MapReduce: Simplified Data Processing on Large Clusters

Tobias Schwarzer, Michael Theil

Hardware-Software-Co-Design

Universität Erlangen-Nürnberg

Tobias.Schwarzer@e-technik.stud.uni-erlangen.de

Michael.Theil@mathe.stud.uni-erlangen.de

Übersicht

- **Motivation**
- Programmiermodell
- Implementierung
- Erweiterungen
- Performance
- Schlussfolgerung

Motivation

- Rechenbeispiel (Stand 2004):
  - ≈ 20 Mrd. Webseiten x 20 KB ≈ 400TB
  - Lesegeschwindigkeit eines Rechners von einer Festplatte ≈ 30-35 MB/s

    => 4 Monate, um Web zu lesen
  - ≈ 1000 Festplatten, um das Web zu speichern
  - Bearbeitung der Daten benötigt zusätzliche Ressourcen

- Wie lässt sich das schneller machen?
  - Verteiltes System: gleiche Datenmenge mit 1000 Rechnern in weniger als 3 Stunden

Motivation

- ABER: Viel Programmierarbeit in verteilten Systemen:
  - Kommunikation und Koordination
  - Wiederherstellung nach Ausfall eines Rechners
  - Statusdaten zur Systemüberwachung
  - Fehlersuche, Debugging
  - Optimierung der Anwendung

- Programmierarbeit muss für jede Anwendung wiederholt werden

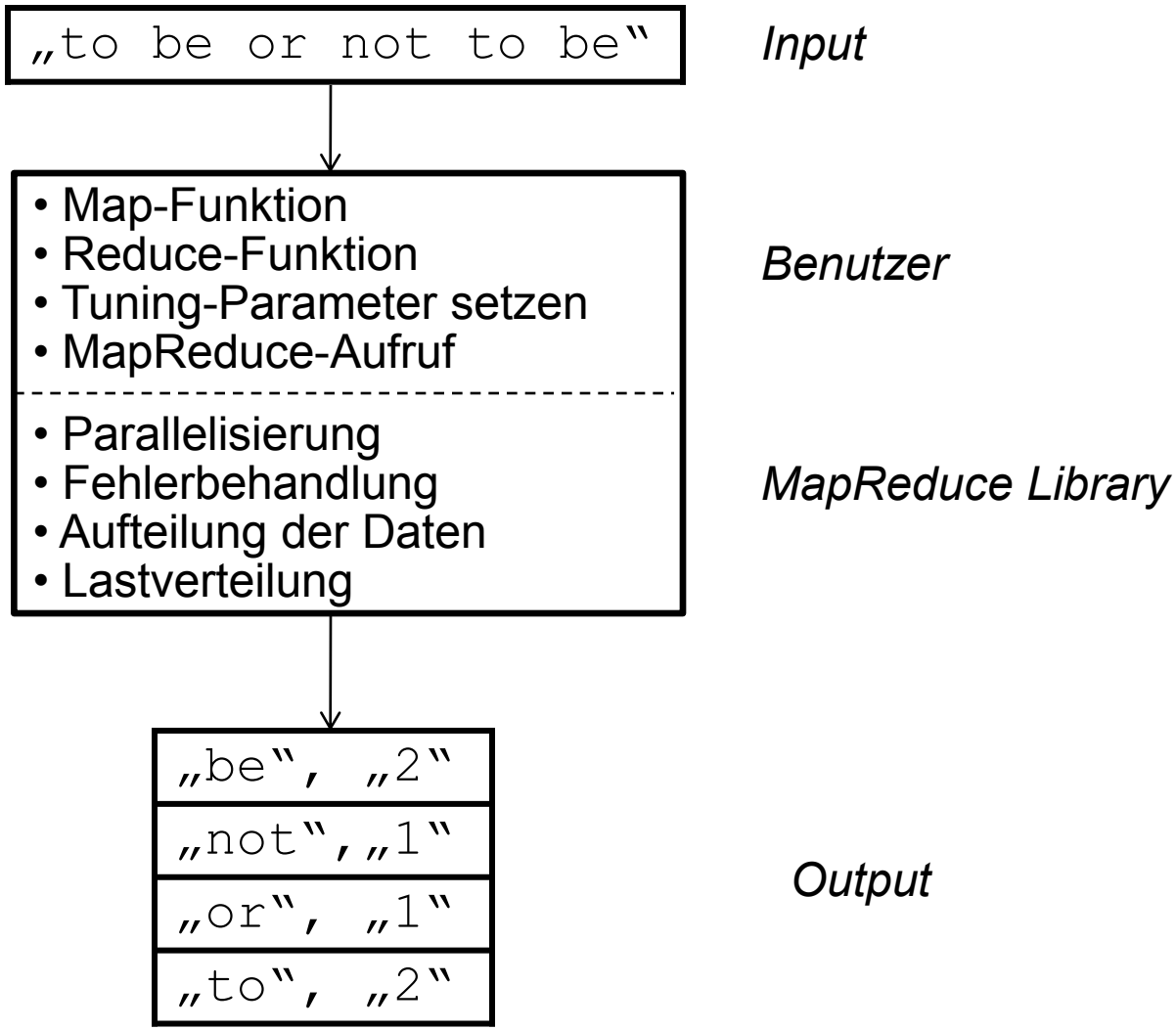
- Lösungsansatz zur Reduzierung des Programmier-aufwands:

MapReduce

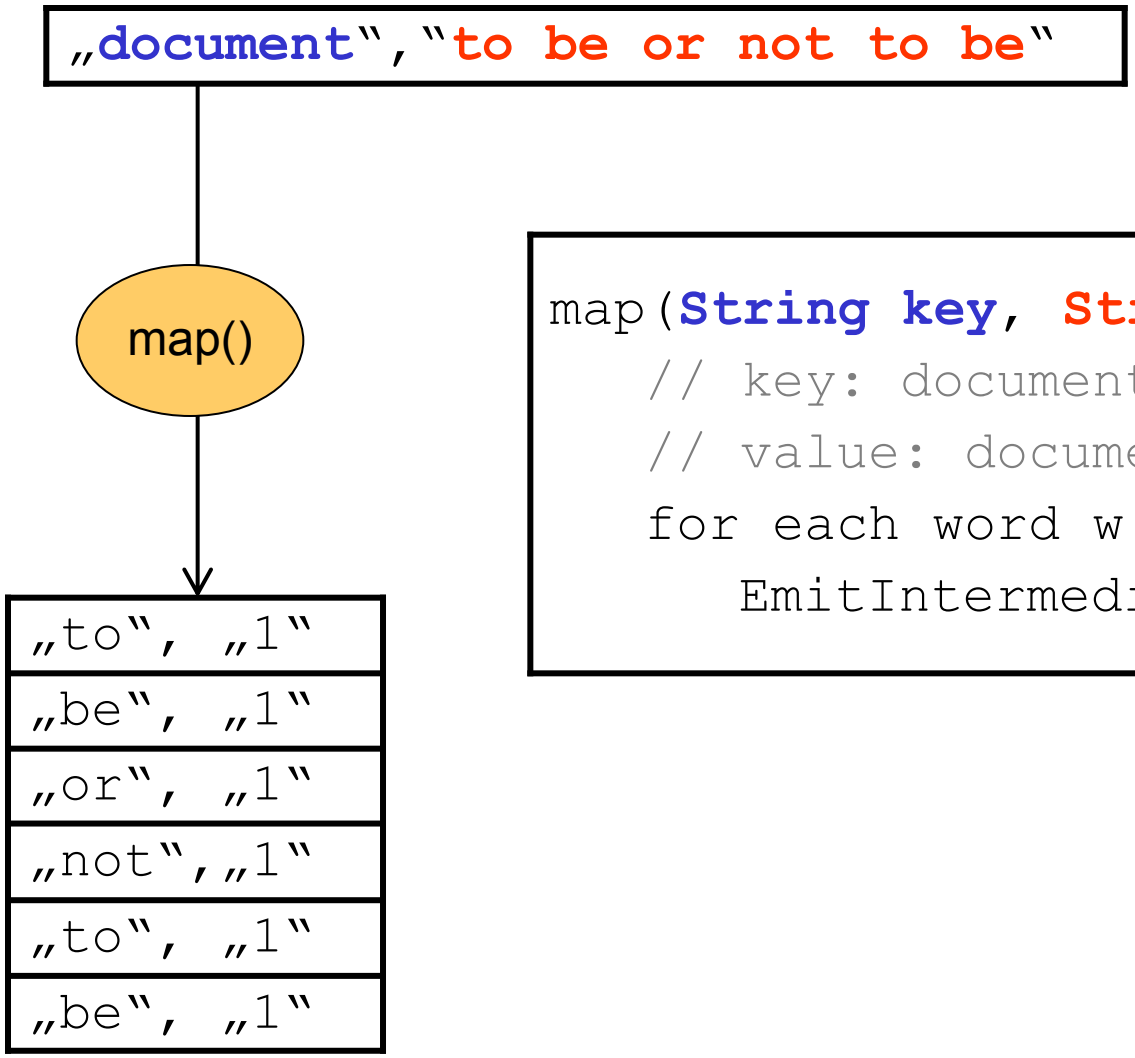
Übersicht

- Motivation
- **Programmiermodell**
- Implementierung
- Erweiterungen
- Performance
- Schlussfolgerung

