

# Kapitel 10: Big Data

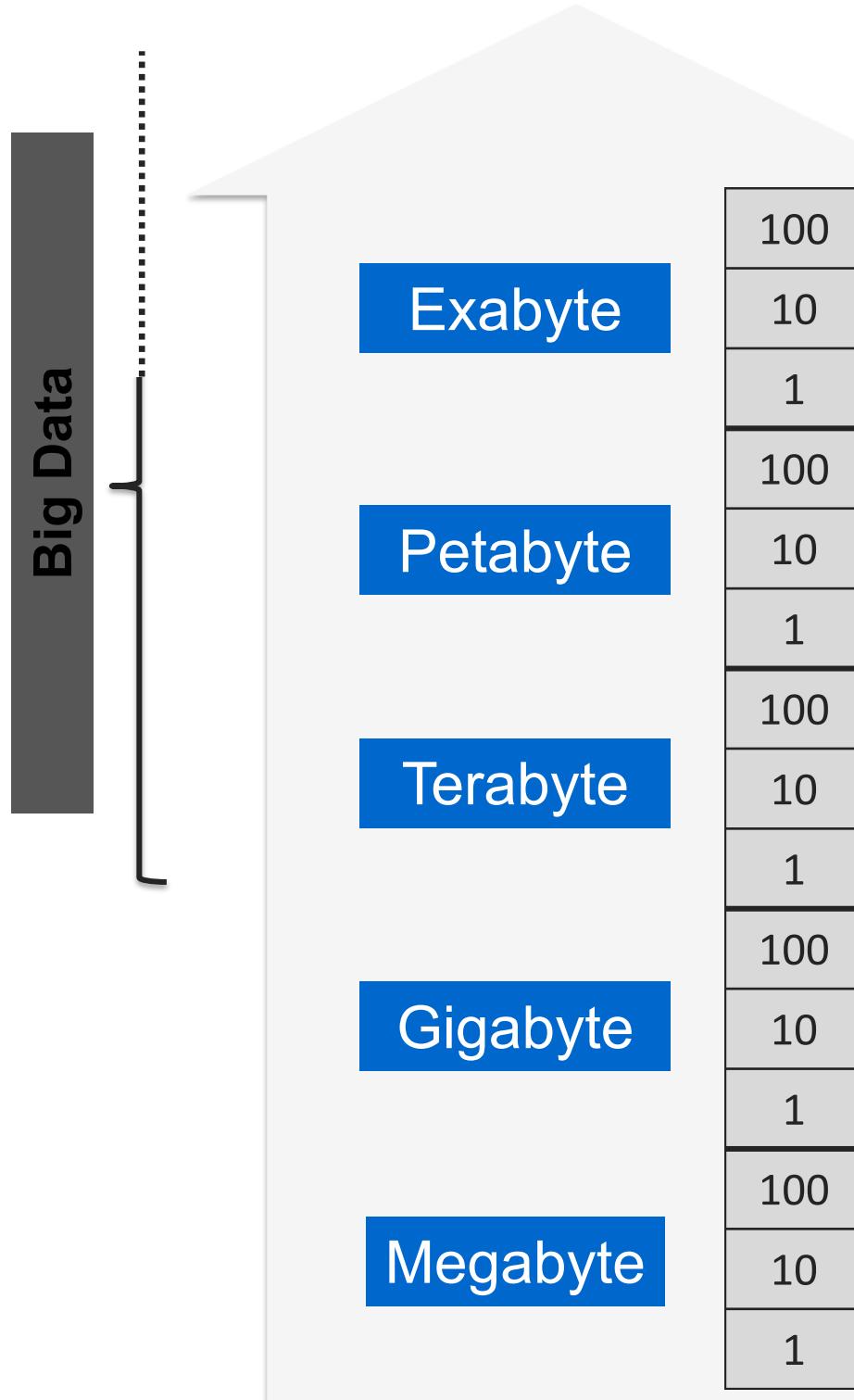
Vorlesung  
**CLOUD  
COMPUTING**

# Big Data

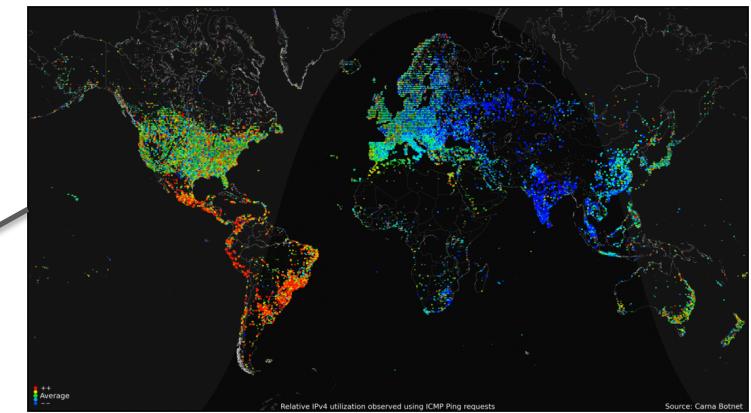
## Big Data

Verarbeitung großer Datenmengen durch:

- verteilte und hochgradig parallelisierte Verarbeitung.
- verteilte und effizient organisierte Datenablagen.



100
10
1
100
10
1
100
10
1
100



Das Internet



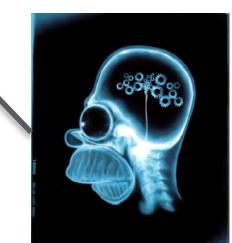
Facebook



Alle Schriften der Menschheit



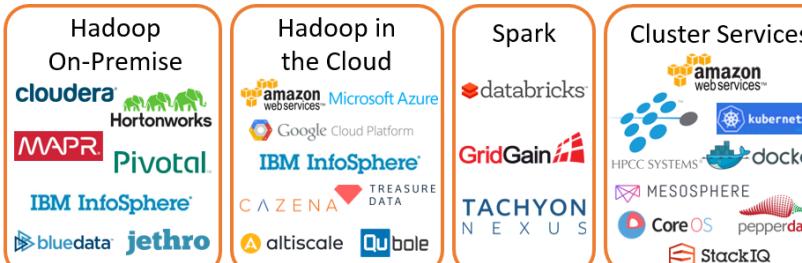
Der Google Index



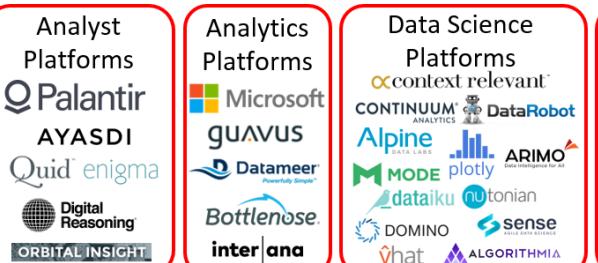
Speichervermögen des menschlichen Gehirns

## **Big Data Landscape 2016 (Version 3.0)**

## Infrastructure



Analytics



## Applications



Cross-Infrastructure/Analytics



Open Source



## Data Sources & APIs



# Wie verwalte und erschließe ich große Datenmengen?

**Die Cloud Computing Antwort:**  
Ich verteile sie auf viele Rechner  
in der Cloud und schaffe eine  
übergreifende  
Zugriffsschnittstelle.



# Große Datenmengen können effizient nur von parallelen Algorithmen verarbeitet werden.

Ein Algorithmus ist genau dann parallelisierbar, wenn er in einzelne Teile zerlegt werden kann, die keine Seiteneffekte zueinander haben.

- Funktioniert gut: Quicksort. Aufwand:  $O(n \log n) \rightarrow O(\log n)$

```
private void QuicksortParallel<T>(T[] arr, int left, int right)
where T : IComparable<T>
{
    if (right > left)
    {
        int pivot = Partition(arr, left, right);
        Parallel.Do(
            () => QuicksortParallel(arr, left, pivot - 1),
            () => QuicksortParallel(arr, pivot + 1, right));
    }
}
```

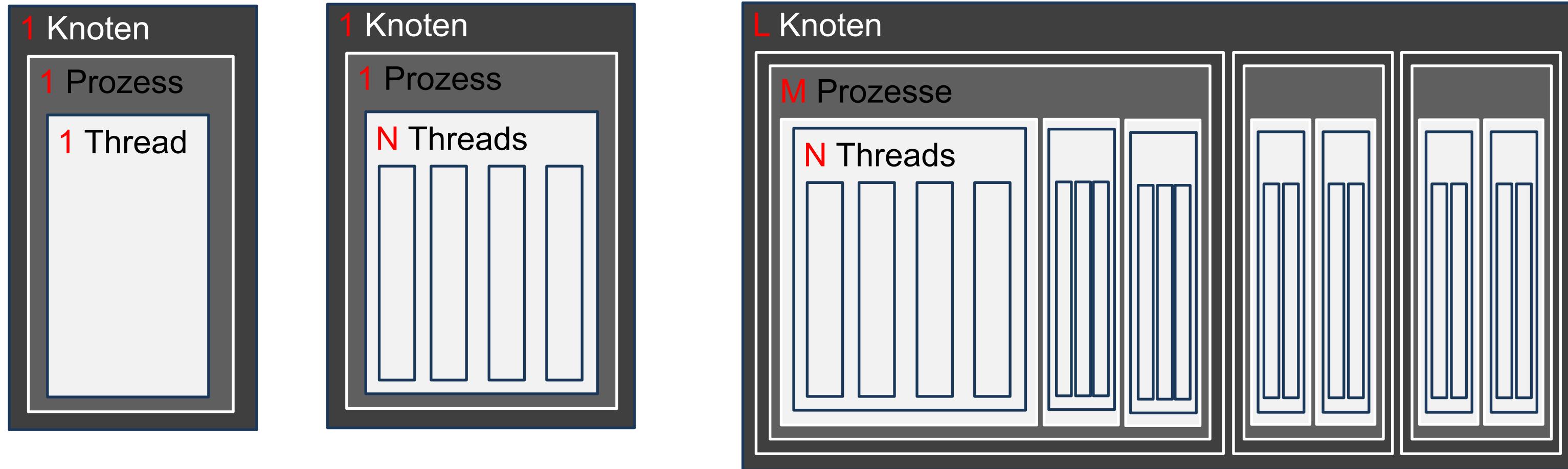
- Funktioniert nicht: Berechnung der Fibonacci-Folge ( $F_{k+2} = F_k + F_{k+1}$ ). Berechnung ist nicht parallelisierbar.

Ein paralleler Algorithmus (**Job**) ist aufgeteilt in sequenzielle Berechnungsschritte (**Tasks**), die parallel zueinander abgearbeitet werden können. Der Entwurf von parallelen Algorithmen folgt oft dem Teile-und-Herrsche Prinzip.

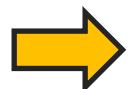
# Parallele Programmierung basiert oft auf funktionaler Programmierung.

- Ein funktionales Programm besteht (ausschließlich) aus Funktionen.
- Eine Funktion ist die Abbildung von Eingabedaten auf Ausgabedaten:  
 $f(E) \rightarrow A$   
Eine Funktion ändert die Eingabedaten dabei nicht.
- Funktionen sind idempotent:
  - Sie erzeugen neben den Ausgabedaten keine weiteren Seiteneffekte.  
→ Funktionen sind somit ideal parallelisierbar und zur Beschreibung von Tasks geeignet.
  - Sie erzeugen für die gleichen Eingabedaten auch stets die gleichen Ausgabedaten.  
→ Funktionen können im Fehlerfall stets neu ausgeführt werden. Parallel Verarbeitung ist aus technischen Gründen oft fehleranfällig. Damit kann eine Fehlertoleranz sichergestellt werden.

# Parallele Programmierung kann sowohl im Kleinen als auch im Großen betrieben werden.



Keine Parallelität



Parallelität im Kleinen

Vorteile im Vergleich:

- Höherer Durchsatz
- Bessere Auslastung der Hardware
- Vertikale Skalierung möglich



Parallelität im Großen

Vorteile im Vergleich:

- Höherer Durchsatz
- Horizontale Skalierung möglich (Scale Out).
- Keine hardwarebedingte Limitierung des Datenvolumens (→ Big Data ready).

