Big-Data-Technologien

Kapitel 4: Hadoop – MapReduce

Hochschule Trier
Prof. Dr. Christoph Schmitz

Überblick

- MapReduce und funktionale Programmierung
- Code-Beispiele
- Serialisierung mit Writables
- Komplexere Operationen: Sortieren, Join
- Verteilter Zustand
- Gute und böse Operationen

Google 2004

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

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

Programs written in this functional style are au... cally parallelized and executed on a large cluster o...

given day, etc. Most such computations are concept... ally straightforward. However, the i... large and the compu... hundr...

5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE

...plexity, we designed a new ...n that allows us to express the simple computa-... tions we were trying to perform but hides the messy de...

Was sind „Map“ und „Reduce“?

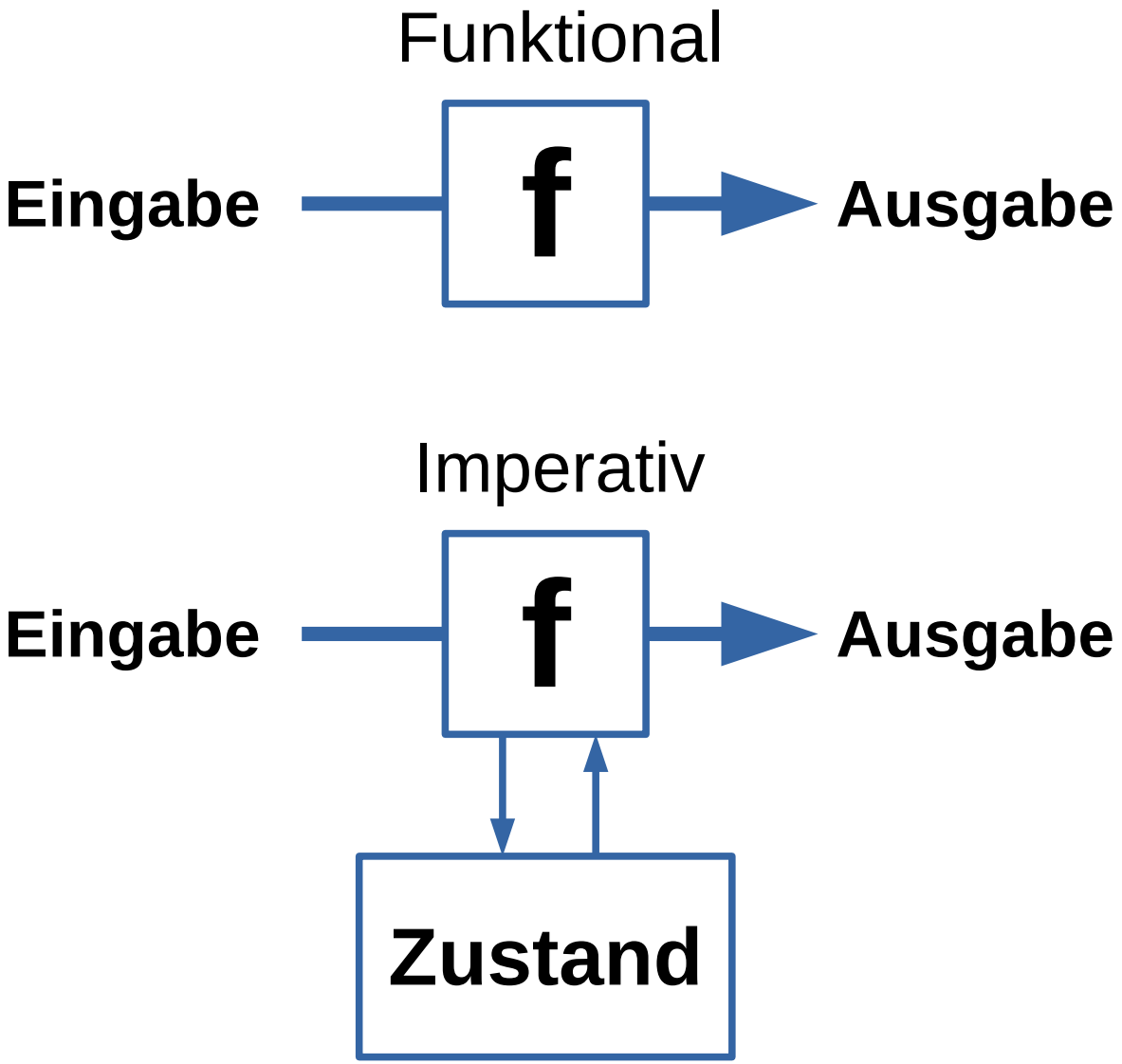
- Konstrukte aus der funktionalen Programmierung

```
(defun f (x) (* x 2))
(defun g (x y) (+ x y))

(defconstant l '(1 2 3 4 5))

(mapcar #'f l)
;; (2 4 6 8 10)

(reduce #'g (mapcar #'f l))
;; 30
```

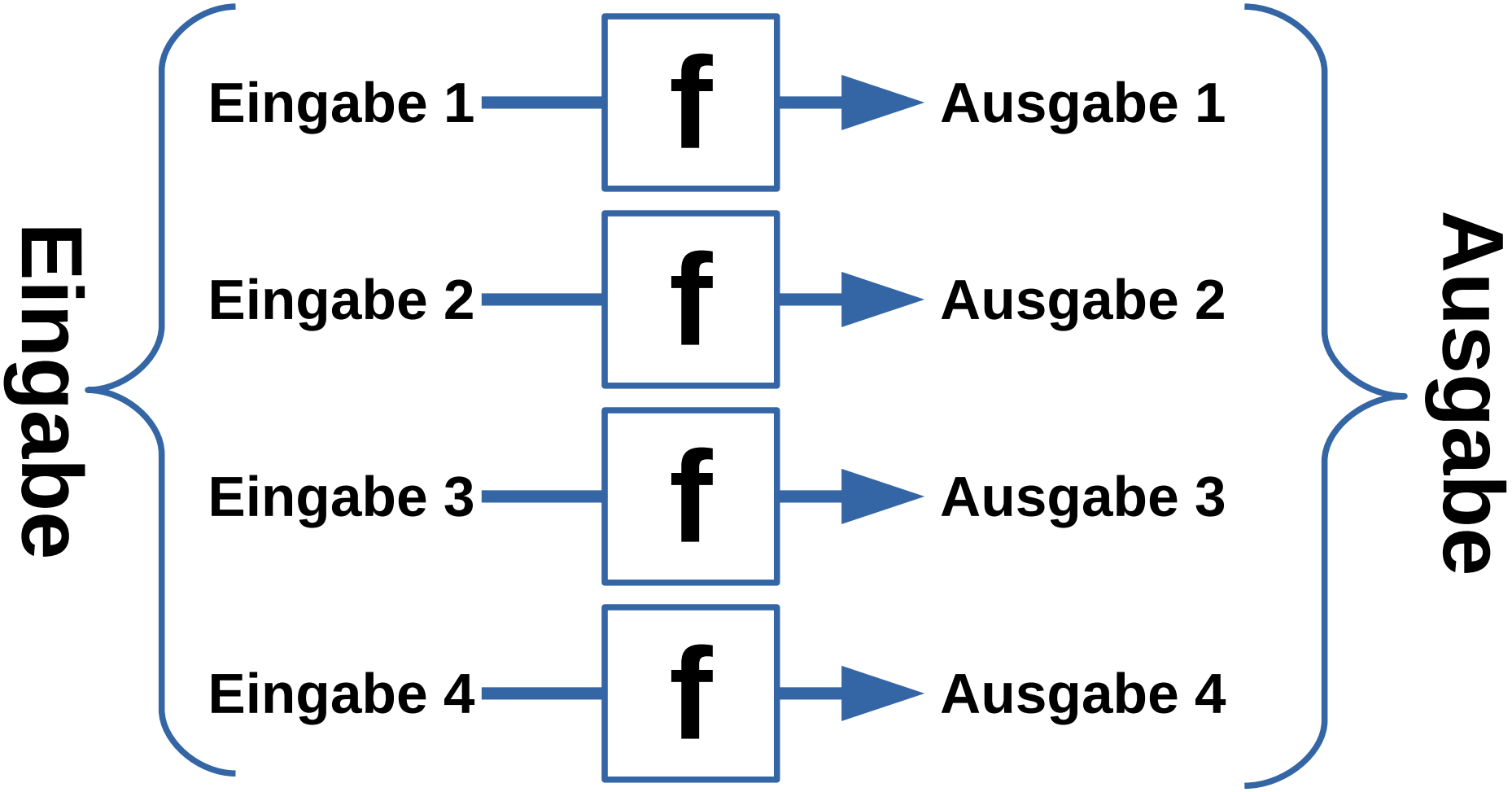
Map:
Transformation

Reduce:
Aggregation

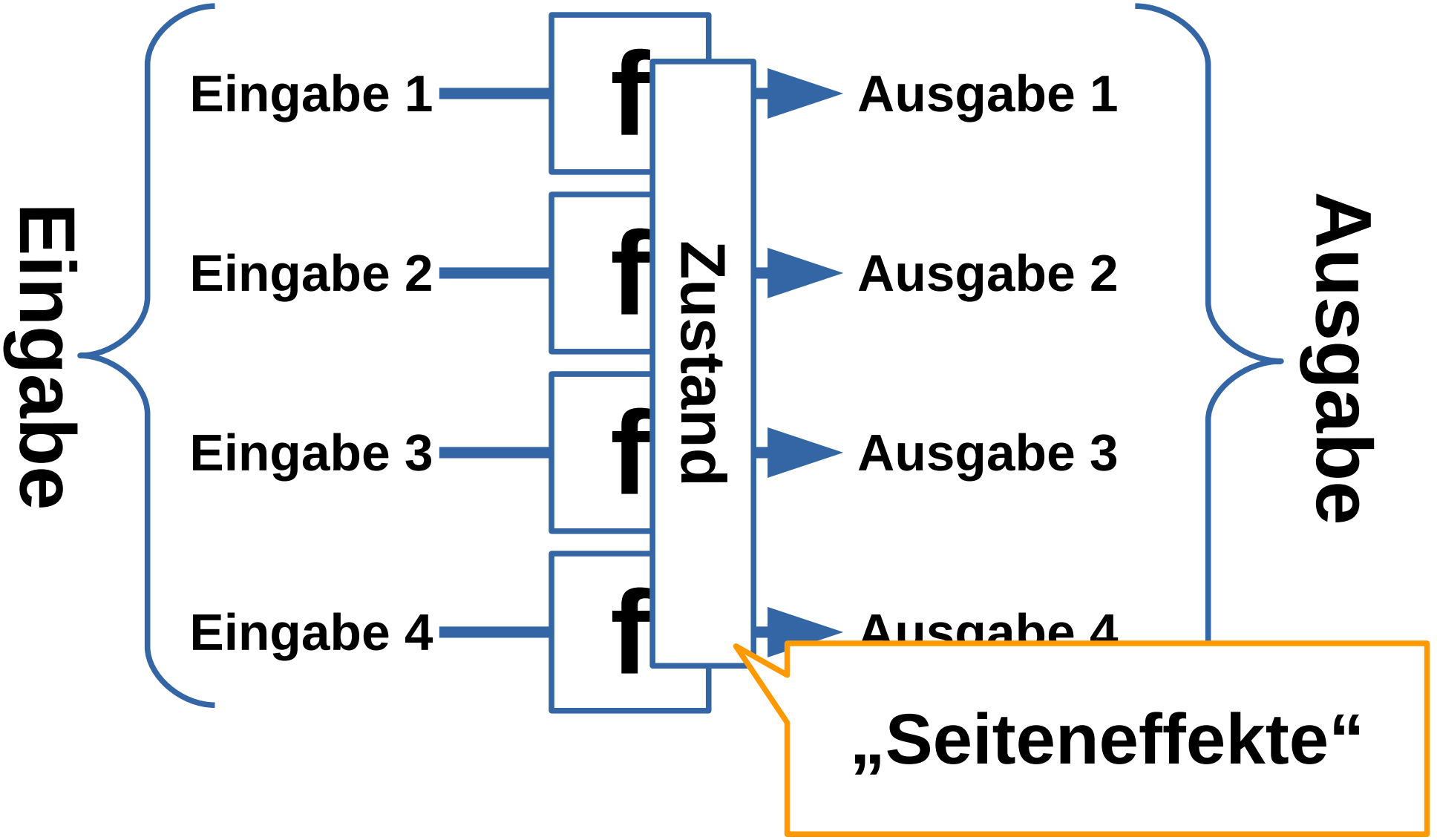
Funktionale und imperative Programmierung



Was hat das mit Big Data zu tun?



