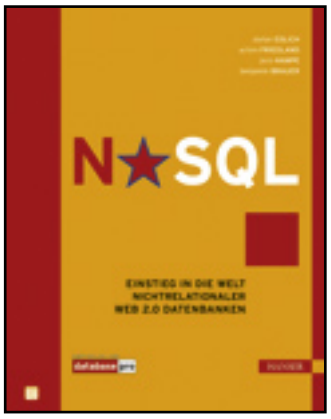
HANSER



Leseprobe

Stefan Edlich, Achim Friedland, Jens Hampe, Benjamin Brauer

NoSQL

Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken

ISBN: 978-3-446-42355-8

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-42355-8

sowie im Buchhandel.

2 NoSQL – Theoretische Grundlagen

Wenn Sie NoSQL-Datenbanken einsetzen wollen, sollten Sie auch mit den wichtigsten Grundbegriffen und theoretischen Ansätzen auskennen, auf denen diese Systeme aufbauen. Einige davon sind quasi traditionelle Verfahren der klassischen Datenbankwelt und Ihnen vielleicht schon bekannt. Daneben gibt es aber in der NoSQL-Welt auch Algorithmen und Protokolle, die bisher noch nicht häufig verwendet wurden. Diese wollen wir in diesem Kapitel vorstellen. Dazu zählen:

- Map/Reduce
- CAP-Theorem/Eventually Consistent
- Consistent Hashing
- MVCC-Protokoll
- Vector Clocks
- Paxos

Mit diesen Grundlagen sind Sie gut gerüstet, wenn Sie sich die NoSQL-Welt erobern wollen, weil die meisten NoSQL-Datenbanken auf diesem minimalen Fundament aufbauen. Eine vollständige Darstellung der Theorie, die den NoSQL-Systemen zugrunde liegt, würde eigentlich ein eigenes Buch erfordern. Im Rahmen unseres Buches können und wollen wir nur die wichtigsten Themen ansprechen.

Wenn Sie noch tiefer in die Thematik einsteigen wollen, als wir es in dieser Einführung tun, müssten Sie sich auch noch mit folgenden Themen auseinandersetzen:

- B- und B*-Trees sowie beispielsweise Quad-Trees, die für Key/Value-Datenbanken, räumliche Indexierung und Kollisionserkennung wichtig sind.
- Transaktionsprotokolle und Consensus-Protokolle; Protokolle wie 2PC, 3PC sind für die klassische Datenbankwelt und auch für NoSQL wichtige Grundlagen. Diese sind aber in der Literatur sehr gut behandelt.
- Die Themen Replikation, Partitionierung und Fragmentierung der Daten spielen für NoSQL ebenfalls eine wichtige Rolle. Zu diesem Bereich gehören evtl. auch verschiedene Quorum-Strategien.

Literatur zu allen obigen Themen finden Sie entweder in Büchern zu Algorithmen oder in umfassenderen Standardwerken zum Thema Datenbanken:

- Gunter Saake, Kai-Uwe Sattler, Algorithmen und Datenstrukturen: Eine Einführung mit Java, dpunkt Verlag, 2010
- Hector Garcia-Molina, Jeffrey D. Ullman und Jennifer Widom, Database Systems. The Complete Book, Pearson International, 2nd Edition, 2008

2.1 Map/Reduce

Um die rasant wachsende Menge von Daten und Informationen effizient verarbeiten zu können, wurden neue alternative Algorithmen, Frameworks und Datenbankmanagementsysteme entwickelt. Bei der Verarbeitung großer Datenmengen in der Größenordnung von vielen Terabytes bis hin zu mehreren Petabytes spielt das in diesem Kapitel beschriebene Map/Reduce-Verfahren eine entscheidende Rolle. Mittels eines Map/Reduce-Frameworks wird eine effiziente nebenläufige Berechnung über solch große Datenmengen in Computerclustern erst ermöglicht.

Entwickelt wurde das Map/Reduce-Framework 2004 bei Google Inc. von den Entwicklern Jeffrey Dean und Sanjay Ghemawat. Eine erste Vorstellung und Demonstration, beschrieben in [Dean04], erfolgte auf der Konferenz „OSDI 04, Sixth Symposium on Operating System Design and Implementation“ in San Francisco, Kalifornien, im Dezember 2004[1]. Im Januar 2010 hat Google Inc. auf das dort vorgestellte Map/Reduce-Verfahren vom US-amerikanischen Patentbüro ein Patent erhalten[2]. Experten sind sich allerdings einig, dass es sich hierbei um ein Schutzpatent für Google selbst handelt und Google nun keine Klagewelle beginnen wird.

Die grundlegende Idee, Komponenten, Architektur und Anwendungsbereiche des Map/Reduce-Verfahrens werden in diesem Abschnitt 2.1 vorgestellt und beschrieben. Die zahlreichen Implementierungen, die seit der ersten Vorstellung dieses Verfahrens entwickelt wurden, werden zur Übersicht kurz aufgelistet. Abgerundet wird dieser Abschnitt durch ein einfaches Einsatzbeispiel, welches zum praktischen Nachvollziehen des beschriebenen Map/Reduce-Algorithmus ermuntern soll.

2.1.1 Funktionale Ursprünge

Die Parallelisierung von Prozessen beginnt bei der Formulierung von Algorithmen. Parallelisierung ist eine Stärke der funktionalen Sprachen. Die Grundidee von Map/Reduce kommt daher auch von funktionalen Programmiersprachen wie LISP[3] und ML[4]. Hinsicht-

[1] *http://labs.google.com/papers/mapreduce.html, 28.02.2010*
[2] Patent Nummer: US007650331; zu finden bei http://patft.uspto.gov
[3] **Lis**t **P**rocessing
[4] **M**eta **L**anguage

lich der Parallelisierung bieten funktionale Sprachen aufgrund ihrer Arbeitsweise Vorteile gegenüber anderen Sprachen. Es entstehen keine Seiteneffekte wie Verklemmungen (*deadlock*) und Wettlaufsituationen (*race conditions*). Funktionale Operationen ändern die vorhandenen Datenstrukturen nicht, sie arbeiten immer auf neu erstellten Kopien vorhandener Daten. Die Originaldaten bleiben unverändert erhalten. Unterschiedliche Operationen auf dem gleichen Datensatz beeinflussen sich somit nicht gegenseitig, da jede Operation auf einer eigenen Kopie der Originaldaten angewendet wird oder bei Datenergänzungen eine neue Datenstruktur erzeugt wird. Ohne Seiteneffekte spielt auch die Ausführungsreihenfolge von Operationen keine Rolle, wodurch die Parallelisierung dieser Operationen möglich wird.

Das Konzept einer Funktion im Sinne der Mathematik ist in der funktionalen Programmierung am klarsten umgesetzt. Hier stellen die Funktionen Abbildungsvorschriften dar. Eine Funktion besteht dann aus einer Reihe von Definitionen, die diese Vorschrift beschreibt. Ein funktionales Programm besteht ausschließlich aus Funktionsdefinitionen und besitzt keine Kontrollstrukturen wie Schleifen. Wichtigstes Hilfsmittel für die funktionale Programmierung ist daher die Rekursion. Funktionen sind in funktionalen Programmiersprachen Objekte, mit denen wie mit Variablen gearbeitet werden kann. Insbesondere können Funktionen als Argument oder Rückgabewert einer anderen Funktion auftreten. Man spricht dann von Funktionen höherer Ordnung.