# Big Data erfordert Parallelität im Großen. Die vier Paradigmen der Parallelität im Großen:



Folgt aus Datenmenge im Vergleich zur Programmgröße

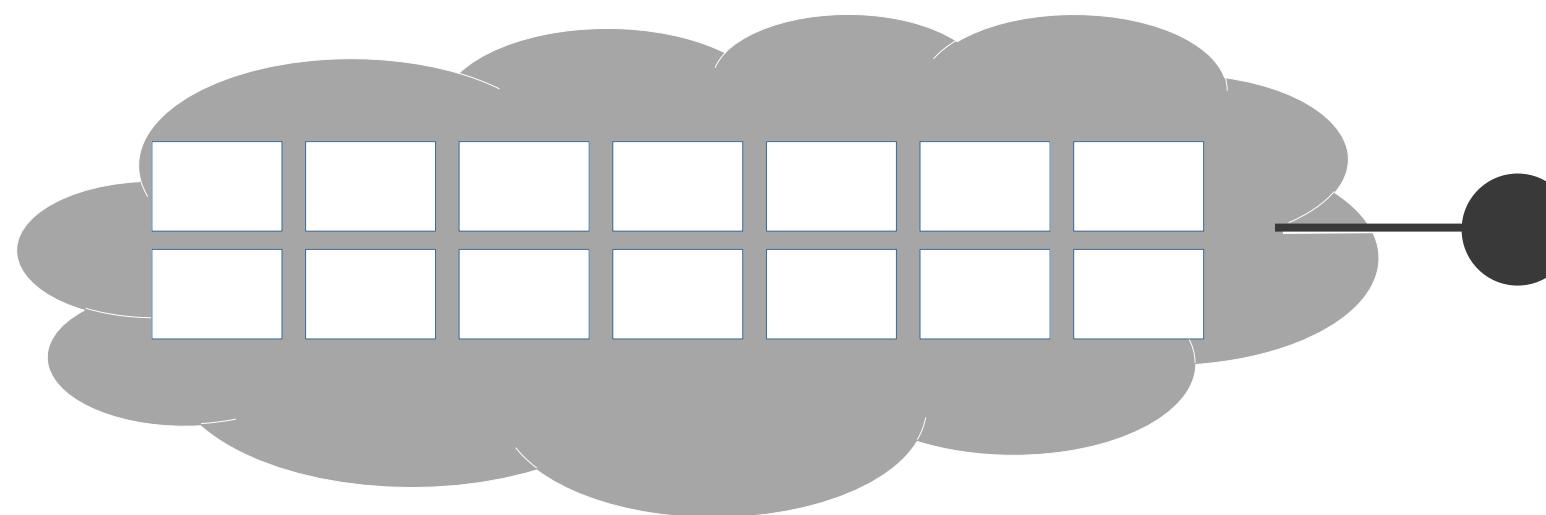
Das Grundprinzip von paralleler Verarbeitung.

Folgt aus Praxisanforderung:  
Viele Knoten bedeutet  
viele Ausfallmöglichkeiten

1. Die Logik folgt den Daten.
2. Falls Datentransfer notwendig, dann so schnell wie möglich:  
In-Memory vor lokaler Festplatte vor Remote-Transfer.
3. Parallelisierung über *Tasks* (seiteneffektfreie Funktionen) und *Jobs* (Ausführungsvorschrift für Tasks) sowie entsprechend partitionierter Daten (*Shards*).
4. Design for Failure: Ausführungsfehler als Standardfall ansehen. Fehler verzeihen und kompensieren.

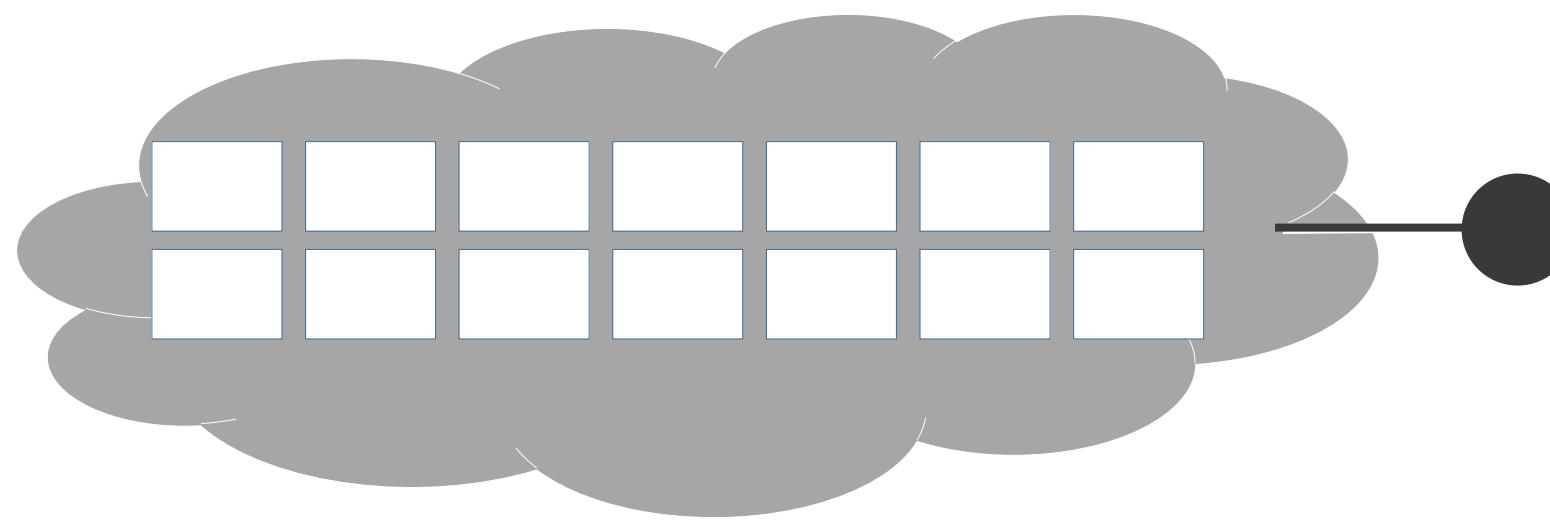
Folgt aus potenziell großer Datenmenge und Verarbeitungsgeschwindigkeit

# Welche Lösungen gibt es dafür im Cloud Computing?



- **Big Data Engines (low level)**
  - MapReduce
  - RDD (Resilient Distributed Dataset)
- **Big Data Datenbanken (high level)**
  - NoSQL Datenbanken
  - NewSQL Datenbanken (NoSQL + SQL)
- Verteilte Dateisysteme
- In-Memory Data Grids / Elastic Memory

# Welche Lösungen gibt es dafür im Cloud Computing?



- **Big Data Engines (low level)**
  - MapReduce
  - RDD (Resilient Distributed Dataset)
- **Big Data Datenbanken (high level)**
  - NoSQL Datenbanken
  - NewSQL Datenbanken (NoSQL + SQL)
- Verteilte Dateisysteme
- In-Memory Data Grids / Elastic Memory

# Die *map* und *reduce* Funktion.

- Die **map** Funktion: Transformation einer Menge von Datensätzen in eine Zwischendarstellung.  
Erzeugt aus einem Schlüssel und einem Wert eine Liste an Schlüssel-Wert-Paaren.

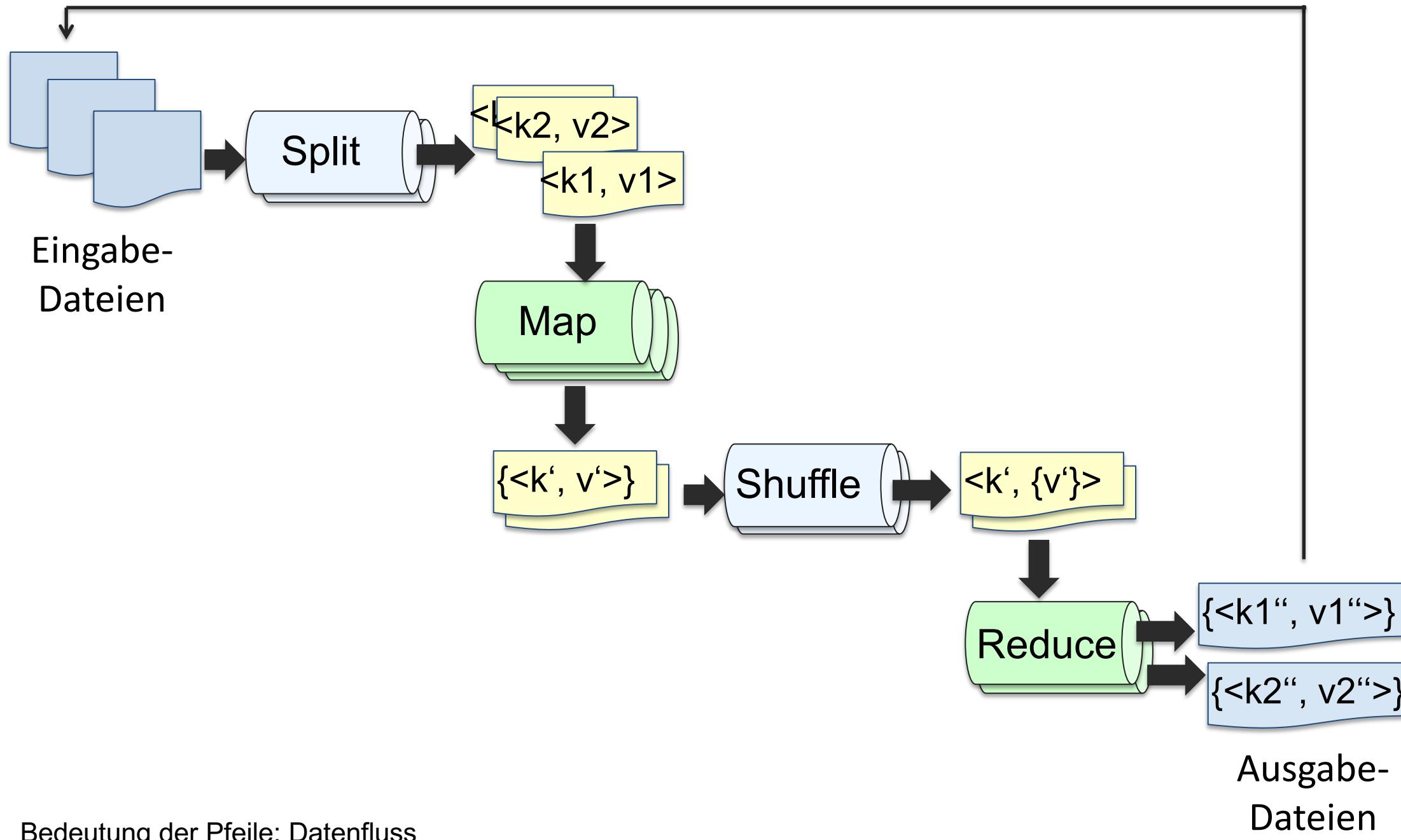
Signatur: **map**(k, v) → list(<k‘, v‘>)

- Die **reduce** Funktion: Reduktion der Zwischendarstellung auf das Endergebnis.  
Verarbeitet alle Werte mit gleichem Schlüssel zu einer Liste an Schlüssel-Wert-Paaren.

