MapReduce

Vereinfachte Datenverarbeitung in großen Rechnerverbünden

Igor
Marijanovic
794894

Fachvortrag
WAR

19.12.2012

Beuth Hochschule für Technik
Berlin

Inhalt

Einleitung

MapReduce

Hauptteil

Implementierung

Verbesserung

Ergebnis

Abschluss

Fazit, Entwicklung und Ausblick

MapReduce

Situation vorher

Google Suchmaschine um 2003

- Viel Rechnerzeit für Erzeugung der Indexdateien
- Berechnung sequentiell
- Hohe Belastung des Netzwerks
- Ablage in verteiltes Dateisystem GFS (Google File System)

MapReduce

Motivation

Verarbeitung von sehr großen Datenbeständen

- Berechnung über tausende CPUs verteilen
- Netzwerklast für Berechnungen reduzieren
- Toleranz gegenüber Rechnerausfällen
- Komplexe Details hinter Framework verbergen

MapReduce

Programmiermodell

- Ursprung: Lisp
- Gebräuchlich in funktionalen Programmiersprachen
- Viele Probleme können mit MapReduce ausgedrückt werden
- Leicht über Knoten verteilbar
- Solide Fehlersemantik

MapReduce

Funktionsweise bei Google

- map(key, value) wird ausgeführt für jedes Element einer Liste
  - Ein Paar (new-key, new-value) wird ausgegeben
- reduce(key, values) wird für **jeden** einzigartigen Schlüssel ausgeführt der von map() ausgegeben wurde
  - Finale Ausgabe

MapReduce

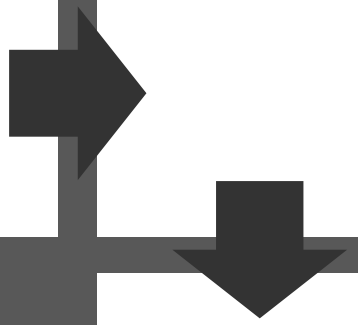
Beispiel: Wortfrequenzen in Dokument ermitteln

- map(key=url, value=content)
  - Gib für jedes Wort *w* in *content* ***(w, „1“)*** aus
- reduce(key=w, values=counts)
  - Summiere alle „1“ in der Liste *values*
  - Liefere Ergebnis *„(wort, summe)“*

MapReduce

Beispiel: Wortfrequenzen in Dokument ermitteln

Ich bin müde
Ich habe fertig



Map

- Ich, 1
- bin, 1
- müde, 1
- Ich, 1
- habe, 1
- fertig, 1

Reduce

- **Ich, 2**
- bin, 1
- müde, 1
- habe, 1
- fertig, 1

MapReduce

Weitere Beispiele

- Zugriffsfrequenz einer URL
  - Map verarbeitet Webserverlogs und liefert *<URL, 1>* pro Request
  - Reduce summiert Werte für gleiche Webseite *<URL, total count>*
- Invertierter Index
  - Map liest Dokumente ein und liefert für jedes Wort *<word, document ID>*.
  - Reduce akzeptiert alle Paare eines bestimmten Wortes, sortiert die *document Ids* und gibt *<word, list(document ID)>* aus.
- Weitere Beispiele in der schriftlichen Ausarbeitung

Inhalt

Einleitung

MapReduce

Hauptteil

Implementierung

Verbesserung

Ergebnis

Abschluss

Fazit, Entwicklung und Ausblick

Implementierung

Überblick

- Google Rechnerverbund aus hunderten bzw. tausenden Rechnern
  - X86 Dualprozessor
  - 2-4 GB Arbeitsspeicher
  - 100 Mbps bzw. 1 Gbps Netzwerkadapter
  - IDE-Festplatte als lokaler Datenspeicher
  - **Hohe Anzahl Rechner = häufige Ausfälle!**

