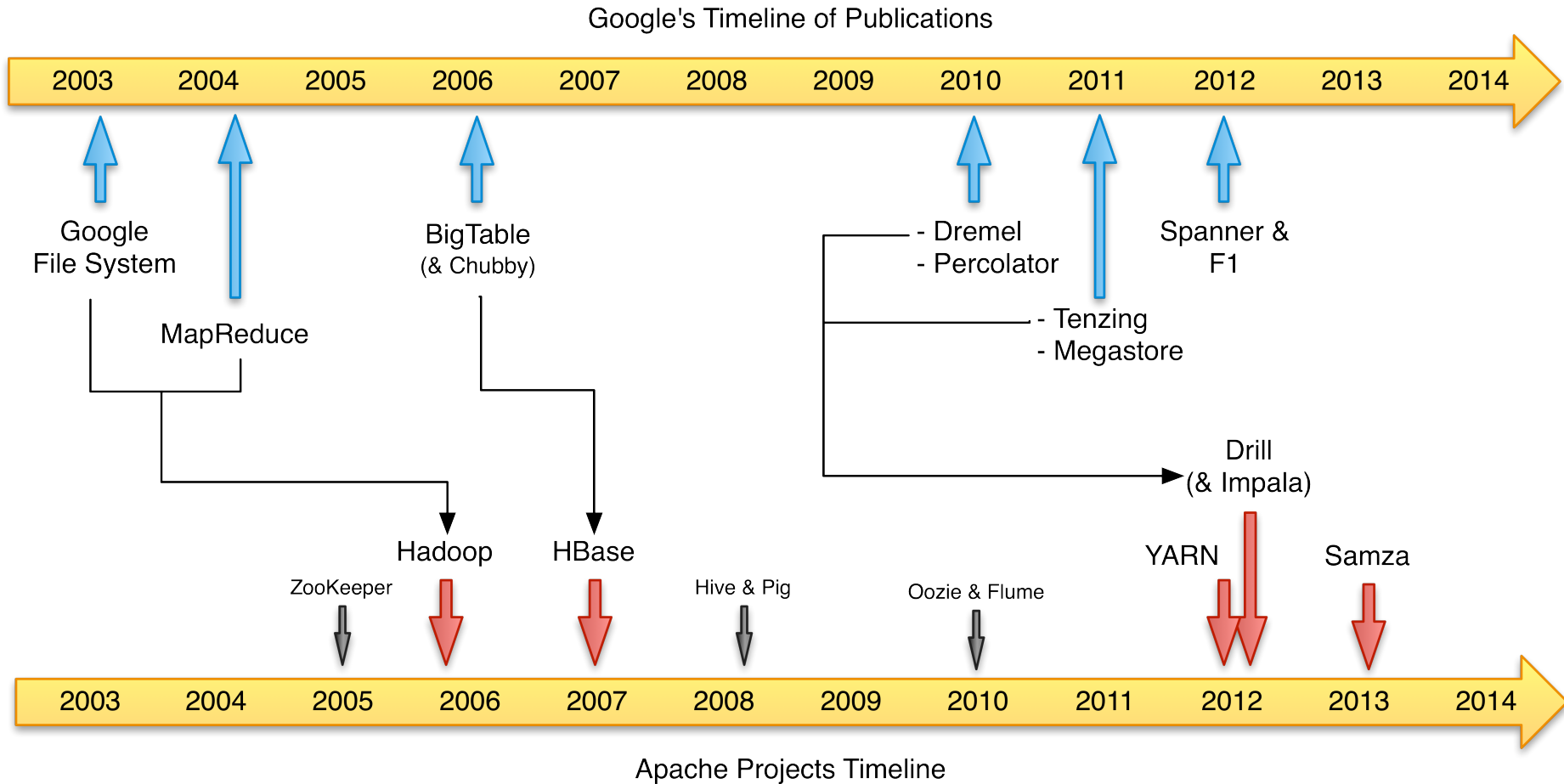
Most interesting to me, however, is that GMR no longer holds such prominence in the Google stack. Just as the enterprise is locking into MapReduce, Google seems to be moving past it. In fact, many of the technologies I'm going to discuss below aren't even new; they date back the second half of the last decade, mere years after the seminal GMR paper was in print.



