Universität Passau

Fakultät für Informatik und Mathematik

Ausarbeitung

MapReduce in der Praxis

Verfasser:

Rolf Daniel

09.12.2010

Zusammenfassung

MapReduce ist ein von *Google* eingeführtes Framework, das Programmierern die Entwicklung von paralleler Software erleichtern soll, indem es die schwierigen Details der Prallelisierung, Fehlertoleranz oder Verteilung der Daten in einer Bibliothek zur Verfügung stellt.

Der Entwickler selbst muss sich nur noch um die Spezifikation der Funktionen *Map* und *Reduce* kümmern. Die Map–Funktion verarbeitet Schlüssel/Wert–Paare und generiert daraus eine Menge von Zwischenergebnissen. Die Reduce–Funktion fügt alle Zwischenergebnisse mit demselben Schlüssel zusammen.

In dieser Arbeit wird das MapReduce–Konzept von Google, vom Programmiermodell über die Implementierung bis hin zu einigen Beispielen, näher vorgestellt. Desweiteren werden drei Implementierungen aus der Praxis betrachtet, die sich von Googles MapReduce nicht grundlegend unterscheiden, jedoch einige Unterschiede aufweisen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

In der heutigen Zeit, müssen große Softwarefirmen mit einer immer größer werdenen Menge an Daten umgehen können. Zum Beispiel muss das Unternehmen Google eine riesige Mengen an Rohdaten verarbeiten, um

- invertierte Indizes zu erstellen
- die Daten in einer graphisch aufbereiteten Form zu repräsentieren (Struktur von Web–Dokumenten als Graph darstellen)
- Statistiken zu erstellen

Die dazu notwendigen Berechnungen sind an sich unkompliziert. Das Problem liegt in der enormen Menge der Eingabedaten. Dadurch müssen die Berechnungen auf hunderttausende von Maschinen verteilt werden, um eine angemessene und möglichst kurze Berechnungszeit zu erreichen.

Eine parallele Verarbeitung der Daten in riesigen Rechner–Clustern (Datenzentren, siehe Abb.1.1) führt jedoch zu neuen Problemen, die die Entwickler von verteilter und paralleler Software bewältigen müssen:

1. Die Verarbeitung der Daten muss parallelisierbar sein.
2. Viele Rechner in einem Cluster führen zu einer höheren Ausfallwahrscheinlichkeit eines Teilsystems.
3. Durch die Verteilung der Daten wird die Netzwerklast erhöht, wenn man z.B. von einem Rechner auf die Daten eines anderen Rechners zugreifen möchte.
4. Die Komplexität für Programme wird durch die Punkte 1.–3. erhöht.

Um diese Probleme in den Griff zu bekommen, hat Google das MapReduce–Konzept [DG08] entwickelt. *MapReduce* ist ein Programmiermodell, das es Entwicklern erlaubt, große Mengen an Daten zu verarbeiten, ohne sich um Dinge wie Parallelisierung der Berechnung, Verteilung der Daten auf die Rechner im Cluster oder Fehlerbehandlung kümmern zu müssen.

In Kapitel 2 werden das grundlegende Programmiermodell und eine Implementierung des MapReduce–Interface genauer vorgestellt. Die gezeigten Beispiele sollen zu einem besseren Verständniss beitragen. Den Schluss des Kapitels bildet die Fehlerbehandlung.

Praktische Umsetzungen des MapReduce–Konzepts werden in Kap.3 anhand von drei Implementierungen vorgestellt: *Disco*, *Hadoop* und *BOOM*. Abschließend folgt eine Zusammenfassung der Arbeit.

Abbildung 1.1: Google–Rack mit 40 Servern
[Quelle: [Sha08]]

Kapitel 2

Googles MapReduce

In diesem Kapitel geht es um das Programmiermodell von Googles MapReduce und eine damit verbundene Implementierung. Ein Paar Beispiele sollen die Anwendungsmöglichkeiten des Konzepts aufzeigen.

2.1 Programmiermodell

Das MapReduce–Framework realisiert eine Funktion. Als Eingabe bekommt die Funktion eine Liste von Schlüssel/Wert–Paaren und als Ausgabe liefert die Funktion wiederum eine Liste von Schlüssel/Wert–Paaren [wik10b]:

$$\begin{aligned} MapReduce \quad : \quad & (K \times V)^* \to (L \times W)^* \\ & [(k_1, v_1), \ldots, (k_n, v_n)] \mapsto [(l_1, w_1), \ldots, (l_m, w_m)] \end{aligned}$$

Dabei enthalten die Mengen K und L Schlüssel und die Mengen V und W Werte. Alle Schlüssel $k \in K$ bzw. $l \in L$ haben jeweils den gleichen Typ. Analog haben alle Werte $v \in V$ bzw. $w \in W$ jeweils den gleichen Typ.

Der Entwickler, der das Framework nutzen möchte, muss nur die Berechnung durch die beiden Funktionen *Map* und *Reduce* spezifizieren [wik10b]:

$$\begin{aligned} Map \quad : \quad & K \times V \to (L \times W)^* \\ & (k, v) \mapsto [(l_1, x_1), \ldots, (l_r, x_r)] \end{aligned}$$

$$\begin{aligned} Reduce \quad : \quad & L \times W^* \to W^* \\ & (l, [y_1, \ldots, y_s]) \mapsto [w_1, \ldots, w_m] \end{aligned}$$

Für die automatische Parallelisierung und Verteilung der Daten, die Fehlerbehandlung und die Kommunikation zwischen den einzelnen Rechnern ist das Framework zuständig. Die gesamte Berechnung lässt sich in zwei Phasen unterteilen:

- Map–Phase
- Reduce–Phase

Map–Phase: Die Map–Funktion erhält als Eingabe ein Paar, bestehend aus einem Schlüssel k und einem Wert v. Dieses Paar wird auf eine Menge von Zwischenergebnissen (l_r, x_r) abgebildet. Dabei sind die Werte x_i vom selben Typ wie die Endergebnisse w_j. Die Map–Funktion wird für jedes Paar in der Eingabeliste aufgerufen. Da diese Aufrufe unabhängig von einander sind, können sie parallel auf einem Rechner–Cluster ausgeführt werden. Das Framework ist zuständig für die Gruppierung der Zwischenergebnisse nach ihrem Schlüssel. Alle Werte y_k, die denselben Schlüssel l besitzen, werden in einer Liste zusammengefasst $(l, [y_1, \ldots, y_s])$.

Reduce–Phase: Die gruppierten Zwischenergebnisse werden dann an die Reduce–Funktion weitergegeben. Das Framework ruft für jedes Paar $(l, [y_1, \ldots, y_s])$ die Reduce–Funktion auf. Diese berechnet dafür einen Ergebniswert w_j. Die einzelnen Werte w_j werden in einer Ausgabeliste $[w_1, \ldots, w_m]$ gespeichert. Da die Aufrufe der Reduce–Funktion ebenfalls unabhängig von einander sind, können sie ebenfalls parallel ausgeführt werden.

2.2 Implementierung

Für die Implementierung des MapReduce–Interfaces gibt es verschiedene Möglichkeiten, die abhängig von der zu Grunde liegenden Umgebung sind. In diesem Abschnitt wird eine Implementierung vorgestellt, die auf einer Umgebung, die weitestgehend bei Google genutzt wird, basiert [DG08][1]:

- großer Cluster, bestehend aus mehreren hundert/tausend Maschinen, die über ein Netzwerk miteinander verbunden sind
- einzelne Maschine: 2–CPU x86, 2–4 GB Arbeitsspeicher, Linux als Betriebssystem
- Standardnetzwerkhardware 100 Mb/s oder 1 Gb/s Ethernet
- Speicherung von Daten lokal auf IDE (Integrated Device Electronics) Festplatten
- GFS (Google File System) zur Verwaltung der Daten
- Scheduling–System für Verarbeitung und Verteilung von Aufgaben

2.2.1 Google File System

Das GFS ist ein skalierbares, verteiltes Dateisystem, das für große, verteilte und datenintensive Applikationen entwickelt wurde [GGL03]. In Abb.2.1 ist die Architektur dieses Dateisystems zu sehen.

[1] Stand 2004

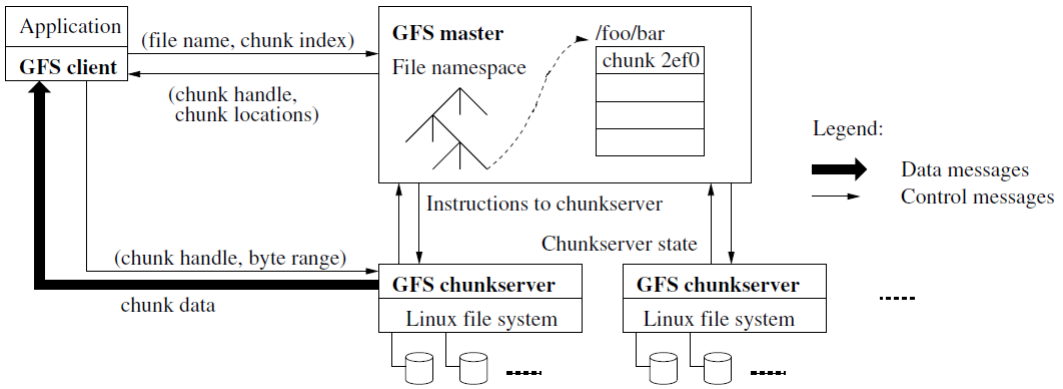

Abbildung 2.1: GFS–Architektur
[Quelle: [GGL03]]

Ein GFS–Cluster besteht aus genau einem *Master* und mehreren *Chunkservern*. Auf diesen Cluster können mehrere *Clients* zugreifen. Der Master, die Chunkserver und die Clients sind typischerweise Linux–Maschinen, wobei sich der Master und ein Chunkserver auch auf der gleichen Maschine befinden können.