Programmiermodell

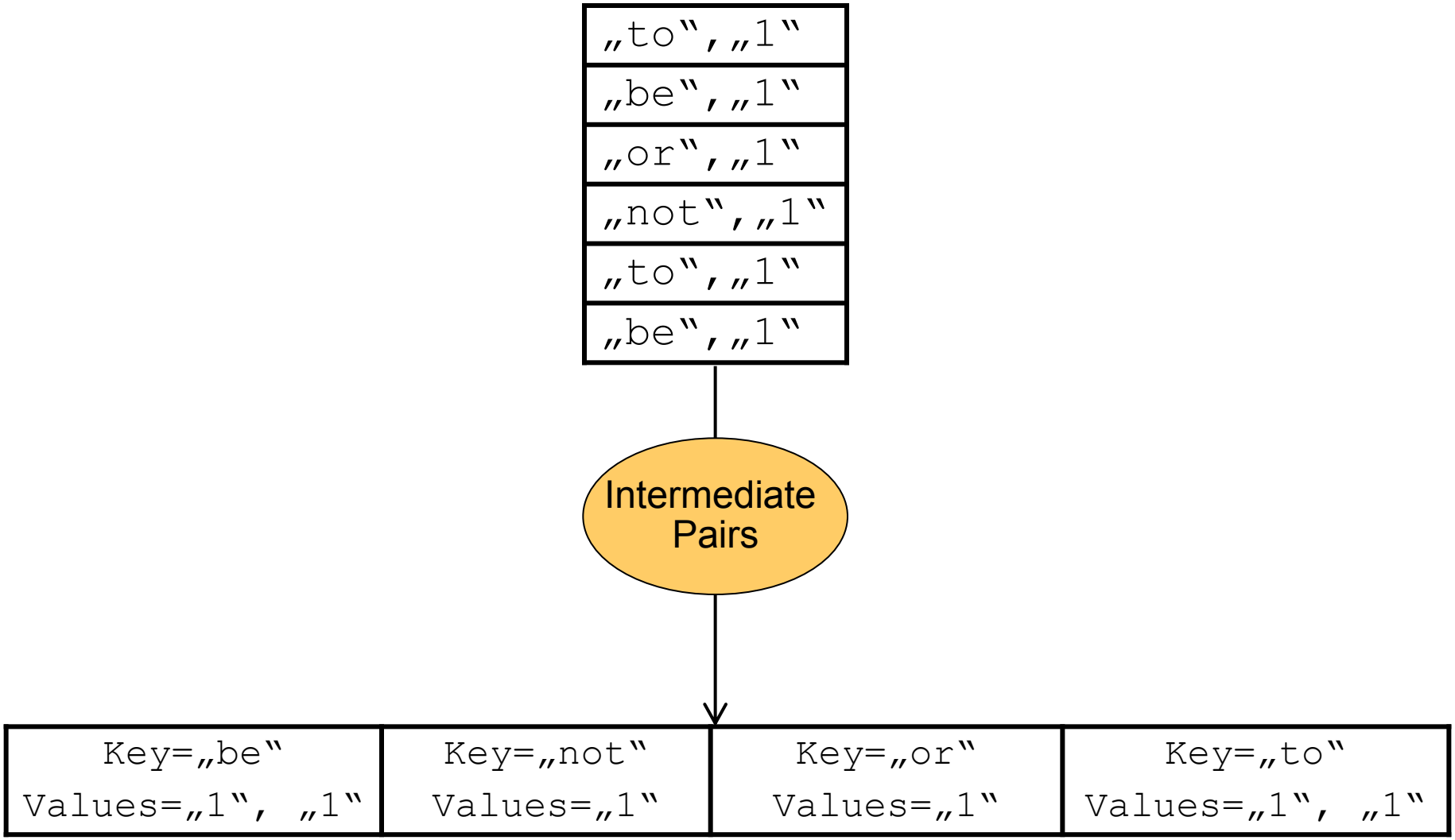


Map Function

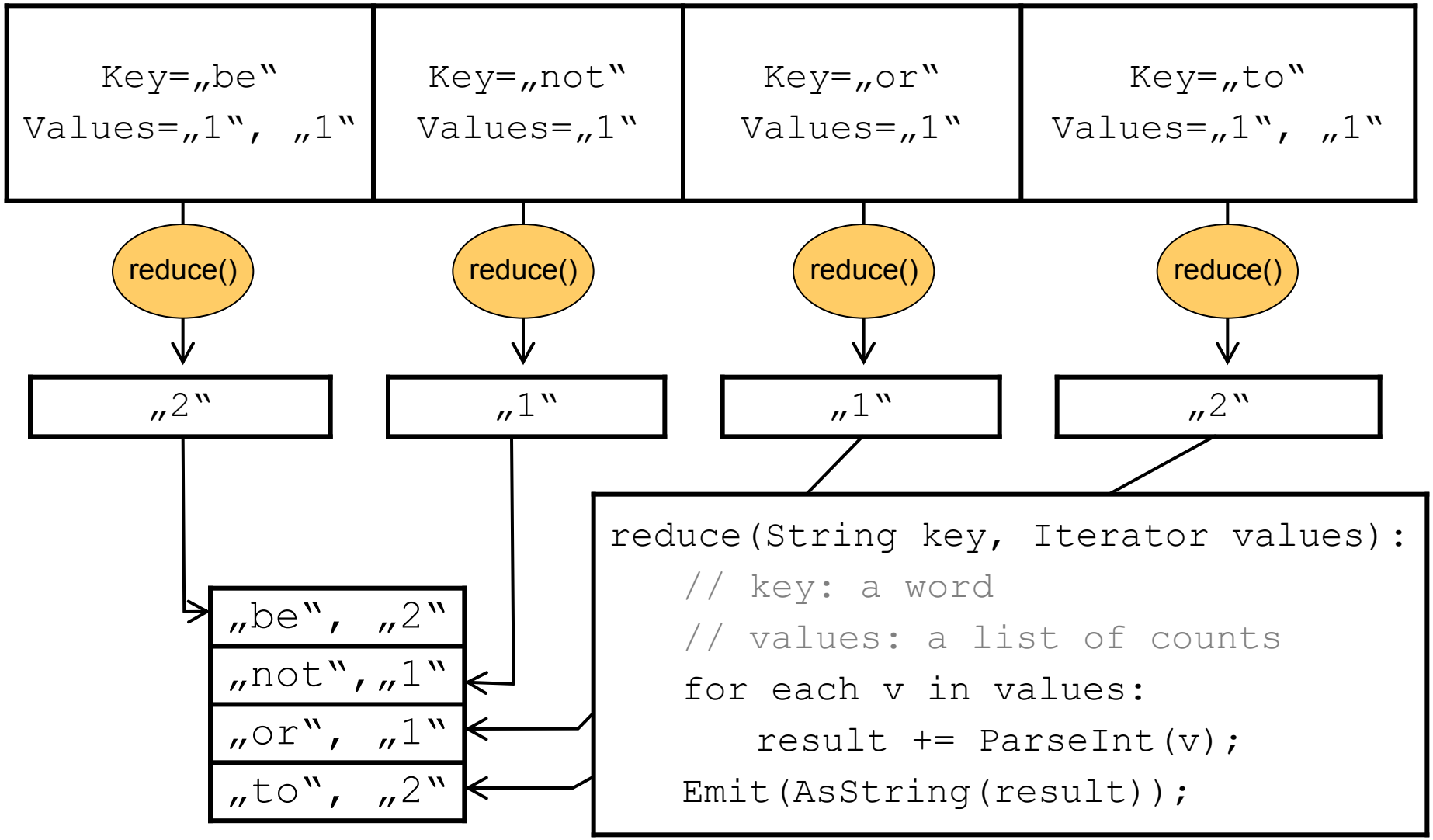



```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, „1“);
```