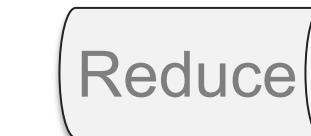
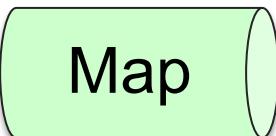
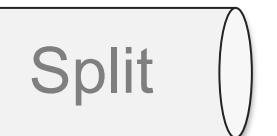
Signatur: **reduce**(k‘, list(v‘)) → list(<k‘‘, v‘‘>)

- Dabei soll gelten:  $|list(<k‘‘, v‘‘>)| \ll |list(<k‘, v‘>)|$

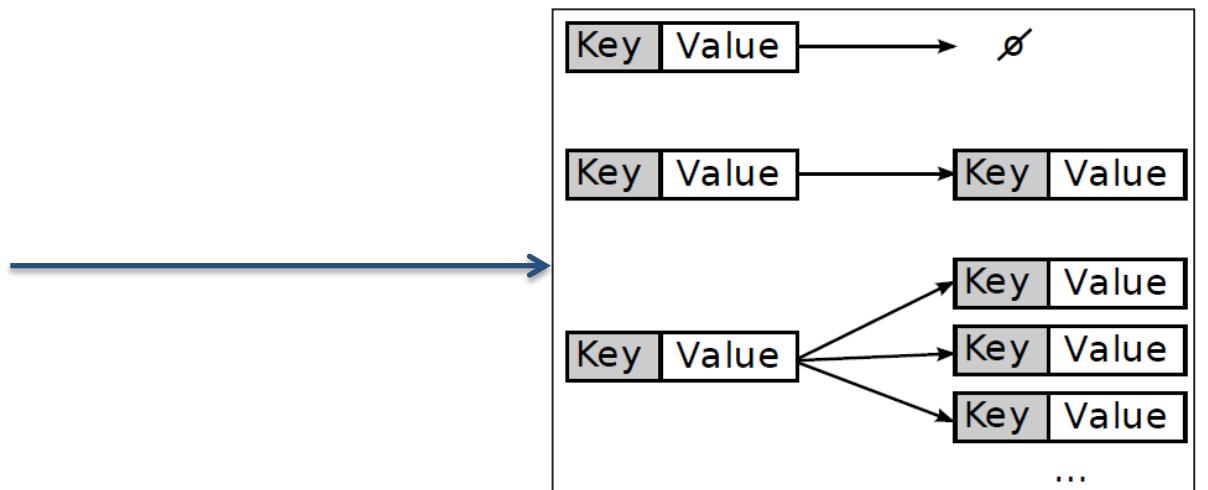
Programme werden in (mehrere) Map-Reduce-Zyklen aufgeteilt.  
Das Framework übernimmt die Parallelisierung.



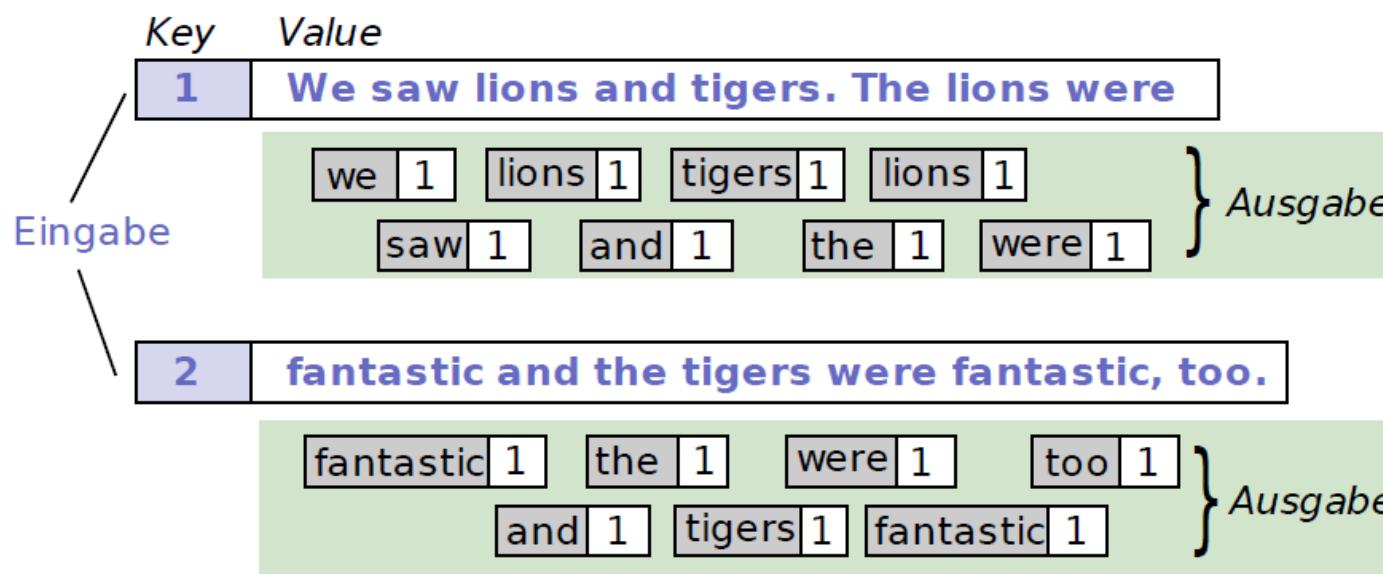
# Die Map-Phase



- Parallel Verarbeitung verschiedener Teilbereiche der Eingabedaten.
- Eingabedaten liegen in Form von Schlüssel/Wert-Paaren vor.
- Abbildung auf variable Anzahl von neuen Schlüssel/Wert-Paaren.  
Dabei sind alle Abbildungsvarianten zulässig:
- Beispiel: WordCount



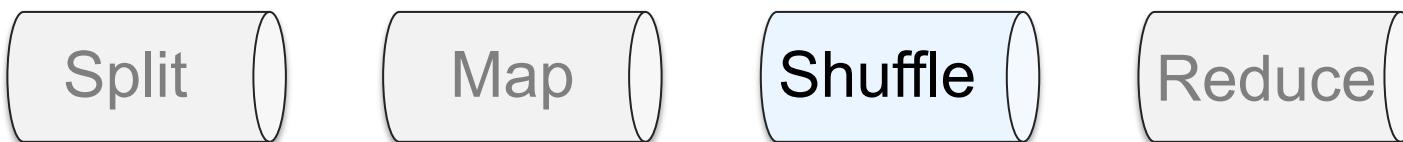
*Ein- und Ausgabe der Map-Phase:*



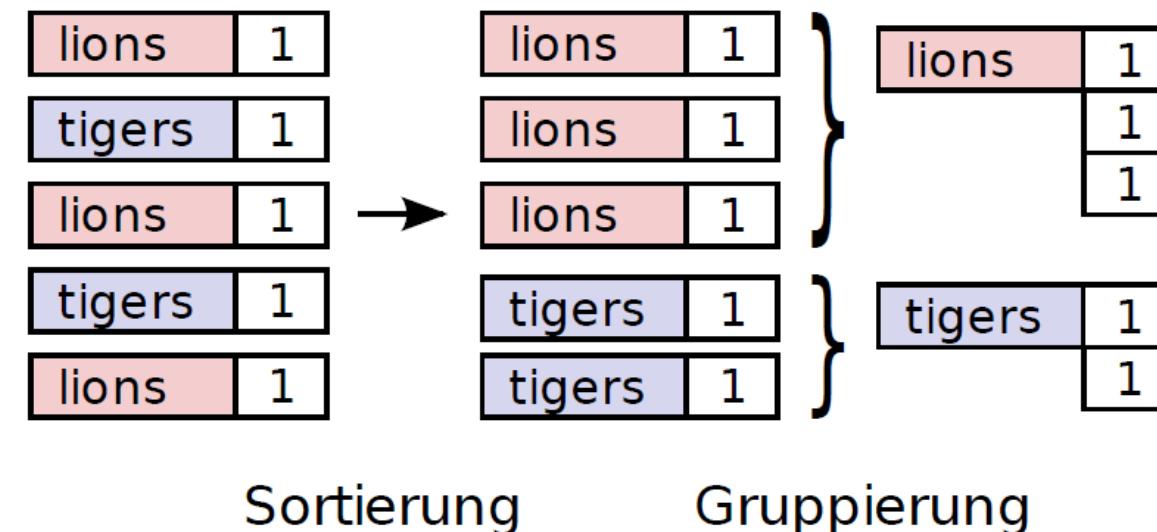
*Pseudocode Map-Phase:*

```
map(String key, String value):  
    //key: document name  
    //value: document contents  
    for each word in value:  
        EmitIntermediate(word, "1");
```

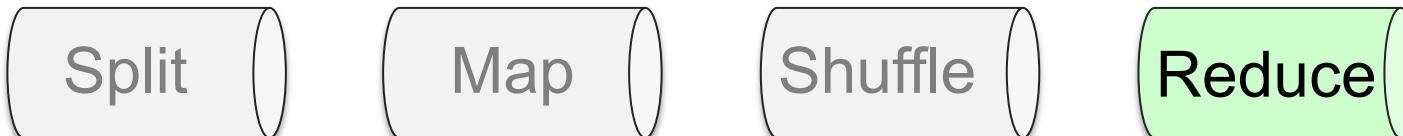
# Die Shuffle-Phase



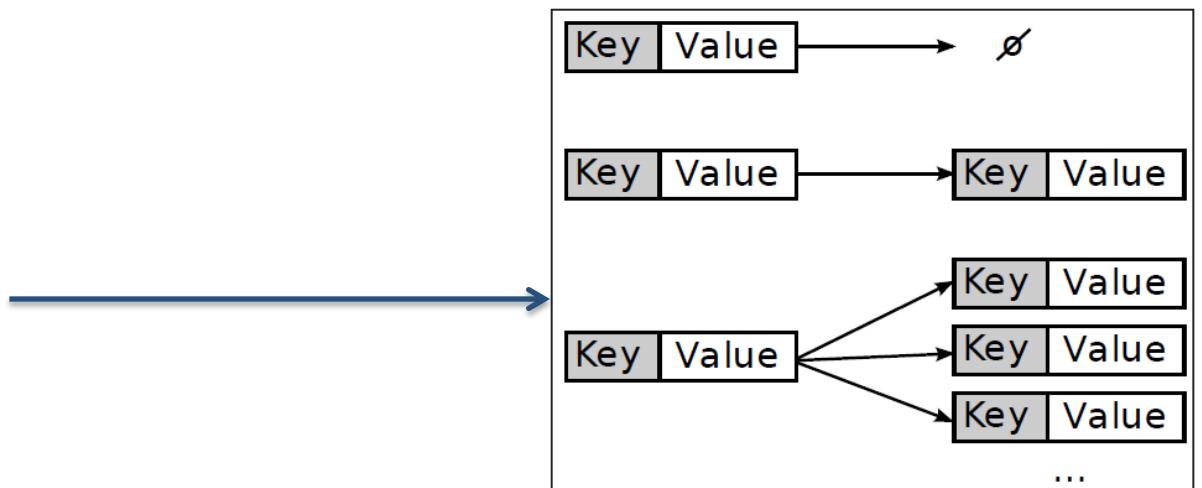
- Verarbeitung der Ergebnisse aus der Map-Phase.
- Ausgaben aus der Map-Phase werden entsprechend ihrem Schlüssel sortiert und gruppiert.
- Im Standard-Fall ist die Shuffle-Phase nicht parallelisiert.
- Sie kann jedoch mittels einer Vor-Sortierung in der Map-Phase über eine Partitionierungsfunktion (z.B. Hash) auf den Schlüssel parallelisiert werden.