Daten, die im Dateisystem gespeichert werden sollen, werden in *Chunks* fester Größe aufgeteilt. Jeder Chunk wird durch einen eindeutigen 64–Bit Identifier identifiziert, den so genannten *chunk handle*. Dieser Identifier wird den einzelnen Chunks während der Erstellung vom Master zugewiesen.

Um die Verfügbarkeit der Daten zu erhöhen, werden die Chunks repliziert und auf verschiedenen Chunkservern lokal gespeichert. Für die Verteilung der Chunks auf die Chunkserver ist der Master zuständig. Dieser muss auch Metadaten des Dateisystems pflegen, wie z.B. die Abbildung von Daten auf Chunks oder den aktuellen Speicherort der Chunks. Der Master kommuniziert periodisch mit den Chunkservern über ein sogenanntes *Heartbeat–Protokoll*, um Instruktionen an die Chunkserver zu schicken, ihren Zustand zu überwachen und Operationen auf Metadaten durchzuführen. Clients kommunizieren direkt mit den Chunkservern, wenn sie Daten lesen oder schreiben wollen.

Wie die einzelnen Komponenten miteinander interagieren soll anhand eines kurzen Beipiels genauer erleutert werden. Hier wird ein Chunk von einem Client gelesen:

1. Der Client schickt eine Anfrage, die den gewünschten Dateinamen und den entsprechenden Chunkindex innerhalb der Datei enthält, an den Master.

2. Der Master antwortet mit einer Nachricht, die den Identifier (chunk handle) und die Speicherorte (chunk locations) des Chunks enthält.

3. Der Client schreibt diese Informationen in seinen Cache und schickt dann eine Anfrage an eine der Maschinen, auf der eine der Chunk–Kopien gespeichert ist (meist die Machine, die am nächsten ist). Die Anfrage enthält den *chunk handle* und einen Byte–Bereich innerhalb des Chunks.

4. Zum Schluss werden die Daten des Chunks an den Client übertragen.

2.2.2 MapReduce: Master und Worker

Die einzelnen Maschinen innerhalb des Clusters haben unterschiedliche Aufgaben und werden deshalb in *Master* und *Worker* unterschieden.

- Master:
  - Es gibt nur einen pro MapReduce–Operation.
  - Er weist den Workern, die gerade keine Arbeit haben, Aufgaben (Tasks) zu.
  - Er Speichert die Zustände der Worker: *idle*, *in–progress* oder *completed*.
  - Er Informiert Reduce–Worker über Zwischenergebnisse, die von den Map–Workern berechnet werden.
- Worker:
  - Ihre Anzahl ist beliebig.
  - Sie Bekommen Tasks und Daten zugewiesen.
  - Sie Führen Map– bzw. Reduce–Tasks durch.

2.2.3 Ablauf einer Berechnung

In diesem Abschnitt wird der Ablauf einer MapReduce–Berechnung genauer betrachtet (siehe Abb.2.2). Die Eingabedaten werden bereits im Vorfeld durch das GFS auf die Maschinen innerhalb des Clusters verteilt. Auf diese Informationen – welche Daten sind auf welcher Maschine lokal gespeichert – kann der Master zugreifen und sie später bei der Verteilung der Aufgaben auf die einzelnen Worker nutzen.

Eine Berechnung besteht aus folgenden Aktionen. Ihre Nummerierung entspricht den Nummern in Abb.2.2:

1. Das Benutzerprogramm kontaktiert den Master und teilt ihm unteranderem mit, welche Eingabedaten für die Berechnung benötigt werden. Die Informationen werden in Form von Metadaten an den Master übertragen. Dabei findet kein großer Datentransfer statt.

2. Der Master weist den Map–Workern je einen Map–Task (mit den zugehörigen Eingabedaten) zu, dabei versucht er einen Worker zu wählen, der bereits eine Kopie der Daten auf seiner lokalen Festplatte gespeichert hat. Gelingt dies nicht, versucht der Master einen anderen Worker, der die gewünschten Daten gespeichert hat, zu finden. Dieser sollte sich in der Nähe (das heisst er ist mit demselben Switch verbunden) befinden. Dadurch werden die meisten Eingabedaten für die MapReduce–Berechnung von den Map–Workern lokal gelesen und müssen nicht über das Netzwerk geholt werden.

3. Die Map–Worker führen auf den Daten, die ihnen vom Master zu gewiesen werden, die Map–Funktion aus und berechnen daraus Schlüssel/Wert–Paare, die als Zwischenergebnisse gepuffert werden.

4. Die gepufferten Zwischenergebnisse werden von den Map–Workern periodisch auf ihre lokale Festplatte gespeichert und für die Reduce–Worker partitioniert. Die Map–Worker teilen dem Master den Speicherort der Daten mit, damit dieser die Informationen an die Reduce–Worker weiterleiten kann.

5. Sobald ein Reduce–Worker Informationen über den Speicherort der Zwischenergebnisse erhalten hat, liest er diese per RPC (Remote Procedure Call) und sortiert sie anhand des Schlüssels.

6. Die Reduce–Worker iterieren über die sortierten Daten und rufen für jeden Schlüssel und die zughörigen Werte die Reduce–Funktion auf. Die Ergebnisse aller Reduce–Funktionen eines Reduce–Workers werden zu einem Endergebniss konkateniert, das innerhalb des GFS gespeichert wird.

Sind alle Map– und Reduce–Tasks abgeschlossen, wird das Benutzerprogramm vom Master informiert, dass der MapReduce–Aufruf beendet ist. Die Ergebnisse der MapReduce–Berechnung befinden sich dann in den Ausgabedateien der einzelnen Reduce–Worker. Pro Reduce–Worker gibt es eine Ausgabedatei. Wie dann mit diesen Ergebnisse fortgefahren wird, ist abhängig von der Anwendung. Oftmals werden sie als Eingabe für eine andere MapReduce–Berechnung verwendet.

2.3 Beispiele

Die Beispiele in diesem Abschnitt sollen die Anwendung des MapReduce–Konzepts genauer darstellen.

2.3.1 Wörter in einem Dokument zählen

Folgende Problemstellung ist gegeben:

> „Man möchte die Häufigkeit des Auftretens von Wörtern in einem Dokument bestimmen.“

Dazu muss der Entwickler die beiden Funktionen *map* und *reduce* implementieren [DG08]. Die Funktionen werden hier nur als Pseudo–Code spezifiziert:

```
map(String key, String value):
  // key: number of the part of a sentence
  // value: content of the part of a sentence
  for each word w in value:
    EmitIntermediate(w,"1");
```

Die Map–Funktion erhält als Eingabe einen Schlüssel, der einen Teil eines Satzes des Dokuments repräsentiert, und den dazugehörigen Inhalt. Die Ausgabe der Funktion bildet ein Paar aus einem Wort und dem dazugehörigen Auftreten (in diesem einfachen Beispiel „1“; wenn also ein Wort in einem Satzteil öfter vorkommt, wird jeweils ein Paar (Wort,“1“) ausgegeben). Die Reduce–Funktion summiert für jedes Wort alle Auftreten.