Intermediate Pairs



Reduce Function



Übersicht

- Motivation
- Programmiermodell
- **Implementierung**
- Erweiterungen
- Performance
- Schlussfolgerung

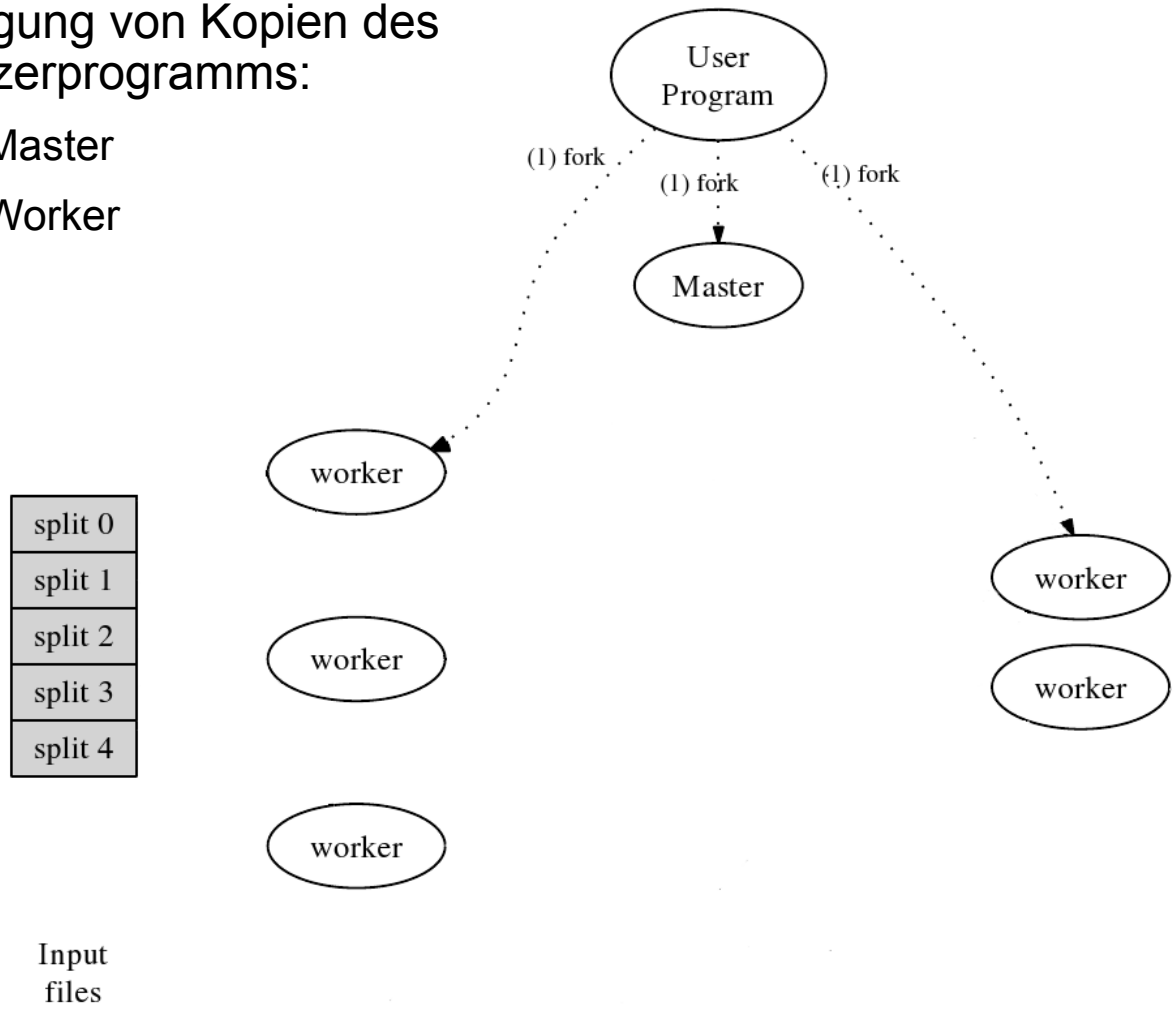
Implementierung

- Implementierung sollte an Entwicklungsumgebung angepasst sein:
  - Small Shared-Memory Machine
  - Großes NUMA Multiprozessorsystem
  - Verteiltes System

- hier: Entwicklungsumgebung von Google
  - Mehrere Cluster aus hunderten bzw. tausenden Rechnern
  - Aufteilung des Speichers auf lokale Festplatten der Rechner
  - Scheduling System, das Arbeitsaufträge auf Rechner verteilt

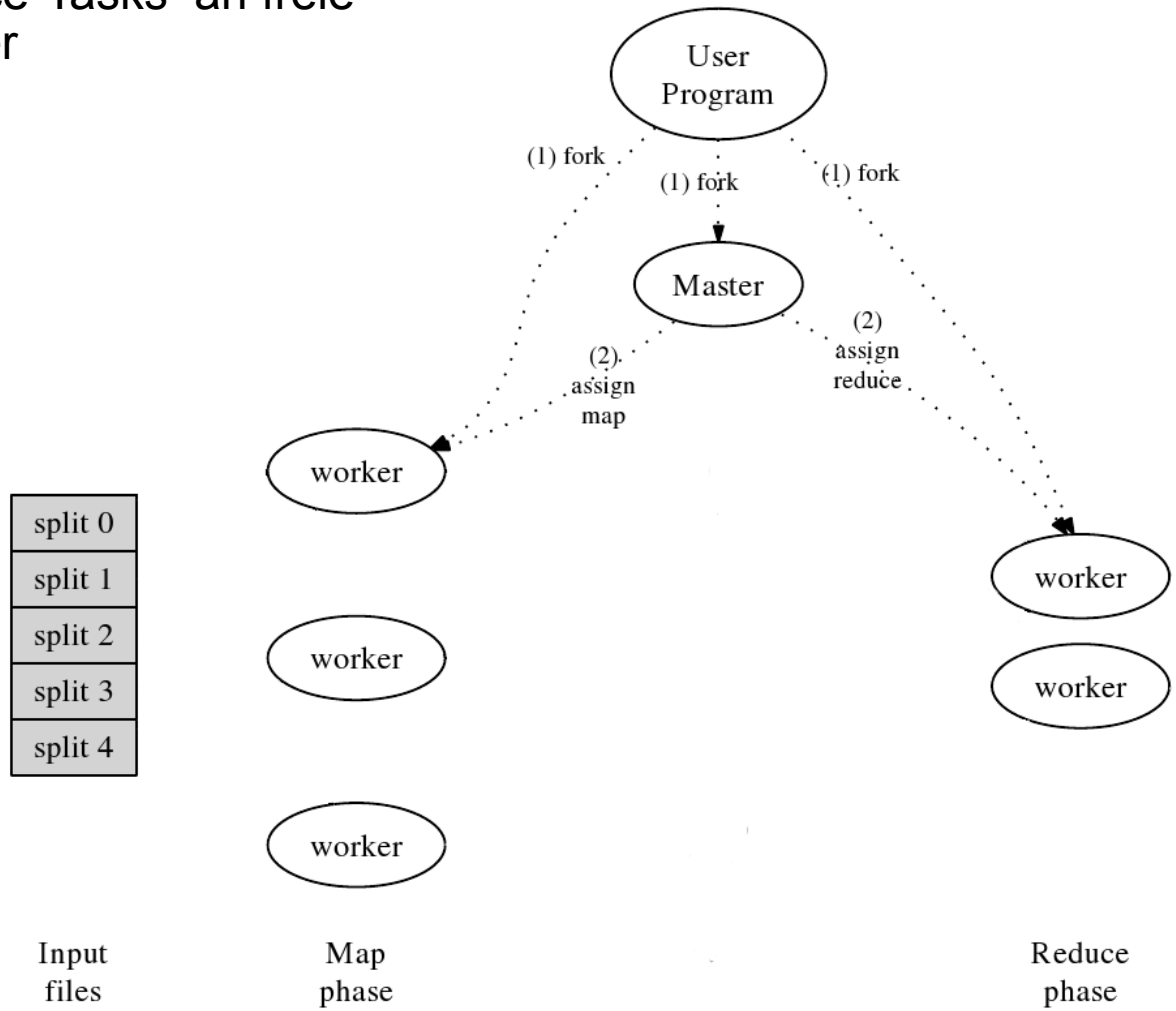
Execution Overview (1)

