

Programmierunterstützung im Kontext von Cloud Computing

Alexander Singer
AlexSinger@gmx.net

ABSTRACT

Mit dem Aufkommen des Cloud Computing wurden große Rechner-Cluster für jedermann verfügbar. Um diese Rechenleistung der Cloud aber auch wirklich ausnutzen zu können, müssen Programme derart geschrieben werden, dass sie auch tatsächlich parallel berechenbar sind. Eine einfache Möglichkeit der fast automatischen Parallelisierung bietet das von Google patentierte [1] Konzept *MapReduce*, das dank einiger Open-Source-Implementierungen für jedermann nutzbar ist.

Die eigentliche Berechnung geschieht in zwei vom Benutzer geschriebenen Funktionen. In der *map*-Funktion werden zuerst die Eingabedaten auf Schlüssel-Wert-Paare abgebildet, um diese anschließend in der *reduce*-Funktion zu einer Ausgabe zusammenzuführen. Die Zuweisung der Funktionen an Rechner des Clusters, die Aufteilung der Daten und die Abstimmung der Rechner übernimmt die Infrastruktur.

Neben dem ursprünglichen Google-Konzept gibt es dabei auch noch mehrere Open-Source-Implementierungen, wie *Disco* [2] oder *Hadoop MapReduce* [3].

Durch diese Unterstützung durch die Infrastruktur begiebt sich der Benutzer von MapReduce auf eine Abstraktionsebene, auf der die Daten und deren Fluss betrachtet werden, anstatt die Koordination von Rechnern oder Prozessen. Ein weiteres Projekt, das sich sehr intensiv mit diesem *datenzentrischen Programmieren* auseinandersetzt ist BOOM [4].

1. EINFÜHRUNG

Da das Datenaufkommen vor allem durch das Internet immer schneller wächst und die Verarbeitung dieser Daten damit immer aufwändiger wird, reichen einzelne Rechner mittlerweile für die Verarbeitung dieser Daten nicht mehr aus. Damit Ergebnisse trotzdem in akzeptabler Zeit verfügbar sind, werden aufwändige Berechnungen daher auf Rechner-Clustern ausgeführt.

Allerdings bringt dies neue Probleme mit sich. Zum einen müssen die Verarbeitungsprogramme stark parallelisiert werden, sodass möglichst alle Rechner im Cluster gleichzeitig arbeiten können, zum anderen erwächst aus der großen Anzahl von Rechnern auch das Problem, dass Ausfälle keine Einzelfälle mehr sind, sondern zur Regel werden.

Die Behandlung dieser Probleme wurde bisher in fast jedem Programm von den Entwicklern selbst geschrieben, um Dinge wie Parallelisierung, Datenverteilung, Lastenausgleich und Fehlertoleranz abzudecken. Durch den dafür nötigen Code wurden selbst Programme, deren eigentliche Berechnung relativ simpel war sehr komplex.

Bis vor ein paar Jahren war dies allerdings nur ein Problem, mit dem sich sehr wenige Entwickler beschäftigen muss-

ten, da sich nur wenige Unternehmen Cluster mit hunderten bis tausenden von Rechnern leisten konnten. Durch das *Cloud Computing* werden diese Kapazitäten jetzt allerdings einem deutlich größeren Kreis von Nutzern zur Verfügung gestellt, weshalb es von Vorteil ist nach Techniken zu suchen, die es Benutzern erlauben, mit möglichst geringem Aufwand große Datenmengen in einem Cluster aus vielen kleineren Rechnern zu bearbeiten.

Vor allem Google, das unter anderem für seine Suchmaschine viele Daten sammelt und in für sie brauchbare Form bringen muss, hat dies erkannt und mit MapReduce ein Framework geschrieben, das dem Entwickler den Großteil dieser Arbeit abnimmt und für eine Vielzahl an häufig benötigten Arbeitsschritten verwendet werden kann. Da mittlerweile dieses MapReduce-Konzept immer beliebter wird und schon einige Cloud-Anbieter wie Amazon ein bereits vorinstalliertes MapReduce-Framework anbieten [5], wird dieses Konzept im Folgenden genauer untersucht.