Die aus der funktionalen Programmierung bekannten Routinen *map()* und *fold(),* auch als *reduce()* bezeichnet, werden in modifizierter Form im Map/Reduce-Algorithmus jeweils nebenläufig in zwei Phasen ausgeführt. Sie zählen zu den Funktionen höherer Ordnung. Wie der Name der ältesten funktionalen Programmiersprache LISP (= **Lis**t **P**rocessing) schon verrät, geht es dabei um die Verarbeitung von Listen. Die Funktion *map()* wendet eine Funktion sukzessive auf alle Elemente einer Liste an und gibt eine durch die Funktion modifizierte Liste zurück. Die Funktion *reduce()* akkumuliert einzelne Funktionsergebnisse der Listenpaare und reduziert sie damit auf einen Ausgabewert. Diese beiden Funktionen werden in modifizierter Ausprägung als Map/Reduce-Algorithmus jeweils parallel auf verschiedenen Knoten im Netzwerk in zwei Phasen hintereinander angewendet. Das Besondere an der Map/Reduce-Formulierung ist, dass sich mit den zwei Phasen jeweils eine Parallelisierungsmöglichkeit ergibt, die innerhalb eines Computer-Clusters für eine beschleunigte Berechnung von sehr großen Datenmengen verwendet werden kann. Bei solchen Datenmengen ist eine Parallelisierung unter Umständen allein schon deshalb erforderlich, weil diese Datenmengen für einen einzelnen Prozess und das ausführende Rechnersystem bereits zu groß sind.

Die *map()*-Funktion der funktionalen Sprachen erhält als Argument eine Funktion `f` und wendet diese auf jedes Element einer übergebenen Liste an. Es ist eine polymorphe Funktion, die beliebige Argumenttypen erhalten kann, wie durch die Typvariablen `a` und `b` in Listing 2.1.1, angegeben ist.

Listing 2.1.1 Haskell-Definition der map-Funktion[5]

```
map :: (a -> b) -> [a] -> [b]
map f []          = []
map f (x : xs)    = f x : map f xs
```

Die erste Zeile in der Funktionsdefinition wird als Typsignatur bezeichnet und beschreibt die Aufgabe der Funktion. Sie wendet die Funktion `(a->b)` auf eine Liste `[a]` an und gibt die Liste `[b]` zurück. Die zweite und dritte Zeile definieren das Verhalten der *map()*-Funktion für verschiede Eingabemuster. Bei Eingabe einer Funktion `f` und einer leeren Liste `[]` wird als Ergebnis auch nur eine leere Liste `[]` zurückgegeben. Die dritte Zeile zeigt, dass bei Eingabe einer Funktion `f` und einer Liste, die durch die Listenkonstruktion `(x:xs)` dargestellt wird, die Funktion `f` auf das erste Listenelement *x* und anschließend dann rekursiv auf die restliche Liste *xs* angewendet wird.

Ein Funktionsaufruf in Haskell mit folgenden Angaben:

```
map (\x -> x^2) [1,2,3,4,5]
```

ergibt das Ergebnis, wie auch in Abbildung 2.1.1 dargestellt:

```
[1,4,9,16,25]
```

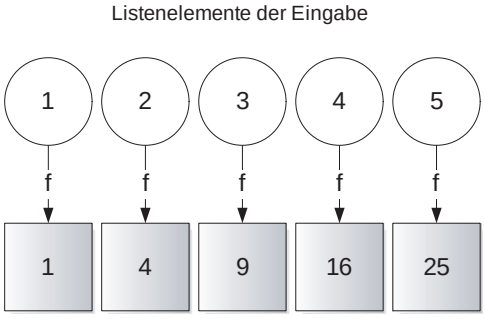


Abbildung 2.1.1
Anwenden der map()-Funktion

Die *map()*-Funktion -lässt sich nach Definition auf beliebige Datentypen anwenden. Das folgende Beispiel wendet die Funktion `toUpper` auf jedes Zeichen des Strings *„nosql“* an. Um die Funktion `toUpper` nutzen zu können, muss sie vorher durch den Befehl `:module +Data.Char` in den Arbeitsbereich importiert werden:

```
:module +Data.Char
map toUpper "nosql"
```

und erzeugt damit die Zeichenausgabe in Großbuchstaben (siehe auch Abbildung 2.1.2):

```
"NOSQL"
```

Die Reihenfolge der Elementeingabe und Elementausgabe bleibt bei dieser *n*-zu-*n*-Transformation erhalten, wobei *n* = Anzahl der Listenelemente ist.

[5] Vgl. [Rech02] Seite 559

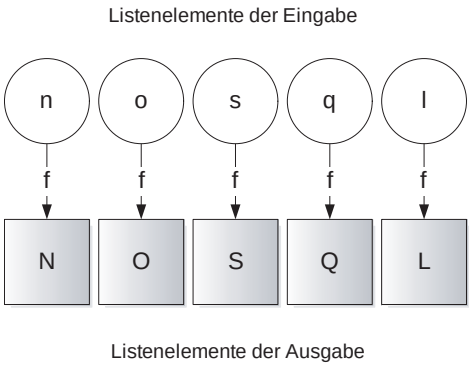


Abbildung 2.1.2
Anwendung der map()-Funktion mit *toUpper*

Die *fold()*-Funktion realisiert quasi eine *n*-zu-1-Transformation und wird in diesem Zusammenhang auch in anderen Sprachen als *reduce-, accumulate-*, *compress-* oder *inject-*Funktion bezeichnet. Das Ergebnis dieser Transformation muss nicht aus einem Element bestehen, es kann auch wiederum eine Liste von reduzierten Elementen sein. In Haskell wie auch in vielen anderen funktionalen Programmiersprachen werden zwei Varianten von *fold()* unterschieden: die *foldl*-Funktion für die Bearbeitung einer Liste von links nach rechts und die *foldr*-Funktion für die Bearbeitung einer Liste von rechts nach links[6].

Listing 2.1.2 Haskell-Definitionen für foldl

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Listing 2.1.3 Haskell-Definitionen für foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Die Typsignaturen der Funktionen in den jeweils ersten Zeilen geben die Aufgaben der Funktionen wieder. Die Argumente der *foldl*-Funktion bestehen aus einer Schrittfunktion `(a -> b -> a)`, einem initialen Wert für einen Akkumulator `a` und einer Eingabeliste `[b]`. Die Schrittfunktion `(a -> b -> a)` wendet den Akkumulator `a` auf ein Listenelement `b` an und gibt einen neuen Akkumulator `a` zurück. Die Schrittfunktion wird nun rekursiv für alle Listenelemente angewendet und liefert den akkumulierten Rückgabewert `a`. Das Verhalten für verschiedene Eingabemuster wird in den jeweils folgenden beiden Zeilen des Listings definiert. Bei Eingabe einer Schrittfunktion `f`, einem Akkumulatorwert `z` und einer leeren Liste `[]` besteht das Ergebnis auch nur aus dem initialen Wert des Akkumulators `z`. Bei Eingabe einer Schrittfunktion `f`, einem Akkumulatorwert `z` und einer nicht leeren Liste `(x:xs)` wird `foldl f` rekursiv auf alle Listenelemente `xs` angewendet, wobei der jeweils neue Anfangswert das Ergebnis der Zusammenlegung des alten ursprünglichen Wertes `z` mit dem nächsten Element `x` ist, ausgedrückt durch `(f z x)`.