- Aufteilung der Eingabedaten
- Erzeugung von Kopien des Benutzerprogramms:
  - Master
  - Worker



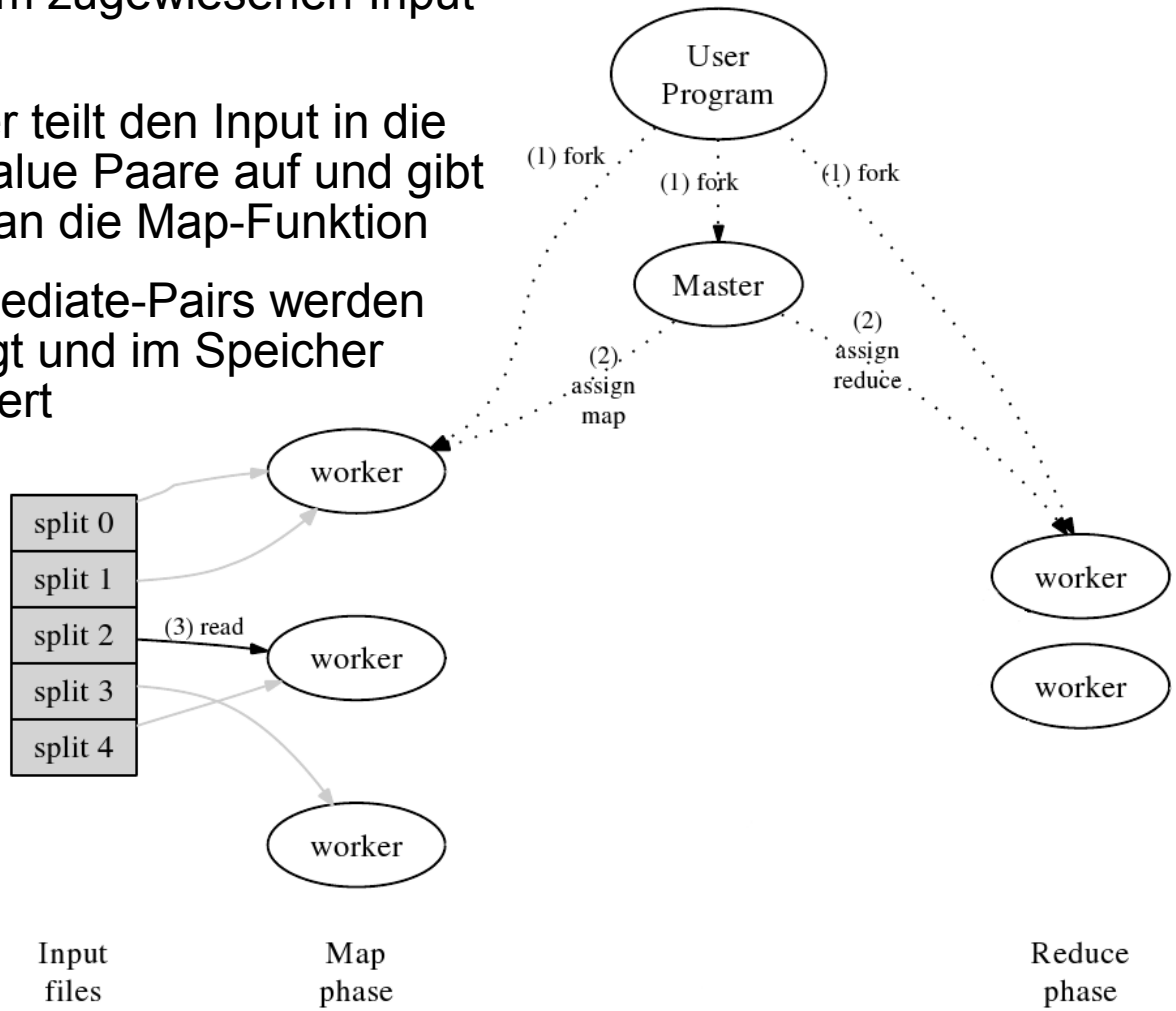
Execution Overview (2)

- Master vergibt Map- bzw. Reduce-Tasks an freie Worker



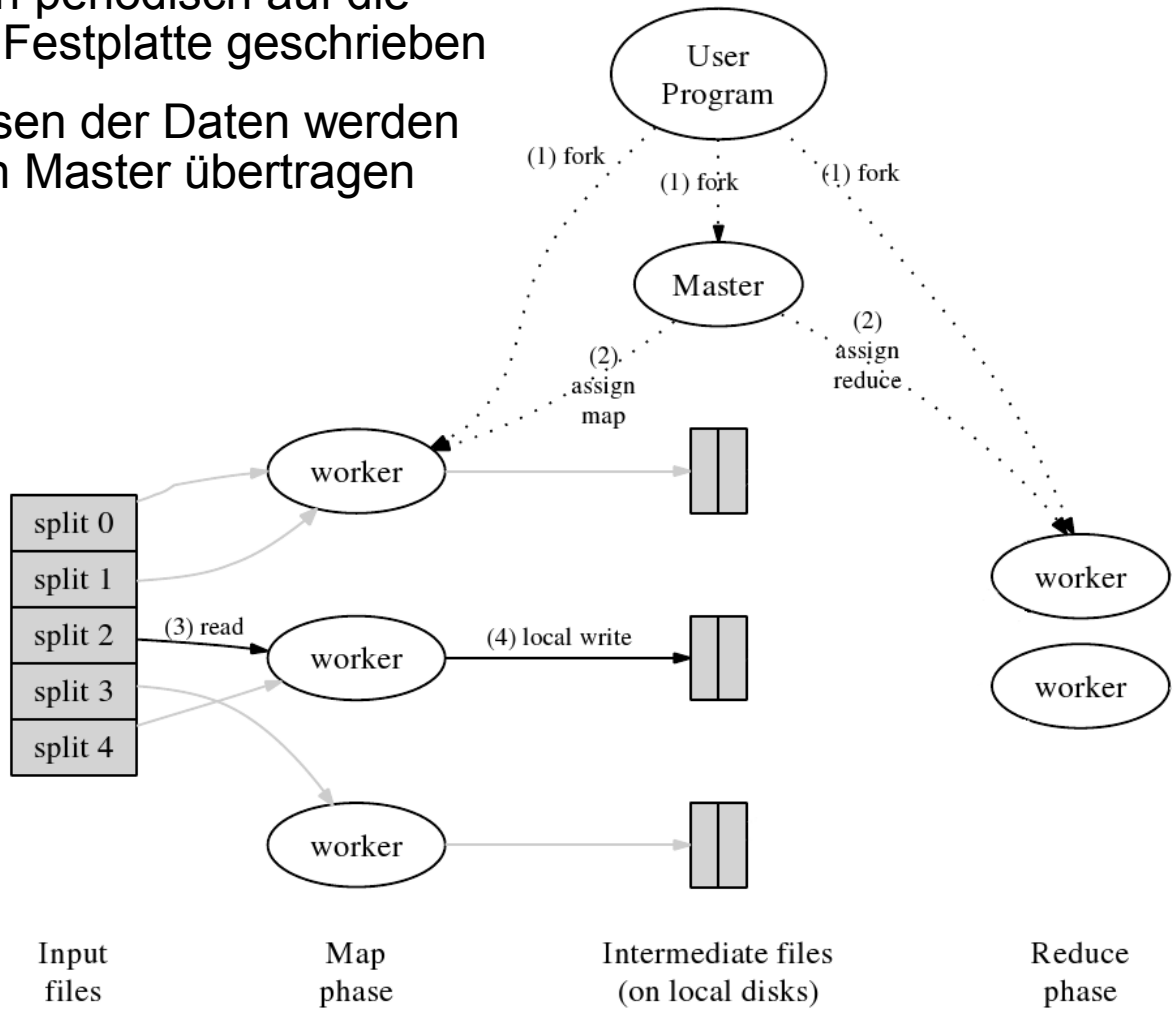
Execution Overview (3)