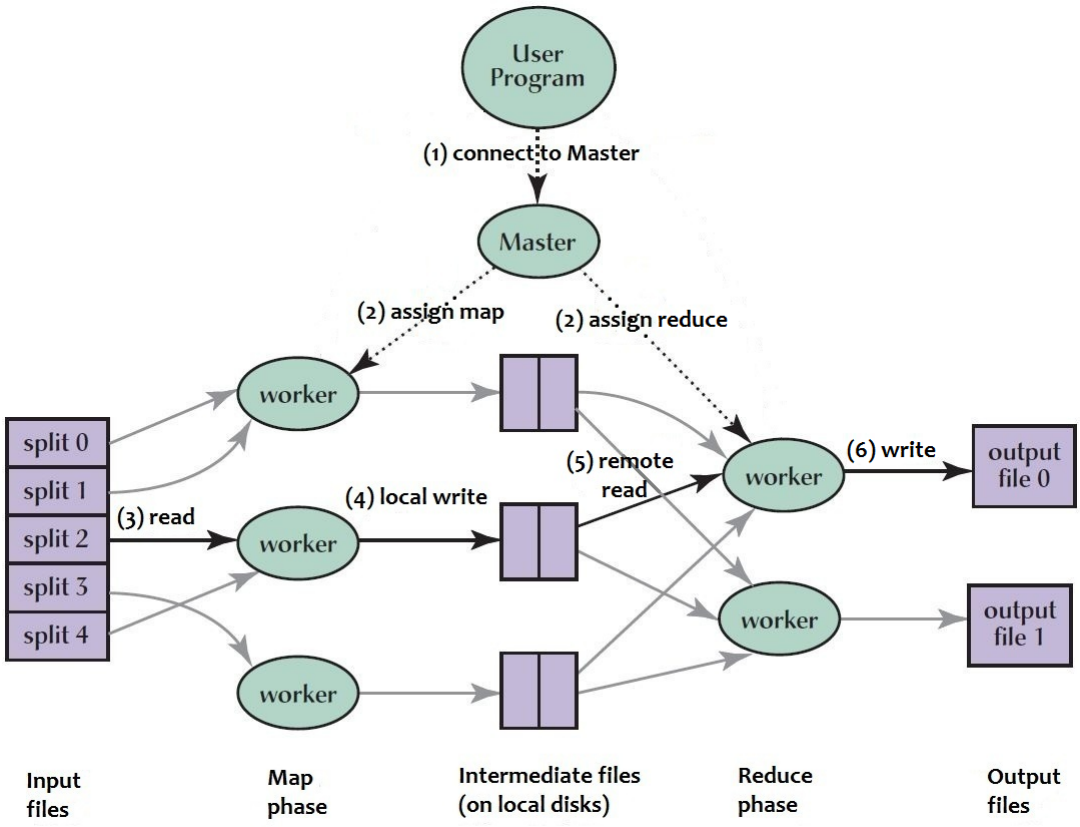

Abbildung 2.2: Ablauf einer MapReduce–Berechnung
[Quelle: [DG08] (leicht abgeändert)]

```
reduce(String key, Iterator values):
  // key: word
  // values: an iterator of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Als Eingabedaten für die MapReduce–Operation sei folgendes Dokument gegeben. Dabei werden Groß– und Kleinschreibung sowie Satzzeichen vernachlässigt:

„Here, each document is split in words, and each word is counted.“

Der erste Schritt ist die Zerlegung des Satzes in mehrere Satzteile, dabei erhält jeder Satzteil eine eindeutige Nummer (Key). Ergebnis sind Schlüssel/Wert–Paare, wobei die Werte dem Inhalt eines Satzteils entsprechen:

- (1,“Here, each“)
- (2,“document is“)
- (3,“split in“)
- (4,“words, and“)
- (5,“each word“)
- (6,“is counted“)

Diese Paare werden nun vom Master auf die verfügbaren Map–Worker verteilt, dabei erhält jeder Map–Worker mehrere Paare. Die einzelnen Map–Worker bilden dann ihre Eingabedaten mit Hilfe der Map–Funktion (vgl. Abb.2.3 obere Hälfte) auf Schlüssel/Wert–Paare ab und puffern diese als Zwischenergebnisse.

Die gepufferten Daten werden periodisch auf die lokale Festplatte der Map–Worker geschrieben und für die Reduce–Worker anhand des Schlüssels partitioniert (vgl. Abb.2.3 untere Hälfte). Dazu wird eine Partitionierungsfunktion genutzt, die abhängig von der Anzahl der Reduce–Tasks (R) ist. Die Anzahl der Reduce–Tasks wird vom Benutzer vor der MapReduce–Berechnung festgelegt (hier $R = 2$). Als Default–Partitionierungsfunktion wird eine Hash–Funktion genutzt (*hash(key) mod R*).

Als nächstes werden die partitionierten Daten, die sich noch auf den Festplatten der Map–Worker befinden, von den Reduce–Workern per RPC gelesen (auch hier wird einem Reduce–Worker vom Master ein Reduce–Task zugewiesen). Abb.2.4 zeigt, wie ein Reduce–Worker die gelesenen Daten sortiert, gruppiert und darauf die Reduce–Funktion anwendet. Die Ergebnisse aller Reduce–Tasks bilden zusammen das Endergebniss der MapReduce–Berechnung. In Abb.2.5 ist noch einmal die gesamte MapReduce–Berechnung zu sehen.

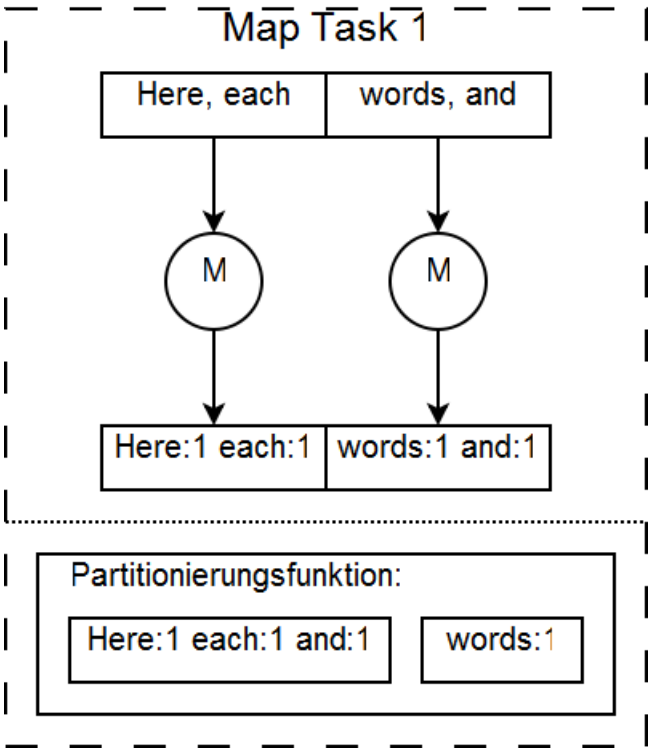

Abbildung 2.3: Map–Worker führt Map–Task aus

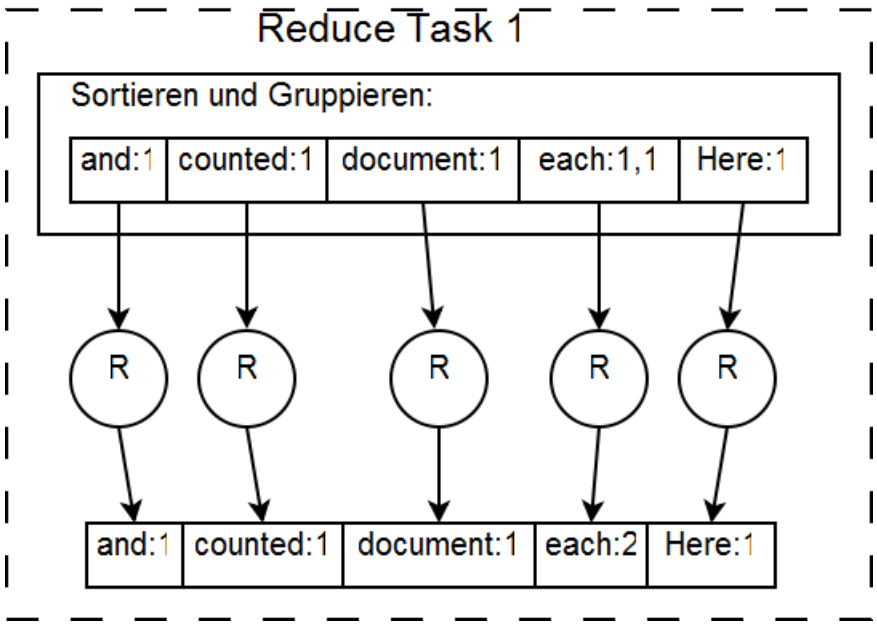

Abbildung 2.4: Reduce–Worker führt Reduce–Task aus

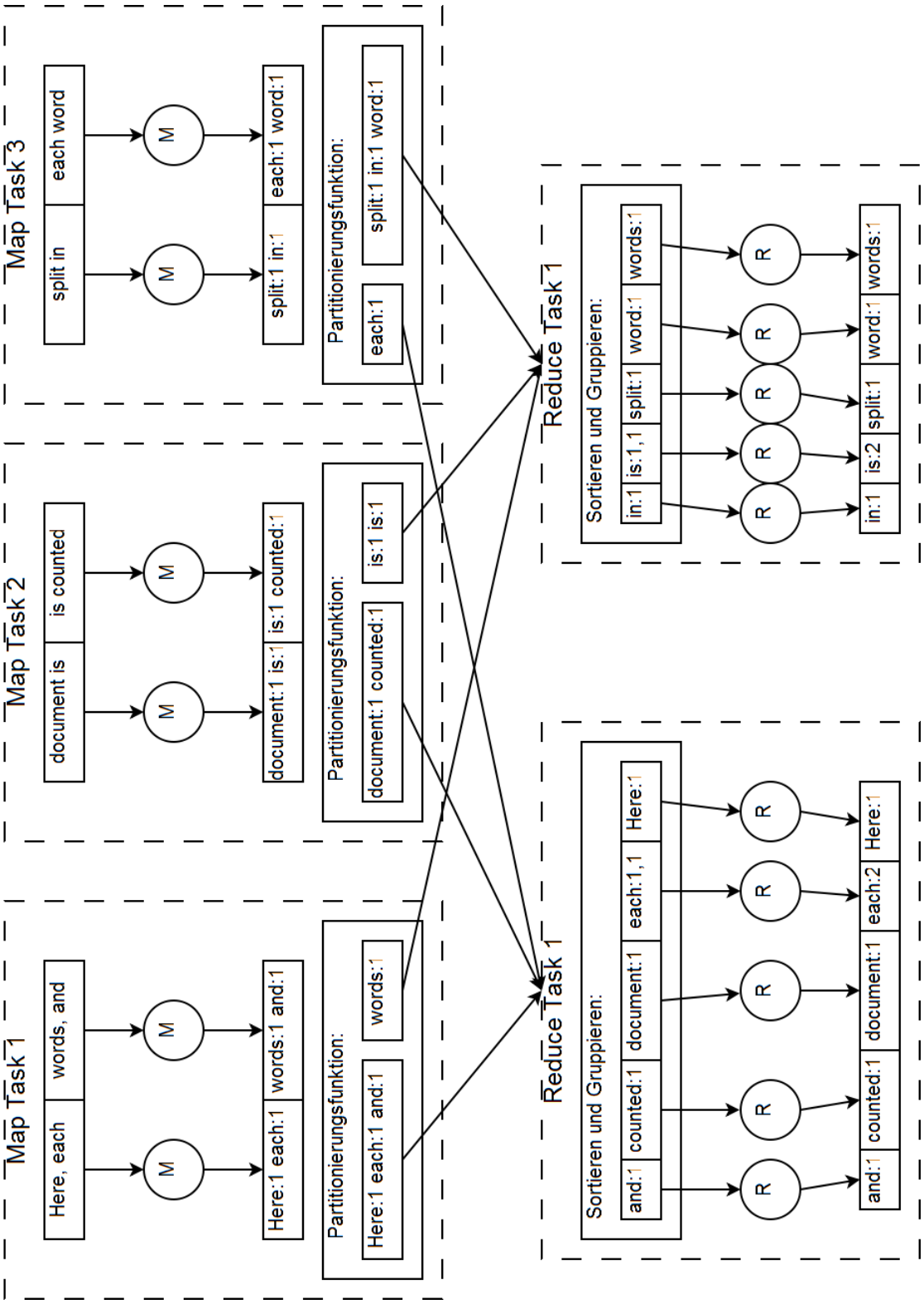

Abbildung 2.5: Wörter in einem Dokument zählen

2.3.2 Weitere Beispiele

Es gibt neben diesem einfachen Beispiel eine Vielzahl weiterer Anwendungsmöglichkeiten für das MapReduce–Konzept:

Verteiltes Grep: Die Eingabedaten werden nach einer bestimmten Zeichenfolge durchsucht. Die Map–Funktion gibt die Zeile aus, in der die Zeichenfolge in den Daten erkannt wird. Die Identitätsfunktion wird als Reduce–Funktion verwendet, d.h. sie bildet die Zwischenergebnisse der Map–Funktionen einfach auf die Endergebnisse ab.

Invertierter Index: Die Map–Funktion erhält als Eingabe ein Dokument und erzeugt eine Sequenz von Paaren $\langle Wort, DokumentID \rangle$. Die Reduce–Funktion erhält alle Paare für ein bestimmtes Wort als Eingabe, sortiert die Dokument–IDs und gibt das Paar $\langle Wort, Liste(DokumentIDs) \rangle$ zurück. Die Ausgaben aller Reduce–Funktionen bilden somit einen invertierten Index.

Term–Vektor: Ein Term–Vektor dient dazu, die häufigsten Schlagwörter, die in einem oder mehreren Dokumenten auftreten, darzustellen. Er besteht aus einer Liste von Paaren $\langle Wort, Auftreten \rangle$. Die Map–Funktion erzeugt für jedes Eingabedokument einen solchen Term–Vektor und gibt ein Paar $\langle Hostname, TermVektor \rangle$ zurück, dabei wird der Hostname aus dem URL (Uniform Resource Locator) des Dokuments extrahiert. Die Reduce–Funktion bekommt für einen Hostnamen eine Menge von Term–Vektoren. Diese werden zusammengefügt, wobei selten auftretende Terme gelöscht werden. Die Ausgabe der Reduce–Funktion bildet dann genau ein Paar, bestehend aus $\langle Hostname, TermVektor \rangle$.

Reverse Web–Link Graph: Für jeden Link (Ziel), denn man auf einer bestimmten Seite (Quelle) findet, erstellt die Map–Funktion ein Paar $\langle Ziel, Quelle \rangle$. Die Reduce–Funktion fügt alle Quell–URLs zu einer Ziel–URL in eine Liste ein und gibt das Paar $\langle Ziel, Liste(Quellen) \rangle$ zurück.