Source: <http://gigaom.com/2012/07/07/why-the-days-are-numbered-for-hadoop-as-we-know-it/>

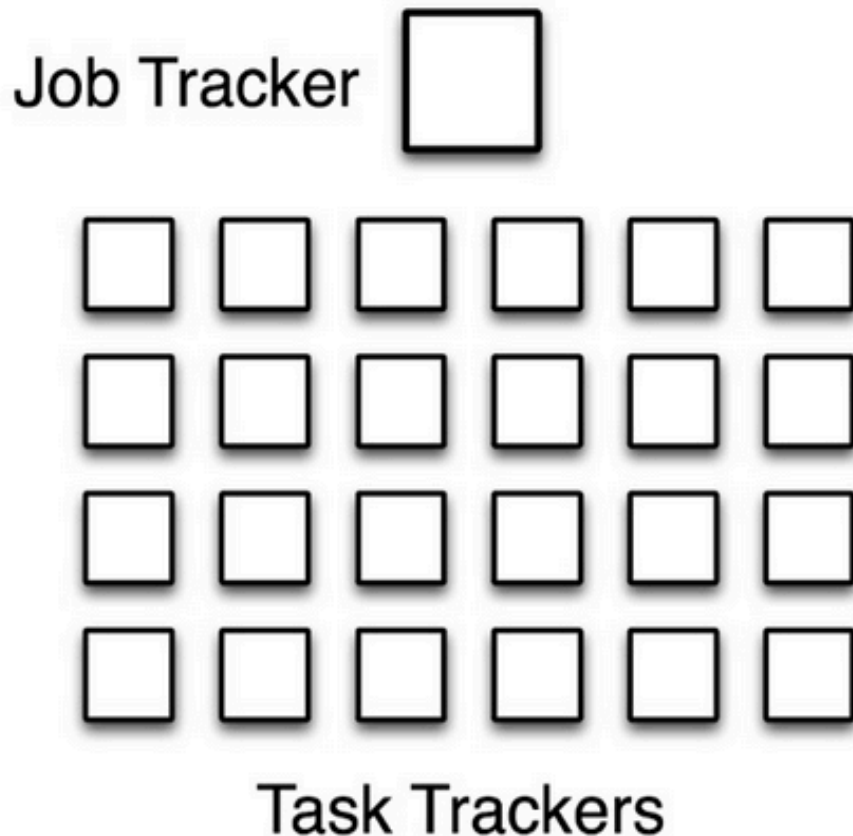


Wdh.: Hadoop Zeitleiste





Zuerst: Was ist MapReduce 1?



Master/Slave Architektur

- Ein zentraler Master: **JobTracker**
- Viele Arbeiterknoten: **TaskTracker**

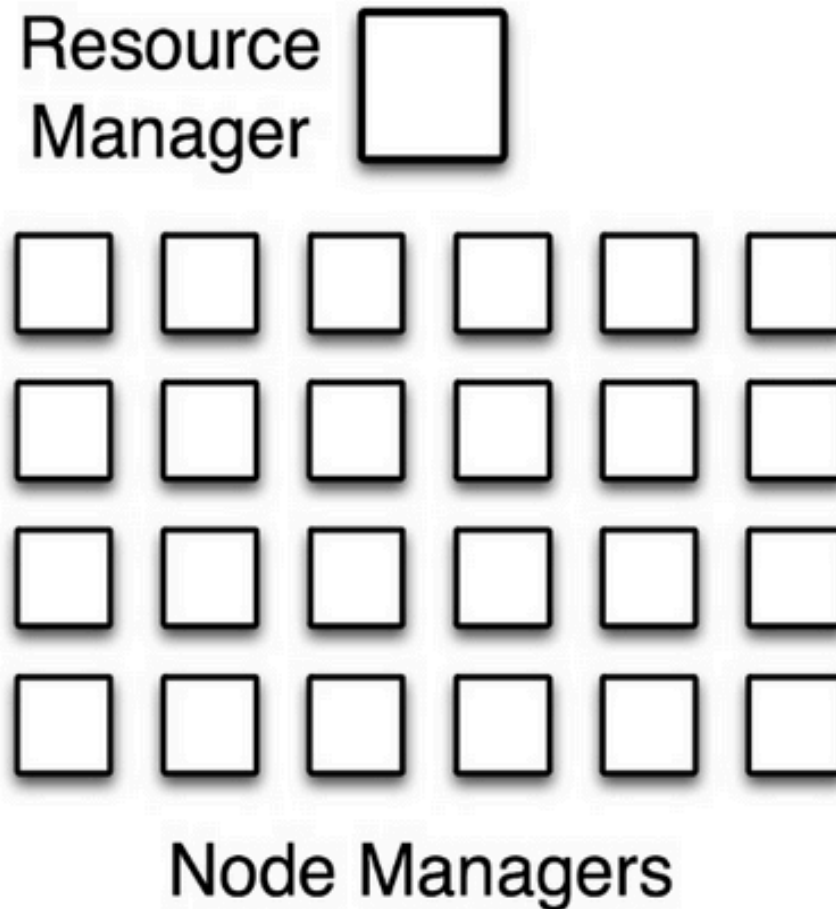


Motivation MR1 zu Ändern

- Skalierung auf >4000 Knoten
- Weniger, dafür größere Cluster
 - Ansonsten existieren wieder Datensilos!
- Hochverfügbarkeit des Job Tracker schwierig
 - Große, komplexe Statusinformationen
- Schlechte Resources Ausnutzung
 - Entweder sind „Slots“ in MR1 für Mapper oder Reducer