[6] Vgl. [Rech02] Seite 559-560

Analog gilt für die Argumente der *foldr*-Funktion die Schrittfunktion `(a -> b -> b)` mit einem initialen Wert für einen Akkumulator `b` und einer Eingabeliste `[a]`. Bei Eingabe einer Schrittfunktion `f` einem Akkumulatorwert `z` und einer leeren Liste `[]` besteht das Ergebnis, so wie auch schon bei der *foldl*-Definition, nur aus dem initialen Wert des Akkumulators `z`. Bei Eingabe einer Schrittfunktion `f`, einem Akkumulatorwert `z` und einer nicht leeren Liste `(x:xs)` wird `f` auf das erste Element `x` der Liste `xs` und dem Ergebnis der Faltung der restlichen Liste angewendet, ausgedrückt durch `f x (foldr f z xs)`.

Das folgende einfache Beispiel wendet den assoziativen Operator `+` auf eine Liste an, der Anfangswert ist hierbei 0:

```
foldl (+) 0 [1,2,3,4,5,6,6,7,8]
foldr (+) 0 [1,2,3,4,5,6,6,7,8]
```

Als Ergebnis erhalten wir, da der Operator `+` assoziativ ist, logischerweise jeweils eine Summe von 42. Die Aufgabe wird aber innerhalb der Funktionen aufgrund ihrer Definitionen jeweils unterschiedlich berechnet. Dieses kann durch einfache Klammerung der einzelnen Operationen wiedergegeben werden:

- Eine Berechnung mit `foldl` entspricht: (((((((0+1) + 2) + 3) + 4)+5)+6)+6)+7)+8
- Eine Berechnung mit `foldr` entspricht: 1 + (2 + (3 + (4 + (5+(6+(6+(7+(8+0)))))))))

Abbildung 2.1.3 stellt die Transformation von fünf Listenwerte auf einen Rückgabewert, für den das Beispiel `foldl (+) 0 [1,2,3,4,5]` gilt, noch einmal grafisch dar.

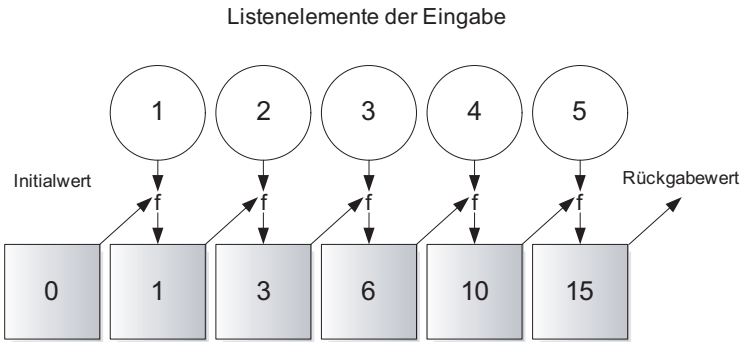


Abbildung 2.1.3
Anwendung der `foldl`-Funktion

Die aufgezeigten einfachen Beispiele wurden mit der funktionalen Programmiersprache Haskell umgesetzt. Sie können und sollten zum Verständnis nachvollzogen werden. Eine aktuelle Version von Haskell und dem Glasgow Haskell Compiler kann man über die offizielle Haskell-Website *http://www.haskell.org/* frei beziehen. Ein freies Online-Buch über die Programmiersprache Haskell findet man auch unter *http://book.realworldhaskell.org/*.

Die beschriebenen Funktionen *map()*, *foldr()* und *foldl()* als reduzierende Funktionen bilden die Basis des Map/Reduce-Verfahrens. Sie werden in den verschiedenen Frameworks aber in angepasster Weise umgesetzt und unterscheiden sich somit von ihren Namensgebern der funktionalen Programmierung. Die beschriebenen Funktionen sind grundsätzlich nicht entwickelt worden, um dann in einem Map/Reduce-Framework eingesetzt zu werden.

Deshalb ist es auch nicht verwunderlich, dass sie in solch einem Framework in modifizierter Form verwendet werden. In einem Aufsatz beschreibt der Microsoft-Entwickler Ralf Lämmel die Unterschiede zwischen den ursprünglichen Map- und Reduce-Funktionen und der Umsetzung in Googels Map/Reduce-Verfahren; sie können in [Lämm07] nachgelesen werden. Wir verlassen nun die Grundlagen des Map/Reduce-Algorithmus aus der funktionalen Programmierung und kommen zur Beschreibung der Arbeitsweise und der Phasen des Map/Reduce-Verfahrens.

2.1.2 Phasen und Datenfluss

Die grundlegende Arbeitsweise eines Map/Reduce-Verfahrens lässt sich sehr gut anhand seines Datenflusses zeigen. Der Datenfluss kann wie in Abbildung 2.1.4 dargestellt in verschiedene Bereiche eingeteilt werden. Es lassen sich folgende Arbeitsphasen unterscheiden:

- Zuerst werden alle Eingabedaten auf verschiedene Map-Prozesse aufgeteilt.
- In der Map-Phase berechnen diese Prozesse jeweils parallel die vom Nutzer bereitgestellte Map-Funktion.
- Von jedem Map-Prozess fließen Daten in verschiedene Zwischenergebnisspeicher.
- Die Map-Phase ist beendet, sobald alle Zwischenergebnisse berechnet worden sind.
- Nun beginnt die Reduce-Phase, in der für jeden Satz an Zwischenergebnissen ein Reduce-Prozess die vom Nutzer bereitgestellte Reduce-Funktion parallel berechnet.
- Jeder dieser Reduce-Prozesse speichert seine Ergebnisdatei im Dateisystem ab.
- Die Reduce-Phase ist beendet, sobald alle Ergebnisdateien abgespeichert worden sind. Somit ist auch die gesamte Durchführung des Map/Reduce-Verfahrens beendet.

Dieses sind die grundlegenden Phasen des Map/Reduce-Verfahrens. Realisiert werden sie in einem Map/Reduce-Framework so wie das von Google in C++ implementierte Framework und das darauf basierende Open Source-Map/Reduce-Framework Hadoop.