2.4 Ausfallbehandlung

Im Hinblick auf die große Anzahl der Maschinen, die für MapReduce–Operationen genutzt werden und die dadurch erhöhte Wahrscheinlichkeit an Ausfällen von Maschinen, muss eine gewisse Fehlertoleranz durch das MapReduce–Framework sichergestellt werden. Daher schickt der Master periodisch an jeden Worker eine Ping–Nachricht. Antwortet ein Worker nicht innerhalb einer gegebenen Zeitspanne, wird er vom Master als „*failed*“ gekennzeichnet. Alle Map–Tasks, die bereits von diesem Worker fertiggestellt wurden oder noch in Bearbeitung sind, müssen zurückgesetzt werden. Dies ist notwendig, da die Zwischenergebnisse lokal auf der Festplatte des Workers gespeichert wurden und ein Reduce–Worker nicht mehr darauf zugreifen kann.

Alle Reduce–Tasks, die schon abgeschlossen wurden, brauchen nicht wiederholt werden, da ihre Ausgabedateien bereits im GFS gespeichert wurden. In Bearbeitung befindliche Reduce–Tasks müssen natürlich schon wiederholt werden. Betroffene Map– bzw. Reduce–Tasks werden vom Master auf andere, verfügbare Worker erneut verteilt und ausgeführt.

Nun kann es auch passieren, dass der Master ausfällt. Dies hat zur Folge, dass die gesamte MapReduce–Operation abgebrochen wird. Es ist möglich, periodisch einen so genannten *Checkpoint* zu erstellen, der den aktuellen Zustand des Masters speichert. Von diesem Zustand aus kann im Fehlerfall ein neuer Master gestartet werden.

Da jedoch die Wahrscheinlichkeit, dass der Master ausfällt, nur sehr gering ist, wird auf diese Möglichkeit verzichtet [DG08].

Kapitel 3

MapReduce–Implementierungen

Dieses Kapitel soll einen kleinen Überblick über vorhandene Implementierungen des MapReduce–Konzepts verschaffen, wobei hier nur auf *Disco*, *Hadoop* und *BOOM* näher eingegangen wird.

3.1 Disco

Das Disco Projekt [NRC10] ist eine Open–Source Implementierung des MapReduce–Konzepts in *Erlang* und *Python* und wurde vom *Nokia Research Center* entwickelt. Disco nutzt ein verteiltes Dateisystem namens DDFS (Disco Distributed Filesystem). Dies ist dem GFS sehr ähnlilch, besitzt aber einige Besonderheiten.

3.1.1 Disco Distributed Filesystem

Das DDFS wurde, ebenso wie das GFS, für die Speicherung und Verarbeitung großer Datenmengen entwickelt. Es ist verantwortlich für die Verteilung, die Replikation, die Persistenz und die Adressierung der Daten, sowie für den Zugriff auf die Daten. Das Disco–Framework speichert die Ergebnisse der MapReduce–Operationen im DDFS und ermöglicht einen einfachen Zugriff darauf.

Eine Besonderheit des Dateisystems ist, dass es *Tag–basiert* ist. Das bedeutet:

- Daten werden nicht in einer Verzeichnishierarchie organisiert, stattdessen werden Mengen von Objekten mit beliebigen Namen gekennzeichnet, um sie später anhand des *Tags* wiederzufinden. Tags können beispielsweise genutzt werden, um unterschiedliche Versionen von Daten zu kennzeichnen.

- Tags können eine Referenz (Link) auf andere Tags enthalten und somit einen gerichteten Graphen, der Metadaten beinhaltet, bilden. Dies erlaubt die Bildung von Tag–Hierarchien.

- Beliebige Attribute, wie z.B. der Datentyp, können in Form von Tags im DDFS gespeichert werden.

- Tags dienen zum Zugriff auf die Daten, die sie referenzieren.

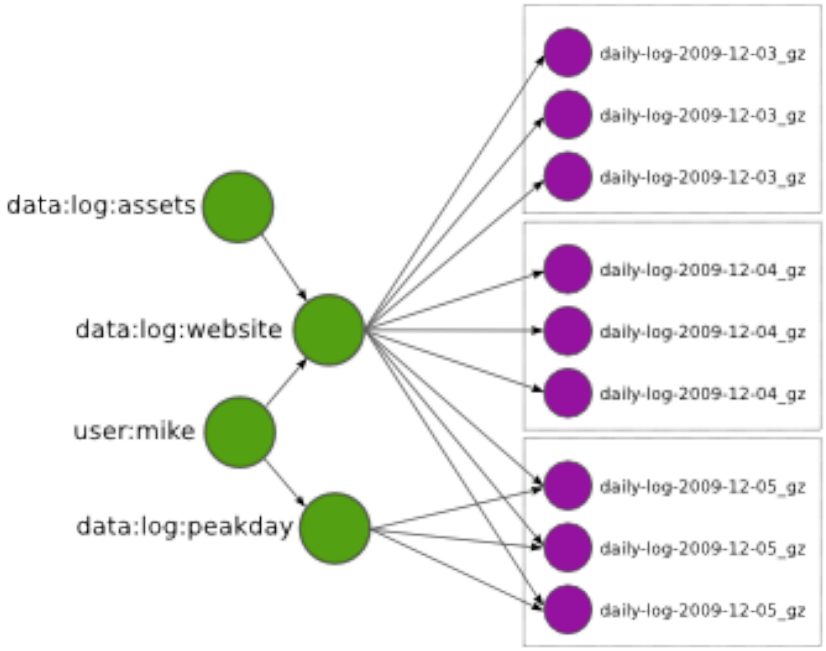

Abbildung 3.1: Blob/Tag–Konzept [NRC10]

In Abb.3.1 werden die Tags mit grünen Kreisen dargestellt. Sie beinhalten Metadaten über die im Dateisystem gespeicherten Daten, hier sind es Log–Dateien (lila Kreise), die eine einfache Abfrage der Log–Informationen erlauben. Eine einzelne Log–Datei wird dabei repliziert (hier: 3–mal) und bildet mit ihren Kopien einen so genannten *Blob* (Kästchen), der auf mehrere Knoten im Cluster verteilt wird. Die Replikation und Verteilung auf verschiedene Knoten gewährleisten eine hohe Verfügbarkeit und Fehlertoleranz des Systems.