Was hat das mit Big Data zu tun?



Funktionale Programmierung und Big Data

- Freiheit von **Seiteneffekten**
  - Kein veränderlicher Zustand
  - Keine (übergreifenden) Variablen
- **Referenzielle Transparenz**
  - Jeder Ausdruck `f(a, b, c)` bedeutet überall das Gleiche
  - `d = f(a, b, c)` bedeutet Gleichheit, nicht Zuweisung
- **Unveränderliche Daten** bevorzugt *(„immutable“)*
  - Jede Kopie ist gleich!

✓ **Parallelisierbarkeit** der Berechnung

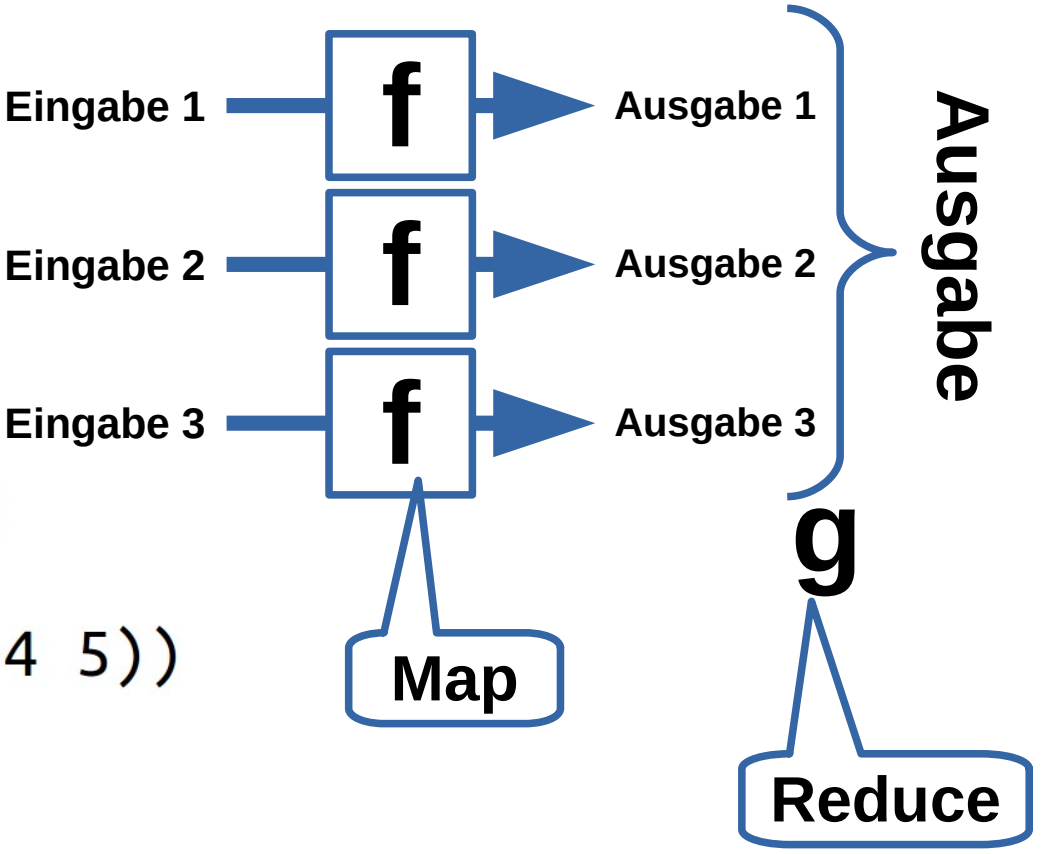
✓ **Replizierbarkeit** von Daten

✓ **Idempotenz:** Mehrfachausführung schadet nicht

✓ **Kein verteilter Zustand**

MapReduce

```
(defun f (x) (* x 2))
(defun g (x y) (+ x y))

(defconstant l '(1 2 3 4 5))

(mapcar #'f l)
;; (2 4 6 8 10)

(reduce #'g (mapcar #'f l))
;; 30
```



MapReduce etwas konkreter

- MapReduce verarbeitet **Name-Wert-Paare** (Key-Value-Pairs) (k, v)
- **Map**
  - verarbeitet jedes (k, v)
  - produziert daraus (k´, v´), (k´´, v´´), ...
- **Reduce**
  - bekommt die Ausgabe von Map
  - **nach Keys gruppiert**
  - produziert daraus Ergebnisse (k´´´, v´´´), ...

MapReduce: WordCount

(1001, **auto auto chemie baum**)
(1002, **auto baum chemie**)

Map

(auto, 1) (auto, 1) (chemie, 1) (baum, 1)
(auto, 1) (baum, 1) (chemie, 1)

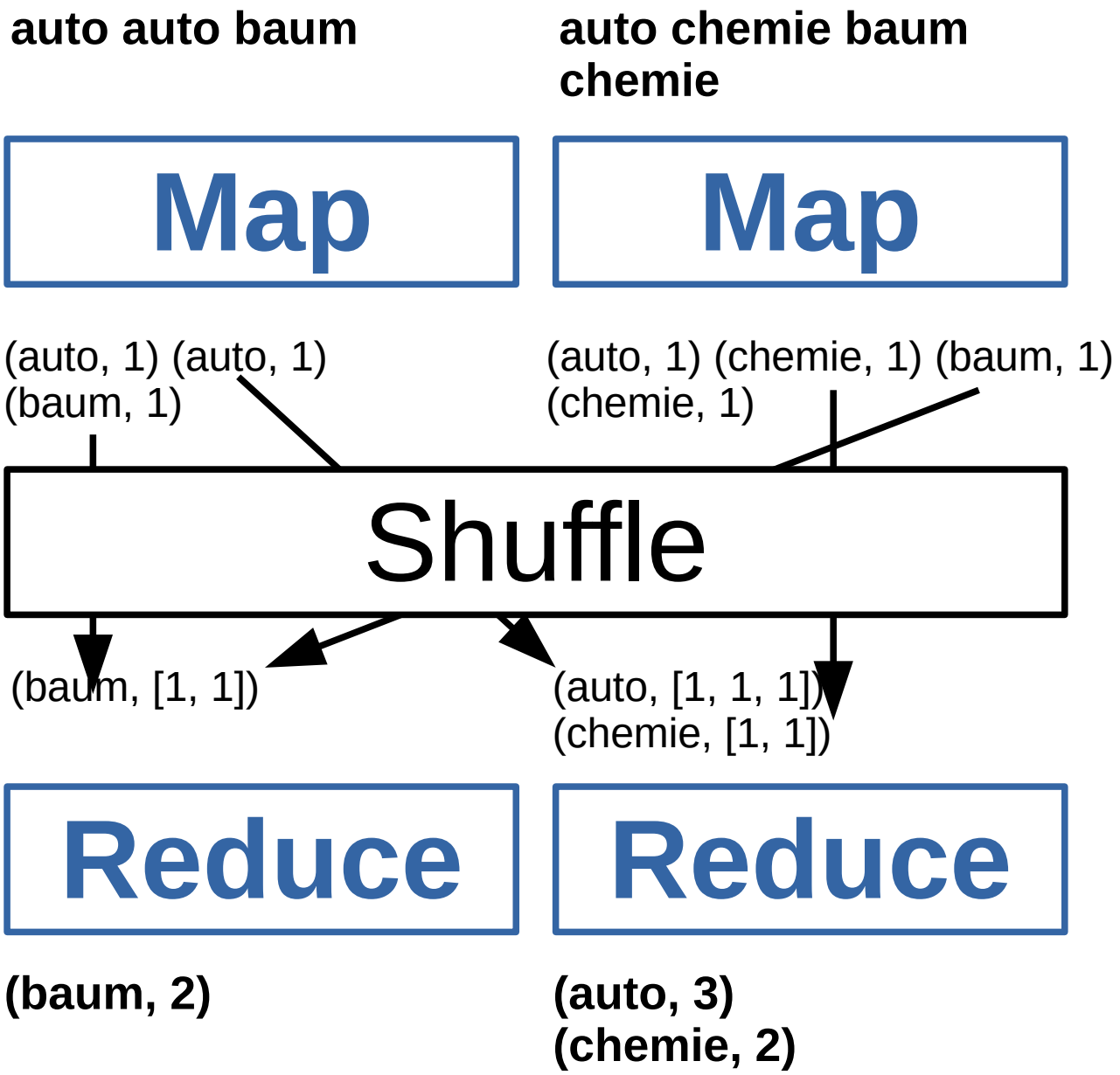
Shuffle

(auto, [1, 1, 1]) (baum, [1, 1])
(chemie, [1, 1])