Die zur Lösung einer Aufgabe benötigte Map-Funktion und Reduce-Funktion muss vom Anwender erstellt werden. Mit ihnen definiert er die Logik seiner Anwendung. Die Verteilung der Daten, der Map-Prozesse, der Reduce-Prozesse sowie die Speicherung der Zwischen- und Endergebnisse werden im Wesentlichen vom Map/Reduce-Framework übernommen. Die Trennung zwischen Anwendungslogik und technischer Seite des Frameworks ermöglicht es dem Anwender, sich auf die Lösung seines Problems zu konzentrieren. Das Framework übernimmt unterdessen intern die schwierigen Details der

- automatischen Parallelisierung und Verteilung der Prozesse,
- Realisierung von Fehlertoleranz bei Ausfall von Hardware und Software,
- I/O-Scheduling,
- Bereitstellung von Statusinformationen und Überwachungsmöglichkeiten.

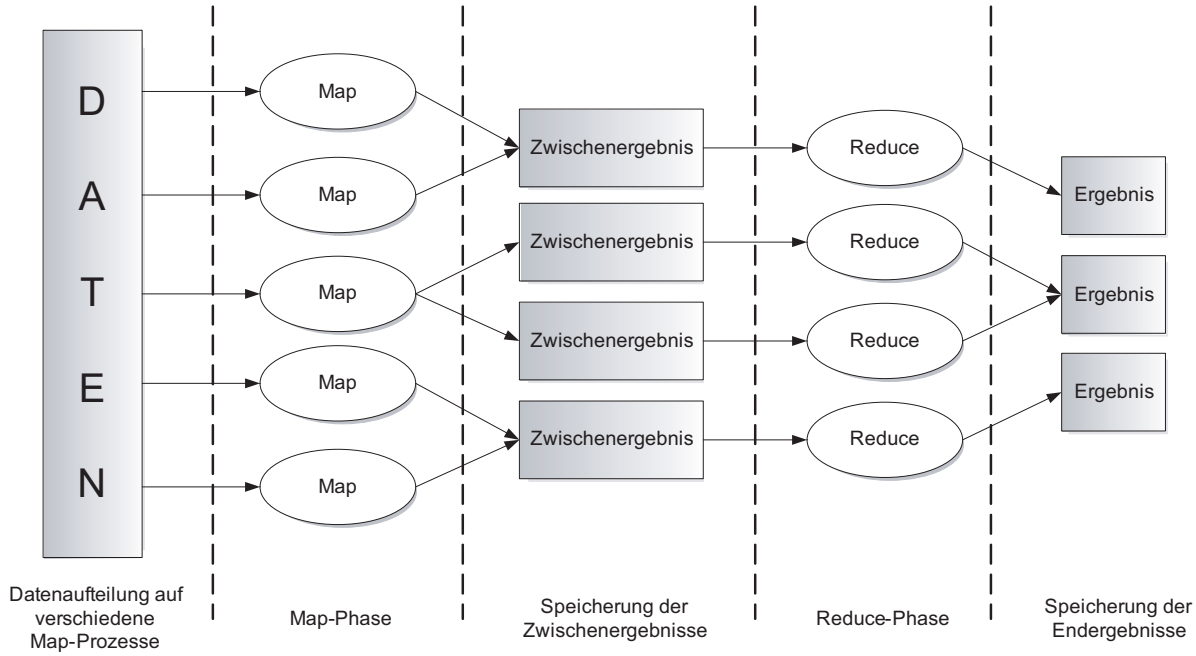


Abbildung 2.1.4 Datenfluss und Phasen des Map/Reduce-Verfahrens

- Der Programmierer muss lediglich die beiden namensgebenden Funktionen *map()* und *reduce()* spezifizieren, welche nachfolgend im Pseudocode angegeben werden.

```
map (in_key, in_value) -> list(out_key, intermediate_value)
reduce (out_key, list(intermediate_value)) -> list(out_value)
```

Die Argumente der `map`-Funktion sind ein Key/Value-Paar: `(in_key, in_value)`. Die `map`-Funktion bildet daraus eine Liste von neuen Key/Value-Paaren: `list(out_key, intermediate_value)`, sie sind die Zwischenergebnisse der Map-Phase, wie auch in Abbildung 2.1.4 dargestellt ist. Sobald alle Zwischenergebnisse vorliegen, wird die Reduce-Phase gestartet. In ihr werden die Werte der Zwischenergebnisse `(intermediate_value)` zu einem bestimmten Schlüssel `out_key` als Liste kombiniert:

```
(out_key,list(intermediate_value))
```

Sie sind die jeweiligen Eingabewerte der `reduce`-Funktionen, die daraus einen Satz von fusionierten Ergebnissen berechnen.

Ein einfaches Beispiel für ein Problem, das mit dem Map/Reduce-Algorithmus gelöst werden kann, ist die Analyse der Worthäufigkeit in einem umfangreichen Text. Seit es in [Dean04] zur Beschreibung des Map/Reduce-Verfahrens herangezogen wurde, hat es sich zum klassischen Beispiel entwickelt. Man stellt sich dazu das Problem der Zählung von gleichen Worten in einer großen Sammlung von Dokumenten vor. Der zur Lösung des Problems notwendige Pseudo-Code sieht dann folgendermaßen aus:

Listing 2.1.4 Pseudo-Code zur Analyse der Worthäufigkeit[7]

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


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

Die `map`-Funktionen erhalten als KeyValue-Paare den Namen und den Inhalt des Dokuments. Beim *Durchlaufen* des Dokuments während der Map-Phase wird dann für jedes Wort `w` das Zwischenergebnis in der Form `(w,"1")` gespeichert. Ist die Map-Phase beendet, so wurde für jedes Wort eine Zwischenergebnisliste angelegt. Bei *n* verschiedenen Wörtern erhält man also *n* Zwischenergebnislisten. In der Reduce-Phase werden diese Zwischenergebnislisten von den `reduce`-Funktionen pro Wort addiert und als Ergebnis zurückgeliefert `Emit(AsString(result))`. Die Ergebnisse können dann in einer Liste mit den Listeneinträgen Wort und Worthäufigkeit zusammengefasst werden.

Weitere Anwendungsbereiche finden Sie in Abschnitt 2.1.4, ein einfaches Anwendungsbeispiel zur praktischen Auseinandersetzung mit einem Map/Reduce-Framework in Abschnitt 2.1.5. Wie schon beschrieben übernimmt das Map/Reduce-Framework die Verteilung der Daten, der Map- und der Reduce-Prozesse sowie die Speicherung der Zwischen- und Endergebnisse. Um diese Aufgaben erledigen zu können, bedarf es weiterer Komponenten. Diese werden im nächsten Abschnitt beschrieben.