- Worker liest den Inhalt aus den ihm zugewiesenen Input-Splits
- Worker teilt den Input in die Key/Value Paare auf und gibt diese an die Map-Funktion
- Intermediate-Pairs werden erzeugt und im Speicher gepuffert



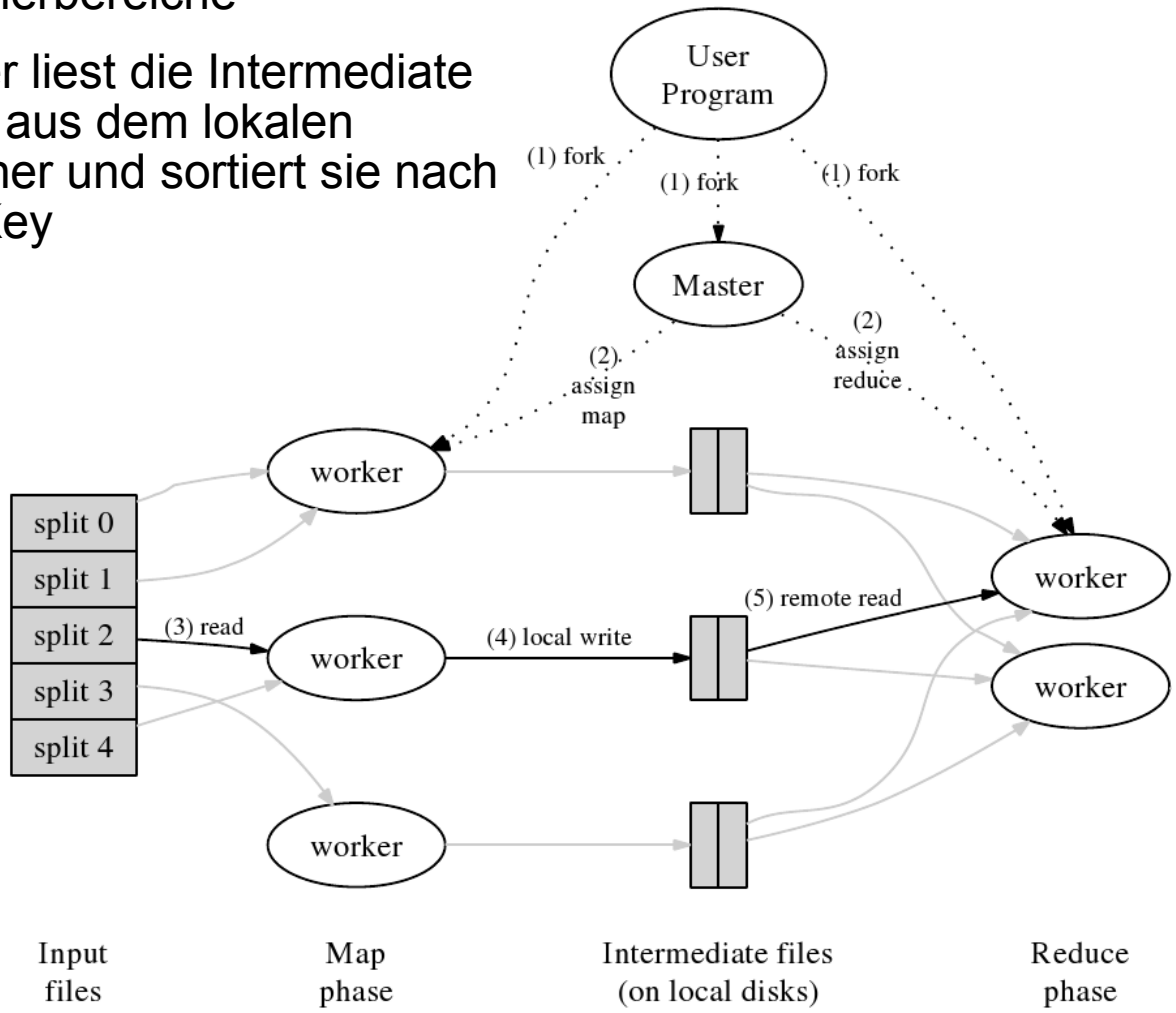
Execution Overview (4)

- Gepufferte Wertepaare werden periodisch auf die lokale Festplatte geschrieben
- Adressen der Daten werden an den Master übertragen



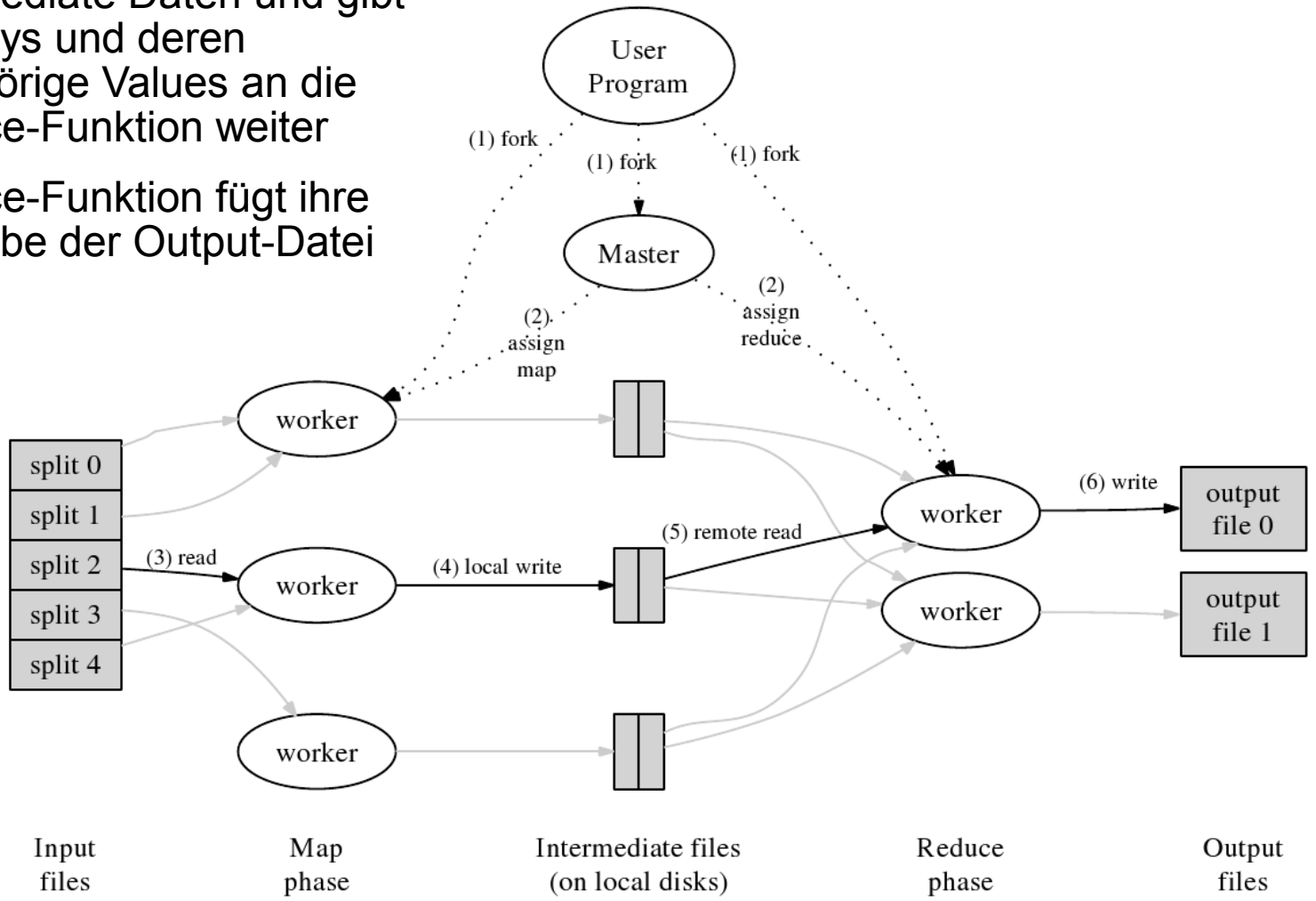
Execution Overview (5)