Implementierung

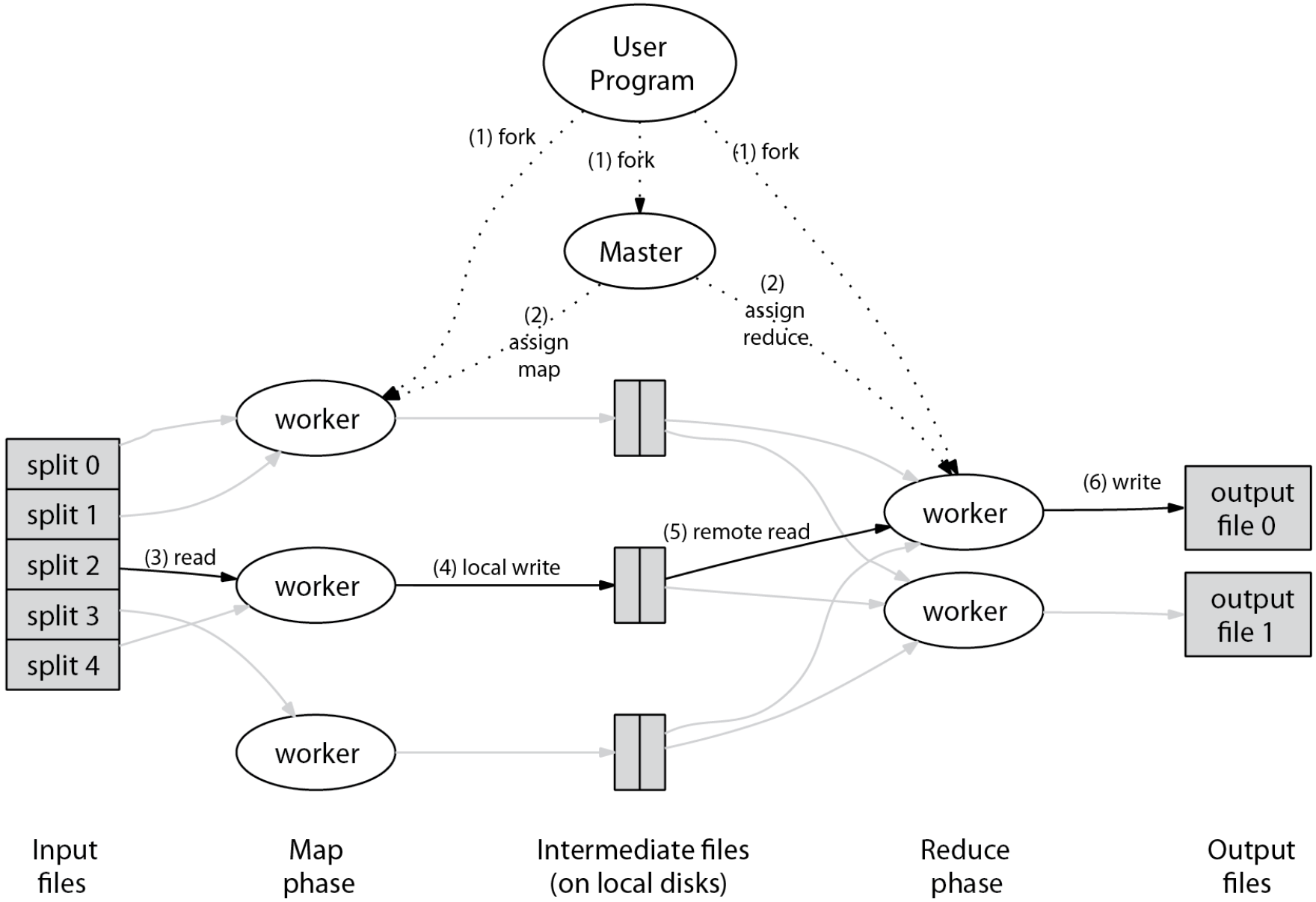
Überblick

- GFS: verteiltes Dateisystem
- Job-Scheduling-System: Jobs bestehen aus Tasks, der Scheduler weist Tasks Rechnern zu
- Implementierung liegt als C++-Bibliothek vor

Implementierung

Ausführung

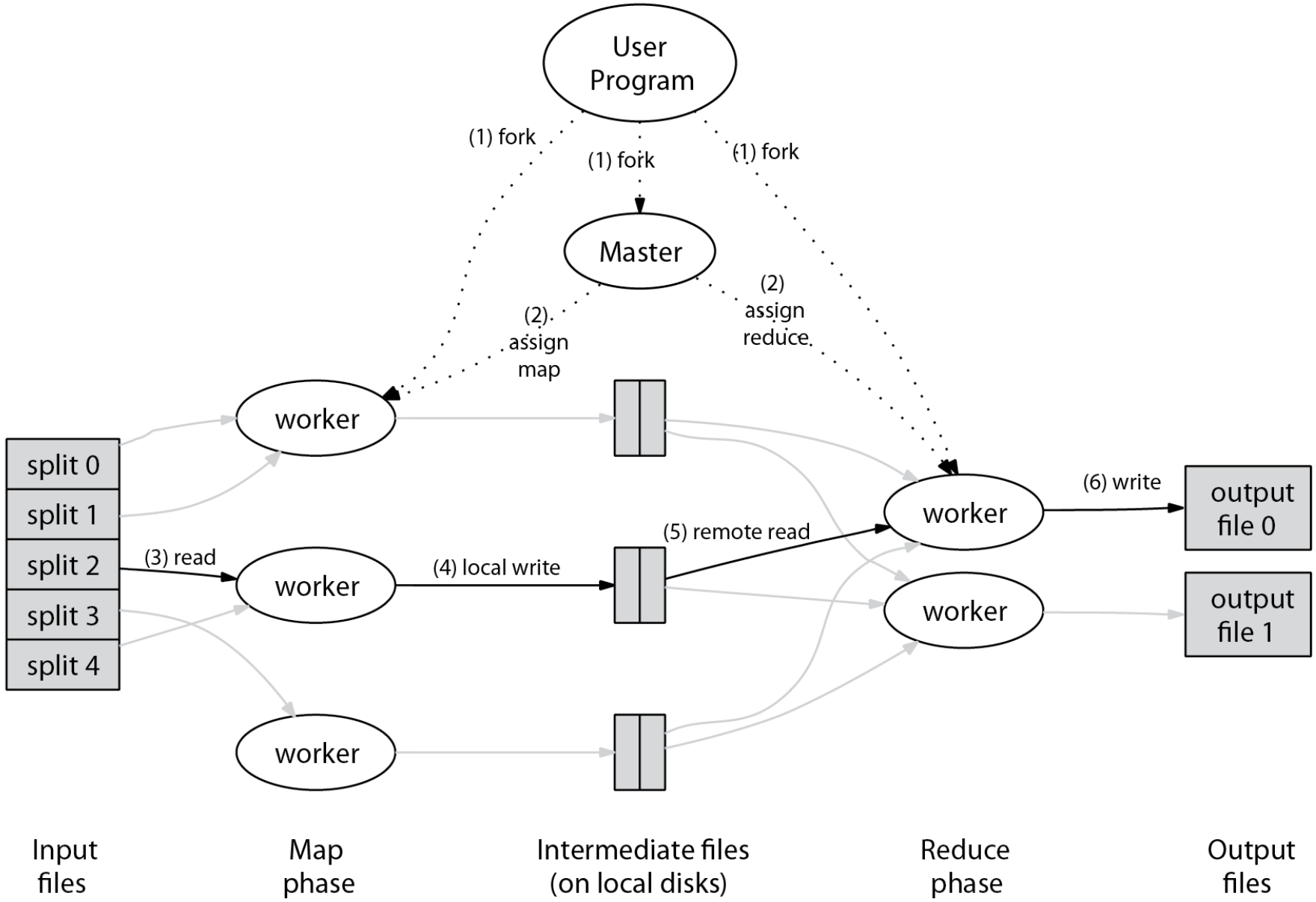
1. Bibliothek zerteilt die Eingabedaten in *M* 16-64 MB große Teile. Viele Kopien des Programms werden gestartet.