2.1.3 Komponenten und Architektur

Nach [Dean04] sind viele verschiedene Implementierungen eines Map/Reduce-Frameworks möglich. Die Wahl eines passenden Frameworks ist vom jeweiligen Anwendungsbereich abhängig. Einige Frameworks sind für die Verarbeitung von Daten im *shared memory* optimiert, andere für die Verarbeitung in *NUMA*[8]-Architekturen oder in großen verteilten Netzwerken. Eine Auflistung verschiedener Map/Reduce-Frameworks ist in Abschnitt 2.1.4 zu finden. Googles Map/Reduce-Framework wurde für die Verarbeitung sehr großer Datenmengen in einem Ethernet-Netzwerk mit Standard-PCs implementiert. Dieses Framework ist für den Einsatz in folgender Umgebung optimiert worden:

- Rechner mit Standard Dual-Prozessoren x86
- 2-4 GB Arbeitsspeicher pro Rechner
- Linux Betriebssystem

[7] [Dean04] Seite 2
[8] Non-Uniform Memory Architecture

- Standardnetzwerkhardware 100 Mbit/s –1 Gbit/s
- Rechnercluster bestehend aus 100 bis mehr als 1.000 Rechnern
- Datenspeicherung auf verteilter Standardhardware mittels GFS[9] (Google File System)
- Die Verarbeitung großer verteilter Datenmengen im Petabyte-Bereich erfordert ein spezielles Dateisystem. Google verwendet hierzu sein eigenes Dateisystem GFS, das für den Umgang mit großen Datenmengen optimiert ist. Ebenso nutzt auch das Open Source-Framework Hadoop eine eigene Implementierung dieses Dateisystems, das HDFS[10] (Hadoop File System). Nach [Grimm09] zeichnen sich diese Dateisysteme durch folgende Eigenschaften aus:
- Verarbeitung großer Datenmengen
- Wenige große Dateien bestehen aus Datenblöcken, typischerweise 64 MByte groß
- Die Datenblöcke liegen redundant (mindestens drei Mal) auf sogenannten Chunk-Servern vor.
- Der Datendurchsatz ist deutlich wichtiger als die Zugriffszeit.
- Streaming-Zugriff auf die Daten. Das heißt insbesondere, dass die Daten meist ganz gelesen werden und die neuen Daten ans Dateiende geschrieben werden.
- Die Daten werden in der Regel einmal geschrieben, aber oft gelesen.
- Das Dateisystem kann mit hohen Fehlerraten umgehen.

Googles Map/Reduce-Framework weist die in Abbildung 2.1.5 dargestellte Architektur und Komponenten auf. Der Datenfluss wird hier noch in erweiterter Darstellung noch einmal etwas genauer betrachtet.

1. Die im Anwendungsprogramm enthaltene Map/Reduce-Library teilt die Eingabedateien in `M` Teile mit einer Größe von 16-64 MB auf. Kopien des Programms werden dann auf mehreren Rechnern innerhalb eines Rechenclusters gestartet.
2. Eine Kopie des Programms übernimmt dabei spezielle Aufgaben, sie wird als *Master* bezeichnet. Die anderen Kopien des Programms bilden die Gruppe der *Worker*, der *Master* weist ihnen die vorhandenen Map-Aufgaben `M` und Reduce-Aufgaben `R` zu.
3. Die Worker, die eine Map-Aufgabe bekommen haben, lesen den korrespondierenden Teil der aufgeteilten Eingabedatei. Ein *Worker* analysiert das Key/Value-Paar der Eingabedaten und verarbeitet sie in der durch den Nutzer definierten Map-Funktion. Die Zwischenergebnisse werden im Speicher als neu erzeugtes Key/Value-Paar gespeichert.
4. Die gespeicherten Zwischenergebnisse werden periodisch auf die lokale Festplatte geschrieben und durch eine Partitionierungsfunktion in `R` Partitionen aufgeteilt. Die Adressen dieser Partitionen werden dem Master mitgeteilt, der nun für die Weiterleitung dieser Adressen an die *Worker* für die Reduce-Aufgabe zuständig ist.

---

9 *http://labs.google.com/papers/gfs.html*
10 *http://hadoop.apache.org/hdfs/*

5. Erhält ein *Reduce-Worker* die Adressdaten von R, so greift er über RPC[11] auf diese Daten zu. Er sortiert die Daten anhand des Schlüssels und gruppiert alle Daten mit gleichem Schlüssel.
6. Der Reduce-Worker iteriert über diese sortierten Zwischenergebnisse und übergibt die Key/Value-Paare für jeden Schlüssel an die Reduce-Funktion, welche die Liste von Werten der Partition zu einem Ausgabewert akkumuliert.
7. Sobald alle Worker ihre Aufgaben beendet haben, wird die Kontrolle vom *Master* wieder an das Anwendungsprogramm übergeben. Nach erfolgreicher Bearbeitung aller Aufgaben sind die Ergebnisse in R Ausgabedateien zu finden. Diese Dateien können zusammengefasst werden oder wiederum die Eingaben für einen weiteren Map/Reduce-Durchlauf bilden.

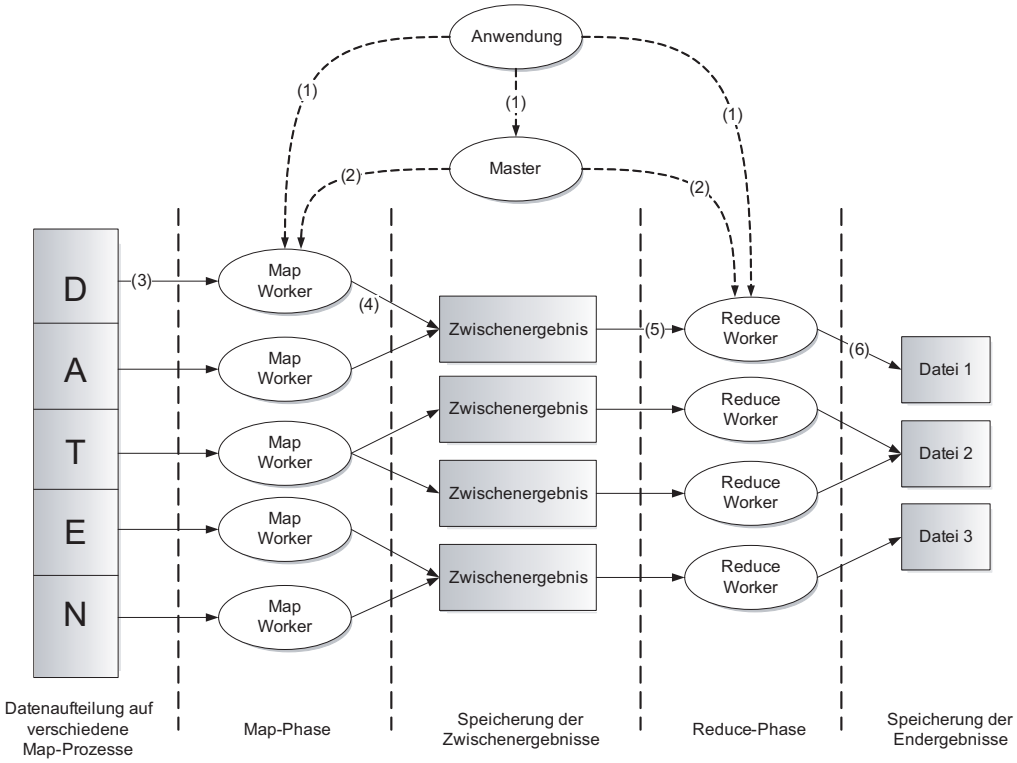


Abbildung 2.1.5 Erweitertes Komponenten- und Datenflussmodell

Im erweiterten Datenflussmodell lassen sich somit die folgenden Komponenten ermitteln:

- Eine im Anwendungsprogramm enthaltene Komponente der Map/Reduce-Library zum Einlesen der Daten zur Datenaufteilung und zur Aufteilung des *Master*- und der *Worker*-Prozesse in einem Rechnercluster

[11] Remote Procedure Call

- Ein *Master*-Prozess zur Verwaltung der *Worker*-Prozesse und zur Überwachung und Verteilung der Map- und Reduce-Aufgaben
- Die jeweiligen Map- und Reduce-Funktionen
- Eine Partitionierungsfunktion zur Aufteilung, Speicherung und Verteilung der Zwischenergebnisse
- Eine Sortierfunktion im *Reduce-Worker*
- Eine Ausgabefunktion zur Speicherung der Endergebnisse
- Je nach Anwendung können auch weitere Komponenten zum Einsatz kommen:
- Eine *Combiner*-Funktion zwischen Map- und Reduce-Phase zur Reduzierung der Netzwerklast durch die Zusammenfassung identischer Ergebnisse der Map-Phase
- Webinterface zur Überwachung und Darstellung von Statusinformationen
- Spezifizierbare Eingabe- und Ausgabefunktionen zur Verknüpfung mehrer Map/Reduce-Durchläufe
- Lokale sequenzielle Map/Reduce-Implementierungen zur Fehleranalyse

2.1.4 Anwendungsbereiche und Implementierungen

Mit einem Map/Reduce-Framework lässt sich eine Vielzahl von Aufgaben lösen, die eine Berechnung großer Datenmengen in einem Verbund von Rechnern erfordern. In [Dean04] (S. 2-3) werden die folgenden Einsatzbereiche skizziert:

- **Verteiltes Suchen (Grep)**: Beim Suchen von Mustern über verteilte Daten liefert die Map-Funktion eine Zeile, wenn der Inhalt der Zeile dem Suchmuster entspricht. Die Reduce-Funktion ist eine Identitätsfunktion welche die Zwischenergebnisse nur zur Liste der Endergebnisse kopiert.
- **Zählen von Zugriffen auf eine URL**: Hier verarbeiten die Map-Funktionen die Zugriffe auf Webseiten aus vorhandenen Aufzeichnungen und geben, ähnlich dem Zählen der Worthäufigkeit, die Key/Value-Paare `(URL,"1")` aus. Die Reduce-Funktionen addieren diese Zwischenergebnisse pro URL und liefern als Ergebnis die Zugriffe pro URL `(URL,total_count)`.
- **Erstellung von Graphen über Verlinkung von Webseiten zu einem Ziel**: Die Map-Funktionen erzeugen Key/Value-Paare in der Form `(target,source)` von jeder Verbindung zu einer Ziel-URL `(target)`, die auf einer Quellen-URL `(source)` gefunden wird. Die Reduce-Funktionen konkatenieren die Liste aller Quellen-URLs in Verbindung mit der Ziel-URL und liefern diese Verbindungen als Ergebnis in der Form von `(target,list(source))`.
- **Ermittlung des Term-Vectors per Host**: Ein *Term-Vector* fasst die wichtigsten Worte zusammen, die in einem Dokument oder in eine Gruppe von Dokumenten auftreten. Dieser Vector wird in Form einer Liste von Wort-Wortfrequenz-Paaren `(word, frequency)` dargestellt. Die Map-Funktionen liefern für jedes zum *Host* gehörende Dokument die Key/Value-Paare `(hostname,term_vector)`. Die Reduce-Funktionen über-

prüfen alle Term-Vectoren pro Dokument und fassen diese Vectoren für die Dokumente zusammen. Dabei werden Terme mit geringer Frequenz verworfen und ein abschließendes Paar `(hostname,term_vector)` geliefert.

- **Wortindex Erstellung**: Die Map-Funktionen analysieren jedes Dokument und liefern Key/Value-Paare in Form von Wort-Dokumentennummer-Paaren `(word, document_ID)`. Die Reduce-Funktionen sortieren diese Zwischenergebnisse für jedes Wort nach der Dokumentennummer und fassen diese Daten als Liste zusammen. Als Ergebnis wird ein invertierter Wordindex geliefert `(word,list(document_ID)`.
- **Weitere Anwendungsbereiche:**
  - Sortieren verteilter Daten
  - Kategorisierung von Daten zur Nachrichtenaufbereitung
  - Auswertung beliebiger Logdateien
  - Aufbereitung von Daten für häufige Aufrufe

Das Map/Reduce-Verfahren ist für viele Anwendungsbereiche effizient anwendbar, bei denen es um die Verarbeitung sehr großer Datenmengen geht. Aus diesem Grunde wurden seit der ersten Vorstellung zahlreiche Implementierungen erzeugt, wie in der folgenden Auflistung zu sehen ist[12]:

- Das Google Map/Reduce Framework ist in C++ implementiert und bietet APIs für Python und Java.
- Hadoop ist ein Apache Open Source-Java-Framework für skalierbare, verteilte Software. Offizielle Website: *http://hadoop.apache.org/mapreduce/*
- Twister ist eine Open Source-Java Map/Reduce-Implementierung.
  Offizielle Website: *http://www.iterativemapreduce.org/*
- Greenplum ist eine kommerzielle Map/Reduce-Implementierung. Sie bietet APIs für Python, Perl, SQL und weitere Sprachen.
  Offizielle Website: *http://www.greenplum.com/technology/mapreduce/*
- Aster Data Systems. Das Map/Reduce-Framework unterstützt Java, C, C++, Perl und Python zum Formulieren von SQL-Funktionen in ANSI SQL.
  Offizielle Website: *http://www.asterdata.com/*
- GridGain bietet eine Open Source-Java-Map/Reduce-Implementierung.
  Offizielle Website:
  http://www.gridgainsystems.com/wiki/display/GG15UG/MapReduce+Overview
- Phoenix ist ein Map/Reduce-Framework für *shared-memory.* Es wurde an der Stanford Universität Kalifornien in der Programmiersprache C entwickelt.
  Offizielle Website: *http://mapreduce.stanford.edu/*
- FileMap ist ein dateibasiertes Map/Reduce-Framework. Die Open Source-Version ist auf GitHub zu finden: *http://mfisk.github.com/filemap/.*