Die Implementierung des DDFS ist der des GFS sehr ähnlich (siehe Abb.3.2). Auch hier wird das Dateisystem durch einen Master koordiniert (bei Bedarf können auch mehrere Master verwendet werden, abhängig von der Anwendung). Dieser ist zuständig für Operationen auf den Metadaten. Neben dem Master gibt es noch die Speicherknoten. Diese besitzen meist mehrere Speichermedien $(vol0, vol1, \ldots, volN)$ und sind für die Speicherung der Tags und Blobs zuständig. Die Auslastung der Speichermedien wird vom DDFS regelmäßig überprüft, um die Daten gleichmäßig im Cluster zu verteilen („Load Balancing“).

Jeder Speicherknoten besitzt einen Cache, indem alle Tags gepuffert werden, die auf dem Knoten gespeichert sind. Bekommt der Master von einem Client eine Anfrage nach einem Tag, das er noch nicht kennt, fragt er bei den Speicherknoten nach, wo sich die Kopien des Tags befinden. Durch den Cache können diese Abfragen schnell abgearbeitet werden.

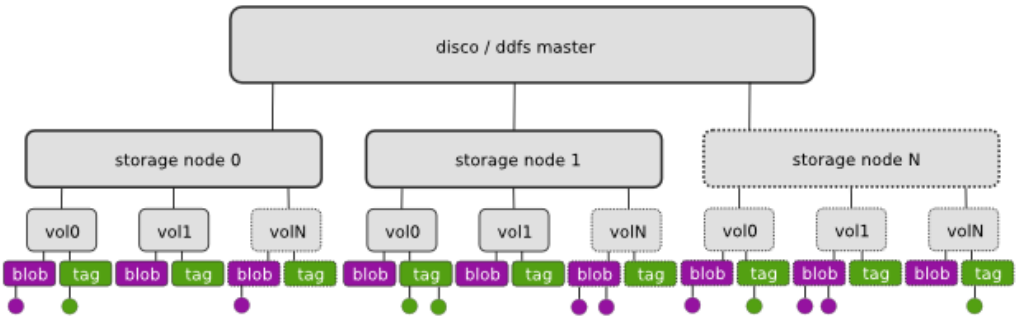

Abbildung 3.2: DDFS–Implementierung [NRC10]

3.1.2 Disco–Architektur

Das Disco–Framework basiert auf einer Master–Slave–Architektur (vgl. Abb.3.3). Die Client–Prozesse (das sind Python–Programme) schicken ihre Aufgaben an den Master, der für die Verteilung der Jobs auf verfügbare Server im Cluster zuständig ist.

Der *Worker Supervisor* ist für die Erstellung und Überwachung der *Python–Worker*, die auf einem Knoten laufen, verantwortlich. Der Master startet auf jedem Knoten des Clusters einen solchen Supervisor. Die eigentlichen Berechnungen (Map– bzw. Reduce–Tasks) werden von den Python–Workern durchgeführt, die alle Ergebnisse lokal speichern. Auf die Ergebnisse können andere Knoten per HTTP (Hypertext Transfer Protocol) zugreifen.

Die Eingabedaten werden im Vorfeld durch das DDFS auf die Knoten im Cluster verteilt. Der Master muss die Aufgaben so verteilen, dass möglichst wenig Daten über das Netzwerk gesendet werden müssen, um den Netzwerk–Traffic gering zu halten.

3.1.3 Discodex – Distributed indices for Disco

Discodex wird dazu verwendet, die Daten im DDFS zu indizieren, damit man in konstanter Zeit darauf zugreifen kann. Discodex arbeitet eigentlich wie ein normaler Index, nur mit zusätzlichen Abstraktionen, um die Handhabung möglichst einfach zu halten.

Mit anderen Worten, Discodex ist ein verteiltes Speichersystem bestehend aus Schlüssel/Werte–Paaren [NRC10]. Diese Paare bilden die Indizes, die im Dateisystem gespeichert werden. Dazu werden die Indizes in so genannte *ichunks* aufgeteilt und durch das DDFS auf die Knoten im Cluster verteilt.

Führt man Discodex aus, heißt das nichts anderes, als das man einen HTTP–Server ausführt. Dieser kann auf eine Datenstruktur *discodb* zugreifen, in der die *ichunks* zusätzlich gespeichert werden. Clients können Daten abfragen, indem sie Anfragen an den Server schicken.

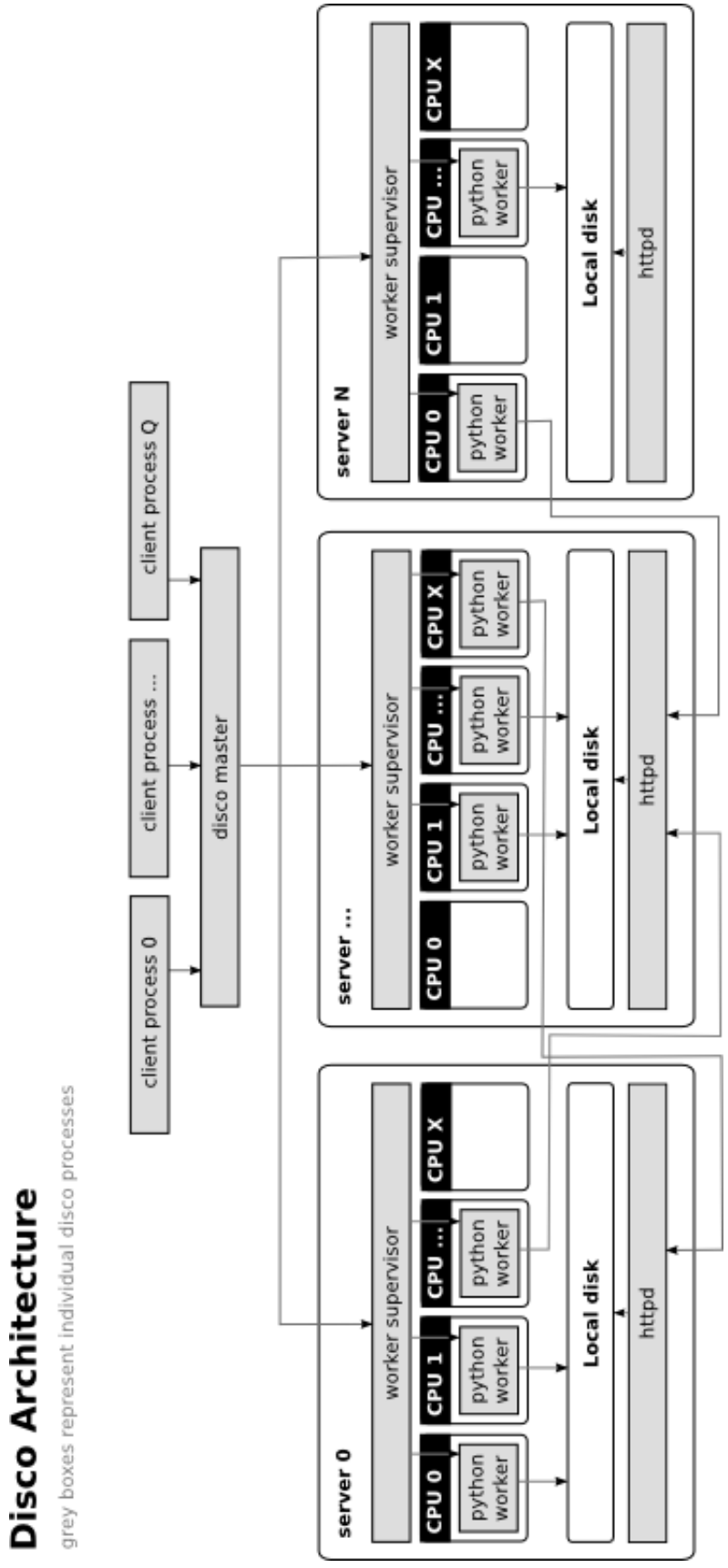

Abbildung 3.3: Disco–Architektur [NRC10]

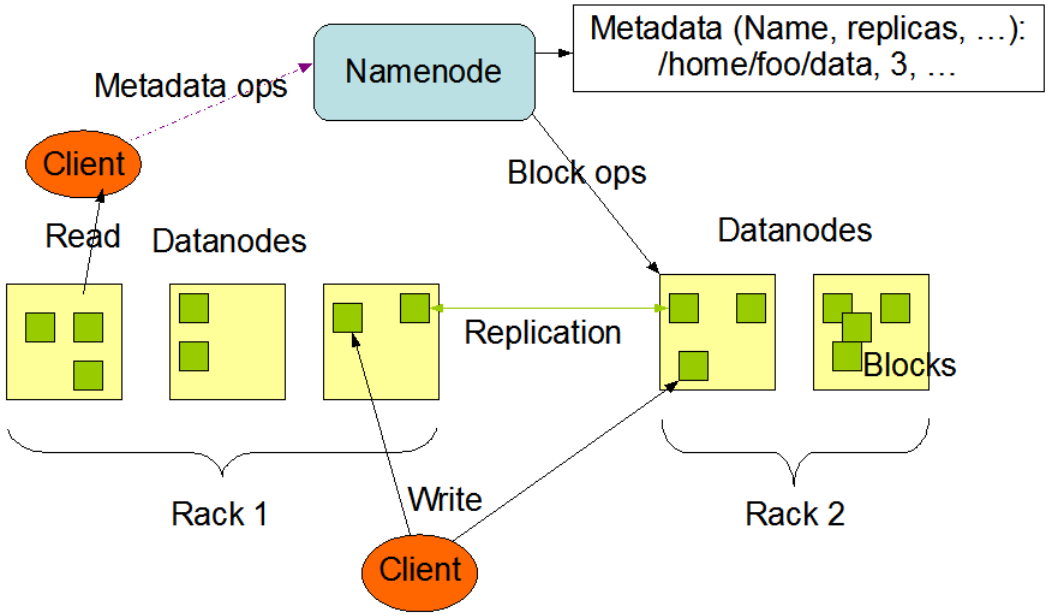

Abbildung 3.4: HDFS–Architektur [had10]

3.2 Hadoop

Hadoop wurde ursprünglich von *Yahoo* entwickelt. Seit Januar 2008 gehört es zu den Projekte der „Apache Software Foundation[1]“. Es ist ein in *Java* geschriebenes Software–Framework. Hadoop basiert auf Googles MapReduce–Konzept und unterstützt datenintensive und verteilte Applikationen. Wie Google und Disco nutzt auch Hadoop ein verteiltes Dateisystem, das HDFS (Hadoop Distributed Filesystem).

3.2.1 Hadoop Distributed File System

HDFS ist ein verteiltes und skalierbares Dateisystem und wurde entwickelt, um auf Standardhardware zu laufen. In Abb.3.4 ist die Architektur des Dateisystems abgebildet. Ein HDFS–Cluster besteht aus einem *NameNode*, der als Master fungiert und zuständig für die Organisation des Clusters ist. Dazu kommen noch beliebig viele *DataNodes* (gelbe Kästchen, typischerweise ein DataNode pro Cluster–Knoten), auf denen die Daten (kleine grüne Kästchen) in Form von Dateien gespeichert werden.

DataNodes haben folgende Aufgaben:

- Öffnen, Schließen und Umbenennen von Dateien
- Beantworten von Lese– und Schreibanfragen der Clients
- Durchführung von Operationen auf den Datenblöcken (Erstellen, Replizieren und Löschen) nach Anweisungen des NameNode

[1] http://www.apache.org/

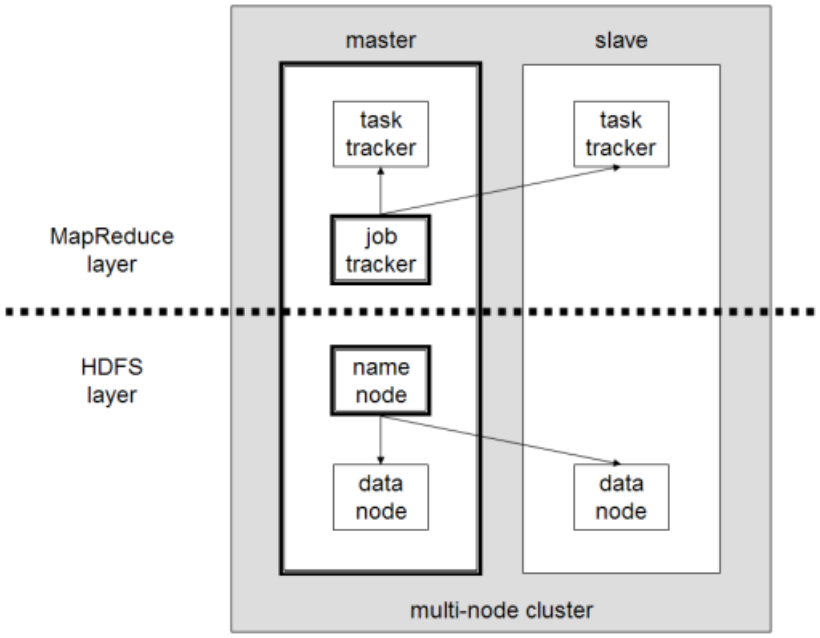

Abbildung 3.5: Aufbau eines Hadoop–Clusters [wik10a]