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Implementierung

Ausführung

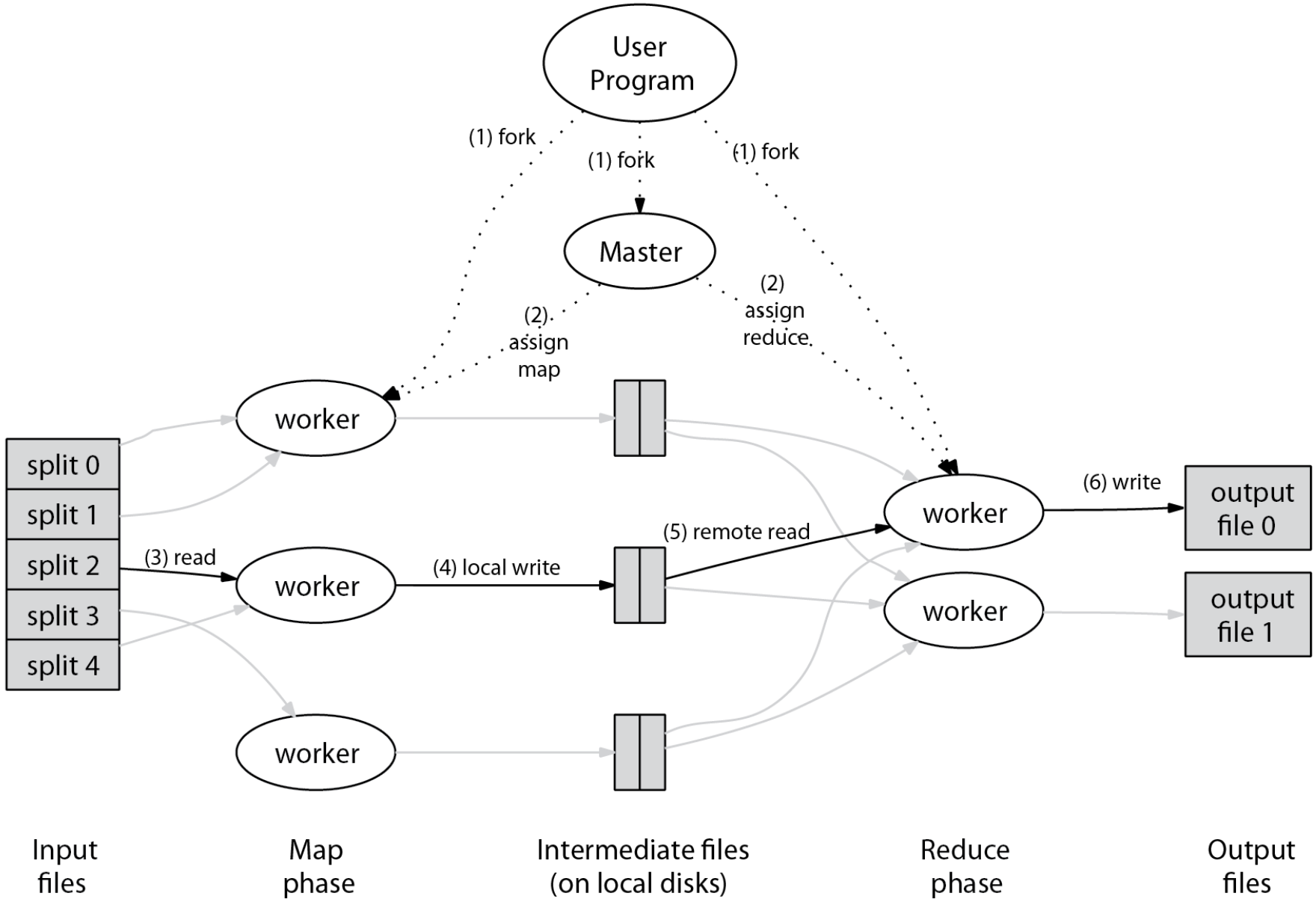
2. Spezielle Kopie Master wird erstellt.
   Alle anderen sind Worker.
   Master weist Map- und Reduce-Tasks freien Workern zu.