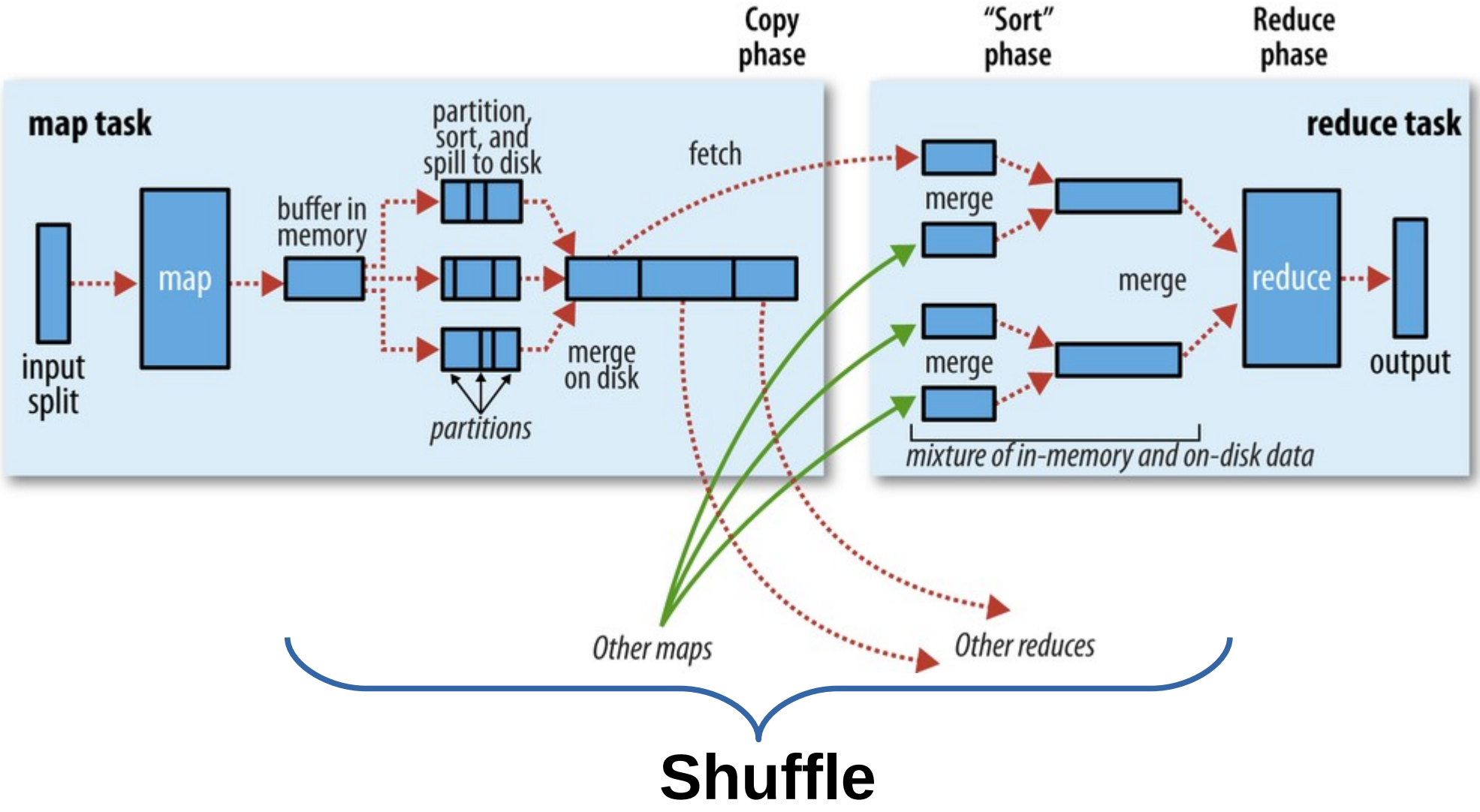
Reduce

(auto, 3) (baum, 2) (chemie, 2)

MapReduce: WordCount

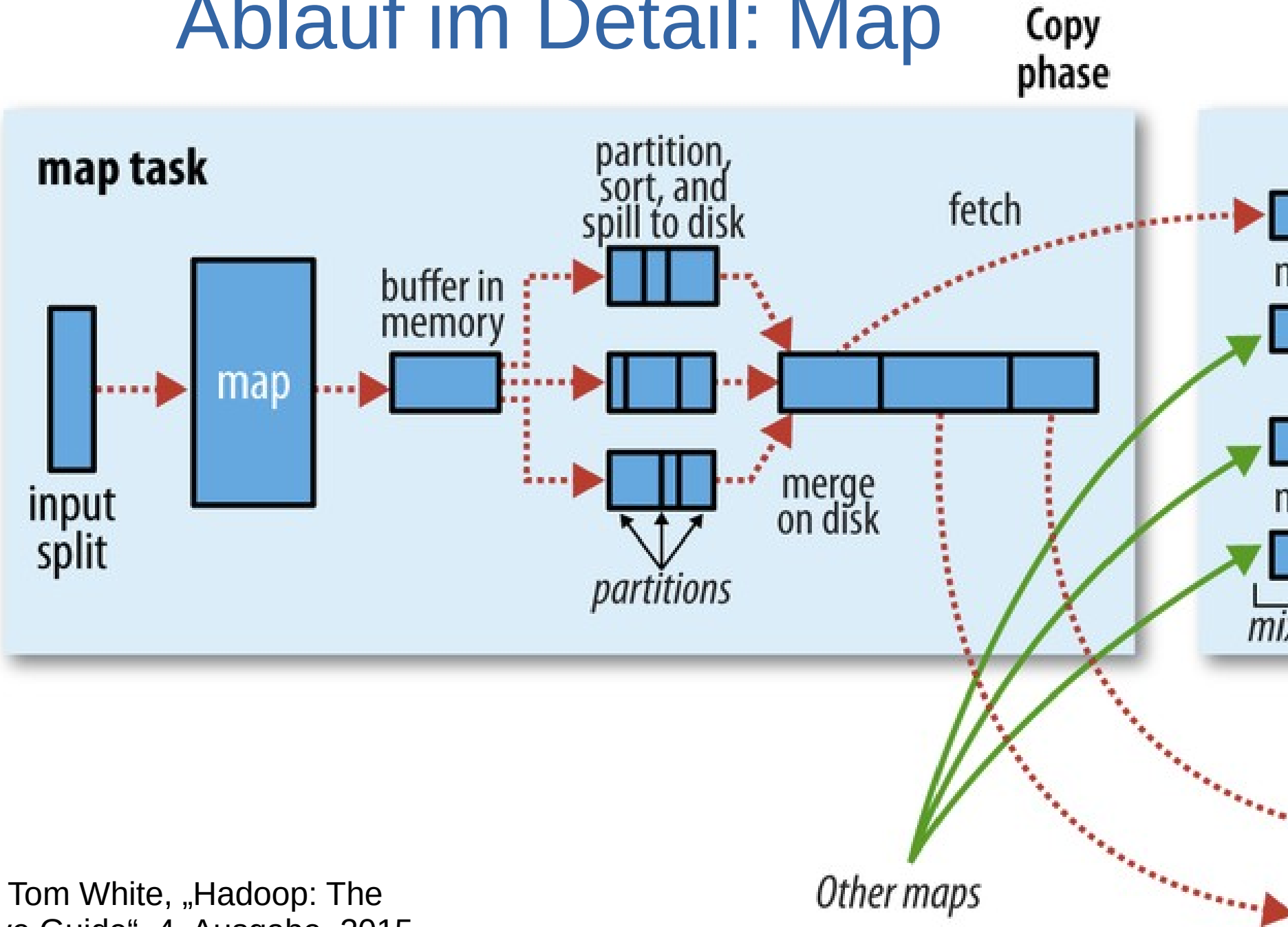


Ablauf im Detail



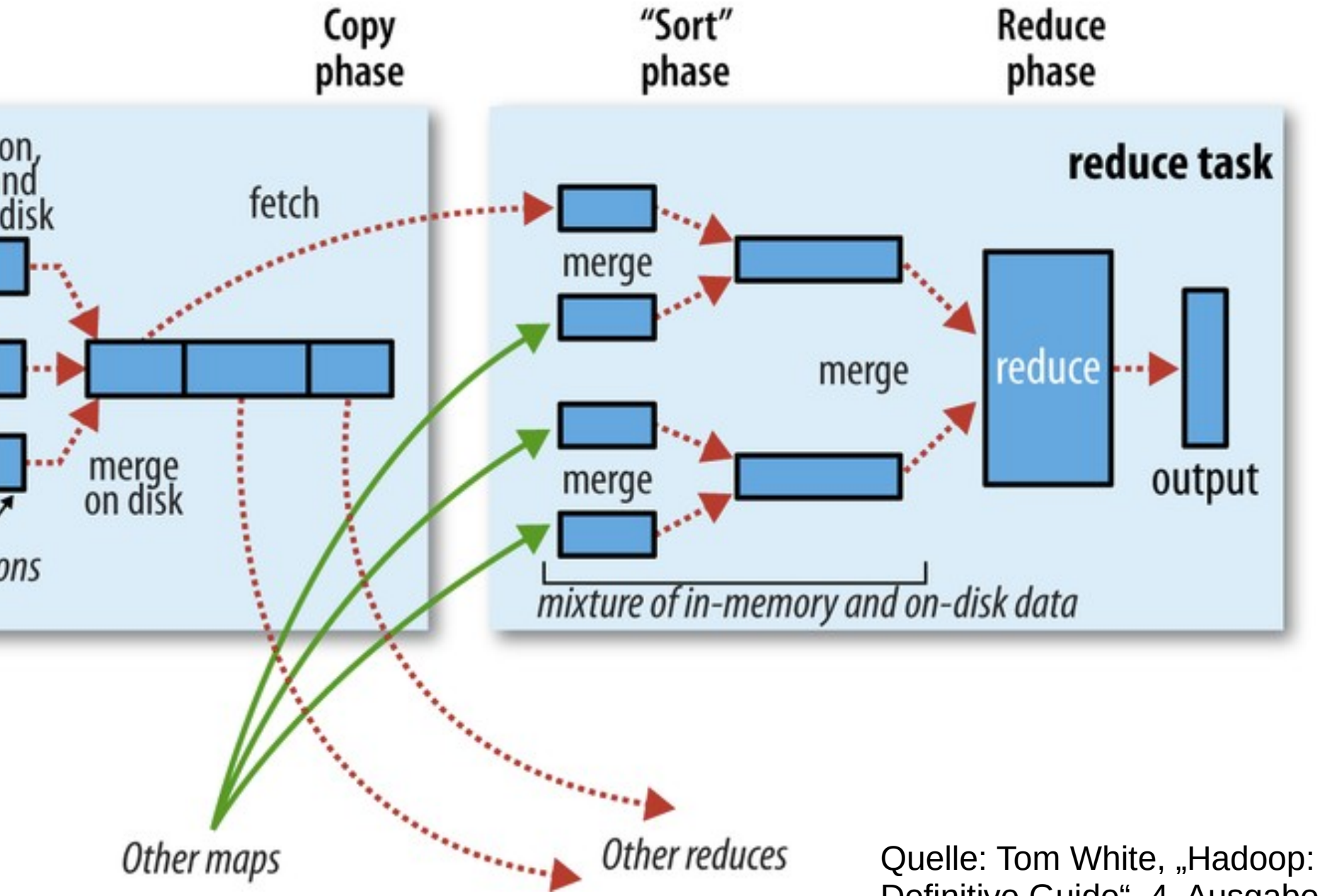
Quelle: Tom White, „Hadoop: The Definitive Guide“, 4. Ausgabe, 2015