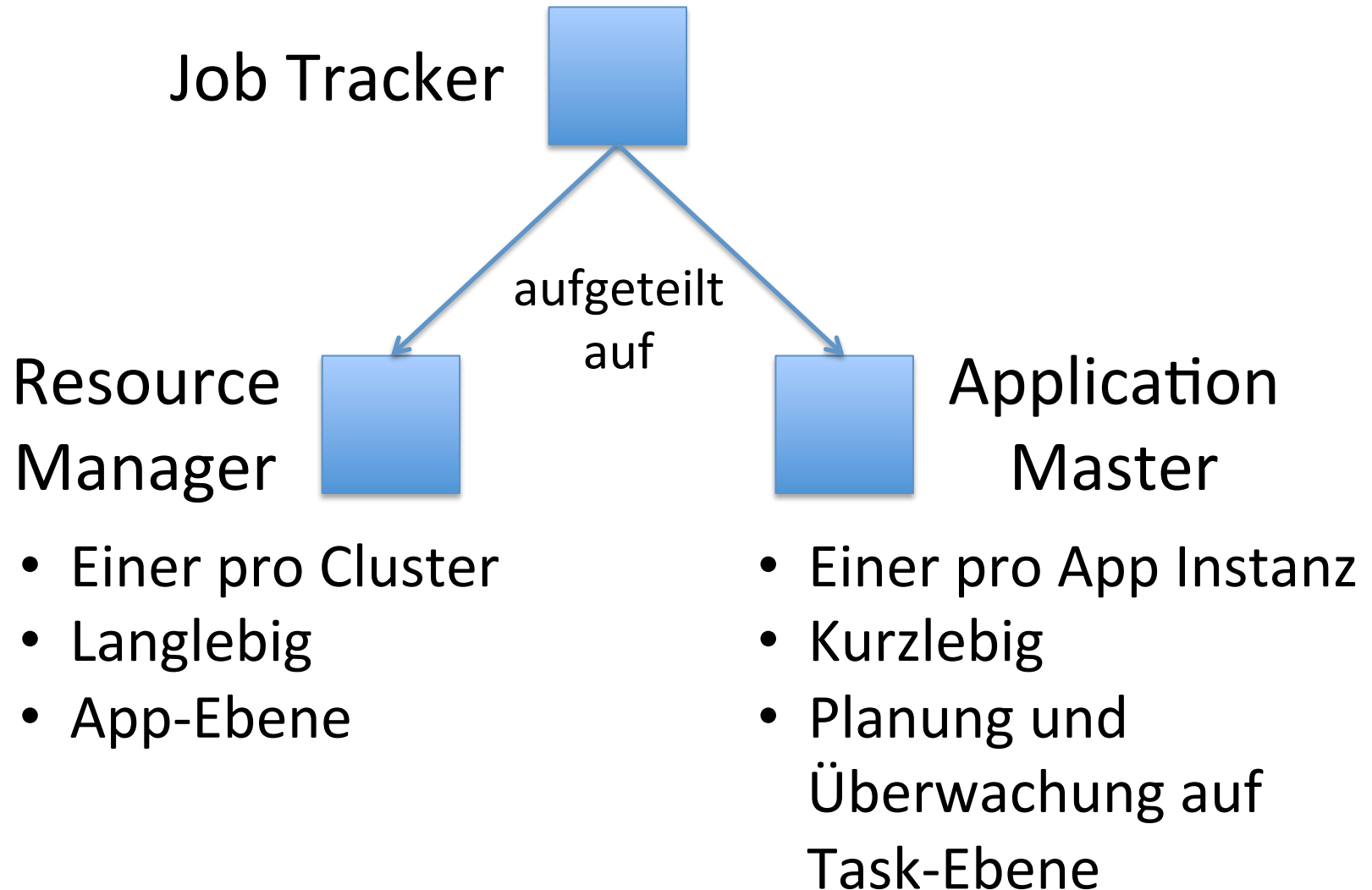


YARN: Yet Another Resource Negotiator





Aufgabenverteilung





Feinstufige Kontrolle

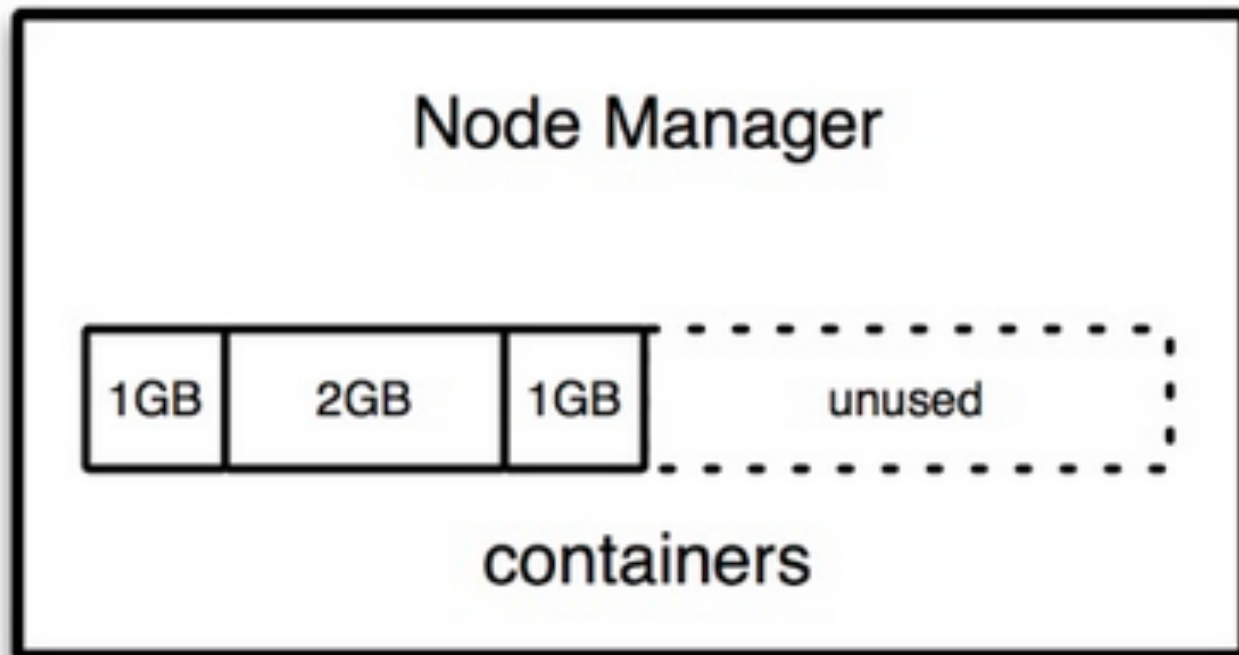
Der neue Node Manager ist ein generischer Task Tracker (aus dem MR1 Modell):

- Task Tracker
 - Feste Anzahl von Map und Reduce Slots
- Node Manager
 - Container mit variablen Ressourcen Begrenzung



Node Manager: Container

Container können mit beliebigen System-ressourcen umgehen (aber zur Zeit begrenzt auf Speicher).