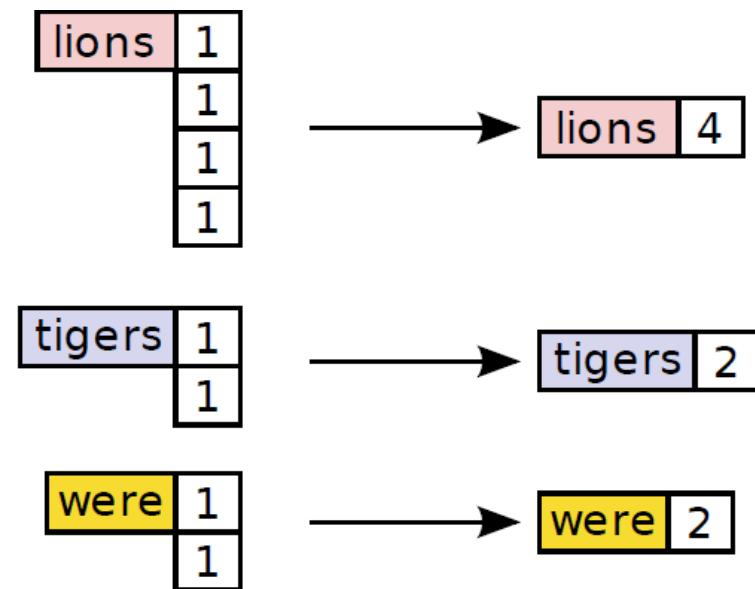
# Die Reduce-Phase



- Parallel Verarbeitung von Ergebnis-Gruppen aus der Map-Phase.  
Es wird pro Reduce-Vorgang genau eine dieser Gruppen verarbeitet.
- Eingabedaten liegen in Form von Schlüssel-Wertlisten vor.
- Abbildung auf variable Anzahl an Schlüssel/Wert-Paaren.  
Dabei sind alle Abbildungsvarianten zulässig:



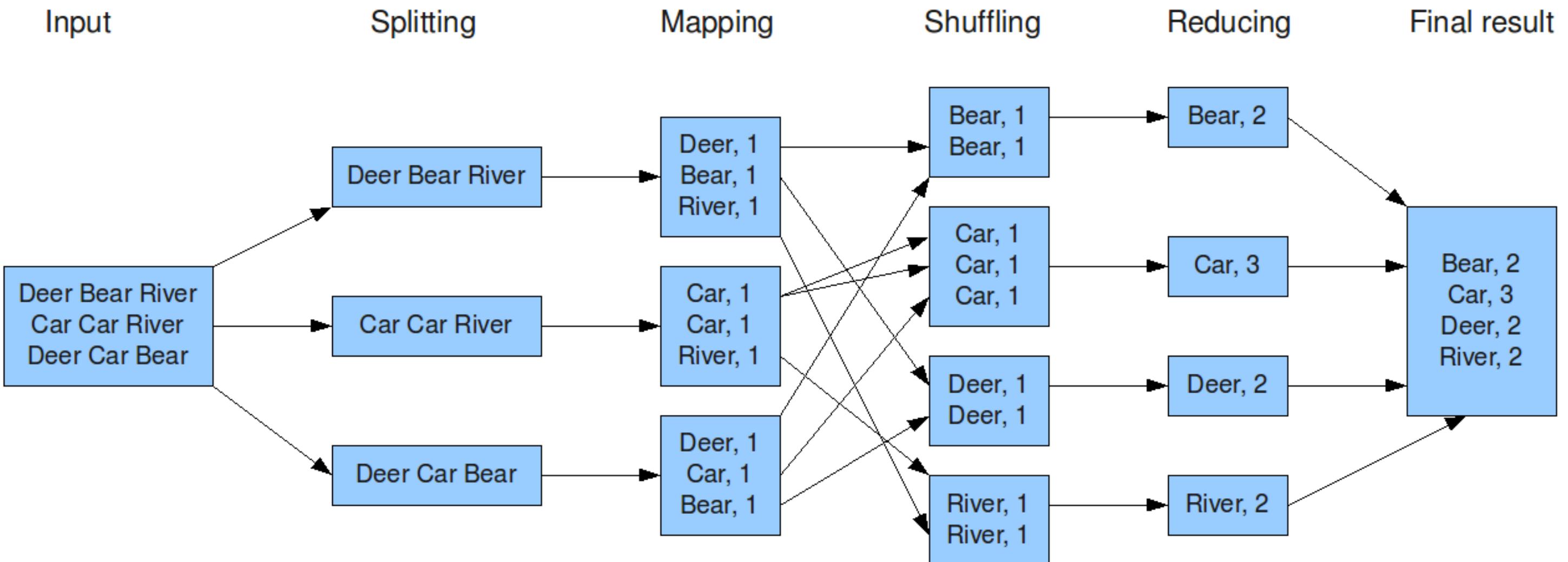
*Ein- und Ausgabe der Reduce-Phase:*



*Pseudocode Reduce-Phase:*

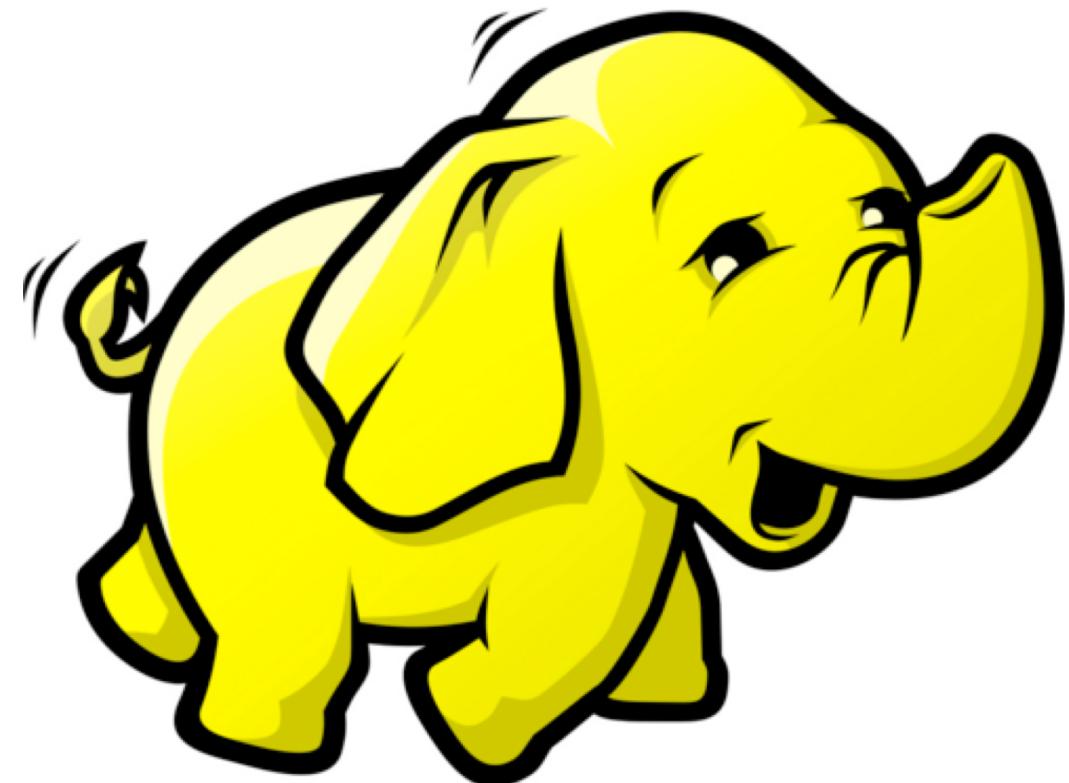
```
reduce(String key, Iterator values):  
    //key: a word  
    //values: a list of counts  
    for each value in values:  
        result += ParseInt(value);  
    Emit(AsString(Key +“, “+result));
```

# Übersicht über alle Phasen



<http://blog.jteam.nl/2009/08/04/introduction-to-hadoop>

# Hadoop



**„Open source platform for  
reliable, scalable, distributed  
computing.”**

# Apache Hadoop

- 2005 implementierte Doug Cutting MapReduce für Nutch (<http://nutch.apache.org>).

Nutch ist eine Open Source Suchmaschine, geschrieben in Java.



- Aus Nutch heraus wurde dann das Projekt Hadoop (<http://hadoop.apache.org>) extrahiert.

Es wurde als Open Source Implementierung des von Google beschriebenen MapReduce-Konzepts entwickelt.  
Die Google-Implementierung ist nicht veröffentlicht.

**„Open source platform for reliable, scalable, distributed computing.“**

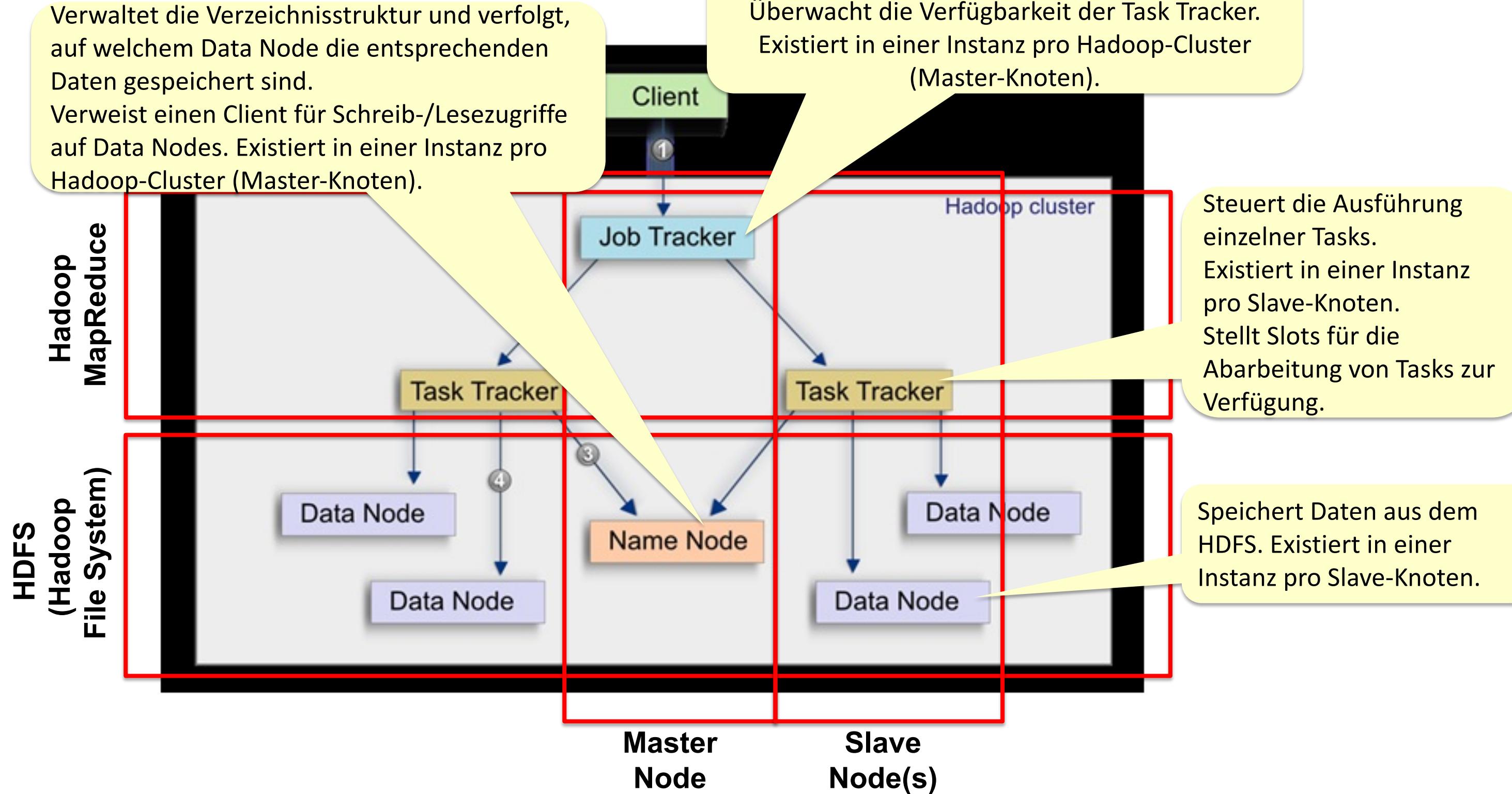
- Hadoop besteht aus zwei wesentlichen Bausteinen:

- Einer Implementierung des Google File Systems (GFS), genannt Hadoop File System (HDFS),
  - sowie einem MapReduce-Framework.

- Seit 2008 ist Hadoop ein Top-Level-Projekt der Apache Software Foundation. Im Juli 2009 hat ein Hadoop-Cluster von Yahoo 100 Terabyte in 2 Stunden und 53 Minuten sortiert (<http://sortbenchmark.org>)



# Die Hadoop Architektur



# Ein Map Task wird in Hadoop über die Schnittstelle Mapper implementiert.

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {  
    void map (KEYIN key, VALUEIN value, Context context){  
        context.write((KEYOUT) key, (VALUEOUT) value);  
    }  
}
```

- Eingabe- und Ausgabe-Datentypen werden mittels Generics an den Mapper gebunden.
- Schlüssel-Typen müssen dabei **WritableComparable** und Wert-Typen **Writable** implementieren. Hadoop stellt eine Reihe an Standard-Datentypen zur Verfügung, die diese Schnittstellen implementieren. Die Java-Standard-Typen sind hier nicht einsetzbar.
- Das Splitting und die De-Serialisierung der Eingabedaten, sowie die Serialisierung und Partitionierung der Ausgabedaten erfolgt „by magic“ im MapReduce Framework. Das Verhalten kann jedoch über Implementierung entsprechender Schnittstellen angepasst werden.
- Über das übergebene **Context**-Objekt können die Zwischenergebnisse übermittelt werden.

# Ein Reduce Task wird in Hadoop über die Schnittstelle Reducer implementiert.