MapReduce teilt die eigentliche Berechnung auf zwei Funktionen, *map* und *reduce*, auf, die beide vom Benutzer geschrieben werden müssen. Die *map*-Funktion wird vom Framework zusammen mit jeweils einem Teil der Eingabedaten verschiedenen Rechnern des Clusters zugewiesen. Auf diesen Rechnern werden daraus Zwischenergebnisse in Form von Schlüssel-Wert-Paaren berechnet, welche wiederum an andere Rechner weitergegeben werden, die eine vom Benutzer geschriebene *reduce*-Funktion erhalten haben. Nachdem alle Zwischenergebnisse die *reduce*-Funktion durchlaufen haben, steht am Ausgang das Gesamtergebnis der Berechnung. Dabei übernimmt das Framework die gesamte Arbeit, die nötig ist, um die vielen Rechner im Cluster auf einander abzustimmen, wie die Aufteilung der Eingabedaten, die Verteilung der *map*- und *reduce*-Funktionen an die verschiedenen Rechner, die Kommunikation der Rechner untereinander und die Behandlung von Ausfällen.

Neben der bereits erwähnten Google-Variante [6] wird dabei noch ein Blick auf die Open-Source-Implementierung *Disco* geworfen, mit der jedermann relativ leicht eine MapReduce-Umgebung aufsetzen kann, sei es in Clouds wie Amazon EC2 [7] oder im lokalen Cluster.

Während sich Disco sehr stark am Google-Konzept orientiert und wie die anderen MapReduce-Implementierungen es dem Benutzer erlaubt in einem datenzentrischen Stil zu programmieren, so versucht das *BOOM-Projekt* anhand von Reimplementierungen der MapReduce- und Dateisystem-Komponenten von Hadoop im datenzentrischen Stil unter Verwendung von Relationen und einer deklarativen Sprache herauszufinden, in wie weit sich dieses datenzentrische Pro-

grammieren zum Entwickeln von Programmen für Cluster eignet [8].

Bei der Betrachtung der verschiedenen MapReduce-Implementierungen fällt auf, dass fast jede Variante für die Verwendung mit einem verteiltem Speichersystem konzipiert ist. Deshalb werden auch diese Speichersysteme und ihre Auswirkungen auf MapReduce-Operationen untersucht. Von besonderem Interesse ist dabei, an welchen Stellen die Art des Speichersystems für die Performanz einer MapReduce-Operation ausschlaggebend ist.

2. GOOGLE MAPREDUCE

Da Google schon immer vor dem Problem stand, Berechnungen auf einem riesigen Datenbestand in vertretbarer Zeit durchführen zu müssen, sah sich das Unternehmen gezwungen, diese Berechnungen an Rechner-Cluster zu vergeben, welche im Fall von Google, wie beim Google File System [9] aus hunderten bis tausenden „Commodity“-Maschinen bestehen. Dabei spielt für diese Berechnungen nicht nur die Prozessorleistung eine Rolle, sondern vor allem auch die Geschwindigkeit, mit der die Eingabedaten gelesen und verteilt werden.

Da dies neue große Probleme wie die Parallelisierung der Bearbeitung oder die optimale Verteilung der Daten und Rechenlast auf die verschiedenen Maschinen mit sich bringt, mussten ständig spezielle Programme entworfen werden, die sich neben der Bereitstellung des eigentlichen Dienstes auch um diese Probleme kümmerten. Ein weiterer Aspekt, der durch die Verwendung dieser großen Anzahl Rechner betrachtet werden musste, ist die nun sehr viel größer gewordene Ausfallwahrscheinlichkeit einzelner oder mehrerer Rechner. Auch dies musste bei jedem Programm, dass eine Berechnung im Cluster durchführen soll, berücksichtigt werden.

Als Reaktion auf diese Probleme suchte man bei Google nach einer Möglichkeit, die Behandlung dieser Probleme auszulagern, sodass sich die Entwickler auf die vergleichsweise einfache Lösung des eigentlichen Problems konzentrieren können.

Dabei wurde erkannt, dass sich viele Berechnungen der bisher verwendeten Anwendungen auf eine Berechnung in zwei Phasen abbilden lassen, in denen zuerst aus den Eingabedaten Zwischenergebnisse erstellt und diese anschließend zusammengeführt werden. Inspiriert von Lisp und anderen funktionalen Programmiersprachen wurden diese beiden Funktionen *map* und *reduce* genannt, woraus der Name *MapReduce* entstand [6].

Zur Gruppe der Anwendungen, die durch eine einfachere Implementierung unter Verwendung dieser *MapReduce*-Frameworks ersetzt werden konnten, gehören Programme, die aus den gesammelten Rohdaten, wie Logs von Seitenaufrufen oder analysierten Dokumenten im Internet, verschiedene Daten errechneten, die für das Unternehmen interessant sind. Beispiele hierfür sind die Berechnung der meistgesuchten Begriffe in der Suchmaschine, Indices zum Auffinden von Webseiten zu bestimmten Begriffen oder die Graphenstruktur von Internetseiten.