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Implementierung

Ausführung

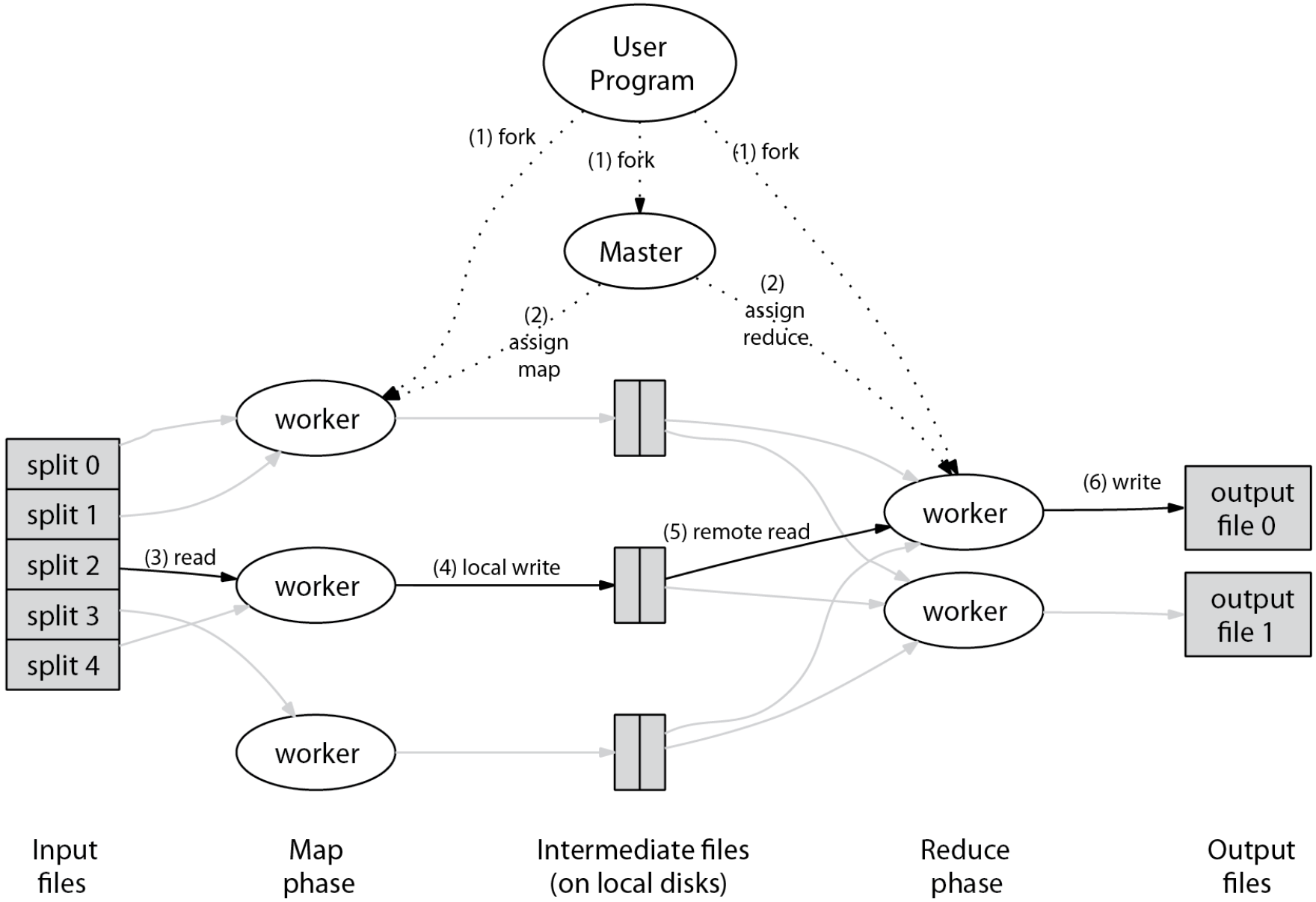
3. Worker mit Map-Task liest korrespondierende Eingabedaten ein.
   Map-Funktion verarbeitet Daten.
   Zwischendaten werden gepuffert.


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Implementierung

Ausführung

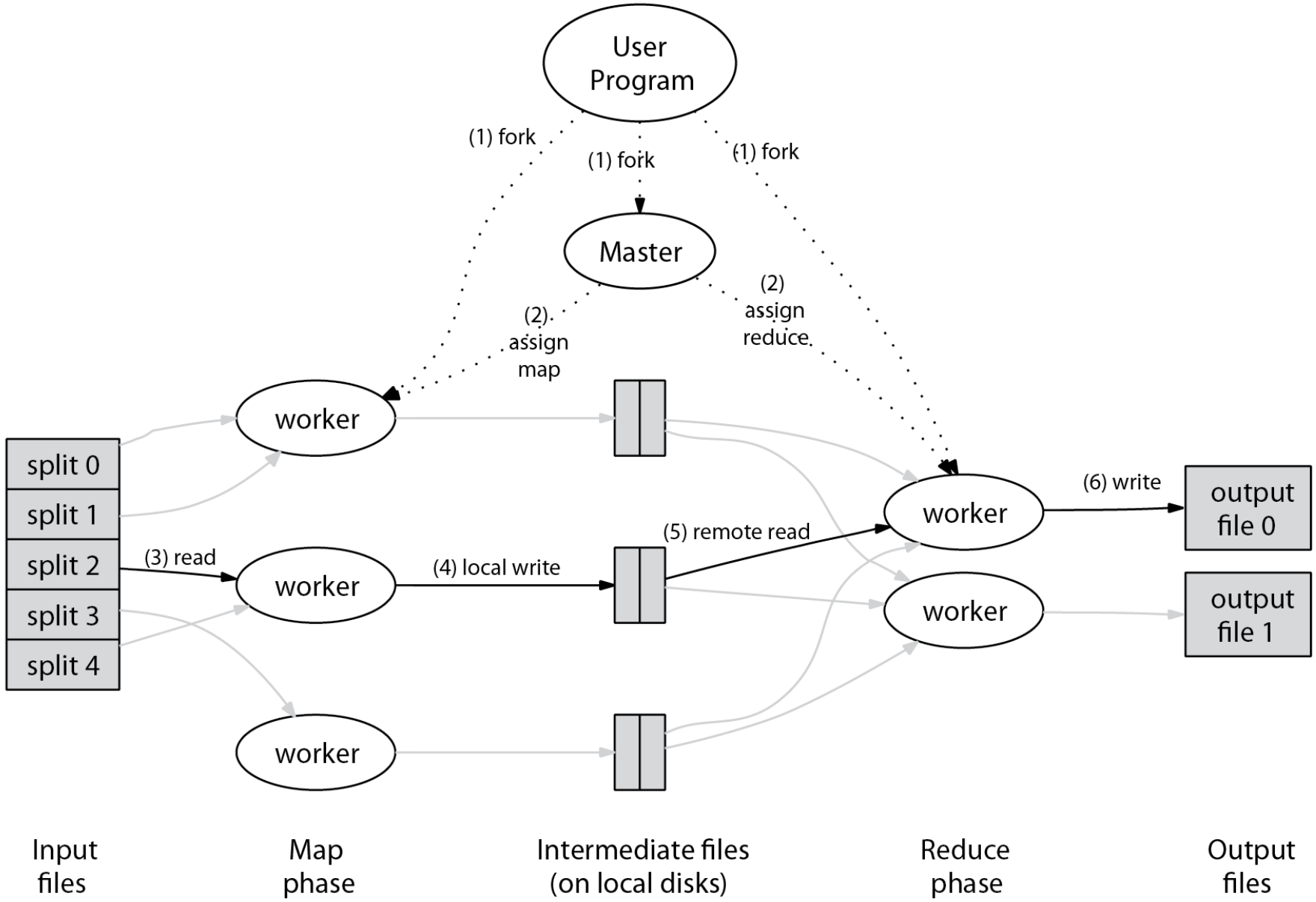
4. Gepufferte Schlüssel/Wert-Paare werden in *R* Teile partitioniert und regelmäßig auf lokaler Festplatte gespeichert.
   Übermittlung der Speicherorte an den Master.
   Master leitet die Information an Reduce-Worker weiter.