Ablauf im Detail: Map



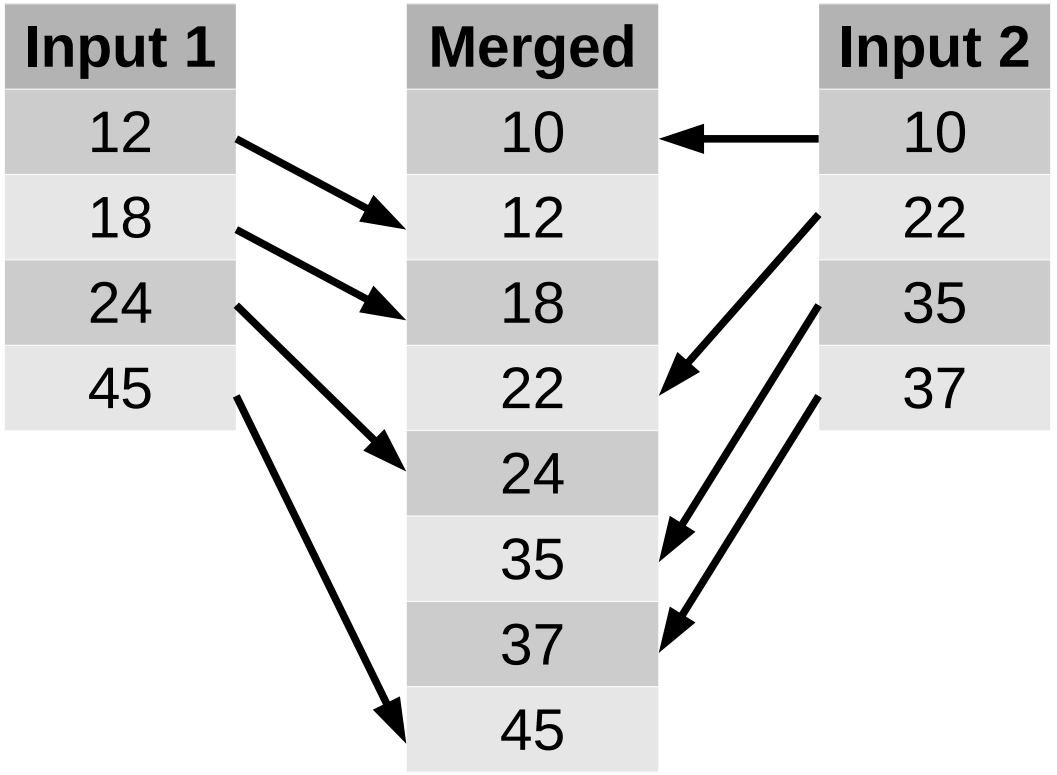
Quelle: Tom White, „Hadoop: The Definitive Guide“, 4. Ausgabe, 2015

Ablauf im Detail: Reduce



Quelle: Tom White, „Hadoop: The Definitive Guide“, 4. Ausgabe, 2015

Auffrischung: Merge Sort



Zeig mir Code!

```
public class WordCountMapper
        extends Mapper<LongWritable, Text, Text, LongWritable> {
    @Override
    protected void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        for (String word: value.toString().split(" ")) {
            context.write(new Text(word), new LongWritable(1));
        }
    }
}
```

(123, auto baum chemie auto)

(auto, 1)
(baum, 1)
(chemie, 1)
(auto, 1)

Zeig mir Code!

```
public class WordCountReducer
        extends Reducer<Text, LongWritable, Text, LongWritable> {
    @Override
    protected void reduce(Text key, Iterable<LongWritable> values, Context context)
            throws IOException, InterruptedException {
        long sum = 0;
        for (LongWritable count: values) {
            sum += count.get();
        }
        context.write(key, new LongWritable(sum));
    }
}
```

(auto, [1, 1, 1, 1, 1, 1])

(auto, 6)

Zeig mir Code!

```
@Override
public int run(String[] args) throws Exception {
    Job job = Job.getInstance(getConf(), "wordcount");

    job.setJarByClass(WordCountMain.class);

    job.setMapperClass(WordCountMapper.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(LongWritable.class);

    job.setReducerClass(WordCountReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(LongWritable.class);

    job.setNumReduceTasks(50);

    job.setInputFormatClass(FileInputFormat.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));

    job.setOutputFormatClass(FileOutputFormat.class);
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    return job.waitForCompletion(true) ? 0 : 1;
}
```

Mapper

Reducer

Eingabe

Ausgabe

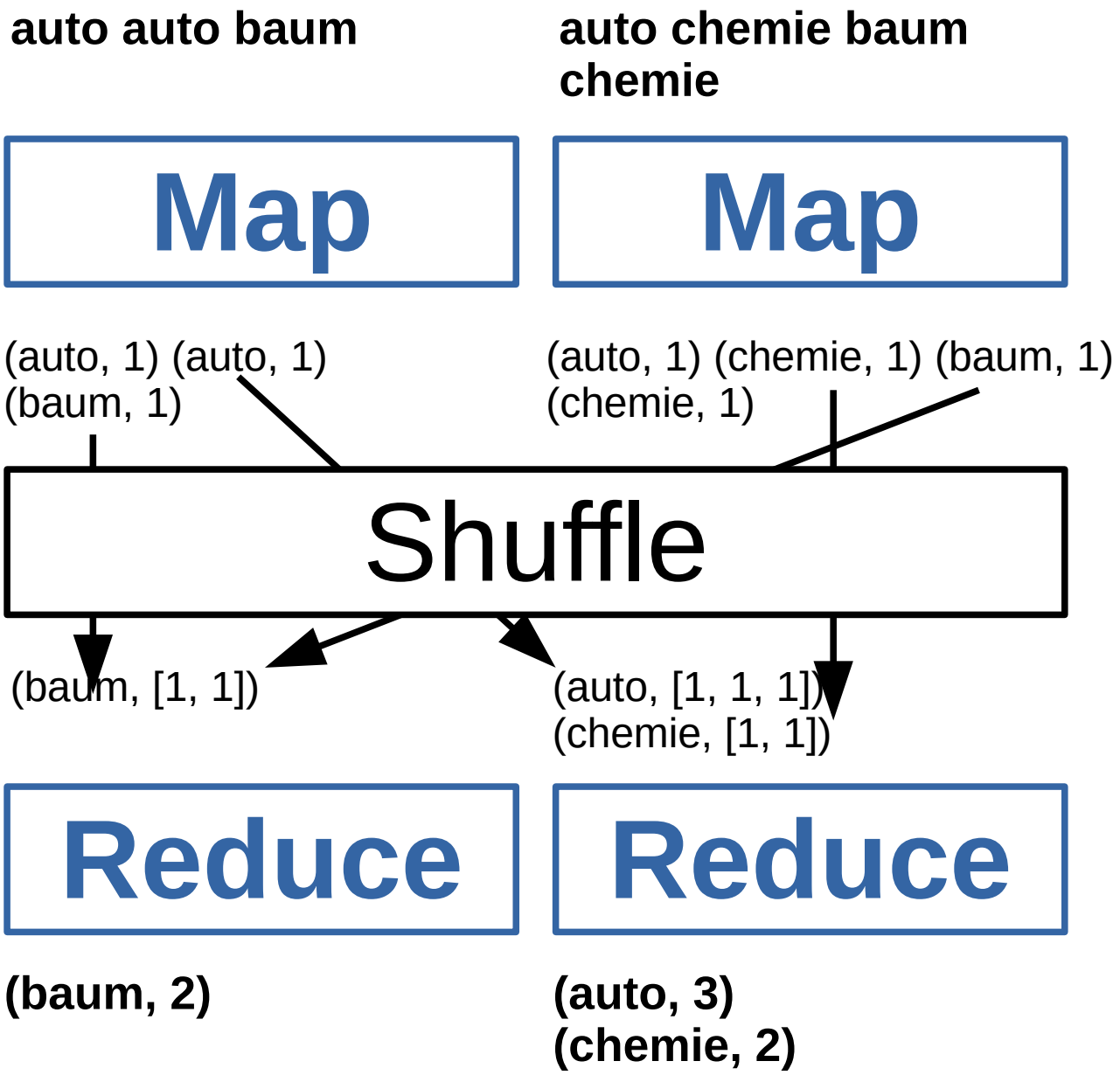
Beispiel starten

```
$ yarn jar wordcount.jar data/input data/output
```

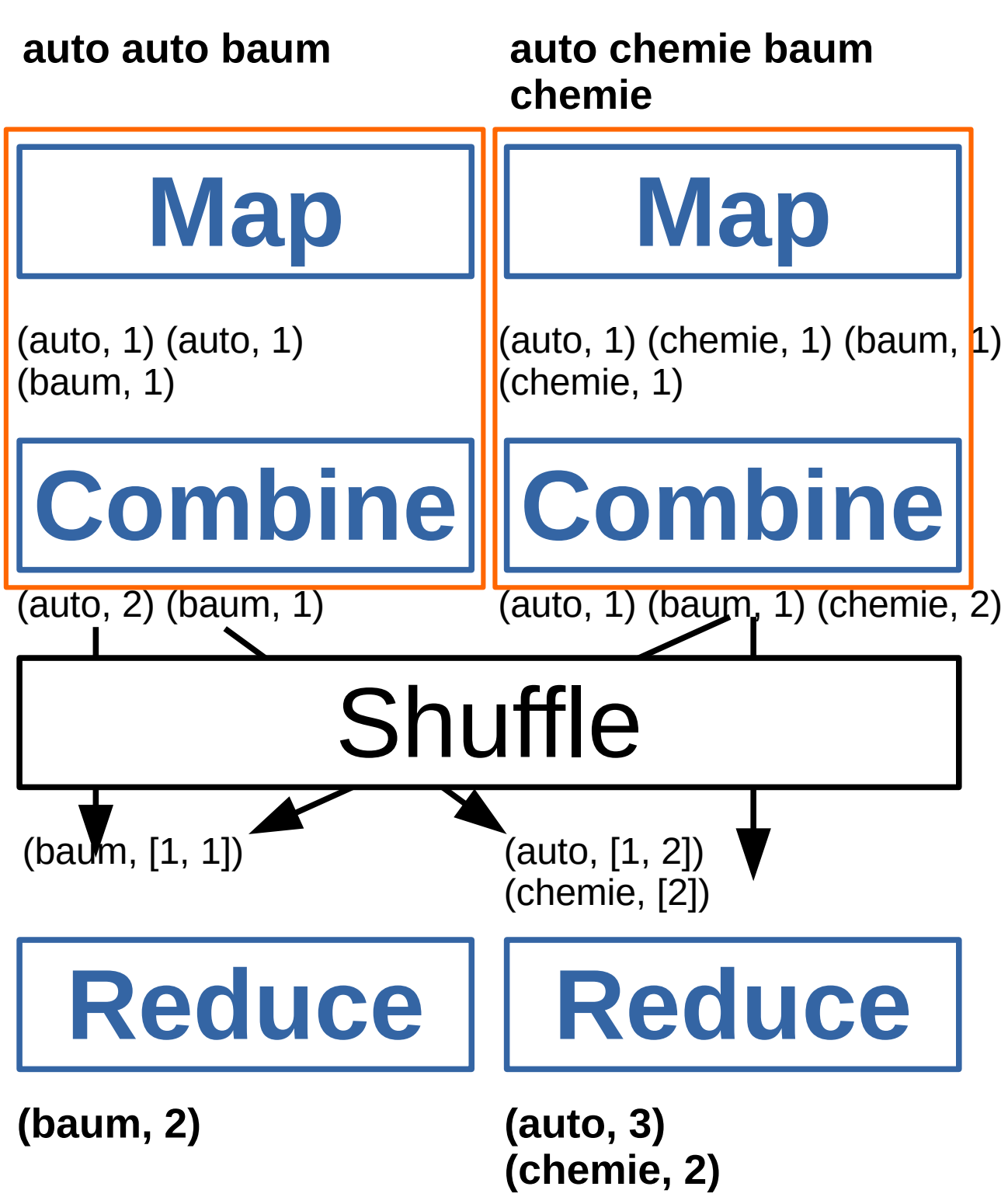
Code („Job-Jar“)

args

MapReduce: WordCount



Optimierung: Combiner



Zeig mir Code!

```
Job job = Job.getInstance(getConf());

job.setMapperClass(WordCountMapper.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(LongWritable.class);

job.setCombinerClass(WordCountReducer.class);

job.setReducerClass(WordCountReducer.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);

...
```

Mapper

Combiner

Reducer

Datentypen

- Beliebige Typen als Keys und Values...
- ... sofern serialisierbar: **Writable**
- vgl. Javas „Serializable“
- Vordefiniert:
  `Text, LongWritable, IntWritable, ...`

```
public interface Writable {
    void write(DataOutput out) throws IOException;
    void readFields(DataInput in) throws IOException;
}
```

Writables: Beispiel

```
public class UidTimestamp implements Writable {
    private String uid;
    private long timestamp;

    @Override
    public void write(DataOutput out) throws IOException {
        WritableUtils.writeString(out, uid);
        out.writeLong(timestamp);
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        uid = WritableUtils.readString(in);
        timestamp = in.readLong();
    }
}
```

Zwischenfazit

- MapReduce: funktionale Konzepte
- Praktisch beliebig verteilbar

- **Map:** Transformiere einzelne Datensätze
- **Reduce:** Aggregiere gruppierte Datensätze

- **Shuffle:** Verbindung zwischen Map und Reduce
- Gruppierung → Sortierung

Weitere Beispiele

- Daten **filtern**
  - Mapper: `if (P(k, v)) { write(k, v); }`
  - Reducer: `-`

- Daten **zählen**
  - Mapper:
```
foreach (input) { sum++; }
  write(„SUM“, sum);
```
  - Reducer:
```
foreach (input) { sum += input; }
  write(„TOTAL“, sum);
```

Komplexere Operationen

- **Sortieren**
  - Shuffle sortiert ohnehin schon!
  - Andere Sortierreihenfolge?
  - Sortieren nach Values statt nach Keys?