- Master übermittelt Worker Speicherbereiche
- Worker liest die Intermediate Daten aus dem lokalen Speicher und sortiert sie nach dem Key



Execution Overview (6)

- Worker iteriert über die Intermediate Daten und gibt die Keys und deren zugehörige Values an die Reduce-Funktion weiter
- Reduce-Funktion fügt ihre Ausgabe der Output-Datei hinzu



Fehlertoleranz

- Worker:
  - Worker wird als *failed* markiert, falls Antwort auf Ping-Anfrage des Masters ausbleibt
  - Folgen für Tasks auf dem Worker:
    - Alle laufenden Tasks werden zurückgesetzt
    - Fertiggestellte Map-Tasks müssen erneut ausgeführt werden, da Zugriff auf lokalen Speicher des Workers nicht mehr möglich ist
    - Fertiggestellte Reduce-Tasks müssen nicht erneut ausgeführt werden (Globaler Output)

- Master:
  - Periodisches Sichern der Master-Data-Structures:
    - Zustand der Tasks: idle, in-progress oder completed
    - Worker-ID
    - Adresse der Intermediate Daten
  - Bei Ausfall: Kopie vom Master wird mit zuletzt gesichertem Status erzeugt

Locality und Backup-Tasks

- Locality
  - Aufteilung der Input-Daten in 64 MB Blöcke
  - Kopien eines jeden Blocks werden auf unterschiedlichen Rechnern gespeichert
  - Master versucht Map-Tasks auf einem dieser Rechner auszuführen

    → Verringerung der Netzwerklast

- Backup-Tasks
  - „straggler“: Rechner, der ungewöhnlich lange Zeit benötigt, um einen der letzten Reduce-Tasks fertig zu stellen
  - Verbesserung durch Backup-Tasks:

    kurz vor Fertigstellung der MapReduce-Operation startet Master Backup-Tasks für übrige „in-progress“ Tasks

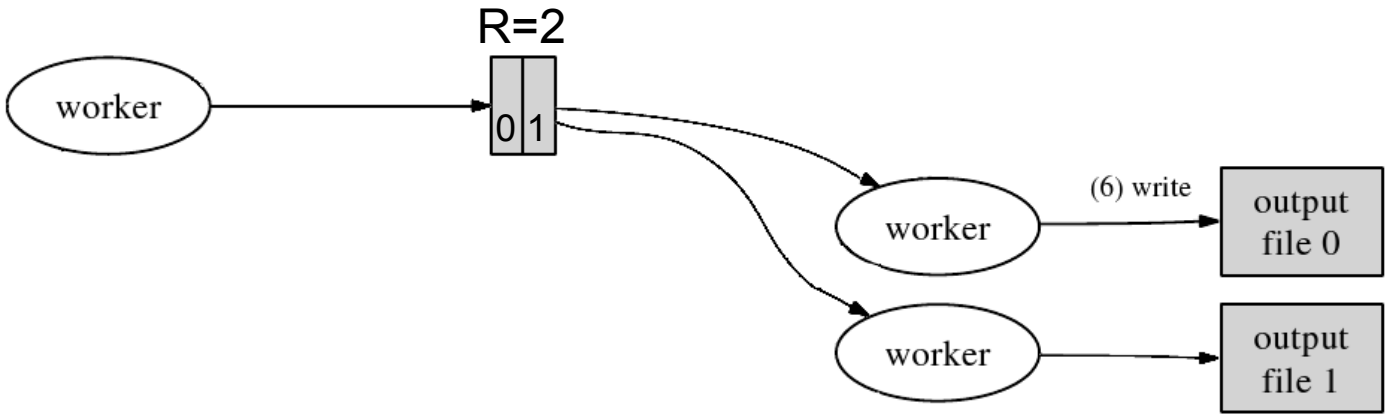
    → Reduzierung der Ausführungszeit von MapReduce

Übersicht

- Motivation
- Programmiermodell
- Implementierung
- **Erweiterungen**
- Performance
- Schlussfolgerung

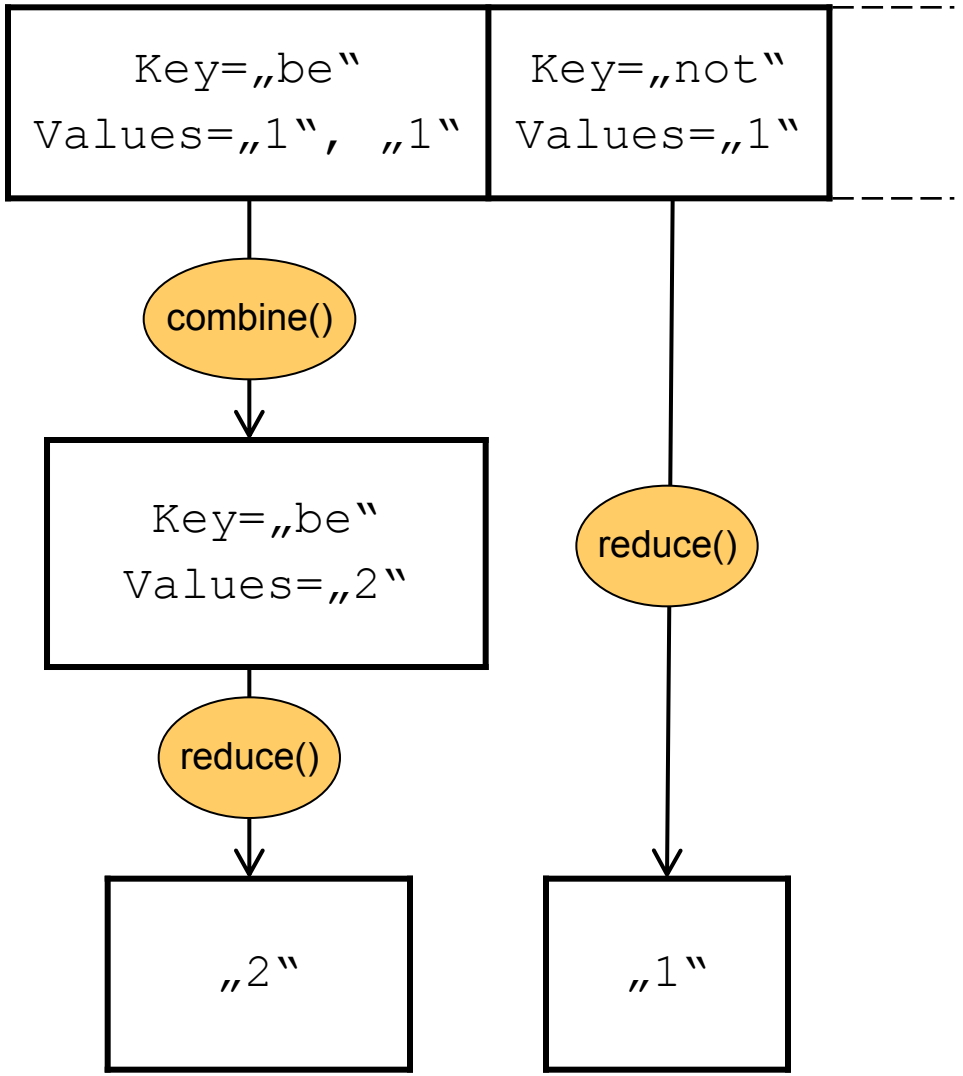
Partitioning Function

- Anwender legt die Anzahl an Reduce-Tasks bzw. Output Files (R) fest
- Daten werden dieser Anzahl entsprechend mittels einer Partitioning Function aufgeteilt
- Standard: `hash(key) mod R` (ausgewogene Partitionen)
- Partitioning Function kann vom Benutzer je nach Anwendung beschrieben werden



Combiner Function

- Zusammenfassen mehrfacher Daten auf dem gleichen Rechner durch Combiner Function
  - ➔ Vermeidung von Wiederholungen in Intermediate keys
  - ➔ Minimierung der Netzwerklast
- Gleicher Code für Combiner und Reduce Function



Andere Erweiterungen

- Skipping Bad Records
  - Bugs im Anwendercode können zum Absturz der Map oder Reduce Function zu einem deterministischen Zeitpunkt führen