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Implementierung

Ausführung

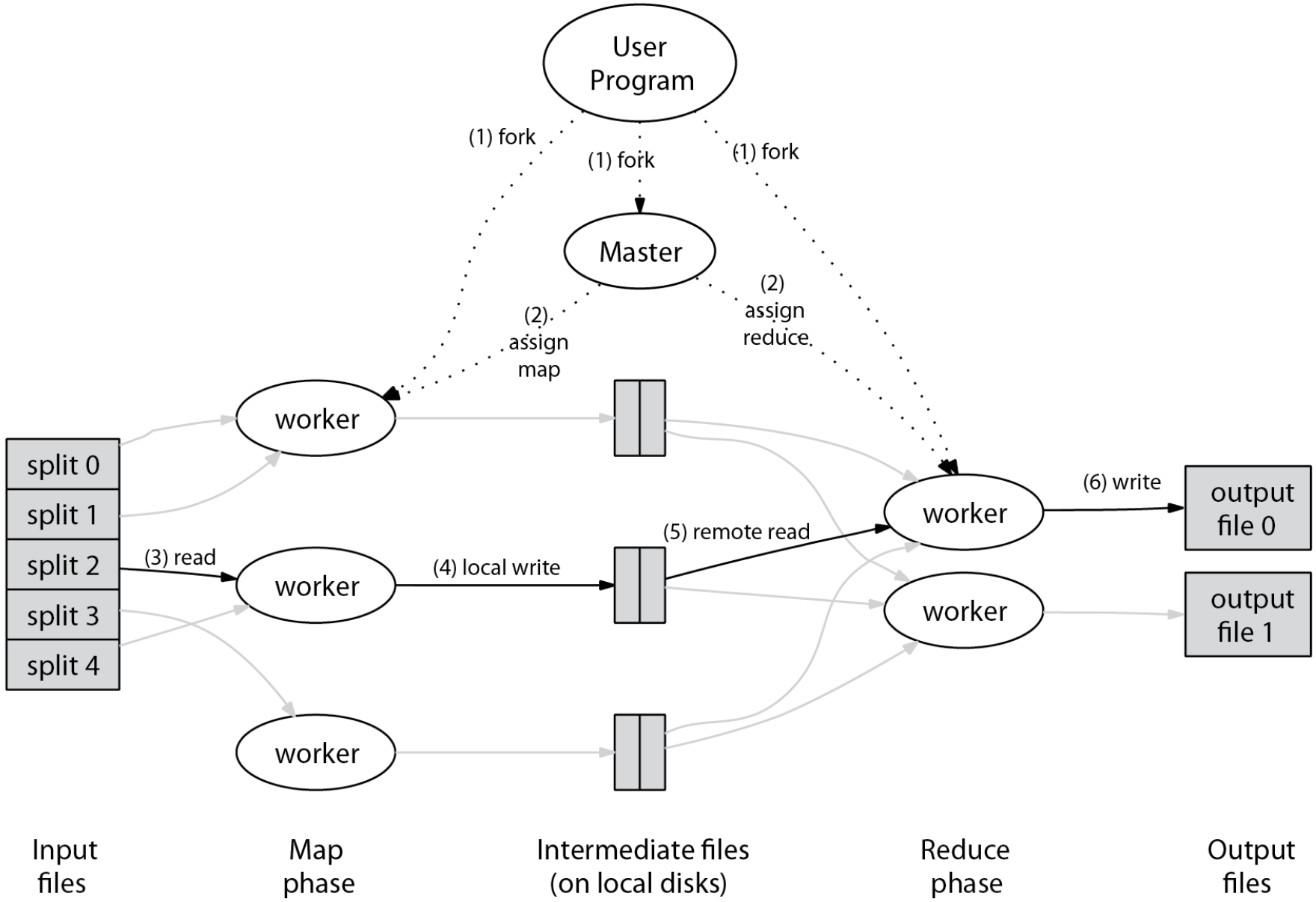
5. Reduce-Worker lesen Daten von Map-Workern via RPC.
   Sortierung der Daten nach Schlüssel.
   Gruppierung übereinstimmender Schlüssel.