[12] Vgl. *http://en.wikipedia.org/wiki/MapReduce, 14.10.2010*

- Map/Reduce wurde auch für die IBM-Cell-Prozessorserie (*Cell Broadband Engine*) entwickelt: *http://www.cs.wisc.edu/techreports/2007/TR1625.pdf.*
- Map/Reduce wurde auch für NVIDIA GPUs (*Graphics Processors*) mittels *CUDA* entwickelt: *http://www.cse.ust.hk/gpuqp/Mars_tr.pdf.*
- Qt-Concurrent bietet eine vereinfachte Version des Frameworks in C++ zur verteilten Bearbeitung von Aufgaben auf einen Multi-Core-Prozessor.
  Offizielle Website: *http://qt.nokia.com/*
  Dokumentation: *http://doc.trolltech.com/4.6/qtconcurrentmap.html*
- CouchDB nutzt ein Map/Reduce-Framework zur Definition von *views* über verteilte Dokumente. Mehr zur NoSQL-Datenbank ist im Kapitel *4.1 CouchDB* zu finden.
- Skynet ist eine Open Source-Ruby-Implementierung von Googles Map/Reduce-Framework. Offizielle Website: *http://skynet.rubyforge.org/*
- Disco ist eine Open Source-Map/Reduce-Implementierung von Nokia. Das Framework ist in *Erlang* geschrieben, Aufgaben werden in *Python* erstellt.
  Offizielle Website: *http://discoproject.org/*
- Qizmt ist ein C#-Open Source-Map/Reduce-Framework auf MySpace.
  Offizielle Website: *http://qizmt.myspace.com/*
- Das Holumbus-Projekt beinhaltet eine *Haskell*-Map/Reduce-Implementierung. Offizielle Website: *http://holumbus.fh-wedel.de*
- BashReduce ist eine Map/Reduce-Implementierung in Form eines *Bash*-Scripts, welches von Erik Frey auf GitHub veröffentlicht wurde.
  BashReduce auf GitHub: *http://github.com/erikfrey/bashreduce*
- Sector/Sphere ist eine Open Source-Map/Reduce-Variante in C++. Website auf *SourceForge*: *http://sector.sourceforge.net*
- Map/Reduce für die Google Sprache Go.
  Auf GitHub: *http://github.com/dbravender/go_mapreduce*
- MongoDB. Auch die document-orientierte NoSQL-Datenbank nutzt das Map/Reduce-Verfahren. Mehr dazu ist in Abschnitt 4.2 *MongoDB* zu erfahren.
- Amazon Elastic MapReduce, ein Map/Reduce-Service der die Hadoop-Implementierung innerhalb der Amazon-Webservice-Umgebung nutzt:
  http://aws.amazon.com/elasticmapreduce

2.1.5 Praktisches Beispiel

Eine Vielzahl von praktischen Beispielen ist mittlerweile im Internet zu finden. Dabei hat sich das Zählen der Wortfrequenz in einem oder mehreren Dokumenten zum klassischen Beispiel entwickelt. In Abschnitt 3.1 *Hadoop/HBase* ist ein weiteres typisches Einsatzbeispiel im Kontext der NoSQL-Datenbanken beschrieben: die Abfrage von NoSQL-Datenbanken. In diesem kleinen praktischen Beispiel wollen auch wir die Wortfrequenz in Dokumenten ermitteln. Wir nutzen dazu eine vereinfachte Version eines Map/Reduce-Frame-

works, welches darauf ausgelegt ist, die Aufgaben innerhalb eines Rechners mit einem Multi-Core-Prozessor auf mehrere *Threads* aufzuteilen. Es handelt sich hierbei um die *QtConcurrent*-API innerhalb der ehemals von der norwegischen Firma *Trolltech* entwickelten plattformunabhängigen C++-Klassenbibliothek *Qt*, die 2008 von der Firma *Nokia* aufgekauft wurde und seitdem unter dem Namen *Qt Development Frameworks* weiterentwickelt und vertrieben wird.

Unter *http://qt.nokia.com/products* kann die aktuelle Version des *Software Development Kits* (SDK) bezogen werden. Neben einer kommerziellen Version kann auch eine kostenfreie Version unter LGPL[13]-Lizenz heruntergeladen werden. Im SDK ist auch die *Qt Creator IDE* enthalten, mit der man das folgende Beispiel bequem erstellen und direkt kompilieren kann. Die Installation gestaltet sich unter Windows recht einfach: Das heruntergeladene *Qt SDK* kann direkt ausgeführt werden, und anschließend wird man durch die einzelnen Installationsschritte geführt. Unter *http://doc.qt.nokia.com/4.6/installation.html* sind weitere Informationen zur Installation auch auf anderen Plattformen zu finden:.

Nun zu unserem Beispiel. Zur Speicherung der Key/Value-Paare bietet sich die assoziative Containerklasse `QMap<K,T>` an, die Key/Value-Paare in aufsteigender Reihenfolge aufnimmt. Da man diese Container vom Typ `<QString,int>` mehrfach benötigt, deklarieren wir für diese Template-Klasse den Alias `KeyValue`.

Listing 2.1.5 typedef-Deklaration für die Key/Value-Paare

```
typedef QMap<QString,int> KeyValue;
```

Den zu analysierenden Text bringen wir in einer Liste von Zeichenketten unter, hierzu können wir die Klasse `QStringList` verwenden.

Listing 2.1.6 Initialisierung der Zeichenkettenliste list

```
QStringList list;
//Gedicht von Heinz Erhart
list << "Das Nasshorn und das Trockenhorn"
     << "spazierten durch die Wueste"
     << "das Trockenhorn verdurstete"
     << "und das Nasshorn sagte"
     << "siehste";
```

Die Map-Funktion teilt den übergebenen Text in einzelne Worte auf, zählt diese und gibt das Ergebnis als Key/Value-Paar zurück. Diese Funktion wird parallel von mehreren *Threads* aufgerufen.

Listing 2.1.7 Definition der Map-Funktion

```
KeyValue mapFunction(const QString &document)
{
    KeyValue keyValue;
    foreach (QString word, document.split(" "))
    keyValue [word] += 1;
    return keyValue;
}
```

[13] GNU Lesser General Public License

Und natürlich wird auch eine Reduce-Funktion benötigt, sie iteriert über die Zwischenergebnisse der Map-Phase mittels des Map-Iterators: `QMapIterator<QString,int> i(intermediateResult);` und berechnet daraus das Endergebnis.

Listing 2.1.8 Definition der Reduce-Funktion

```
void reduceFunction(KeyValue &finalResult,
                    const KeyValue &intermediateResult)
{
    QMapIterator<QString, int> i(intermediateResult);
    while (i.hasNext()) {
        i.next();
        finalResult[i.key()] += i.value();
    }
}
```

Schließlich wird alles mit der Map/Reduce-Funktion des Qt-Frameworks zusammen ausgeführt, indem der Funktion die Textliste sowie die beiden erstellten Funktionen übergeben wird.

Listing 2.1.9 Aufruf der Map/Reduce-Funktion des Qt-Frameworks

```
KeyValue finalResult = mappedReduced(list, mapFunction, reduceFunction);
showResult(finalResult);
```

Das Endergebnis wird durch die Funktion `showResult(finalResult)` ausgegeben (siehe Listing 2.1.13).

Listing 2.1.10 Definition der Funktion zur Ausgabe der Key/Value-Paare

```
void showResult(const KeyValue &result)
     {
         QMapIterator<QString, int> iter(result);
         while (iter.hasNext()) {
             iter.next();
             std::cout << iter.key().toStdString() << ": "
                       << iter.value() << std::endl;
         }
         std::cout << std::endl;
     }
```

In Listing 2.1.11 wird das gesamte Programm dargestellt. Zusätzlich benötigt das Qt-Framework noch eine Projektdatei. Diese wird in Listing 2.1.12 dargestellt. Das Beispiel lässt sich am einfachsten mit der *Qt Creator IDE* erstellen: über den Menüpunkt `Datei->Neu->leeres Qt4-Projekt`, Namen angeben, z.B. `wordfrequenz`. Die Qt-Projektdatei wird dann direkt erstellt und kann editiert werden. Über `Datei->Neu->C++ Quelldatei` kann dann auch die Datei `main.cpp` erstellt werden. Über den Menüpunkt `Erstellen` kann das Programm bequem erstellt und ausgeführt werden.