    → MapReduce Operation wird nicht abgeschlossen
  - Optionaler Modus, in dem diese Records entdeckt und verworfen werden

- Input and Output Types
  - Hinzufügen von selbst definierten In- und Output-Types mit Hilfe eines *reader* Interface

- Local Execution
  - Alternative Implementierung: Sequentielle Ausführung von MapReduce auf lokalem Rechner

    → Jedes Debugging oder Testing Tool einsetzbar

Übersicht

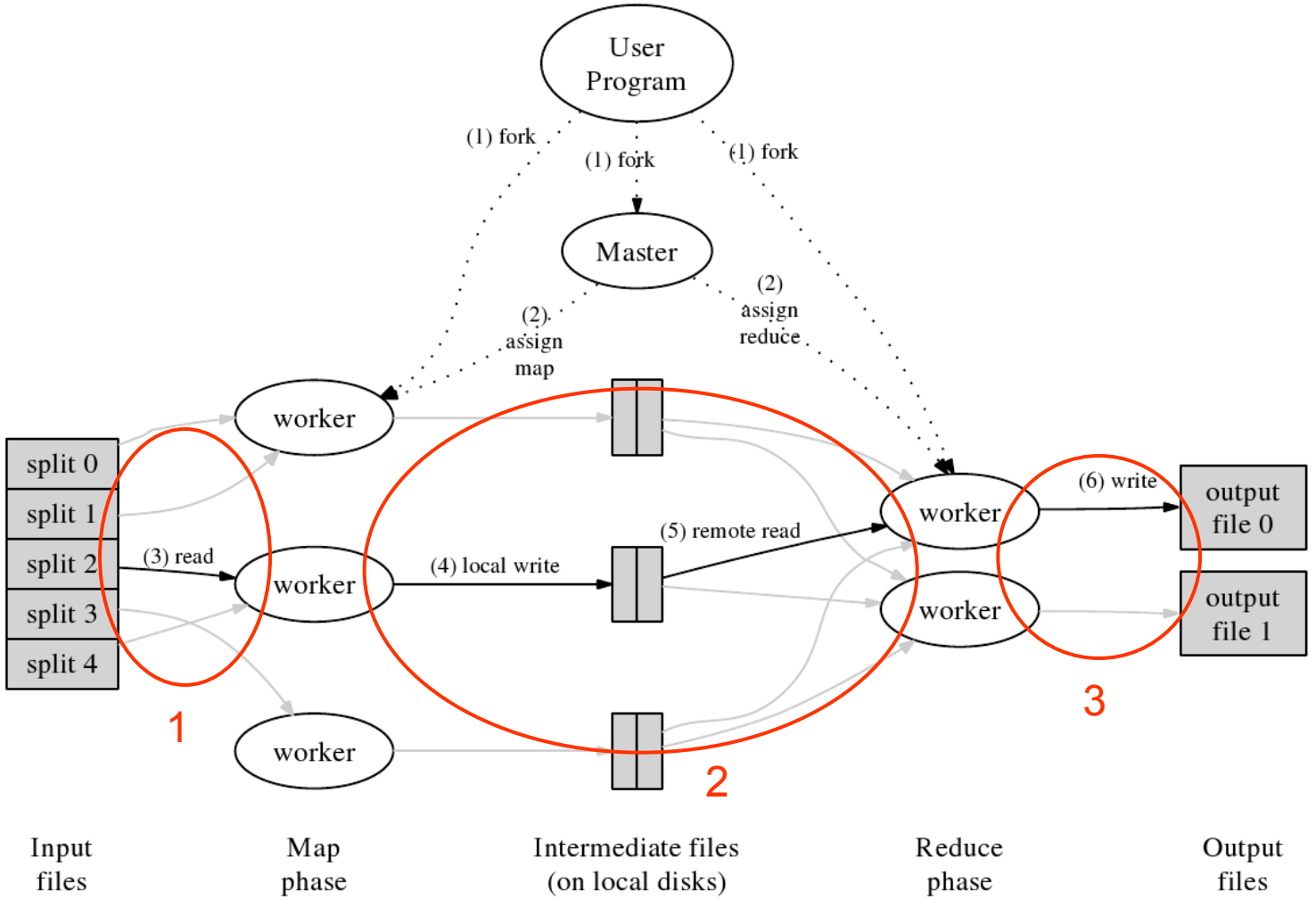
- Motivation
- Programmiermodell
- Implementierung
- Erweiterungen
- **Performance**
- Schlussfolgerung

Performance - Test (2004)

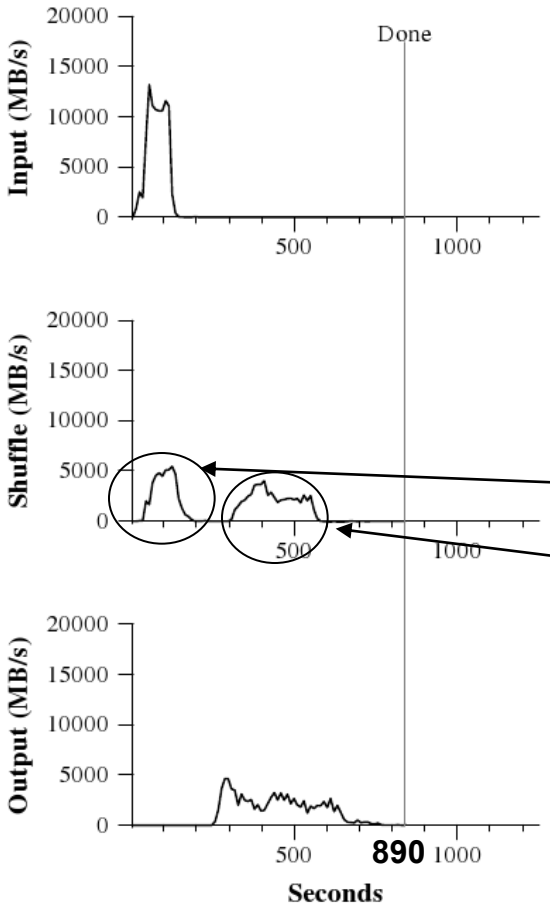
- Aufgabe:
  Sortierung von 10^{10} 100-Byte Records (≈ 1 TB)

- Cluster Konfiguration
  - 1800 Rechner mit 2 GHZ Intel Xeon Prozessoren mit Hyper-Threading
  - 4GB Arbeitsspeicher, davon 1-1,5 GB für andere Tasks reserviert
  - 160GB IDE Festplatten
  - 100-200 Gbit/s Netzwerk

Performance – Schritte 1-3



Normale Ausführung



(a) Normal execution

1. Lesen des Inputs durch Map-Worker:

- Lesen hauptsächlich von lokaler Festplatte

  → Leserate sehr hoch (Spitzenrate: 13 GB/s)