```
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {  
    void reduce (KEYIN key, Iterable<VALUEIN> values, Context context){  
        for (VALUEIN value : values) {  
            context.write((KEYOUT) key, (VALUEOUT) value);  
        }  
    }  
}
```

- Eingabe- und Ausgabe-Datentypen werden analog zum Mapper über Generics gebunden. Es gelten dabei die selben Regeln.
- Die Bereitstellung der Eingabedaten inkl. Sortierung und Gruppierung sowie die Serialisierung der Ausgabedaten erfolgt im MapReduce Framework „by magic“. Das Verhalten kann jedoch über Implementierung entsprechender Schnittstellen angepasst werden.
- Über das übergebene **Context**-Objekt können die Endergebnisse übermittelt werden.

# Na dann ist doch alles Prima?

## ■ Kritik?

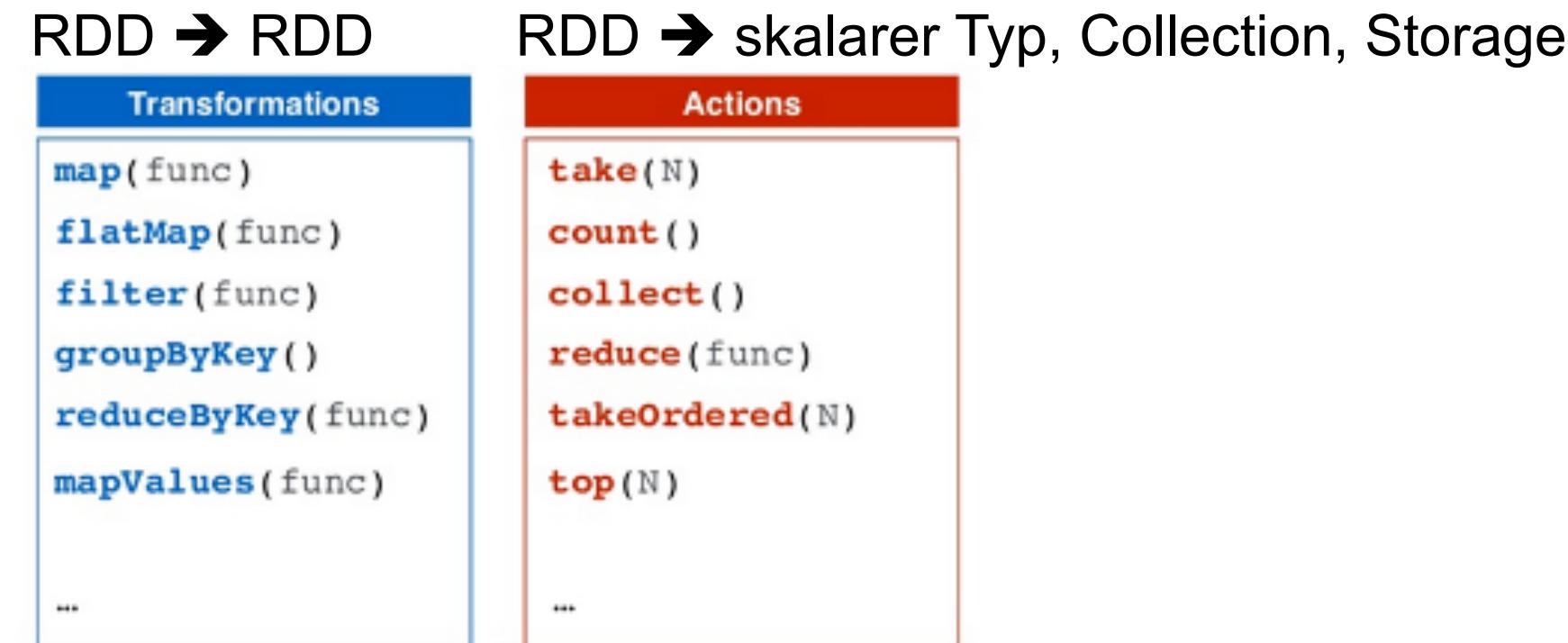
- Alles ist Map -> Reduce -> Map -> Reduce.
- API ist ziemlich Low-Level.
- Kein Schema.
- Keine Möglichkeit einen Zwischenstand mehrfach zu Verarbeiten.
- ...

# Apache Spark

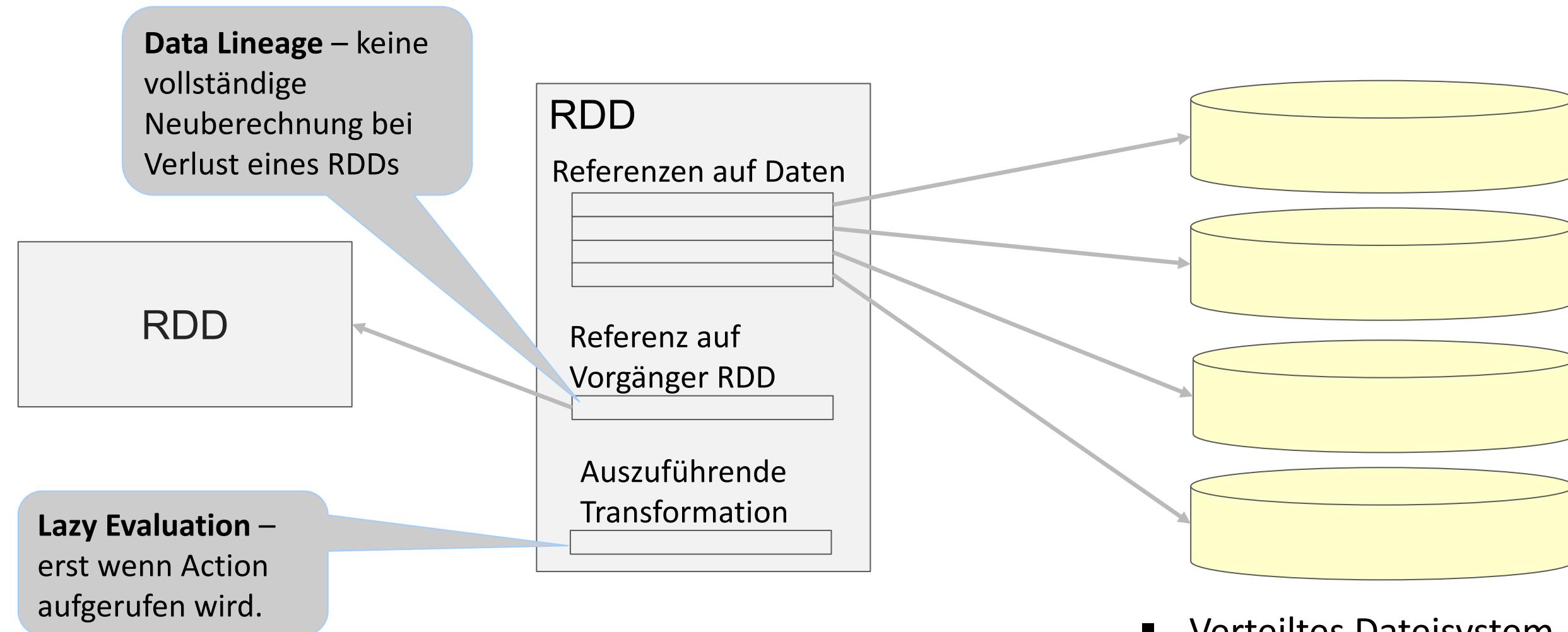


# Die Resilient Distributed Dataset (RDD) Datenstruktur

Eine RDD ist in der Außensicht ein klassischer Collection-Typ mit Transformations- und Aktionsmethoden.



# Die Anatomie eines RDDs.



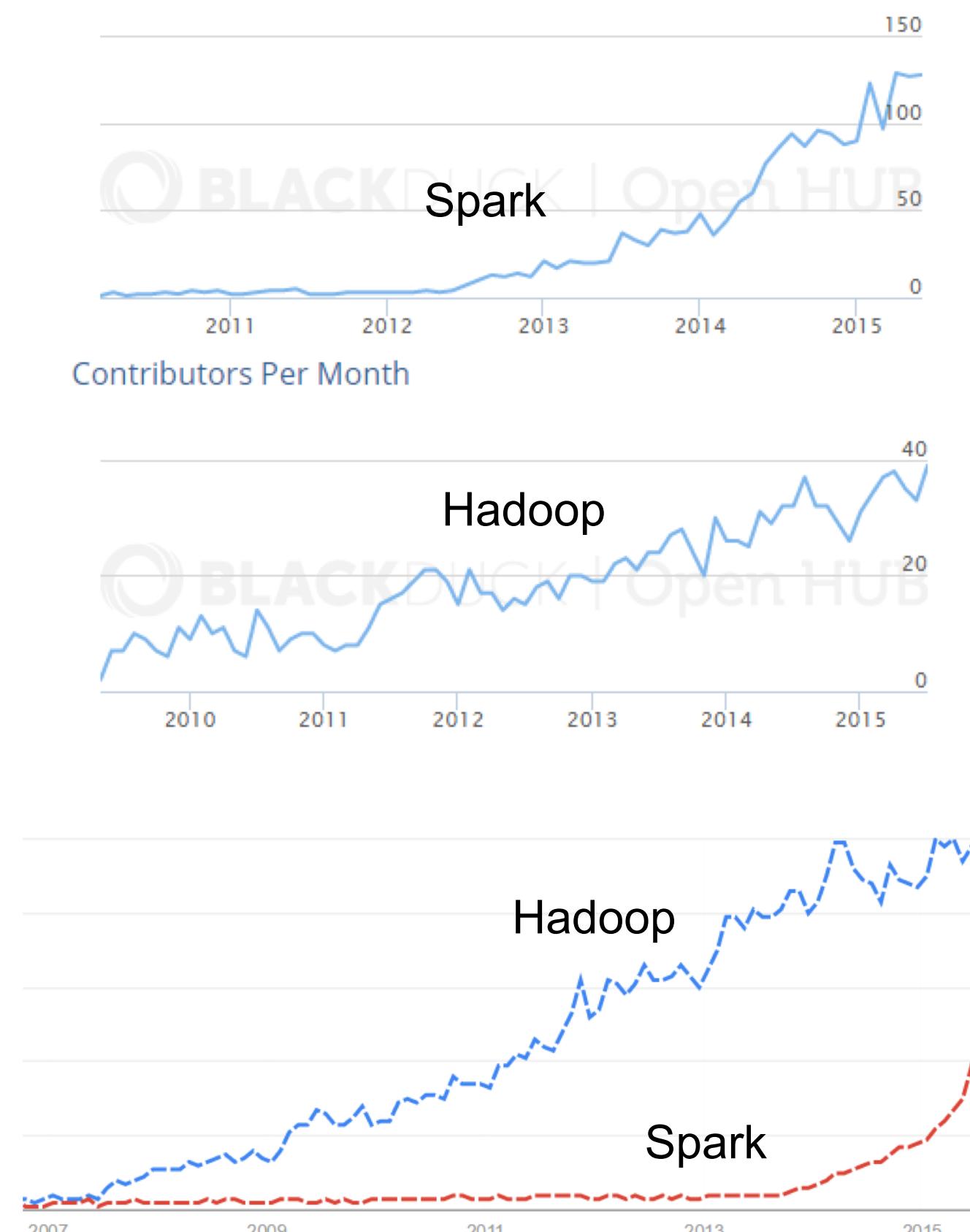
- Verteiltes Dateisystem
- Hauptspeicher
- Beliebiger anderer Datenspeicher

# Spark läuft Hadoop aktuell deutlich den Rang ab.

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	<b>1.42 TB/min</b>	<b>4.27 TB/min</b>	<b>4.27 TB/min</b>
Sort rate/node	<b>0.67 GB/min</b>	<b>20.7 GB/min</b>	<b>22.5 GB/min</b>

<http://sortbenchmark.org>

Contributors Per Month



# Daten verarbeiten: Mehr als Map und Reduce.

## Filter