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Implementierung

Ausführung

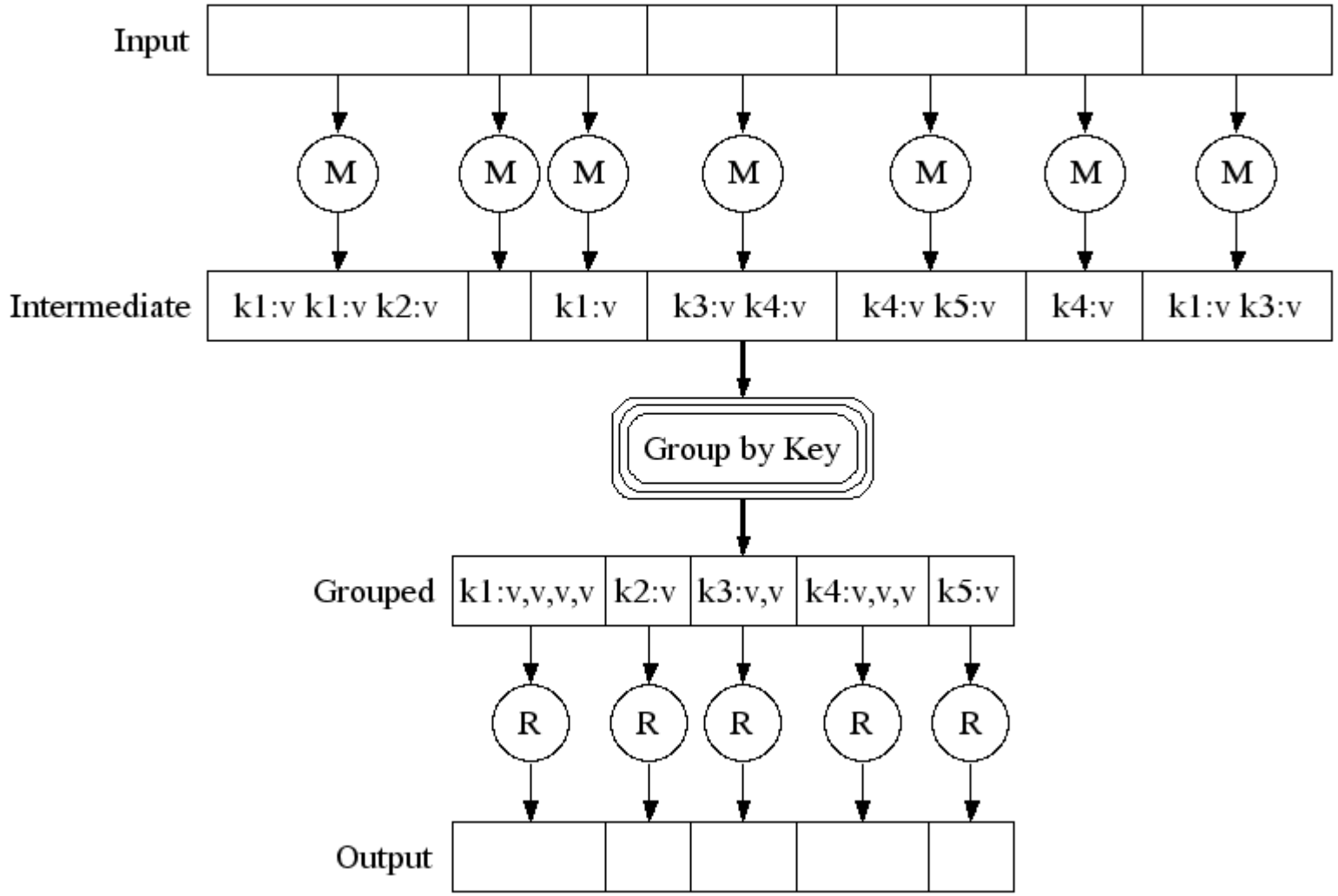
6. Reduce-Worker iteriert über sortierte Daten aus (5).
   Für jedes einzigartige Schlüssel/Wert-Paar wird der Schlüssel und dazugehörige Werte an die Reduce-Funktion übergeben.
   Die Ausgabe wird an die finale Ausgabedatei der Partition angehängt.


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Implementierung

Ausführung

7. Master weckt nach Abschluss aller Tasks das Hauptprogramm. Rückkehr in den Programmcode des Nutzers.


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

Implementierung

Fehlertoleranz

Ausfall Worker

- Master erhält keine Pingantwort von Worker
- Master markiert Worker als *failed*
- Map-Tasks des *failed* Workers werden anderen Workern zugewiesen
- Neuausführung abgeschlossene Map-Tasks -> kein Zugriff auf Festplatte
- Reduce-Tasks werden über den Ausfall informiert -> Einlesen von anderer Quelle

Robust: 1600 von 1800 Rechnern ausgefallen -> erfolgreiche Ausführung

Implementierung

Fehlertoleranz

Ausfall Master

- Unwahrscheinlich, da Master nur einmal vorhanden
- Implementierung bricht Berechnung ab

Inhalt

Einleitung

MapReduce

Hauptteil

Implementierung

Verbesserung

Ergebnis

Abschluss

Fazit, Entwicklung und Ausblick

Verbesserung

Backup-Tasks

Nachzügler: Rechner, die besonders lange für letzte Tasks einer Berechnung brauchen

- Fehlerhafte Festplatte
- Rechner ausgelastet

Zur Vorbeugung werden Tasks in Bearbeitung anderen Workern zugewiesen

- Der Task, der zuerst abgeschlossen wird, liefert das Ergebnis

Effekt

- Signifikante Reduzierung der Berechnungszeit

Verbesserung

Lokalitätsoptimierung

Master Planungsstrategie

- Fragt GFS nach Speicherorten von Kopien der Eingabeblöcke
- Map-Tasks normalerweise in 64MB (GFS-Blockgröße) Teile partitioniert
- Planung von Map-Tasks, so dass die GFS-Eingabeblöcke auf dem selben Rechner oder im selben Rack sind

Effekt

- Tausende Rechner lesen Daten von lokaler Festplatte
- Netzwerklast wird reduziert

Verbesserung

Überspringen fehlerhafter Einträge

Abbruch einer MapReduce-Operation durch bestimmten Eintrag

- Lösung: Debuggen und Reparieren
  - Nicht immer möglich (Bibliotheken von Drittanbietern)

Bei Segmentierungsfehler

- Sende ein UDP-Paket an den Master
- Füge die Sequenznummer des verarbeiteten Eintrags hinzu

Master empfängt zwei Fehler für gleichen Eintrag

- nächster Worker wird angewiesen, den Eintrag zu überspringen

Verbesserung

weitere

Partitionierungsfunktion

- Standardfunktion verwendet Hashing -> gut ausbalancierte Partitionen

Garantie der Reihenfolge

- Zwischendaten werden aufsteigend nach Schlüssel verarbeitet
- Ermöglicht schnelle Sortierung

Kombinierfunktion

- Lokale Zusammenfassung der Zwischendaten
- Reduziert die Netzwerklast

Verbesserung

weitere

Zähler

- Vorgegebene und selbstdefinierte Zähler zur Plausibilitätsprüfung

Statusinformationen

- Master betreibt HTTP-Server
- Darstellung von Informationsseiten zu MapReduce-Jobs