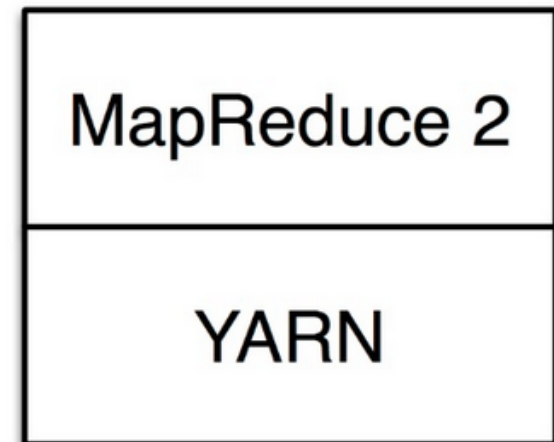




YARN + MapReduce 2

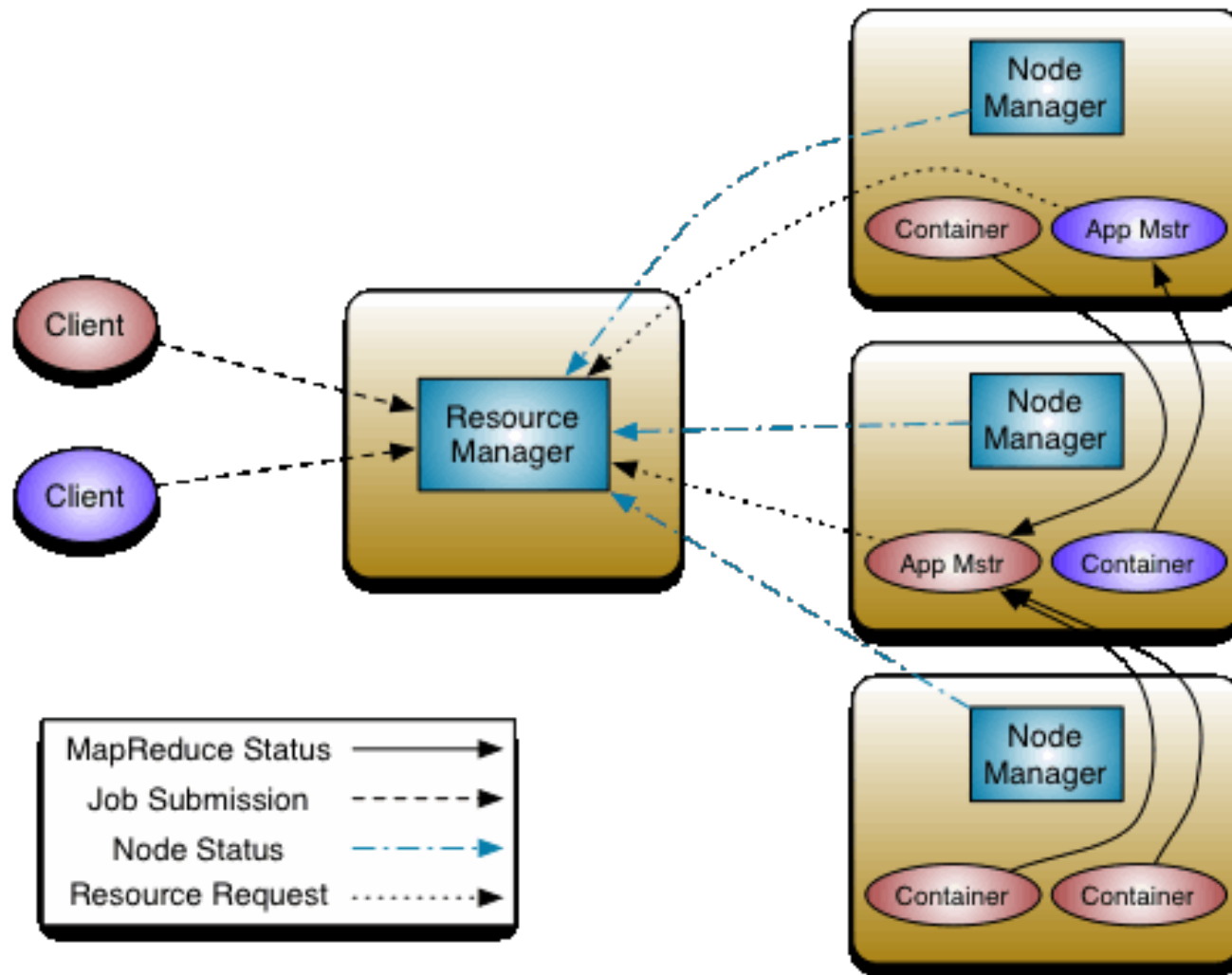
YARN führt MapReduce als eine eigene Anwendung aus. Dies kann man mit dem Linux Modell vergleichen:

- MR ist *User Space*
- YARN ist *Kernel Space*





Apache YARN Komponenten





YARN Anwendungen

Es gibt bereits einige Anwendungen für YARN, neben MapReduce, zum Beispiel:

- Distributed Shell
- Open MPI
- Master-worker
- Apache Giraph, Hama
- Spark
- HBase, MemCached, ...



Einheit 2

An dieser Stelle endet die zweite Einheit mit einer Einführung in Big Data Engineering. In der nächsten Einheit werden wir tiefer in MapReduce einsteigen und uns Algorithmen und bekannte Anwendungsfälle anschauen.

Bis bald!



Übung 2

Ziele:

- MapReduce WordCount schreiben und ausführen
- MapReduce Programm schreiben, welches eine Test Weblog Datei erstellt
 - Common Format
 - Variable Ausgabeformate (Text, SequenceFile)
- MapReduce Programm schreiben, welches die Weblogs summiert nach URI
 - Variation von Job Parameter testen (z. B. Parallelität)



Übung 2

Code:

<https://github.com/larsgeorge/fh-muenster-bde-lesson-2>



Quellen

- MapReduce Konzepte
 - <http://developer.yahoo.com/hadoop/tutorial/module4.html>
 - „Hadoop - The Definitive Guide“ von Tom White
<http://shop.oreilly.com/product/0636920021773.do>