```
val numAs = logData.filter(line => line.contains("a")).count()  
val numBs = logData.filter(line => line.contains("b")).count()  
val numABs = logData.filter(line => line.contains("a"))  
            .filter(line => line.contains("b")).count()
```

## Map

```
val lengths = logData.map(line => line.length)
```

## Reduce

```
val maxLength = lengths.reduce(Math.max)
```

## Sort

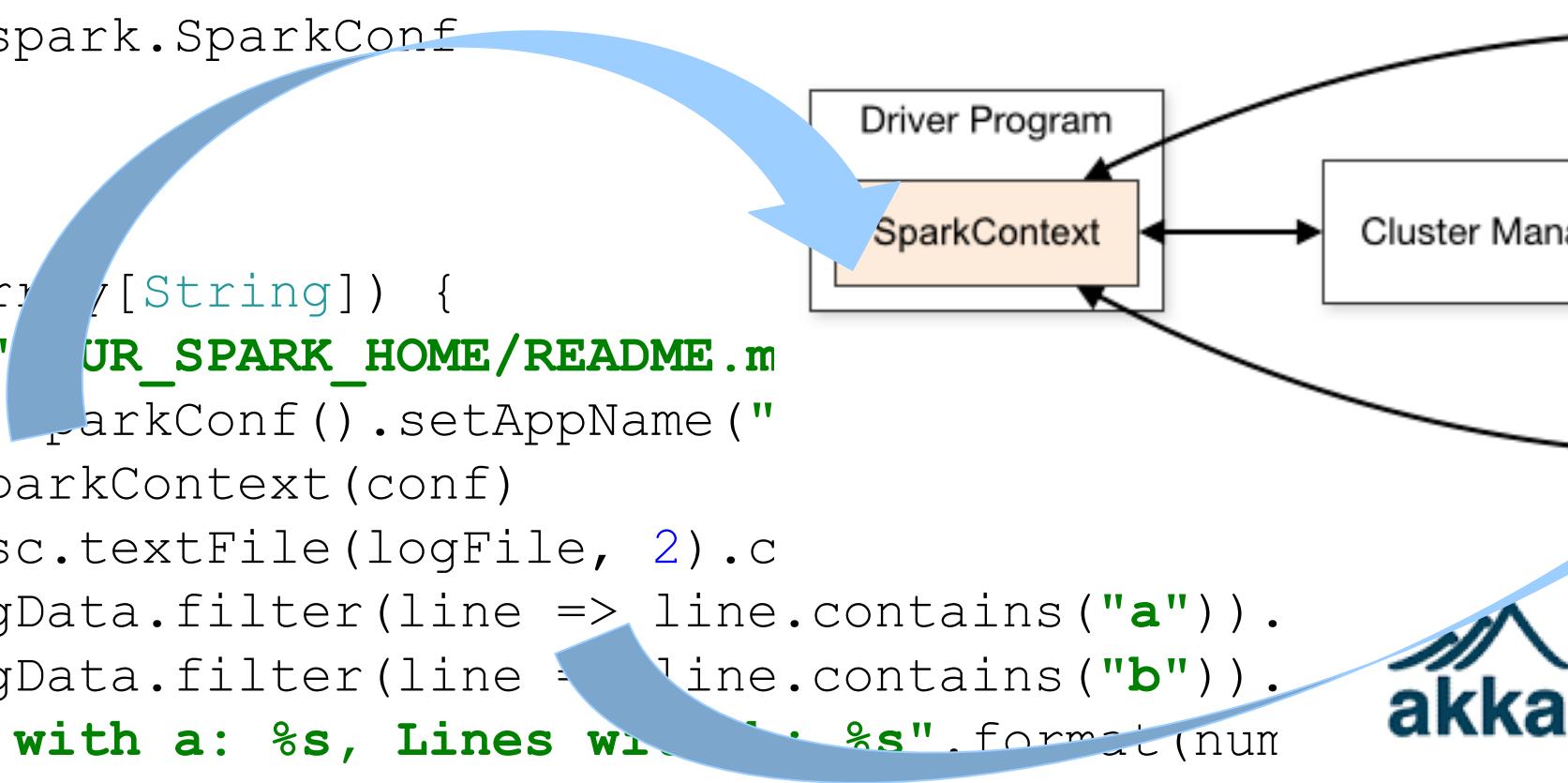
```
val sorted = logData.sortBy(l => l.length)
```

Transformations	Actions
map(func)	take(N)
flatMap(func)	count()
filter(func)	collect()
groupByKey()	reduce(func)
reduceByKey(func)	takeOrdered(N)
mapValues(func)	top(N)
...	...

# Wie funktioniert das?

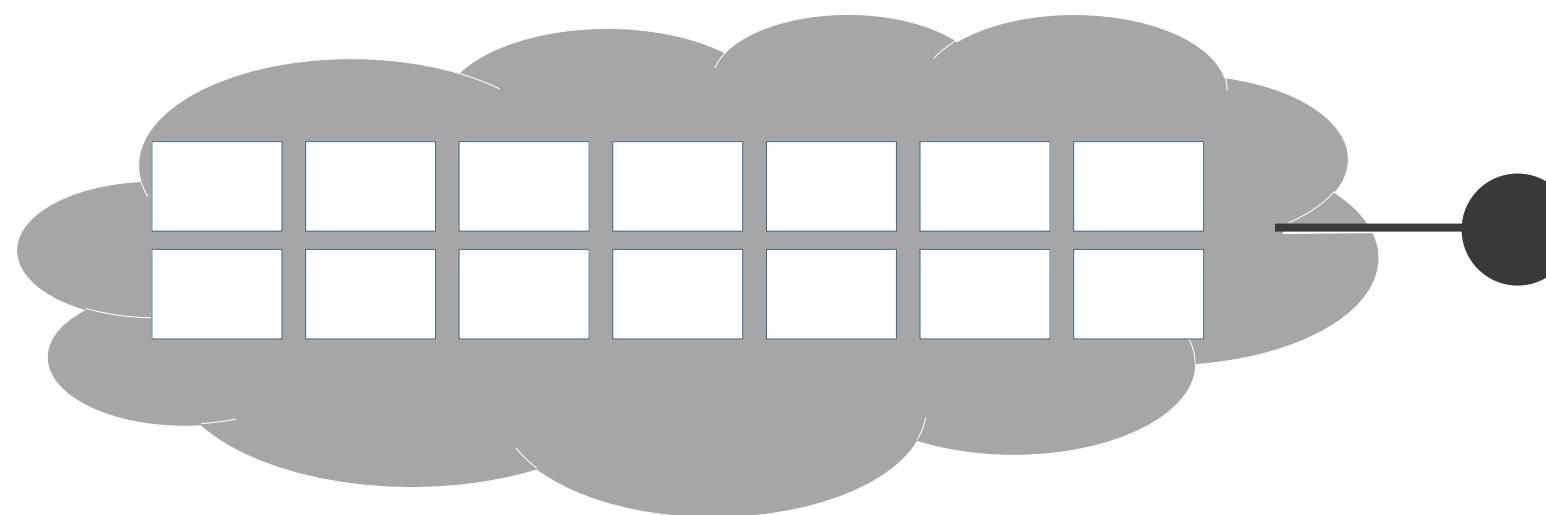
```
/* SimpleApp.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf

object SimpleApp {
  def main(args: Array[String]) {
    val logfile = "YOUR_SPARK_HOME/README.md"
    val conf = new SparkConf().setAppName("Simple App")
    val sc = new SparkContext(conf)
    val logData = sc.textFile(logfile, 2).c
    val numAs = logData.filter(line => line.contains("a")).rdd.count()
    val numBs = logData.filter(line => line.contains("b")).rdd.count()
    println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))
  }
}
```



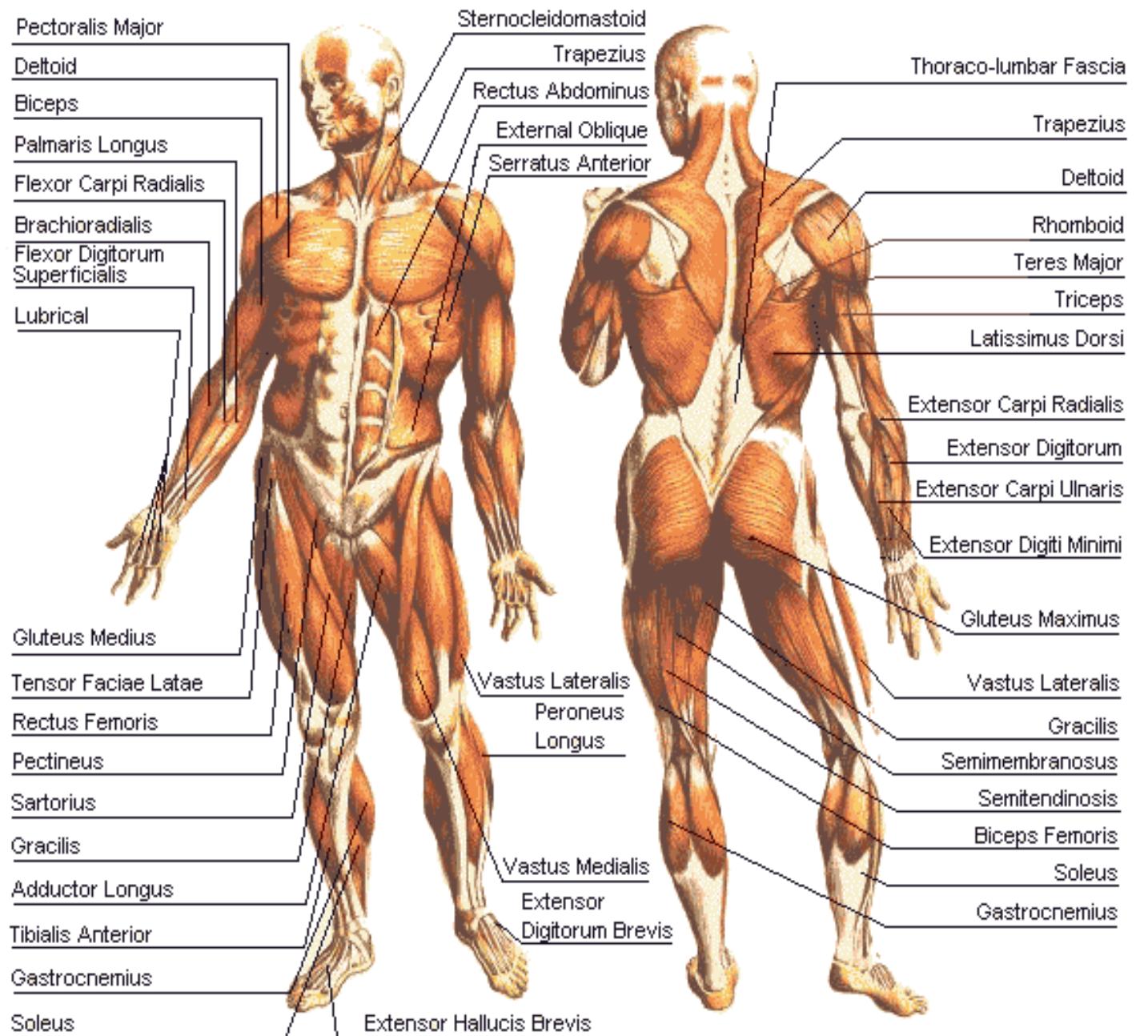
akka

# Welche Lösungen gibt es dafür im Cloud Computing?



- **Big Data Engines (low level)**
  - MapReduce
  - RDD (Resilient Distributed Dataset)
- **Big Data Datenbanken (high level)**
  - NoSQL Datenbanken
  - NewSQL Datenbanken (NoSQL + SQL)
- Verteilte Dateisysteme
- In-Memory Data Grids / Elastic Memory

# Die Anatomie von Big Data Datenbanken

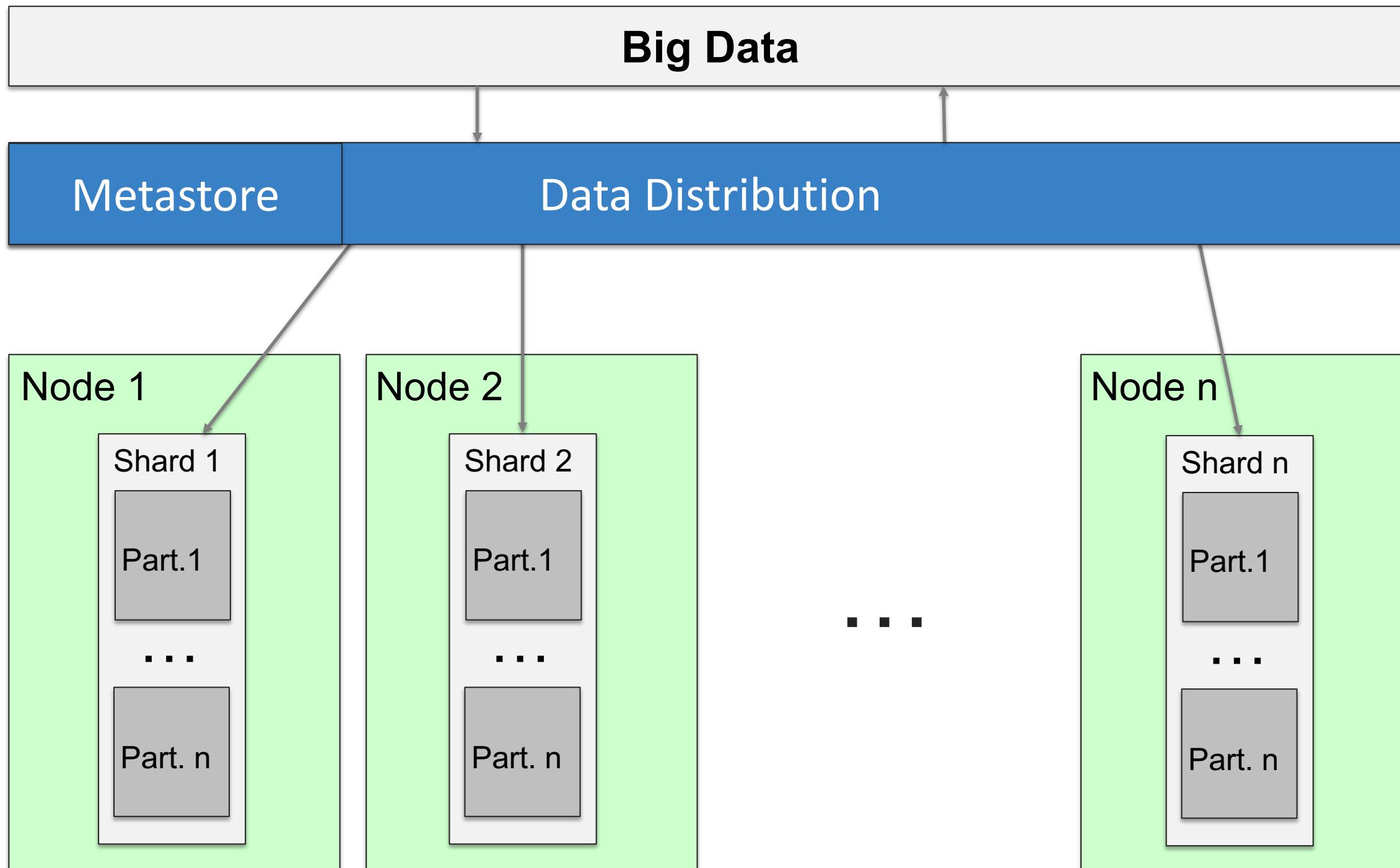


Query Distribution

Data Distribution

Data Persistence

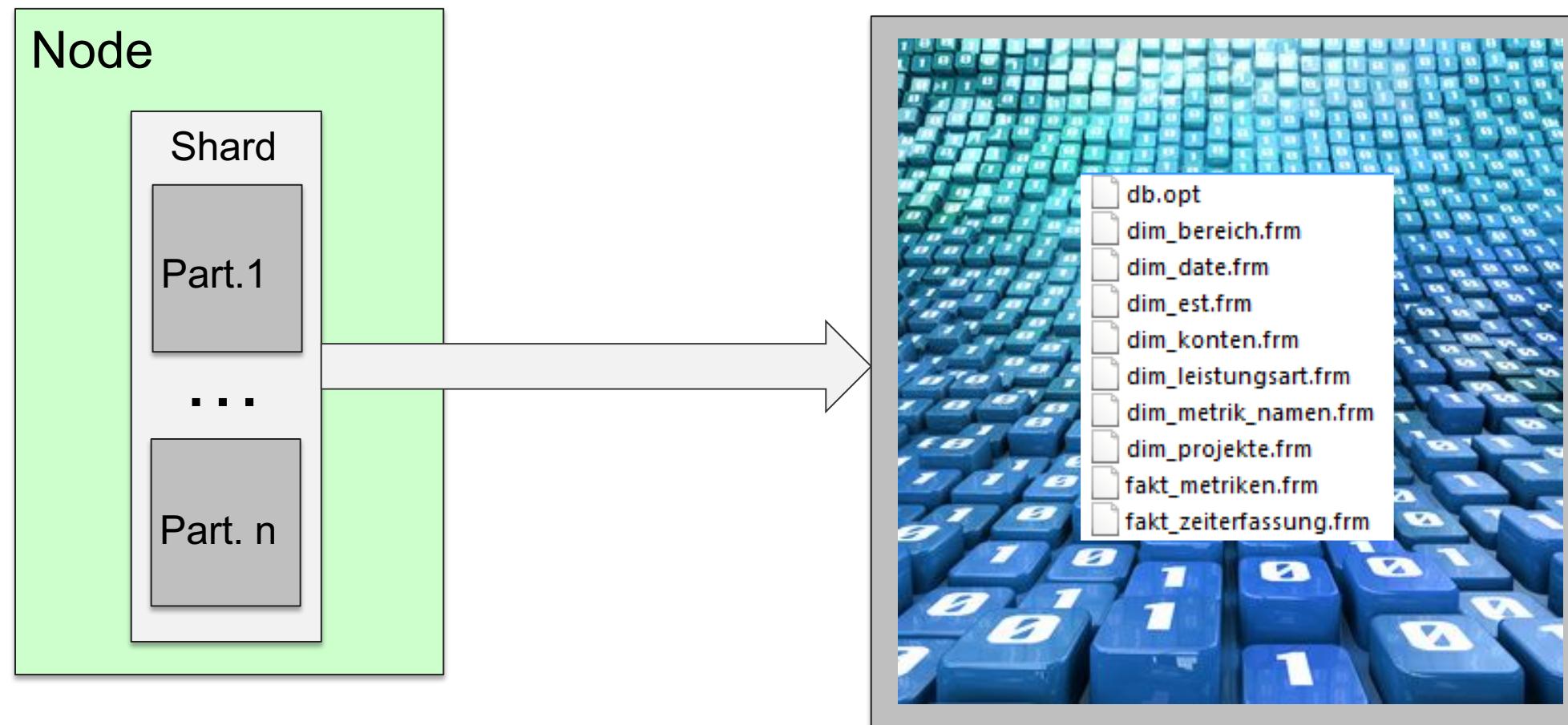
# Sharding and Partitioning: Verteilung und Stückelung von großen Datenmengen.



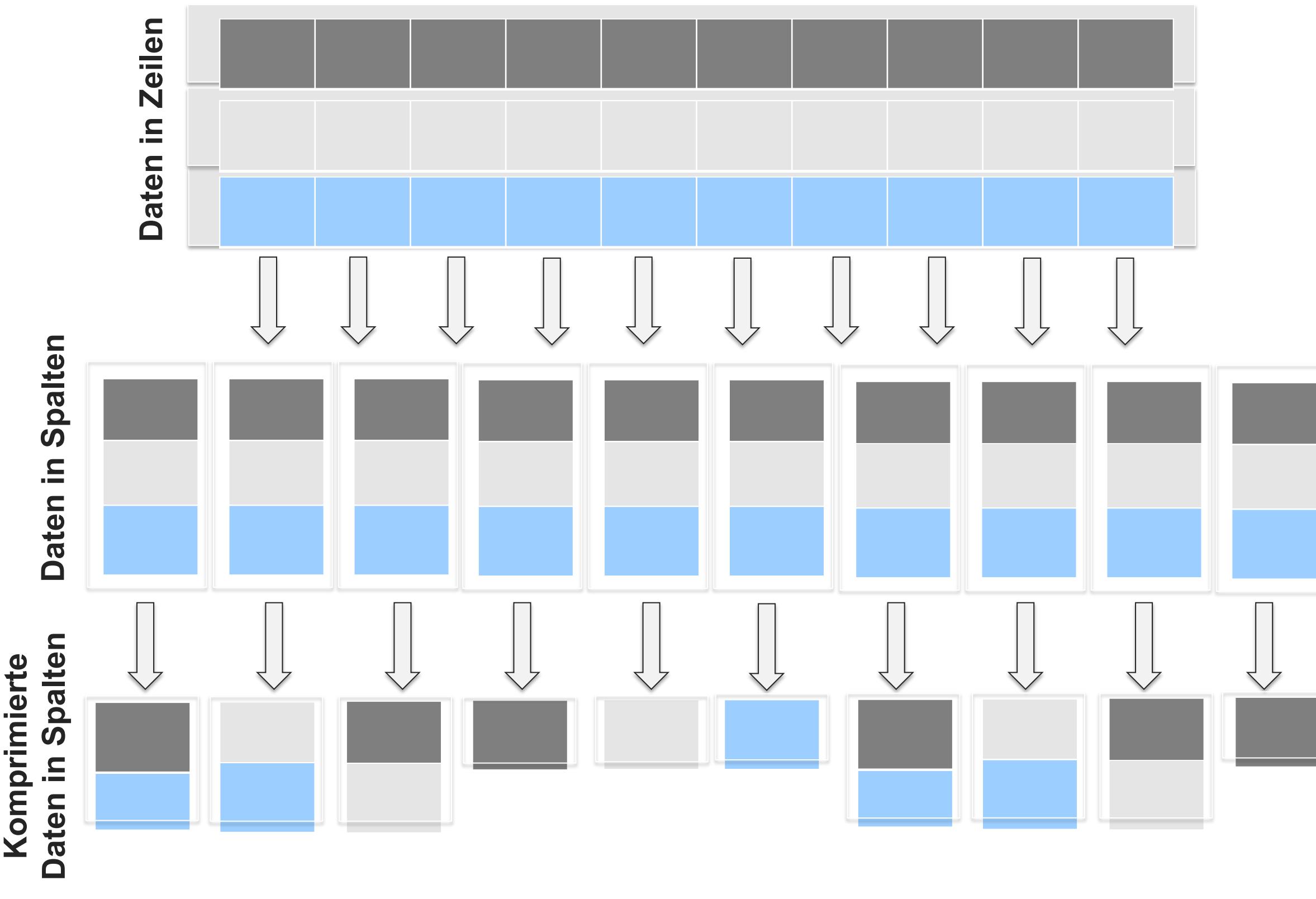
(Re-) Sharding- und  
Partitioning-Funktion:  