Weitere Verbesserungen in der schriftlichen Ausarbeitung

Inhalt

Einleitung

MapReduce

Hauptteil

Implementierung

Verbesserung

Ergebnis

Abschluss

Fazit, Entwicklung und Ausblick

Ergebnis

Testaufbau

Verbund aus 1800 Rechnern

- 4 GB Speicher
- Dualprozessor 2 GHz Xeon mit Hyperthreading
- 1 Gbps Netzwerkadapter
- Zwei 160 GB IDE-Festplatten
- 100 Gbps Bandbreite am Netzwerkswitch

Ergebnis

Testprogramme

Grep

- Durchsucht 10^{10} Einträge mit 100 Byte Länge
- Match für Suchmuster nur alle 92.337 Einträge
- Eingabe 15.000 Teile zu 64 MB (M = 15.000)
- Ausgabe einer Datei (R = 1)

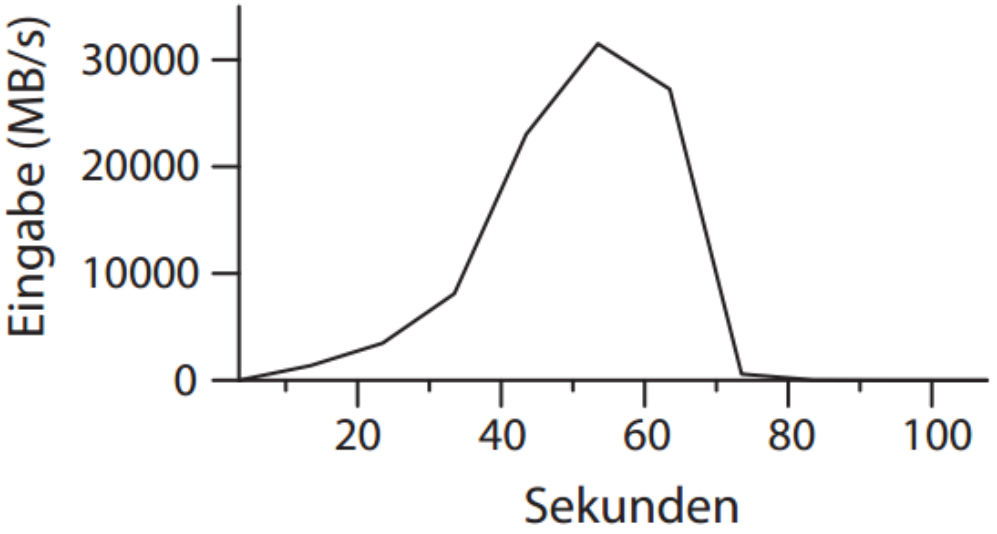
Sort

- Sortiert 10^{10} Einträge mit 100 Byte Länge
- Eingabe 15.000 Teile zu 64 MB (M = 15.000)
- Ausgabe über Partitionierungsfunktion in 4000 Dateien (R = 4000)

Ergebnis

Grep

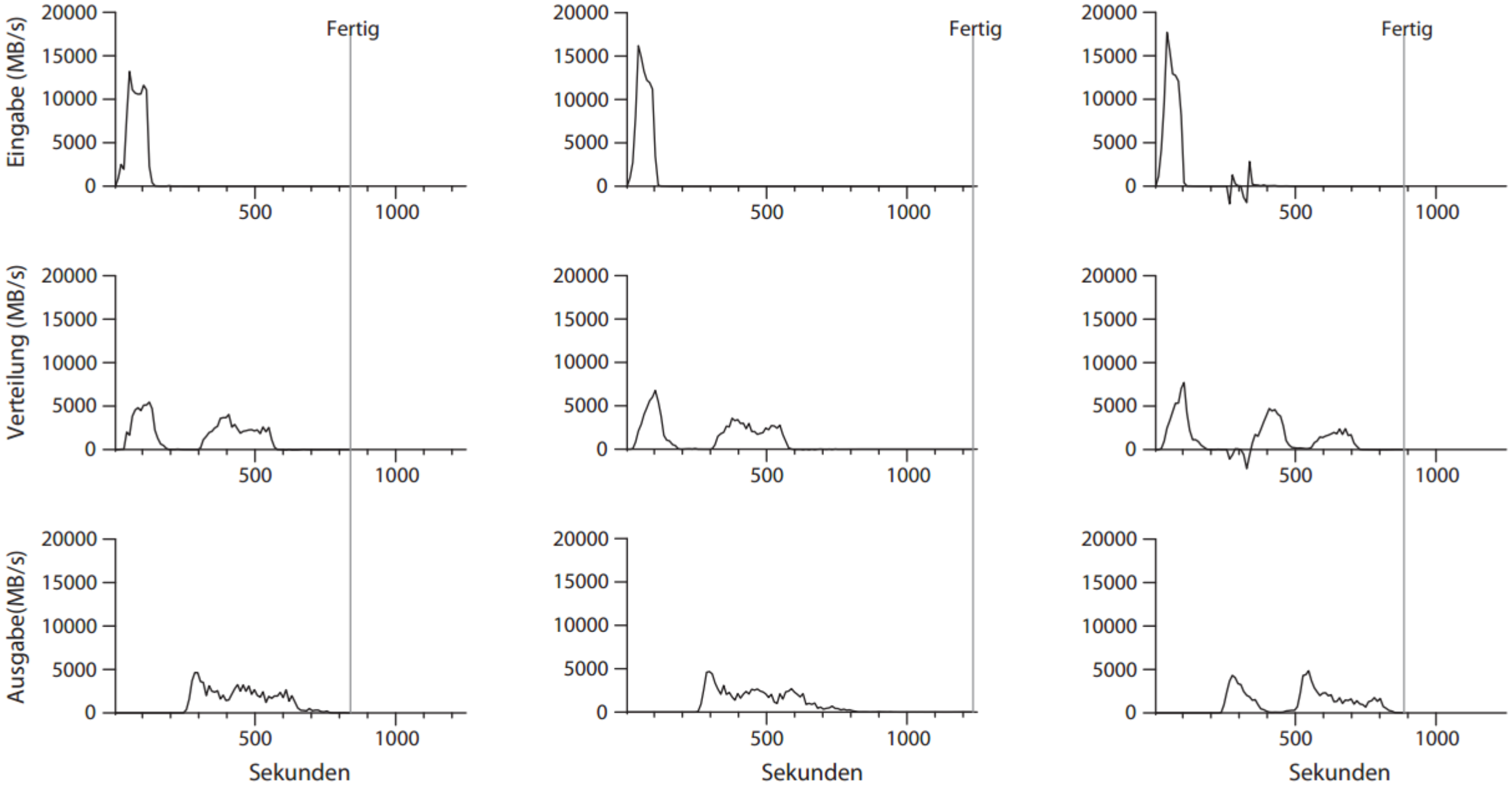
- Höchststand 30 GB/s mit 1764 Rechnern
- Einlesedauer 80 Sekunden
- Gesamtzeit 150 Sekunden