2.1 Programmiermodell

Das Ziel von Google war es nun, dass der Entwickler nur noch diese beiden Phasen, die für die gewünschte Verarbeitung der Daten allein verantwortlich sind, in Form von zwei Funk-

tionen schreiben muss. Diese werden dann an das Framework übergeben, welches sich um den ganzen Rest kümmert.

In der *map*-Funktion bildet man dabei einen Eingabe-Datensatz auf eine temporäre Menge an Schlüssel-Wert-Paaren ab und gibt diese an die *reduce*-Funktion weiter.

Diese *reduce*-Funktion läuft dann per Iterator über diese Paare und fügt die Werte eines Schlüssels zusammen, um eine geringere Wertemenge zu erhalten.

2.1.1 Beispiel: Wörter zählen

Für ein Programm, das das Auftreten von Wörtern in einem Dokument zählt, könnten folgende *map*- und *reduce*-Funktionen definiert werden.

In Pseudocode:

```
map(string docText) :
    für jedes Wort w in docText:
        gebe aus: Paar(w,1)

reduce(Wort w, Iterator iter) :
    int anzahl=0;
    für jedes int i aus iter:
        anzahl = anzahl + i;
    gebe aus: Paar(w, anzahl)
```

Die *map*-Funktion bekommt die Eingabedaten in Form eines Textes. Über dessen einzelne Worte wird iteriert und zu jedem Wort ein Schlüssel-Wert-Paar mit dem jeweiligen Wort als Schlüssel und dem Wert „1“ ausgegeben. Diese Paare werden vor der *reduce*-Funktion sortiert, die dann zu jedem Schlüssel einen Iterator über alle Werte dieses Schlüssels bekommt. Diese Werte werden alle aufsummiert und das Ergebnis als neues Schlüssel-Wert-Paar ausgegeben.

2.1.2 Beispiel: verteiltes grep

Eine weitere Anwendung für *MapReduce* ist ein *verteiltes grep*. Hier werden in der *map*-Funktion die Zeilen der Eingabedaten durchlaufen und die gesuchten Zeilen als Ausgabe weitergeleitet. Die *reduce*-Funktion ist in diesem Beispiel nur eine Weiterleitung ihrer Eingabedaten an den endgültigen Ausgang der *MapReduce*-Operation und berechnet selbst nichts.

Allerdings tritt in dem von Google in [6] vorgestellten Konzept das Problem der Reihenfolge der Ergebnisse auf. Vor jeder *reduce*-Funktion werden die Werte eines Schlüssels sortiert, wodurch die Ausgabezeilen nicht mehr in der gleichen Reihenfolge stehen, wie die Eingabezeilen. Andere *MapReduce*-Implementierungen wie Hadoop umgehen dieses Problem, indem die *reduce*-Phase komplett abgeschaltet werden kann und damit auch der Sortiervorgang entfällt. Als Endergebnis dient dann die Ausgabe der *map*-Funktion.

2.1.3 Beispiel: Indexerstellung

Ein drittes Beispiel für die Anwendungsmöglichkeit von *MapReduce* ist die Erstellung eines Indexes für bestimmte Wörter auf Internetseiten. Die Gesamteingabedaten bestehen aus einer großen Menge Seiten und deren Inhalt, die aufgeteilt und dann als Seitenadresse-Inhalt-Paar an die *map*-Funktion übergeben werden. Dieser Inhalt wird durchsucht und bei einem gefundenen Schlagwort ein Schlüssel-Wert-Paar mit dem Schlagwort als Schlüssel und der Seite als Wert an den Ausgang der *map*-Funktion gegeben.

Die *reduce*-Funktion erhält als Übergabeparameter den Schlüssel, in diesem Fall das Schlagwort, sowie einen Iterator über alle Seiten, auf denen dieses Schlagwort gefunden wurde. In dieser *reduce*-Funktion werden die iterierten Seitenadressen zu einer Liste zusammengefügt und als Schlagwort-Adressliste-Paar ausgegeben.

2.2 Architektur

Der Aufbau eines *MapReduce*-Systems hat eine gewisse Ähnlichkeit zum Aufbau des *Google File Systems* [9]. Auch hier hat sich Google entschieden ein Master-Worker-Konzept zu verwenden.