 $f(\text{Daten}) \rightarrow \text{Shard}$   
 $f(\text{Daten}) \rightarrow \text{Partition}$ .  
+ Replikationsstrategie.  
+ Konsistenzstrategie.

# Wie werden große Datenmengen technisch so gespeichert, dass eine schnelle Scan-Geschwindigkeit erreicht wird?

Dateien im (verteilten) Dateisystem



# Spalten-orientierte Datenspeicherung.

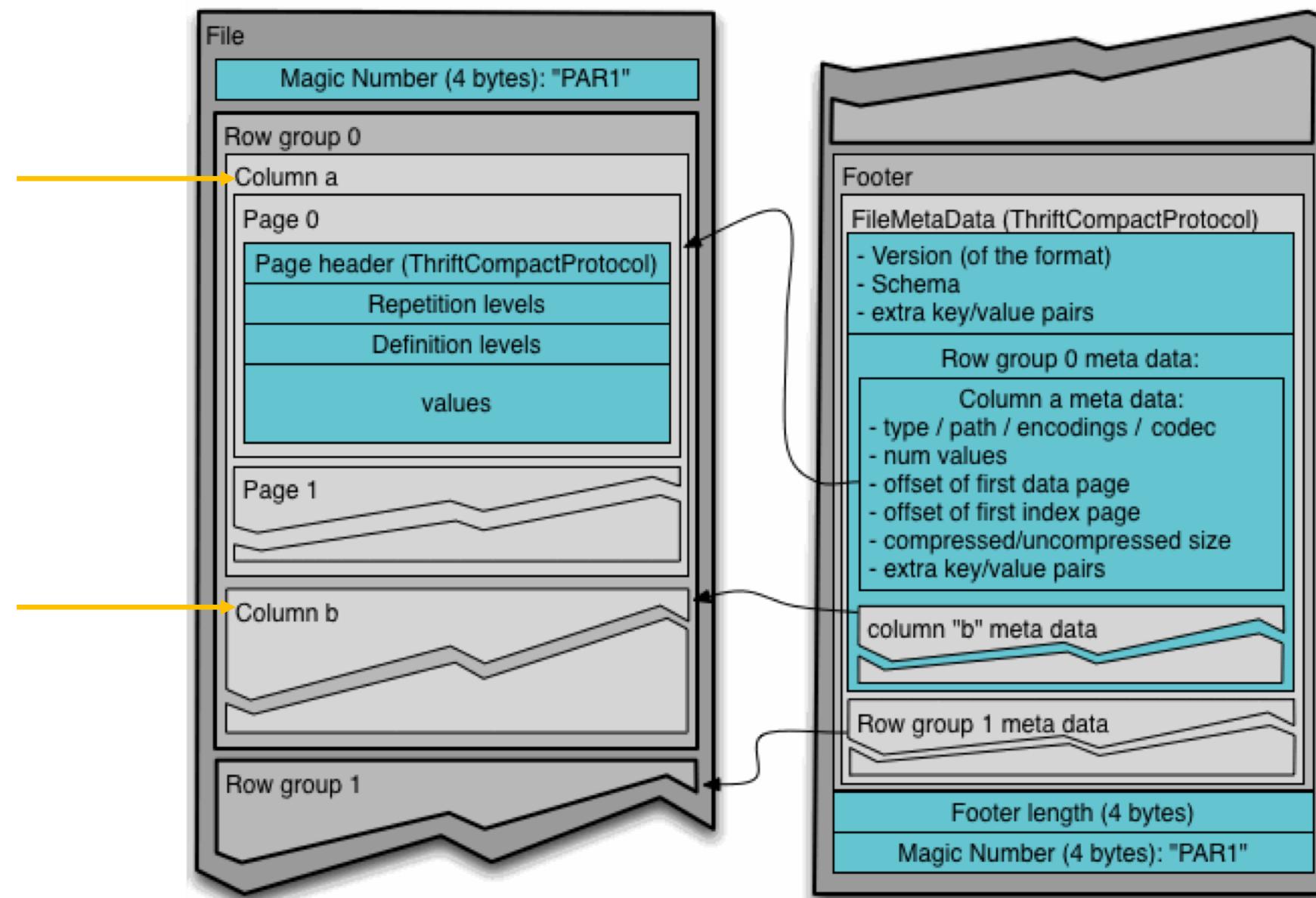


**The fastest I/O is the one that never takes place:** Es werden nur diejenigen Spalten gelesen, die benötigt werden (gerade bei breiten Tabellen wichtig)

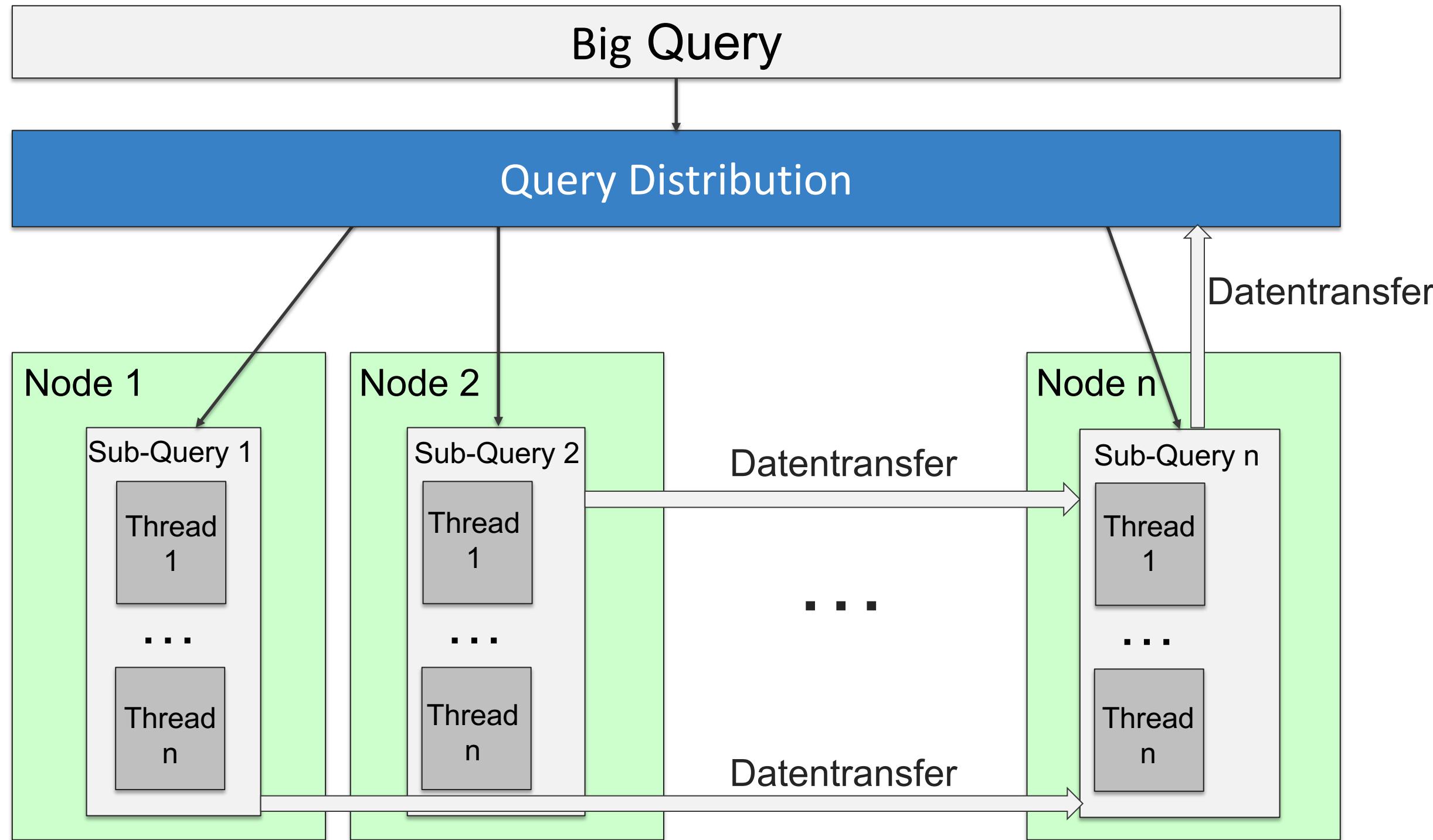
**Kompression** (funktioniert bei Spalten besser als bei Zeilen):

- Datentyp-spezifisch (z.B. Dictionaries)
- Allgemein (z.B. Snappy)
- + ggF. Spalten-Index

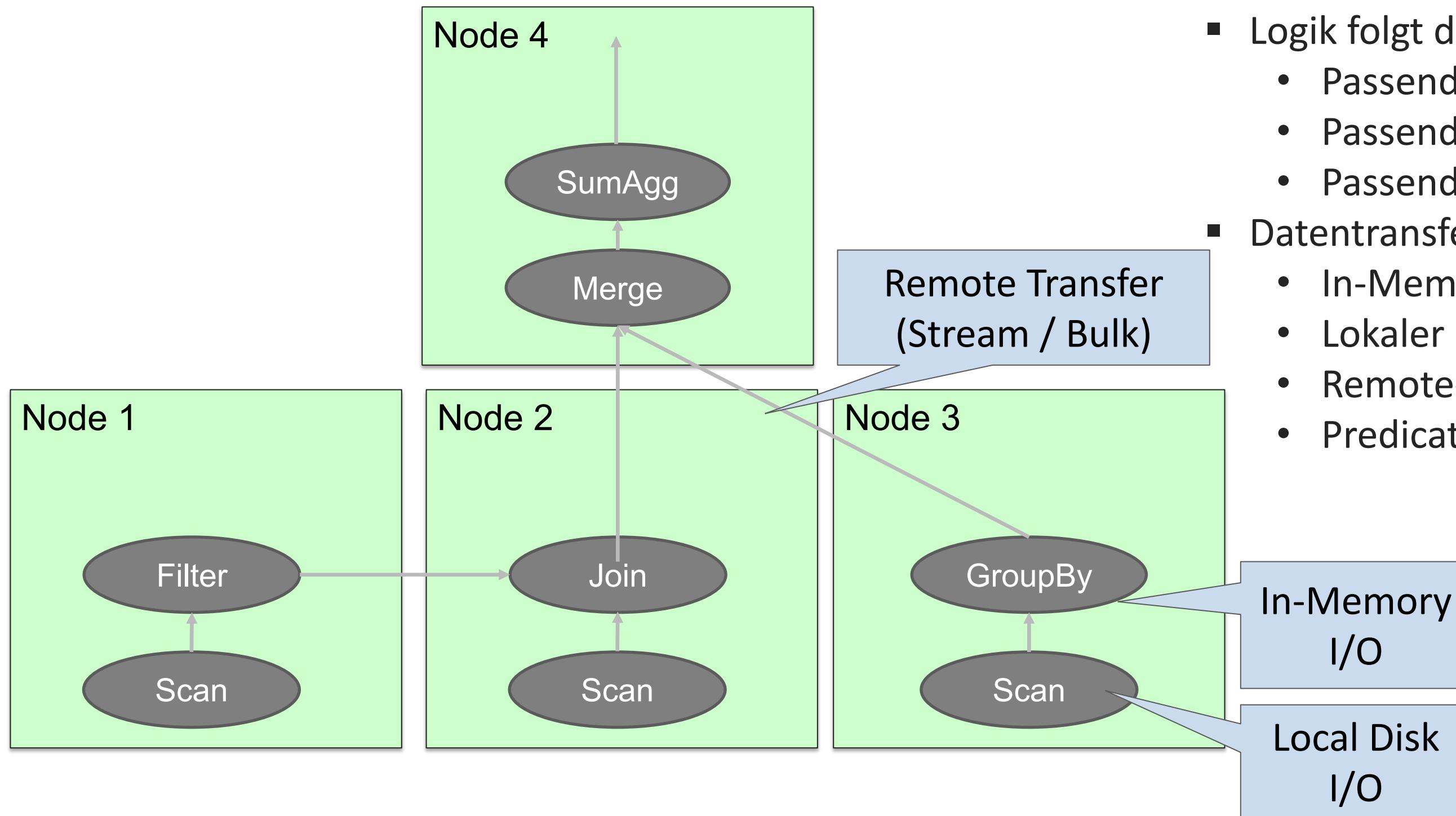
# Beispiel: Parquet



# Verteilte und parallelisierte Ausführung von Abfragen.



# Ein verteilter Ausführungsplan: Ein azyklischer Funktionsgraph.



- Logik folgt den Daten:
  - Passende Sharding-Funktion
  - Passende Partitioning-Funktion
  - Passende Replikation
- Datentransfer-Optimierung:
  - In-Memory vor ...
  - Lokaler Disk I/O vor ...
  - Remote-Transfer.
  - Predicate Pushdown.