- **Joins**
  - Verschiedene Varianten
  - Hier: **Sort-Merge**-Join auf der Reducer-Seite

Sortieren: Secondary Sort

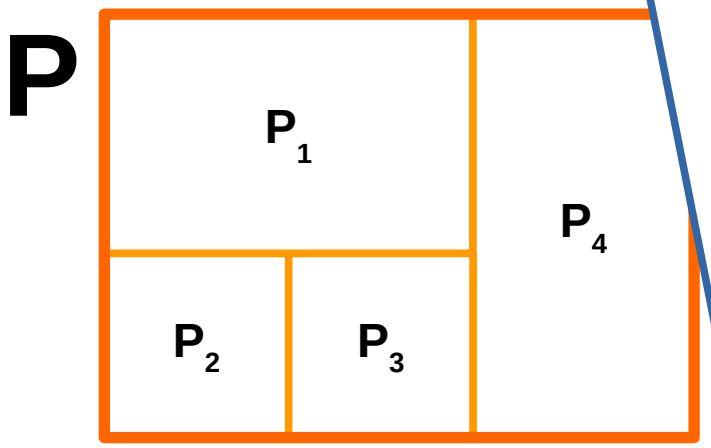
- Beispiel: Logdaten mit User-ID und Zeitstempel

```
uid1    2017-02-05 ...
uid2    2017-02-05 ...
uid1    2017-02-06 ...
uid2    2017-02-04 ...
```

- Gewünscht:
  - Reducer **gruppiert** pro User-ID
  - **Sortiert** nach Zeitstempel

Sortieren: Secondary Sort

- Eingriffe an verschiedenen Stellen möglich:
  - **Partitionierung:** Verteilen auf Reducer
  - **Gruppierung:** Zusammenfassung der Tupel
  - **Sortierung:** Sortierung innerhalb der Gruppen



Partitionierung:

$P = P_1 \cup P_2 \cup \ldots \cup P_n$

$\forall i \neq j: P_i \cap P_j = \emptyset$

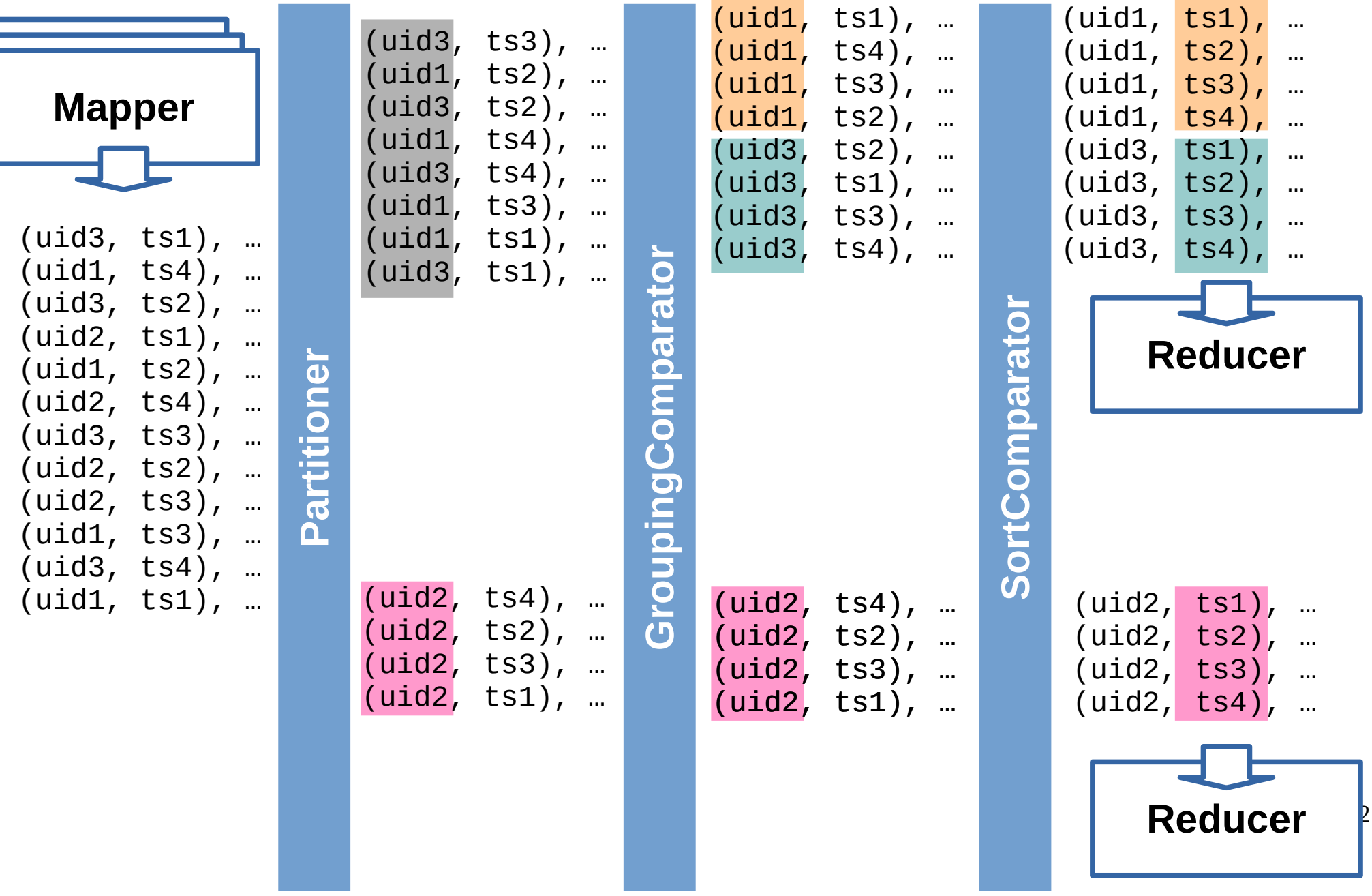
Sortieren: Secondary Sort

- Lösung: **zusammengesetzte Schlüssel**

```
((uid1, 2017-02-05), ...)
((uid2, 2017-02-05), ...)
((uid1, 2017-02-06), ...)
((uid2, 2017-02-04), ...)
```

- **Partitionieren:** nach User-ID
- **Gruppieren:** nur ersten Teil nutzen
- **Sortieren:** nutze kompletten Schlüssel

Secondary Sort im Detail



Secondary Sort im Detail

```
job.setPartitionerClass(UidPartitioner.class);
job.setGroupingComparatorClass(UidComparator.class);
job.setSortComparatorClass(UidTimestampComparator.class);
```

Secondary Sort im Detail

```
public class BuggyUidPartitioner extends Partitioner<UidTimestamp, Text> {
    @Override
    public int getPartition(UidTimestamp key, Text value, int numPartitions) {
        int hashCode = key.getUid().hashCode();
        return Math.abs(hashCode) % numPartitions;
    }
}
```

```
antiUser.hashCode() == Integer.MIN_VALUE
Math.abs(antiUser.hashCode()) == Integer.MIN_VALUE
Math.abs(antiUser.hashCode()) < 0 !
```

Der AntiUser

Secondary Sort im Detail

```
public class UidPartitioner extends Partitioner<UidTimestamp, Text> {
    @Override
    public int getPartition(UidTimestamp key, Text value, int numPartitions) {
        int hashCode = key.getUid().hashCode();
        if (hashCode == Integer.MIN_VALUE) {
            hashCode = 0;
        }
        return Math.abs(hashCode) % numPartitions;
    }
}
```

Secondary Sort im Detail

```
public class UidComparator extends WritableComparator {

    public UidComparator() {
        super(UidTimestamp.class, true);
    }

    @SuppressWarnings("rawtypes")
    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        UidTimestamp uid1 = (UidTimestamp) a;
        UidTimestamp uid2 = (UidTimestamp) b;

        return uid1.getUid().compareTo(uid2.getUid());
    }
}
```

Secondary Sort im Detail

```
public class UidTimestampComparator extends WritableComparator {

    public UidTimestampComparator() {
        super(UidTimestamp.class, true);
    }

    @SuppressWarnings("rawtypes")
    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        UidTimestamp uid1 = (UidTimestamp) a;
        UidTimestamp uid2 = (UidTimestamp) b;

        return ComparisonChain.start()
                .compare(uid1.getUid(), uid2.getUid())
                .compare(uid1.getTimestamp(), uid2.getTimestamp())
                .result();
    }
}
```

Wiederholung: Secondary Sort

- Eingriffe an verschiedenen Stellen möglich:
  - **Partitionierung:** Verteilen auf Reducer
  - **Gruppierung:** Tupel zusammen verarbeiten
  - **Sortieren:** Sortierung innerhalb der Gruppen

Joins in MapReduce

- **Join**
  - Zusammenführen von Daten mit gleichen Schlüsseln

Name	Alter
Alice	23
Bob	39
Charlie	34

Name	Haarfarbe
Alice	rot
Bob	blond
Charlie	braun
Diana	schwarz

Name	Alter	Haarfarbe
Alice	23	rot
Bob	39	blond
Charlie	34	braun

Implementierungsvarianten: A ⋈ B

- **Nested Loop Join**
  - foreach (a in A)
    foreach (b in B)
    if (a == b) output(...)

O(m·n)

- **Hash Join**
  - Lies A in Hashtabelle H
  - foreach (b in B)
    if (b in H) output(...)