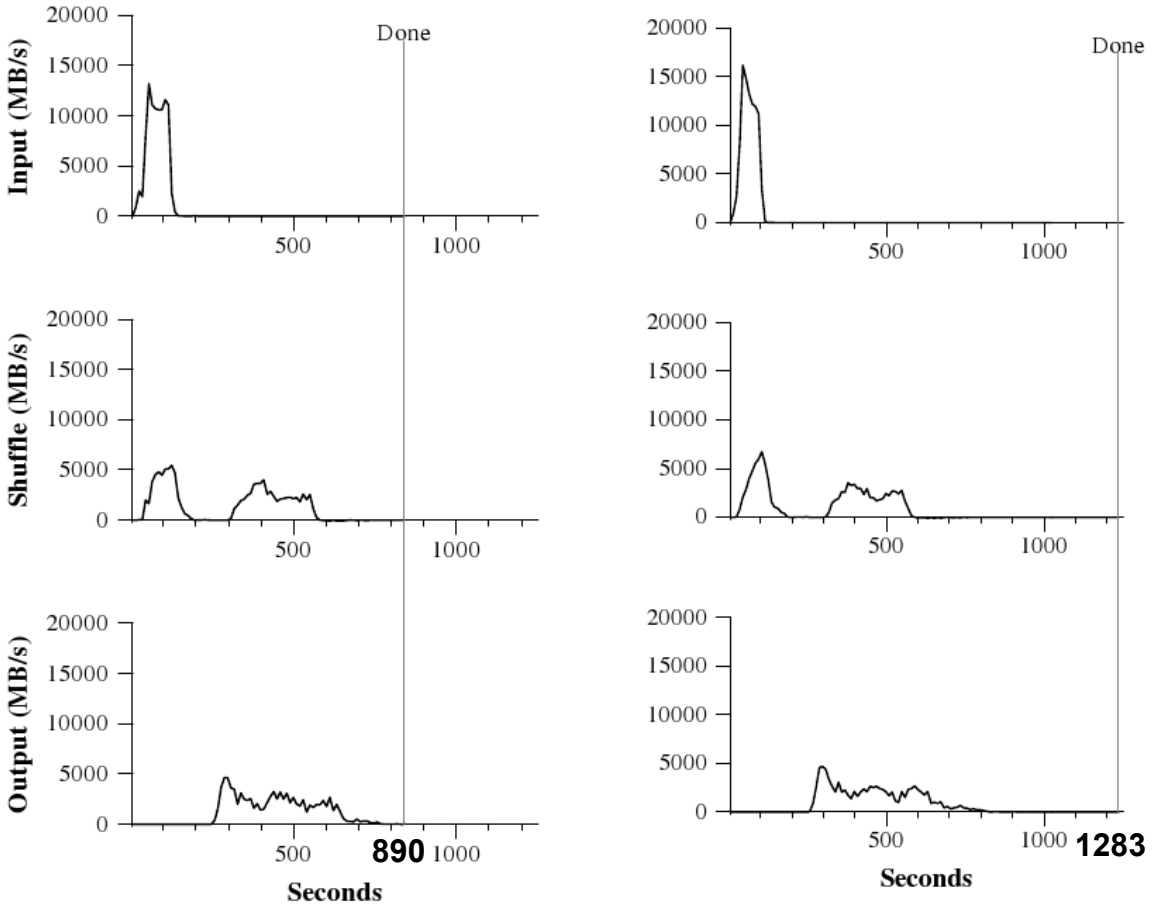
2. Senderate von Map-Tasks zu Reduce-Tasks (Shuffle):

- Begrenzung auf 1700 Reduce-Tasks zur selben Zeit
  → Aufteilung in 2 Teile
- 1.Teil beginnt, sobald 1. Map-Task fertig
- 2.Teil beginnt, sobald 1. Reduce-Task fertig

3. Schreibrate der Daten zum Output:

- Delay zwischen Shuffle und Schreiben aufgrund der Sortierung der Daten
- Fertigstellung bei 890s

Ohne Backup-Tasks



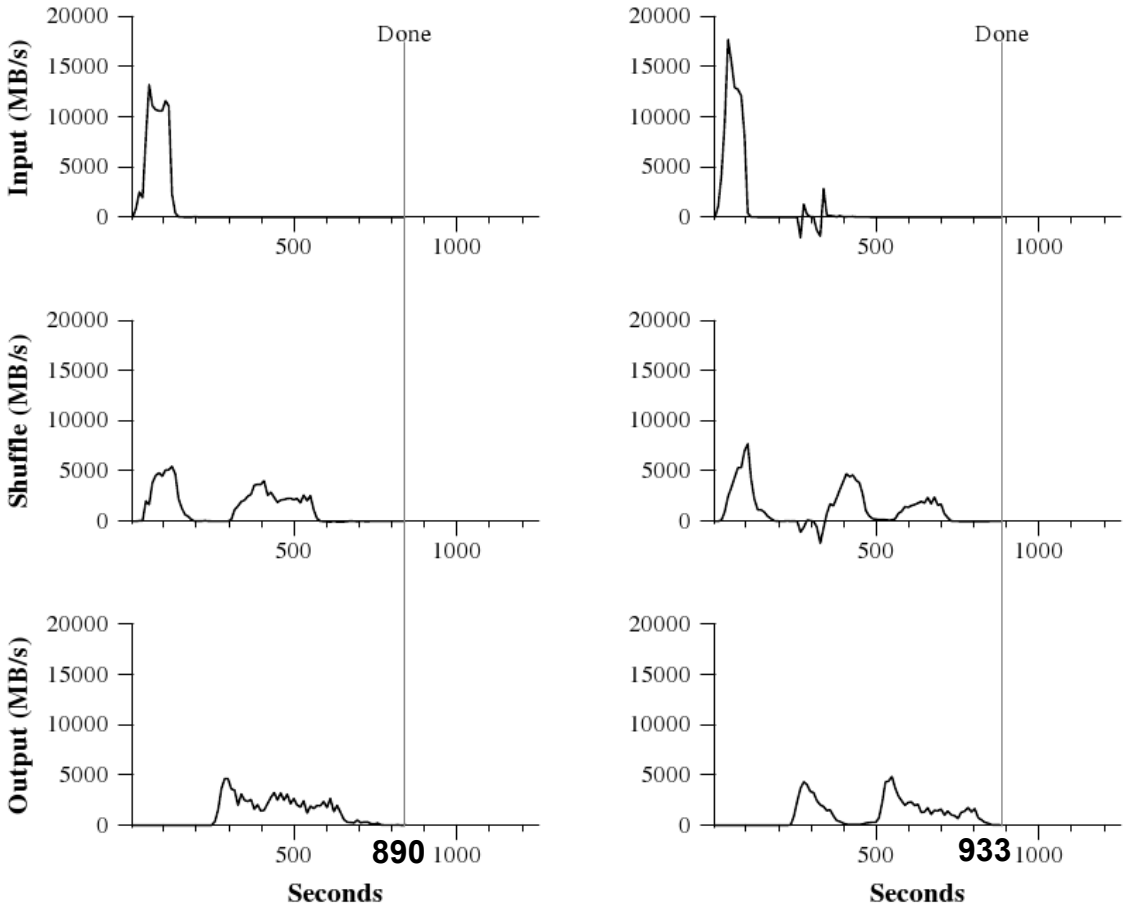
(a) Normal execution

(b) No backup tasks

Ohne Backup-Tasks:

- ähnlich der normalen Ausführung
- Nach 960s: alle bis auf 5 Reduce-Tasks beendet
- Fertigstellung nach 1283s (44% länger)

Tasks killed



(a) Normal execution

(c) 200 tasks killed

Tasks killed:

- 200 von 1746 Workerprozessen gekilled
- Auswirkung: negative Inputrate, da fertig gestellte Arbeit neu ausgeführt werden muss
- Fertigstellung nach 933s (5% länger)

Übersicht

- Motivation
- Programmiermodell
- Implementierung
- Erweiterungen
- Performance
- **Schlussfolgerung**

Schlussfolgerung

- Gründe für MapReduce:
  - Programmiermodell einfach zu verwenden auch für Programmierer ohne Erfahrung mit Parallelen und Verteilten Systemen
  - Vielzahl an Problemstellung mit MapReduce lösbar
    - Sortieren von Daten
    - Google Projekte (z.B. Indexing System bei der Websuche)
    - Data Mining oder Maschinelles Lernen
  - Implementierung von MapReduce für viele unterschiedliche Entwicklungsumgebungen möglich
- Master als Bottleneck

Vielen Dank für eure Aufmerksamkeit!

Referenzen

- [1] Dean, Jeffrey/Ghemawat, Sanjay. „MapReduce: Simplied Data Processing on Large Clusters.“ *OSDI 2004*. San Francisco: USENIX Association, 2004. 137-150.

- [2] Kleber, Michael. „The MapReduce Paradigm“. Google, Inc., 2008. http://sites.google.com/site/mriap2008/intro_to_mapreduce.pdf

Andere Erweiterungen (2)

- Status Information
  - Auf dem Master läuft ein interner HTTP Server, der Status Pages ausgibt
  - Status Pages zeigen den Fortschritt der Berechnung
  - Vorhersagen über Restdauer und Geschwindigkeit möglich
- Counters
  - User definiert Counter Objekte und inkrementiert diese bei bestimmten Ereignissen in Map und/oder der Reduce Function
  - Worker gibt Counter periodisch an Master weiter
- Ordering Guarantees
  - Abarbeitung der Intermediate Pairs erfolgt aufsteigend nach dem Key (Sortierung)
- Side-effects
  - Hilfsdateien als zusätzliche Ausgaben von Map- und Reduce-Operatoren