Es gibt zwei verschiedene Arten von Daten, die im HDFS gespeichert werden. Die Metadaten des Dateisystems, die auf dem NameNode gespeichert werden, und die normalen Daten, die in mehrere Blöcke aufgeteilt und auf verschiedene DataNodes im Cluster verteilt werden. Dabei wird jeder dieser Blöcke zunächst repliziert (standardmäßig sind es 3 Kopien), bevor er auf die DataNodes verteilt wird, um eine hohe Fehlertoleranz zu erreichen.

Die DataNodes informieren den NameNode periodisch, welche Blöcke sie gerade gespeichert haben. Erhält der NameNode diese Nachricht von einem DataNode nicht innerhalb eines festen Zeitintervalls, nimmt er an, dass der DataNode nicht mehr verfügbar ist. Das hat zur Folge, dass die betroffenen Datenblöcke kopiert und auf andere Knoten verteilt werden müssen, um die Fehlertoleranz weiterhin sicherzustellen. Bisher funktioniert dies leider noch nicht automatisch.

HDFS–Clients kontaktieren den NameNode, um Operationen auf den Metadaten durchzuführen, und die DataNodes, um Operationen auf den Daten durchzuführen.

3.2.2 Hadoop–Architektur

Ein Hadoop–Cluster lässt sich in zwei Schichten einteilen (vgl. Abb.3.5). Ein Knoten des Clusters hat innerhalb der beiden Schichten unterschiedliche Aufgaben. In der HDFS–Schicht wird er als Knoten des verteilten Dateisystems betrachtet (siehe Abschnitt 3.2.1) und in der MapReduce–Schicht als Knoten zur Durchführung einer MapReduce–Berechnung. In der MapReduce–Schicht gibt es analog zum HDFS einen Master–Knoten und mehrere Slave–Knoten.

Der Master–Knoten wird auch als *JobTracker* bezeichnet. Er ist für die Steue-

rung der Slave–Knoten zuständig, die als *TaskTracker* bezeichnet werden. Clients schicken MapReduce–Jobs an den JobTracker. Die Jobs werden in mehrere Map– bzw. Reduce–Tasks aufgeteilt und vom JobTracker an verfügbare TaskTracker verteilt. Bei der Verteilung der Tasks muss der JobTracker folgende Dinge berücksichtigen:

- Er weiß, welche Daten auf welcher DataNode durch das HDFS lokal gespeichert wurden. Daher versucht er die Tasks so zu verteilen, das sie einer DataNode mit zugehörigem TaskTracker zugewiesen werden, die die für den Task benötigten Daten bereits gespeichert hat.

- Gelingt das nicht, versucht der JobTracker eine DataNode auszuwählen, die sich in der Nähe (im gleichen Rack) der Daten befindet.

Dadurch wird die Menge der Daten, die über das Netzwerk verschickt werden müssen, erheblich reduziert und somit werden Netzwerkressourcen eingespart.

Jeder TaskTracker, dem ein Map–Task zugewiesen wird, liest die benötigten Daten aus dem HDFS, führt dann die vom Benutzer spezifizierte Map–Funktion aus, partitioniert die berechneten Schlüssel/Wert–Paare für die Reduce–Tasks und speichert sie lokal auf seiner Festplatte.

Die TaskTracker, denen ein Reduce–Task zugewiesen wird, holen sich die partitionierten Daten von allen Map–TaskTrackern, sortieren die Daten, führen die vom Benutzer spezifizierte Reduce–Funktion aus und speichern die Ergebnisse im HDFS.

Falls ein TaskTracker ausfällt, müssen die von ihm durchgeführten Tasks von anderen verfügbaren TaskTrackern erneut ausgeführt werden. Damit der JobTracker den Ausfall eines TaskTrackers erkennen kann, wird ein *Heartbeat–Protokoll* verwendet. Erhält der JobTracker von einem TaskTracker innerhalb einer gewissen Zeit keine Nachricht mehr, kennzeichnet er diesen als „*failed*“.

Die Heartbeat–Nachrichten haben noch eine weitere Aufgabe. Jeder TaskTracker besitzt eine feste Anzahl an *Slots* für die Ausführung von Map– bzw. Reduce–Tasks. Mit Hilfe der Nachrichten teilt ein TaskTracker dem JobTracker mit, wieviele freie Slots er gerade zur Verfügung hat und der JobTracker kann diese Informationen bei der Verteilung der Tasks berücksichtigen.

3.3 BOOM

Das BOOM–Projekt[2] erforscht die Entwicklung von Software für Rechencentern mit Hilfe einer datenzentrischen Programmiersprache [BOO]. Um die Realisierbarkeit dieses Ansatzes zu evaluieren, wurde *BOOM Analytics* entwickelt. Dabei wurde einerseits der MapReduce–Kern von Hadoop, in die deklarative Programmiersprache *Overlog* portiert. Mit Hadoops MapReduce–Kern ist die Funktionalität des *JobTrackers* gemeint (vgl. Abschnitt 3.2.2). Andererseits wurde Hadoops HDFS komplett neu in Overlog und Java implementiert.

[2] BOOM: Berkeley Orders Of Magnitude