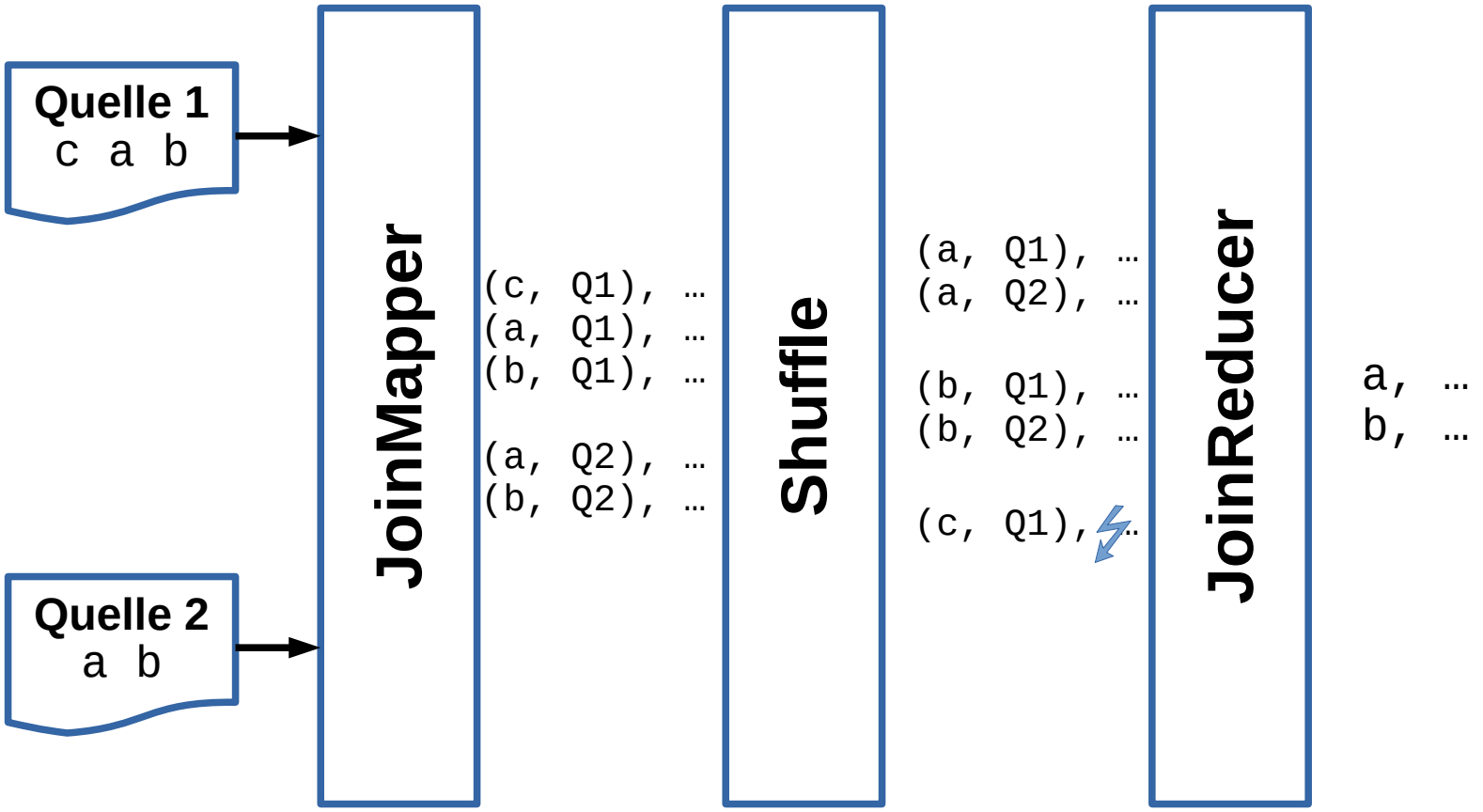
O(m+n)

Implementierungsvarianten: A ⋈ B

- **Sort-Merge-Join**
  - $S_A \leftarrow$ sort(A)
  - $S_B \leftarrow$ sort(B)
  - result $\leftarrow$ merge(S_A, S_B)

O(m log m + n log n)

Sort-Merge-Join (Reduce-Side Join) in MapReduce

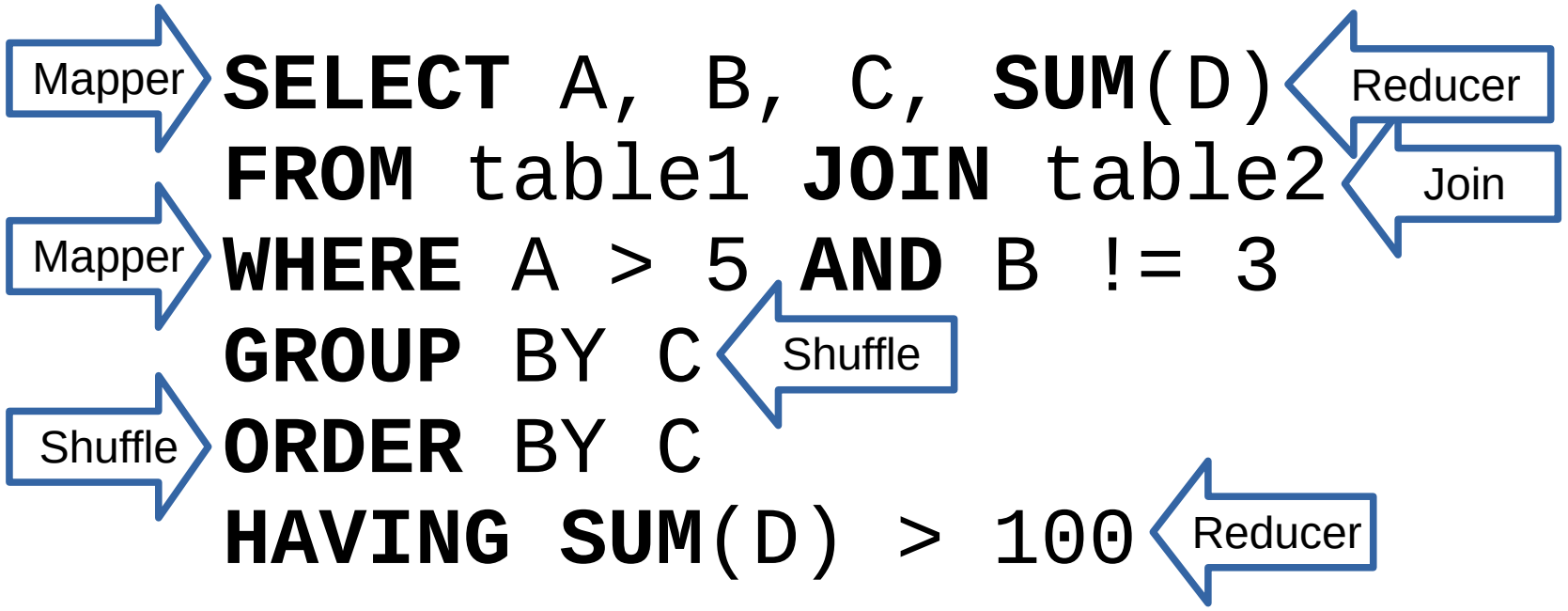


Joins in MapReduce: Fazit

- Sort-Merge-Join lässt sich direkt auf MapReduce abbilden
  - **Markieren** der Datenquellen im Mapper
  - **Sortieren** im Shuffle
  - **Merge** im Reducer