Ergebnis

Sort Testszenarien

- Normale Ausführung (Terasort 2004 Bestwert)
- Verzicht auf Backup-Tasks (Nachzügler)
- Absichtliches beenden von 200 Tasks (Robustheit)



(a) normale Ausführung

(b) keine Backup-Tasks

(c) Kill von 200 Tasks

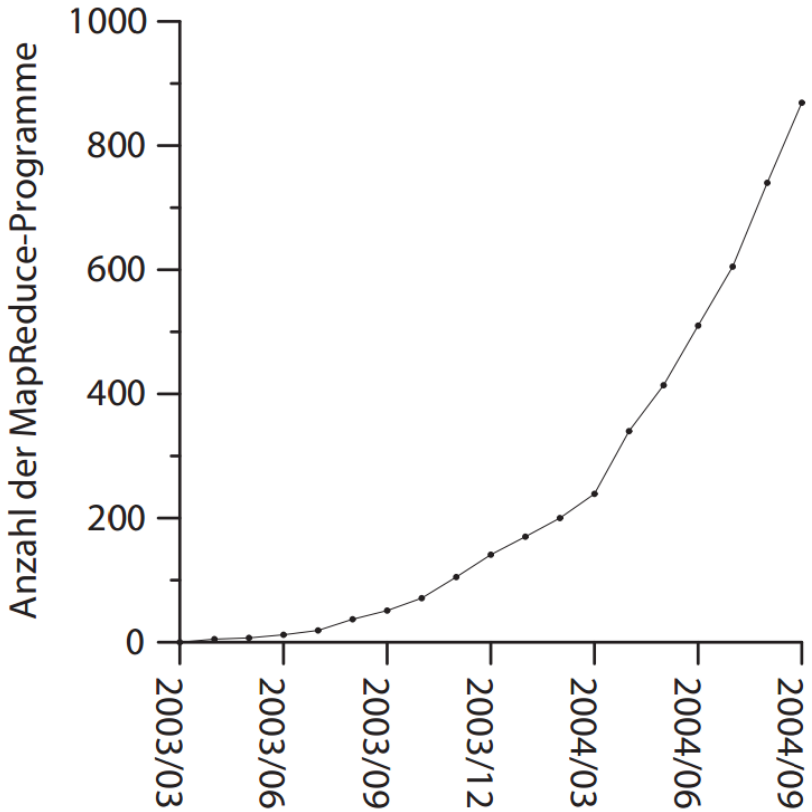
Ergebnis

Google MapReduce-Jobs (August 2004)

Anzahl der Jobs	29.423
Durchschn. Ausführungszeit	634 Sekunden
Maschinentage	79.186 Tage
Gelesene Eingabedaten	3.288 TB
Produzierte Zwischendaten	758 TB
Geschriebene Ausgabedateien	193 TB
Durchschn. Worker pro Job	157
ausgefallene Worker pro Job	1,2
Durchschn. Map-Tasks pro Job	3.351
Durchschn. Reduce-Tasks pro Job	55
Map Implementierungen	395
Reduce Implementierungen	269
Map/Reduce Kombinationen	426

Ergebnis

MapReduce-Programme in Google's Codebasis



Ergebnis

Vorteile von MapReduce

Indexerstellung

- Programmcode ist einfacher, kürzer und leichter zu verstehen (z.B. für eine Phase 700 statt 3800 Zeilen)
- Nicht relevante Berechnungen können separat ausgeführt werden. (ermöglicht einfache Änderungen)
- Indizierung leichter zu benutzen, da Probleme von der MapReduce-Bibliothek behandelt werden.
- Mehr Performanz durch Hinzufügen von Rechnern in Verbund

Inhalt

Einleitung

MapReduce

Hauptteil

Implementierung

Verbesserung

Ergebnis

Abschluss

Fazit, Entwicklung und Ausblick

Fazit, Entwicklung und Ausblick

Fazit

- MapReduce nahezu beliebig skalierbar
- Eingabedaten nur limitiert durch Dateisystem
- Effektive Parallelisierung von Berechnungen auf sehr großen Datensätzen

Fazit, Entwicklung und Ausblick

Ereignisse seit Veröffentlichung 2004

Seit 2008 MapReduce-Framework Hadoop

- Schirmherrschaft: Apache Software Foundation
- Programmiersprache Java
- Verteiltes Dateisystem HDFS
- Hohe Marktakzeptanz (Einsatz bei Facebook, Yahoo, etc.)
- Version 2.0 geplant für Anfang 2013

2010 erhält Google ein Patent auf MapReduce

- Lizenz an die ASF zur freien Nutzung der Technologie

Fazit, Entwicklung und Ausblick

Ausblick

- MapReduce wird weiter an Bedeutung bei der Verarbeitung großer Datenmengen (Big Data) gewinnen
- Komplexe Algorithmen (Apache Mahout) erweitern die Datengewinnung
- Kombination mit verteilten Datenbanken ermöglicht gezielte Abfragen (Google Spanner, Apache Hbase)

Abschluss

Q & A

Ende

Vielen Dank für Ihre Aufmerksamkeit!