Listing 2.1.11 Die gesamte Qt-Quelldatei main.cpp

```
#include <QList>
#include <QMap>
#include <QString>
#include <QStringList>
#include <QApplication>
#include <QDebug>
#include <iostream>
```

```
#include <qtconcurrentmap.h>

#ifndef QT_NO_CONCURRENT
// Namensraum für die Funktion mappedReduced im Qt-Framework
using namespace QtConcurrent;

// Deklaration des Containers der Key/Value-Paare
typedef QMap<QString,int> KeyValue;

//Die mapFunction teilt den übergebenen Text in einzelne Worte auf,
//zählt diese und gibt das Ergebnis als Key/Value-Paar zurück. //Diese
Funktion wird parallel von mehreren Threads aufgerufen.
KeyValue mapFunction(const QString &document)
{
    KeyValue keyValue;
    foreach (QString word, document.split(" "))
        keyValue [word] += 1;
    return keyValue;
}

//Die Reduce-Funktion iteriert über die Zwischenergebnisse
//der Map-Phase und berechnet daraus das Endergebnis.
void reduceFunction(KeyValue &finalResult,
                    const KeyValue &intermediateResult)
{
    QMapIterator<QString, int> i(intermediateResult);
    while (i.hasNext()) {
        i.next();
        finalResult[i.key()] += i.value();
    }
}

//Funktion zur Ausgabe der Key/Value-Paare
void showResult(const KeyValue &result)
{
    QMapIterator<QString, int> iter(result);
    while (iter.hasNext()) {
        iter.next();
        std::cout << iter.key().toStdString() << ": "
                  << iter.value() << std::endl;
    }
    std::cout << std::endl;
}

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QStringList list;
    //Gedicht von Heinz Erhardt
    list << "Das Nasshorn und das Trockenhorn"
         << "spazierten durch die Wueste"
         << "das Trockenhorn verdurstete"
         << "und das Nasshorn sagte" << "siehste";

    //Aufruf der Map/Reduce-Funktion des Qt-Frameworks
    KeyValue finalResult = mappedReduced(list, mapFunction,
                                         reduceFunction);

    //Ausgabe der Endergebnisse
    showResult(finalResult);
}

#else
int main()
{
    //Ausgabe bei nicht unterstützten Plattformen!
    qDebug() << "Qt Concurrent is not yet supported on this platform";
}
#endif
```

Listing 2.1.12 Qt-Projektdatei wordfrequenz.pro

```
TEMPLATE = app
TARGET +=
DEPENDPATH += .
INCLUDEPATH += .

# Input
 SOURCES += main.cpp
 CONFIG += console
```

Listing 2.1.13 Konsolenausgabe des Endergebnisses

```
Das: 1
Nasshorn: 2
Trockenhorn: 2
Wueste: 1
das: 3
die: 1
durch: 1
sagte: 1
siehste: 1
spazierten: 1
und: 2
verdurstete: 1
```

Dieses kleine Beispiel sollte die grundlegende Vorgehensweise beim Erstellen einer Anwendung für ein Map/Reduce-Framework aufzeigen und zum Experimentieren anregen. Die meisten Beschreibungen der verschiedenen Map/Reduce-Frameworks enthalten ähnliche Beispiele, die zum Ausprobieren einladen.

2.1.6 Zusammenfassung

Das Map/Reduce-Verfahren spielt im Kontext der NoSQL-Datenbanken eine zentrale Rolle. Damit lassen sich große verteilte Datenmengen bei paralleler Ausführung effizient durchsuchen. NoSQL-Datenbanken wie z.B. CouchDB, MongoDB, Riak und HBase nutzen das Map/Reduce-Verfahren zur Abfrage ihrer Datenbankeinträge. Die Ursprünge des Map/Reduce-Verfahrens liegen in der funktionalen Programmierung, welches die Parallelisierung innerhalb der Map/Reduce-Phasen ermöglicht. Die zur Lösung einer Aufgabe benötigte Map-Funktion und Reduce-Funktion muss vom Anwender erstellt werden. Mit ihnen definiert er die Logik seiner Anwendung. Die Verteilung der Daten, von Map- und Reduce-Prozessen sowie die Speicherung der Zwischen- und Endergebnisse werden im Wesentlichen vom Map/Reduce-Framework übernommen. Das Framework ermöglicht außerdem:

- Parallelisierung
- Fehlertoleranz
- Datentransfer
- Lastverteilung
- Monitoring

Zur Verarbeitung großer verteilter Datenmengen im Petabyte-Bereich wird ein spezielles Dateisystem benötigt, nämlich ein verteiltes Dateisystem wie GFS oder HDFS, das für den

Umgang mit großen Datenmengen optimiert ist. Die Verarbeitung und Speicherung großer Datenmengen in verteilten Systemen wird weiter zunehmen. Ein Map/Reduce-Framework ermöglicht es, diese großen Datenmengen effizient zu durchsuchen und zu verarbeiten. Zahlreiche Implementierungen sind inzwischen entstanden.

Links & Literatur

[Dean04] Jeffrey Deans und Sanjay Ghemawat: MapReduce: Simplified Data Processing on LargeClusters. In OSDI'04, 6th Symposium on Operating Systems Design andImplementation, Sponsored by USENIX, in cooperation with ACM SIGOPS, 2004. *http://labs.google.com/papers/mapreduce.html*

[Grimm09] Rainer Grimm: Funktionale Programmierung 3 – Das MapReduce-Framework, Linux Magazin 17.11.2009, *http://www.linux-magazin.de/Online-Artikel/Funktionale-Programmierung-3-Das-MapReduce-Framework?category=380*

[Rech02] Peter Rechenberg, Gustav Pomberger: Informatik Handbuch. 2. Auflage. Hanser Fachbuch, Februar 2002.

[Lämm07] Ralf Lämmel: Google's MapReduce Programming Model – Revisited. Data Programmability Team, Microsoft Corp., Redmond, WA, USA. Science of Computer Programming, Volume 70, Issue 1, 1 January 2008, Received 9 February 2006; revised 10 July 2007; accepted 10 July 2007. Available online 18 July 2007.

Haskell – Homepage zum Buch: *http://book.realworldhaskell.org/*

Hadoop – Dokumentation: *http://hadoop.apache.org/mapreduce/*

Wikipedia – *http://en.wikipedia.org/wiki/MapReduce*

Wikipedia – *http://de.wikipedia.org/wiki/MapReduce*

Heise online – *http://www.heise.de/newsticker/meldung/Google-laesst-Map-Reduce-patentieren-908531.html*

Linux Magazin – *http://www.linux-magazin.de/NEWS/Online-Artikel-Googles-MapReduce-Framework-und-Hadoop*