- Hash-Join möglich
- Sortierte Daten → Map-Side (Merge-)Join

SQL-Operationen in MapReduce

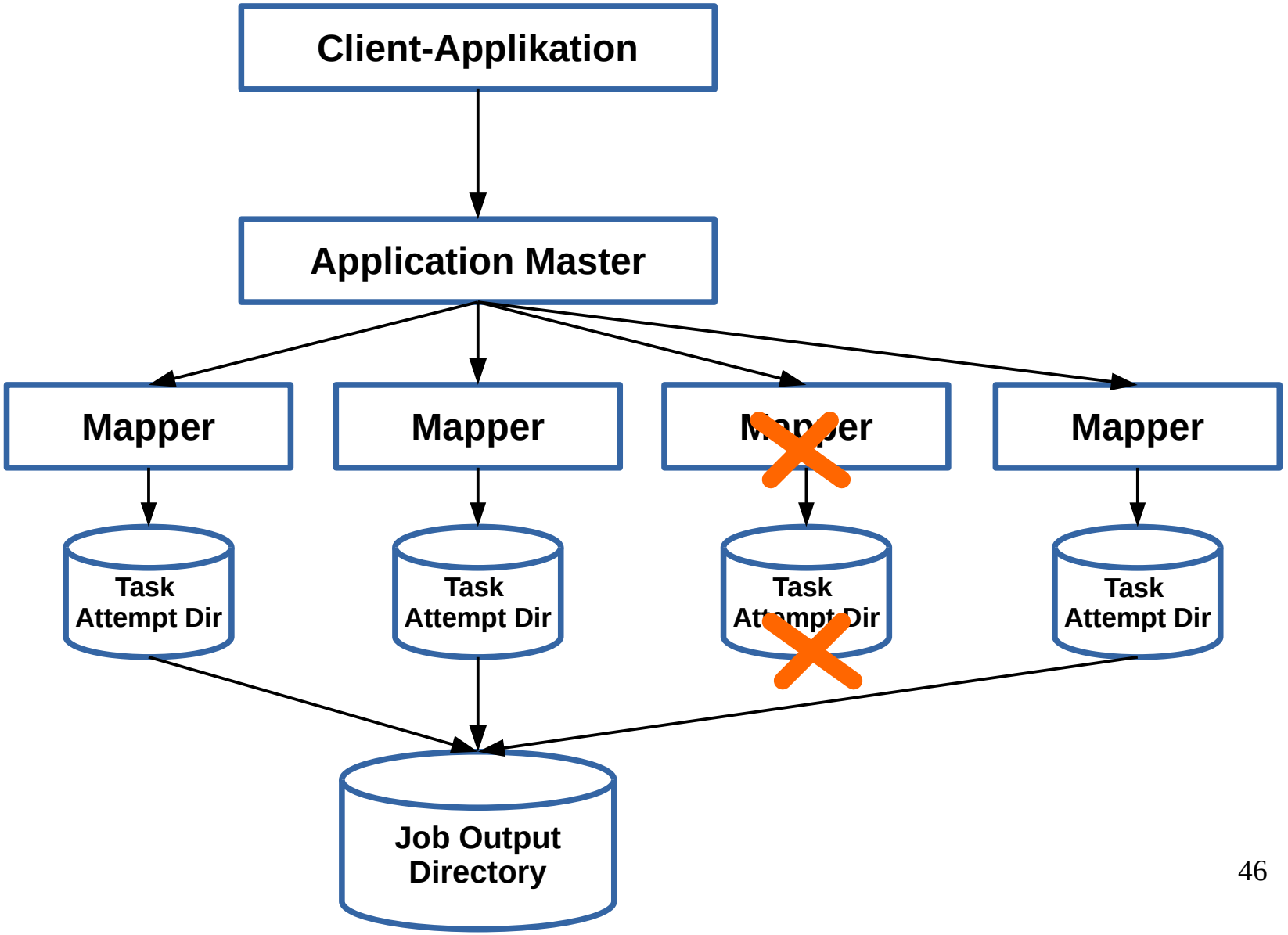


Ausführung von MR-Jobs

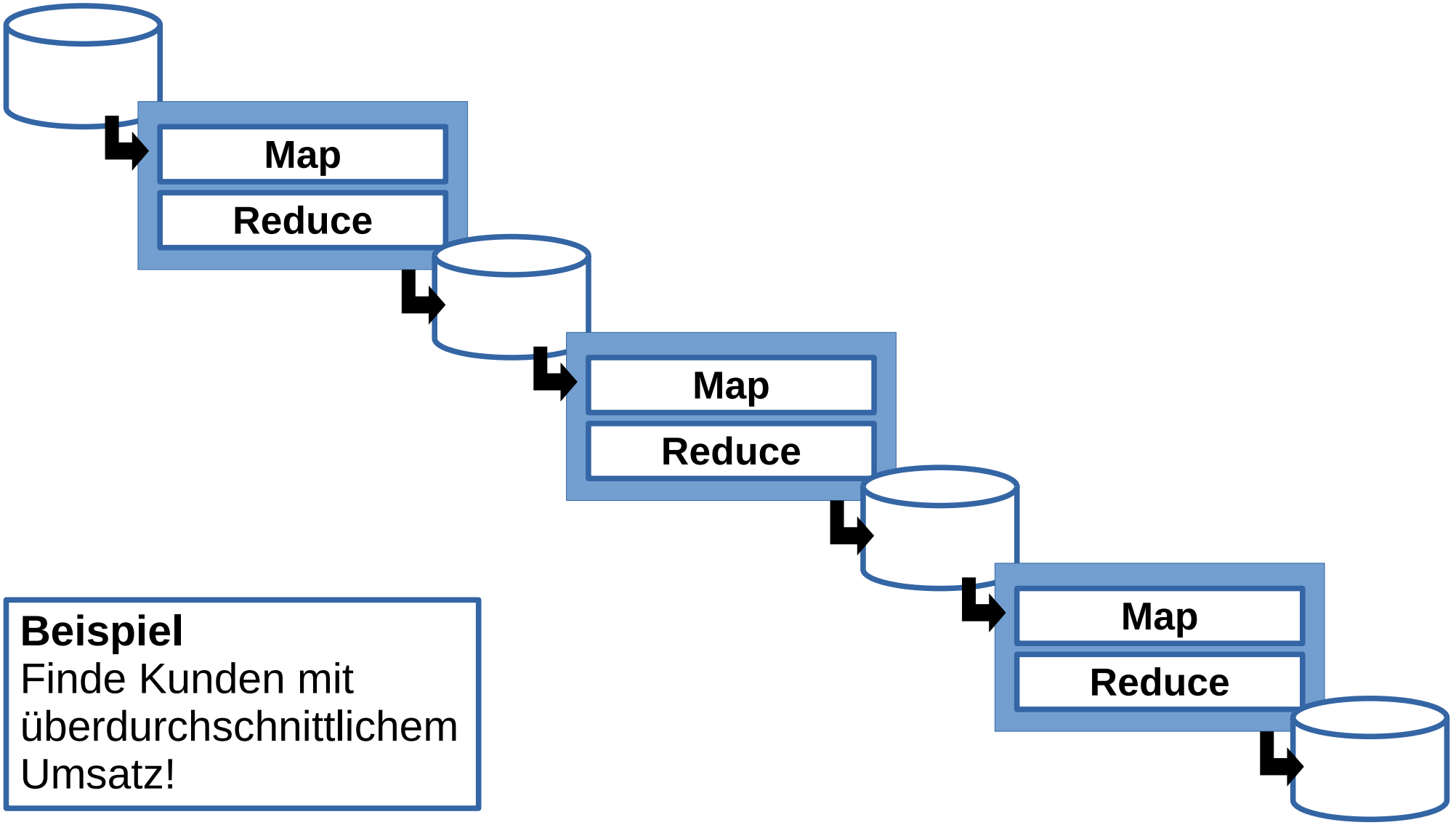
- **Job** = Map + Reduce auf gegebenen Daten
- Daten zerfallen in **Splits**
  - Dateien → HDFS-Blöcke
- Pro Split ein **Map-Task**
- **Shuffle**
- Konfigurierbare Anzahl von **Reduce-Tasks**

- **Anzahl Tasks:** $\approx n \cdot$ #CPU-Kerne
- Fragmentierung vermeiden!

Und wenn etwas schiefgeht?



Wenn MapReduce nicht reicht?



Beispiel
Finde Kunden mit überdurchschnittlichem Umsatz!

Doch noch verteilter Zustand?

- Verteilter Zustand (1): **Configuration**

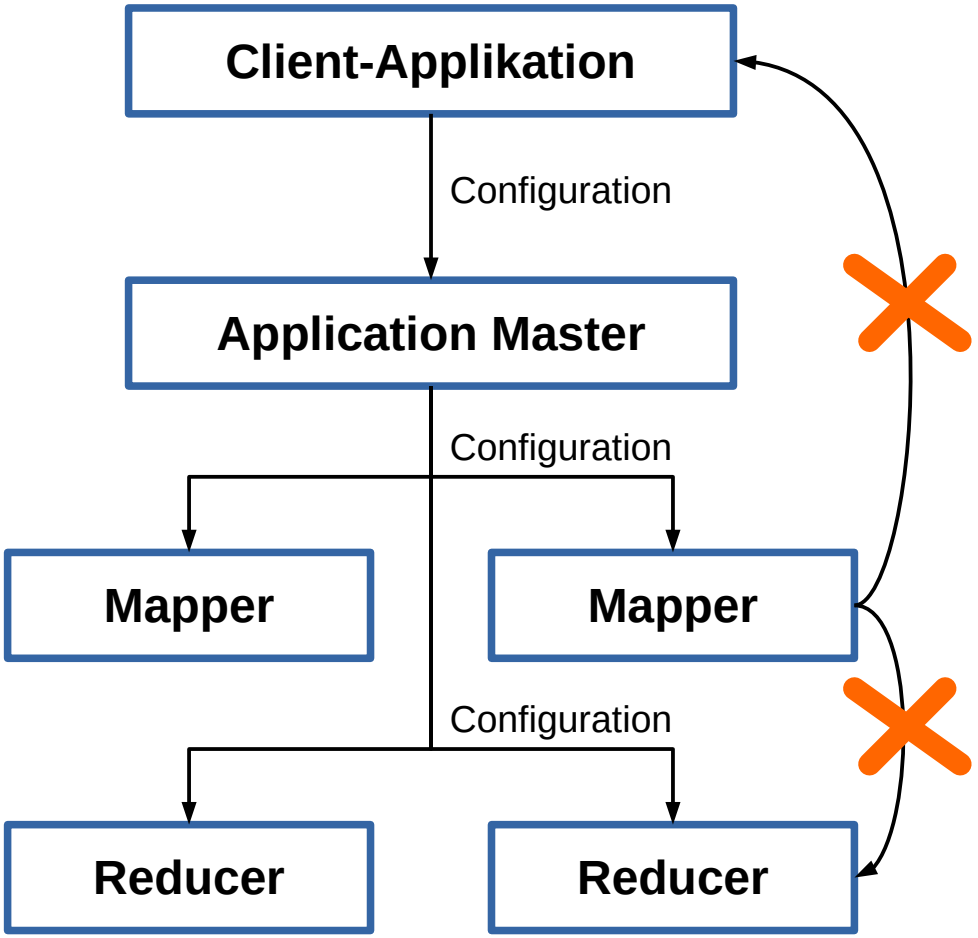
```
Configuration conf = getConf();
conf.set("my-parameter-1", "foobar");
conf.setInt("my-parameter-2", 4567);
Job job = Job.getInstance(conf, "my-job");
```

Client-Applikation

```
public class MyMapper extends Mapper<LongWritable, Text, Text, LongWritable> {
    private String myParameter1;
    private int myParameter2;

    @Override
    protected void setup(Context context)
            throws IOException, InterruptedException {
        Configuration conf = context.getConfiguration();
        myParameter1 = conf.get("my-parameter-1");
        myParameter2 = conf.getInt("my-parameter-2", -1);
    }

    protected void map(LongWritable key, Text value, Context context)
}
```

Mapper/Reducer/…

Configuration ist eine Einbahnstraße



Doch noch verteilter Zustand?

- Verteilter Zustand (2): **Counter**
  - Verteilte Zähler
  - Einzige Operation:

```
Counter myCounter = context.getCounter("my-group", "my-counter");
myCounter.increment(100);
```

- Ergebnis wird vom Application Master gesammelt
- Addition ist gut zu verteilen
  - Assoziativ
  - Kommutativ
  - (Inverse)

Doch noch verteilter Zustand?

Counter Group	Counters			
	Name	Map	Reduce	Total
File System Counters	FILE: Number of bytes read	0	0	0
	FILE: Number of bytes written	187,408	0	187,408
	FILE: Number of large read operations	0	0	0
	FILE: Number of read operations	0	0	0
	FILE: Number of write operations	0	0	0
	HDFS: Number of bytes read	4,286	0	4,286
	HDFS: Number of bytes written	8,101	0	8,101
	HDFS: Number of large read operations	0	0	0
	HDFS: Number of read operations	5	0	5
	HDFS: Number of write operations	2	0	2
	Name	Map	Reduce	Total
Job Counters	Launched map tasks	0	0	1
	Rack-local map tasks	0	0	1
	Total megabyte-milliseconds taken by all map tasks	0	0	10,148,864
	Total time spent by all map tasks (ms)	0	0	9,911
	Total time spent by all maps in occupied slots (ms)	0	0	9,911
	Total vcore-milliseconds taken by all map tasks	0	0	9,911
	Name	Map	Reduce	Total
Map-Reduce Framework	CPU time spent (ms)	1,060	0	1,060
File Input Format Counters				
File Output Format Counter				

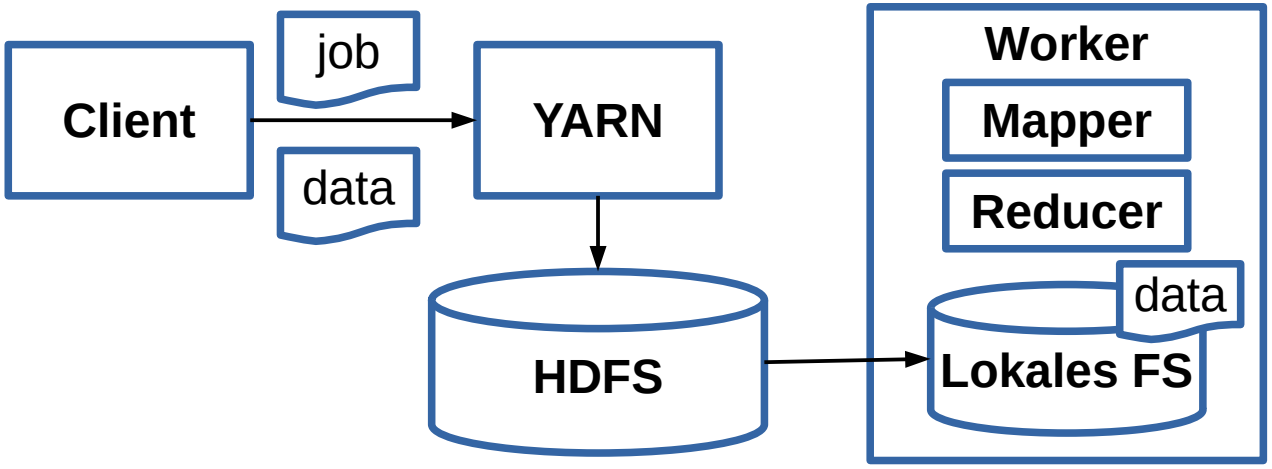
Name	Map
FILE: Number of bytes read	0
FILE: Number of bytes written	187,408
FILE: Number of large read operations	0
FILE: Number of read operations	0
FILE: Number of write operations	0
HDFS: Number of bytes read	4,286
HDFS: Number of bytes written	8,101
HDFS: Number of large read operations	0
HDFS: Number of read operations	5
HDFS: Number of write operations	2

Doch noch verteilter Zustand?