```
path(@From, To, To, Cost)
        :- link(@From, To, Cost);
path(@From, End, To, Cost1+Cost2)
        :- link(@From, To, Cost1),
           path(@To, End, NextHop, Cost2);

WITH path(Start, End, NextHop, Cost) AS
  ( SELECT link.From, path.End,
           link.To, link.Cost+path.Cost
      FROM link, path
     WHERE link.To = path.Start );
```

Abbildung 3.6: Berechnung von Pfaden ausgehend von Links in Overlog und Übersetzung der zweiten Regel in SQL [ACC+10]

Ein Vorteil dieser Portierung bzw. Neuimplementierung von Hadoop ist eine kompaktere und leicht erweiterbare Codebasis. Diese Aussage und wie BOOM Analytics funktioniert, soll im Folgenden genauer betrachtet werden. Zunächst aber einige Bemerkungen zu den dabei verwendeten Programmiersprachen Datalog und Overlog.

3.3.1 Datalog und Overlog

Datalog ist auf relationalen Tabellen definiert und eine rein logische Anfragesprache, die keine Änderungen an den gespeicherten Tabellen vornimmt. Ein Datalogprogramm besteht aus einer Menge an Datalog–Regeln, die wie folgt aussehen:

$$r_{head}(\langle col - list \rangle) \ :- \ r_1(\langle col - list \rangle), \ldots, r_n(\langle col - list \rangle)$$

Jeder der Terme r_i repräsentiert eine Relation, die entweder eine Datenbanktabelle darstellt oder das Ergebnis anderer Regeln. Die Spalten einer Relation werden als Liste aufgefasst, wobei die einzelnen Variablennamen durch ein Komma getrennt sind. Die Terme rechts von dem Zeichen $:-$ bilden den Rumpf der Regel, die Relation links davon bildet den Kopf. In SQL (Structured Query Language) entspricht die rechte Seite dem FROM und WHERE und die linke Seite dem SELECT Statement. In Abb.3.6 ist ein Beispiel für Overlog–Regeln und eine entsprechende SQL–Anfrage zu sehen.

Jede Datalog–Regel ist eine logische Aussage. Der Regel–Kopf enthält nur die Tupel, die von dem Regel–Rumpf generiert werden können. Dazu werden die Spalten der Realtionen im Rumpf betrachtet. Stimmen die Variablennamen der Spalten überein, werden die entsprechenden Einträge der Realtionen „gejoint“ (siehe Abb.3.6). In SQL entspricht dies einem *JOIN*.

Da man mit Datalog keine Änderungen an den gespeicherten Tabellen vornehmen kann, nutzt BOOM die ereignisorientierte und deklarative Programmiersprache Overlog, die Datalog auf folgende drei Arten erweitert:

- Notation zur Spezifizierung des Speicherorts der Daten (entspricht dem „@“ in Abb. 3.6)

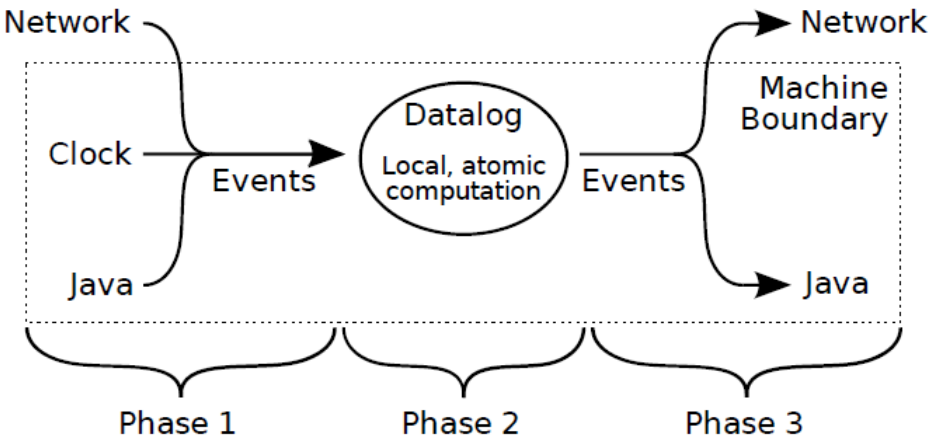

Abbildung 3.7: Overlog–Zeitschritt [ACC+10]

- SQL–basierte Erweiterungen wie Primärschlüssel und Aggregation
- Modell zur Verarbeitung und Generierung von Änderungen an Tabellen (Overlog–Zeitschritt)

Änderungen an Tabellen werden mit Hilfe eines Overlog–Zeitschritts (siehe Abb.3.7) beschrieben. Ein Overlog–Zeitschritt wird von einem Knoten im Cluster durchgeführt. Innerhalb eines Zeitschritts sieht ein Knoten nur die Tupel, die er lokal gespeichert hat, also seinen lokalen Zustand. Zeitschritte lassen sich in drei Phasen einteilen[3]:

1. Phase: Eingehende Events werden in Tupel umgewandelt. Diese Tupel entsprechen Einträgen, die in Tabellen eingefügt, gelöscht oder aktualisiert werden sollen.

2. Phase: Datalog–Regeln werden bis zu einem Fixpunkt ausgeführt. Dieser Fixpunkt ist erreicht, wenn alle Regeln rekursiv evaluiert und keine neuen Ergebnisse mehr generiert worden sind.

3. Phase: Der lokale Zustand wird aktualisiert. Diese Aktion erfolgt atomar und ist dauerhaft. Danach werden ausgehende Events an das System gesendet.

3.3.2 BOOM–FS

Im Gegensatz zur Portierungsstrategie bei BOOM–MR, wird für HDFS und BOOM–FS eine Neuimplementierung in Overlog als Strategie gewählt. BOOM–FS kann entweder zusammen mit Hadoops MapReduce oder mit BOOM–MR verwendet werden. Die Neuimplementierung erfolgt in zwei Schritten:

[3] Zu beachten ist, dass Datalog nur auf statischen Tabellen definiert ist. Aber die Phasen 1 und 3 erlauben es Overlog–Programmen den Zustand zu verändern.

Zustand des Dateisystems

Zuerst werden die Metadaten des Dateisystem in eine Menge von Relationen überführt. Der NameNode muss sicherstellen, dass die Relationen stets konsistent sind.

File: Diese Relation enthält für jede Datei oder jedes Verzeichnis, das im BOOM–FS gespeichert ist, einen Eintrag.

Fqpath: Pfadangaben der Dateien und Verzeichnisse werden durch diese Relation repräsentiert.

Fchunk: Die Menge der Chunks, die eine Datei bilden, werden hier in einem Eintrag gepflegt. Da die Ordnung auf den Chunks eine Rolle spielt, aber Relationen ungeordnet sind, erhalten die Chunks eine aufsteigende ID. Clients, die die Chunks anfordern, können sie anhand der ID sortieren und erhalten somit die richtige Reihenfolge.

Datanode: In dieser Relation ist für jeden verfügbaren DataNode ein Eintrag gespeichert. Kommen neue verfügbare DataNodes hinzu, werden neue Einträge in die Tabelle eingefügt. Genauso werden nicht mehr verfügbare DataNodes aus der Tabelle entfernt. Für die Aktualisierung, die über Heartbeat–Nachrichten erfolgen, ist der NameNode zuständig.

Hb_chunk: Ein Eintrag in dieser Relation enthält alle Chunks, die auf einem verfügbaren DataNode gespeichert sind. Der NameNode ist für die Aktualisierung verantwortlich.

Kommunikationsprotokolle

Für die Kommunikation zwischen Clients, NameNode und DataNodes nutzt BOOM–FS drei verschiedene Kommunikationsprotokolle. Das Metadaten–Protokoll, das Heartbeat–Protokoll und das Daten–Protokoll. Die ersten beiden Protokolle sind durch Overlog–Regeln implementiert. Nur das letzte Protokoll ist in Java implementiert. Im Folgenden werden die Protokolle näher betrachtet:

Metadaten–Protokoll: Für jedes Kommando dieses Protokolls gibt es eine Regel auf der Seite des Client. Diese legt fest, wie neue Metadaten beim NameNode gespeichert oder existierende geändert werden sollen. Das heißt, es werden Tupel in die entsprechenden Tabellen eingefügt, geändert oder gelöscht. Zu dieser Regel gibt es zwei entsprechende Regeln auf der Seite des NameNode. Eine der beiden Regeln spezifiziert das Ergebnis, das bei Erfolg zum Client gesendet werden soll. Die andere ist für die Fehlerbehandlung zuständig, indem sie eine Fehlernachricht generiert und an den Client sendet.