- Verteilter Zustand (3): **Distributed Cache**

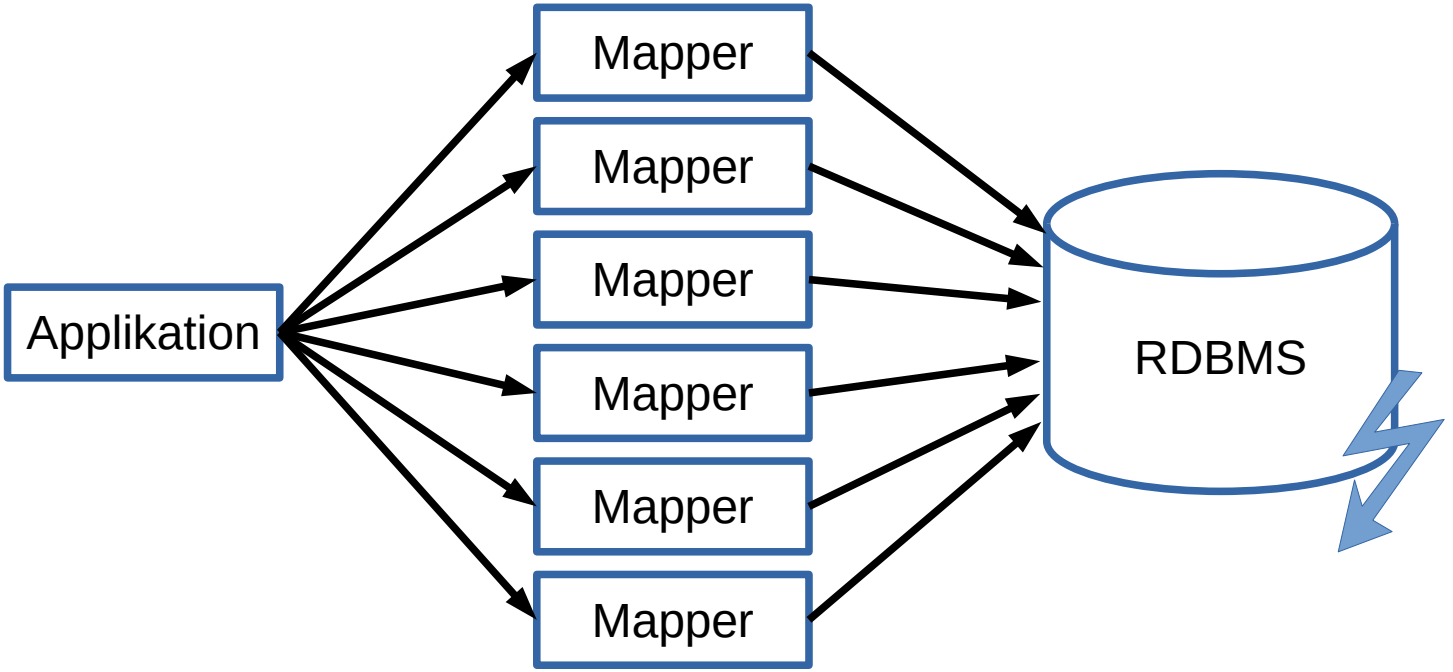
```
job.addCacheFile(new URI("file:///mapping-table.txt"));
job.addCacheFile(new URI("file:///complex-config.xml"));
```

- Anwendungen:
  - Komplexere Konfiguration
  - Kleine Datensätze



Doch noch verteilter Zustand?

- So nicht…!

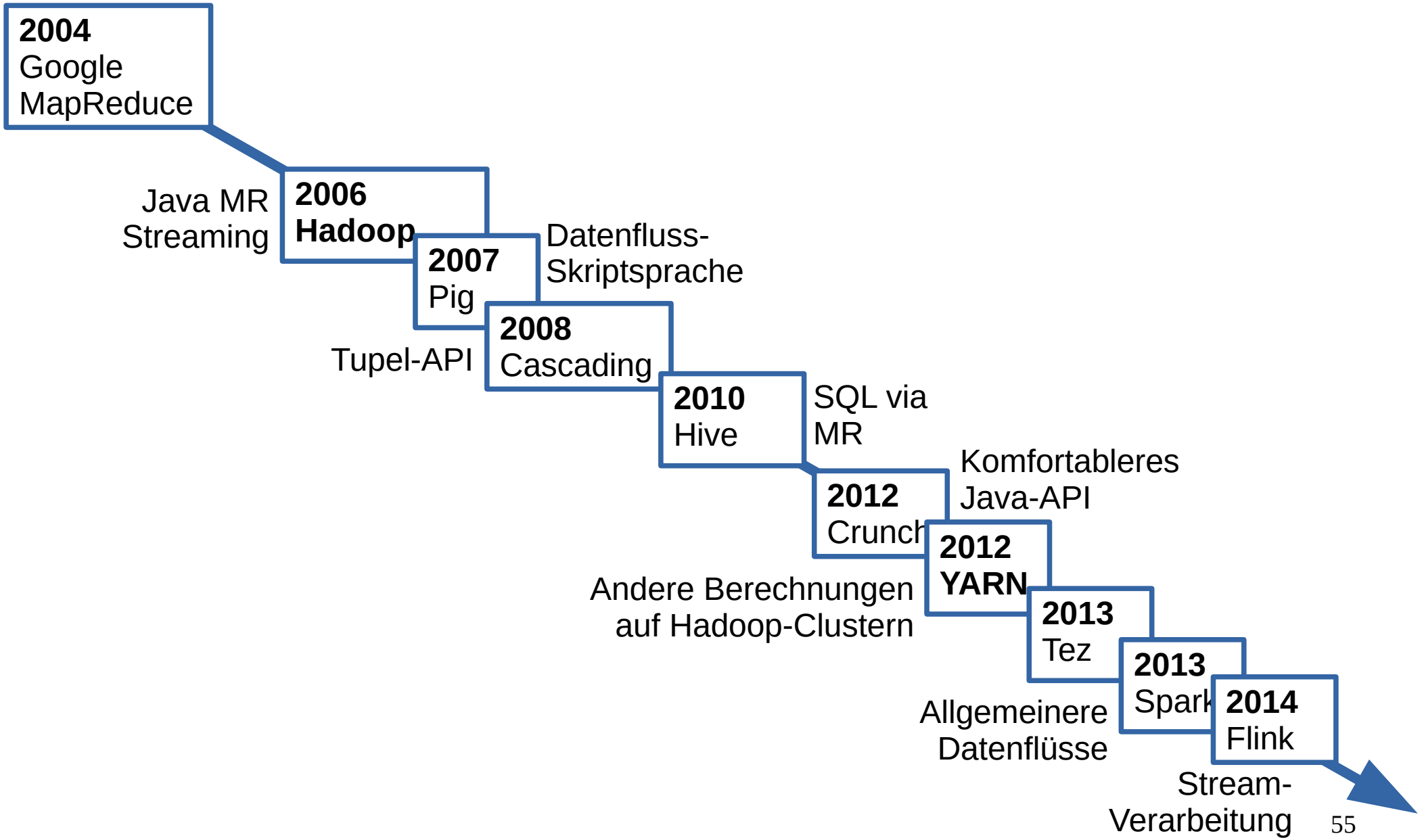


- Ausnahme: verteilte DBMS, die dafür gemacht sind

Zusammenfassung: Verteilter Zustand

- Verteilter Zustand ist immer noch böse!
- **Configuration**
  - Properties werden verteilt
  - Kein Rückkanal
- **Counter**
  - Nur Zähloperationen
  - Sammeln im Application Master
  - Keine Verteilung an andere Mapper/Reducer/…
- **Distributed Cache**
  - Verteilen von kleinen Datensätzen via HDFS

Historie/Konkurrenz



MapReduce: Pro und Contra

- ✔ Einfach verteilbar
- ✔ Hoher Durchsatz möglich
- ✔ > 10 Jahre Praxiserfahrungen
- ✔ Extrem robust

- ✘ Eingeschränktes Programmiermodell
- ✘ Nur zwei Operationen pro Job
- ✘ Viele persistierte Zwischenergebnisse

Vogelperspektive: Gutartige und bösartige Operationen

Filter
Map/Transform
Count

Group
Reduce
Unique

Sort
Join

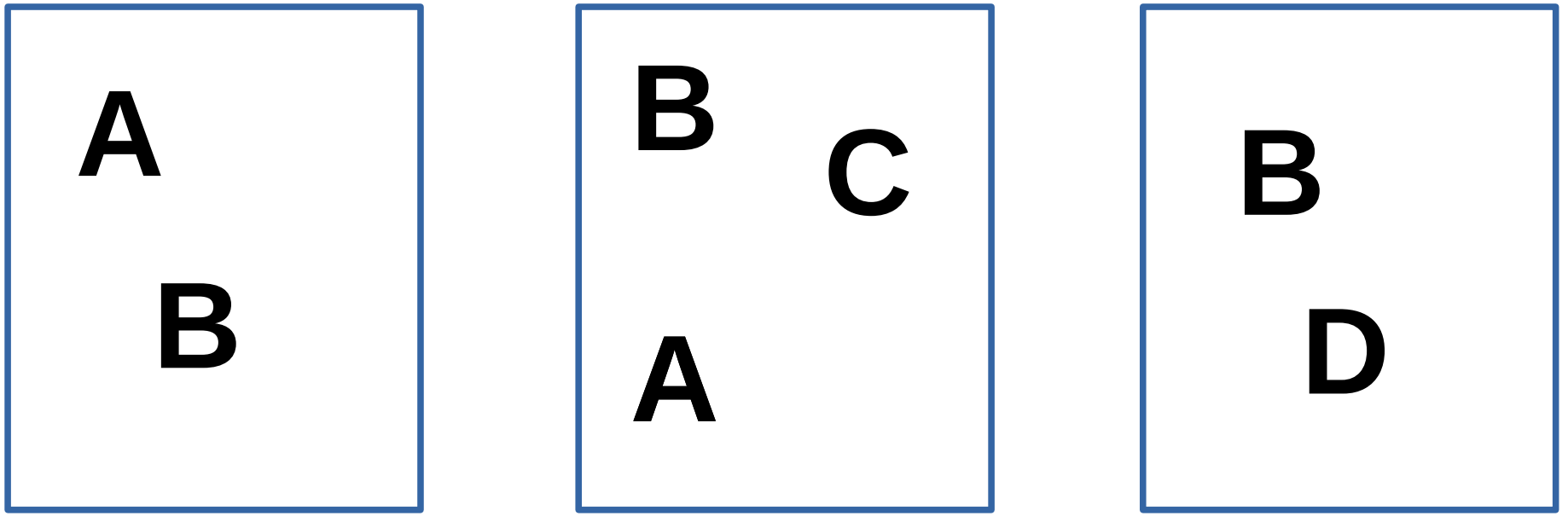
Count Distinct
Top-k
Quantile

Effizient
Leicht zu implementieren
Gut parallelisierbar

„Embarrassingly parallel“

Teuer
Umständlich zu implementieren
Schlecht parallelisierbar
Keine Zwischenergebnisse möglich

Ausblick: Kardinalität von Mengen („Count Distinct“)



$|M_1| = 2$ $|M_2| = 3$ $|M_3| = 2$

$$|M_1 \cup M_2 \cup M_3| = f(|M_1| \ldots$$

Wie viele User waren in den letzten 30 Tagen auf unserer Webseite?