Heartbeat–Protokoll: Heartbeat–Nachrichten von den DataNodes an den NameNode haben ein ähnliches Anfrage/Antwort–Verhalten wie die Nachrichten beim Metadaten–Protokoll. Sie werden jedoch nicht durch Events aus dem Netzwerk ausgelöst, sondern erfolgen periodisch.

Name	*Description*	*Relevant attributes*
job	Job definitions	jobid, priority, submit_time, status, jobConf
task	Task definitions	jobid, taskid, type, partition, status
taskAttempt	Task attempts	jobid, taskid, attemptid, progress, state, phase, tracker, input_loc, start, finish
taskTracker	TaskTracker definitions	name, hostname, state, map_count, reduce_count, max_map, max_reduce

Abbildung 3.8: BOOM–MR Relationen definieren Zustand des JobTrackers [ACC+10]

Daten–Protokoll: Diese Protokoll dient zur Übertragung der Chunks zwischen Client und DataNodes. Implementiert ist es zwar in Java, es wird jedoch durch Overlog–Regeln überwacht und gesteuert.

3.3.3 BOOM–Architektur

Wie am Anfang des Kapitels bereits erwähnt, wird im Rahmen von BOOM Analytics der MapReduce–Kern nach Overlog portiert. Zunächst muss der Kernzustand, der vom JobTracker gepflegt wird, identifiziert werden. Dieser Zustand beinhaltet zwei Datenstrukturen: Eine für die Überwachung und Steuerung des Systemstatus, und eine für die Nachrichten, die vom JobTracker empfangen und gesendet werden. Dieser Zustand wird in eine relationale Repräsentation überführt, die in Abb.3.8 zu sehen ist. Dabei bilden die unterstrichenen Attribute den Primärschlüssel der jeweiligen Relation.

Job: Diese Relation enthält für jeden Job, der an den JobTracker gesendet wird einen Eintrag. Das Attribut **jobConf** enthält ein Java–Objekt und ist eine Altlast von Hadoop. Es umfasst die Konfiguration eines Jobs.

Task: Mit Hilfe dieser Relation werden alle Tasks innerhalb eines Jobs identifiziert.

TaskAttempt: Da ein Task mehrmals ausgeführt werden kann, beispielsweise weil er fehlgeschlagen ist, enthält diese Relation einen Eintrag für jeden solchen Versuch.

TaskTracker: Zur Identifizierung der einzelnen TaskTracker im Cluster wird diese Relation genutzt.

Durch die Anwendung von Overlog–Regeln können diese vier Tabellen geändert werden. Somit wird eine Scheduling–Strategie für den JobTracker realisiert. Die Scheduling–Entscheidungen sind in der Tabelle *taskAttempt* kodiert, denn

System	*Lines in Patch*	*Files Modified by Patch*
Hadoop	2102	17
BOOM-MR	82	2

Abbildung 3.9: Modifizierung der MapReduce Scheduler durch LATE [ACC+10]

hier werden die einzelnen Tasks den TaskTrackern zugewiesen. Eine Scheduling–Strategie ist eine Menge von Regeln, die mit der Tabelle *tastTracker* „gejoint“ wird, um die DataNodes zu finden, die noch freie Slots für Map– bzw. Reduce–Tasks besitzen. Durch das Einfügen eines Tupels in die Tabelle *taskAttempt* wird der entsprechende Task zur Ausführung gebracht. Neue Scheduling–Strategien, wie der LATE–Scheduler [ZKJ+08], lassen sich in BOOM–MR leichter und mit weniger Coedaufwand als bei Hadoop implementieren (vgl. Abb.3.9).

Kapitel 4

Fazit

Viele reale Probleme lassen sich mit Hilfe des MapReduce–Konzepts beschreiben und lösen. Durch die Einteilung der Berechnungen in zwei Phasen, eine Map– und eine Reduce–Phase, die parallel ausgeführt werden können, lässt sich die gesamte Berechnung um ein Vielfaches beschleunigen. Da die Menge der zu bearbeitenden Daten (beispielsweise bei Google) immer größer wird, ist eine Parallelisierung oftmals erforderlich, weil einzelne Prozesse allein gar nicht mit den Daten umgehen können.

Das Programmiermodell wird bereits für viele Zwecke genutzt, was durch die Anzahl der vorhandenen Implementierungen deutlich wird. Es wurde schon in *C++*, *C#*, *Erlang*, *Java*, *Ocaml*, *Python*, *Ruby*, *F#*, *R* und vielen anderen Programmiersprachen implementiert. Google nutzt das MapReduce–Konzept beispielsweise für seine Internetsuchmaschine, zur Sortierung von Daten, für Analysen im Rahmen des Data Mining und für Machine Learning.

Da die Netzwerkbandbreite in großen Clustern eine sehr knappe Ressource darstellt, wird versucht, die Menge der Daten, die über das Netzwerk gesendet wird, so weit wie möglich zu reduzieren. Um dies zu erreichen, wird ein verteiltes Dateisystem benötigt, das für die Verteilung der Daten zuständig ist (vgl. Kapitel 3: GFS, DDFS, HDFS, BOOM-FS).

Ein weiterer Vorteil des Modells ist seine Abstraktion und die damit verbundene Einfachheit seiner Nutzung. Auch Programmierer mit nur wenig Erfahrung auf dem Gebiet der Parallelprogrammierung können einfach parallele, verteilte Systeme entwickeln, da das Framework Details wie Parallelisierung, Fehlertoleranz und Verteilung der Daten übernimmt.

Literaturverzeichnis

[ACC+10] Alvaro, Peter ; Condie, Tyson ; Conway, Neil ; Elmeleegy, Khaled ; Hellerstein, Joseph M. ; Sears, Russell: Boom analytics: exploring data-centric, declarative programming for the cloud. In: Morin, Christine (Hrsg.) ; Muller, Gilles (Hrsg.): *EuroSys*, ACM, 2010. – ISBN 978–1–60558–577–2, 223–236

[BOO] BOOM: *Berkeley Orders Of Magnitude*. http://boom.cs.berkeley.edu/

[DG08] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: simplified data processing on large clusters. In: *CACM* 51 (2008), Nr. 1, 107–113. http://doi.acm.org/10.1145/1327452.1327492

[GGL03] Ghemawat, Sanjay ; Gobioff, Howard ; Leung, Shun-Tak: The Google file system. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (19th SOSP'03)*. Bolton Landing, NY, USA : ACM SIGOPS, Oktober 2003, S. 29–43

[had10] *Hadoop*. http://hadoop.apache.org. Version: 2010

[NRC10] Nokia-Research-Center: *disco: massive data - minimal code*. http://discoproject.org/. Version: 2010

[Sha08] Shankland, Stephen: *Google spotlights data center inner workings*. http://news.cnet.com. Version: 2008

[wik10a] *Hadoop*. http://en.wikipedia.org/wiki/Hadoop. Version: 2010

[wik10b] *MapReduce*. http://de.wikipedia.org/wiki/MapReduce. Version: 2010

[ZKJ+08] Zaharia, Matei ; Konwinski, Andy ; Joseph, Anthony D. ; Katz, Randy H. ; Stoica, Ion: Improving MapReduce Performance in Heterogeneous Environments. In: Draves, Richard (Hrsg.) ; Renesse, Robbert van (Hrsg.): *OSDI*, USENIX Association, 2008. – ISBN 978–1–931971–65–2, 29–42