Der Master, von dem es pro *MapReduce*-Operation nur eine einzige Instanz gibt, ist dafür verantwortlich, einzelnen Workern ihre jeweiligen Aufgaben zuzuteilen und deren Bearbeitung zu überwachen. Indem er regelmäßig mit jedem Worker kommuniziert, erhält er Informationen zu deren Zuständen und kann damit Einplanungsentscheidungen treffen oder Ausfälle behandeln. Außerdem ist er auch noch für die Weitergabe des Speicherorts der *map*-Zwischenergebnisse an die Worker mit *reduce*-Auftrag verantwortlich.

Die Worker, von denen es beliebig viele geben kann, warten darauf, vom Master ihre Aufgabe und Eingabedaten zu bekommen. Da es, um besseren Lastausgleich zu erzielen, deutlich mehr *map*- und *reduce*-Tasks gibt als Worker, werden diese Tasks dynamisch an jeweils freie Worker zugewiesen.

2.3 Ausführung

Um die *MapReduce*-Bibliothek zu nutzen, erstellt der Benutzer zuerst die beiden oben genannten Funktionen *map* und *reduce*, die die eigentliche Berechnung beinhalten. Diese übergibt er an die Bibliothek und spezifiziert zusätzlich einige gewünschte Parameter, wie zum Beispiel die maximale Anzahl zu verwendender Maschinen, Ein- und Ausgabedateien oder den maximalen Speicherverbrauch.

Die im Folgendem aufgeführten geklammerten Nummern entsprechen den Schritten in Abbildung 1.

Wird die *MapReduce*-Operation gestartet, wird zuerst die spezifizierte Anzahl an Worker-Prozessen auf Rechnern des Clusters für die jeweiligen *map*- und *reduce*-Aufgaben sowie ein zusätzlicher Master erstellt (1).

Die Worker bekommen vom Master in Form der *map*- oder *reduce*-Funktionen ihre Aufgabe zugeteilt (2) und lesen den ihnen zugeteilten Teil der Eingabedateien (3). Aus diesem Teil der Eingabe werden vom *map*-Worker temporäre Schlüssel-Wert-Paare erstellt und gespeichert (4). Diese Speicherstelle wird dem Master mitgeteilt, der sie wiederum an einen bereitstehenden Reduce-Worker weitergibt. Dieser liest die Paare per Fernaufruf (5), sortiert sie anhand der vorhandenen Schlüssel und gibt diese in Form eines Iterators an die, vom Benutzer geschriebene, *reduce*-Funktion.

Die Ausgabe dieser *reduce*-Funktion wird von den jeweiligen Workern in eine eigene Ausgabedateien geschrieben (6), sodass am Ende genauso viele Ausgabedateien wie *reduce*-Worker vorhanden sind. Da in der Regel diese Ausgabedateien von einer weiteren *MapReduce*-Operation oder einem anderen verteiltem Programm, das mit verteilten Eingabedaten umgehen kann, verwendet werden, wird das Zusammenführen der Ausgabe in eine einzige Datei weggelassen.

2.4 Ausfallbehandlung

Die Erfahrungen von Google bei der Durchführung von *MapReduce*-Operationen haben gezeigt, dass bei Operatio-

nen auf etwa 160 handelsüblichen Rechnern und einer durchschnittlichen Operationsdauer von 10 Minuten im Schnitt ein Worker-Prozess ausfällt [6]. Dies hat zur Folge, dass die Behandlung von Worker- und Rechnerausfällen sehr wichtig wird. Dabei muss auch unterschieden werden, ob ein Worker ausfällt oder sogar der Master.

Damit Worker-Ausfälle erkannt werden, schickt der Master regelmäßig eine Anfrage an jeden Worker und markiert diesen als ausgefallen, falls die Antwort ausbleibt. Der Zustand des betroffenen *map*- oder *reduce*-Tasks wird in den Ausgangszustand zurückgesetzt und kann neu eingeplant werden. Ein von einem ausgefallenen Worker bereits abgeschlossener *map*-Task wird dabei auch noch einmal wiederholt, da das *map*-Ergebnis auf der lokalen Festplatte des Worker-Rechners gespeichert wird und daher bei einem Rechnerausfall nicht mehr zu erreichen ist.

Sollte der Master ausfallen, so verzichtet Google hier auf Wiederherstellungsmechanismen, und lässt die komplette *MapReduce*-Operation wieder von vorne beginnen. Der Grund für diese Herangehensweise liegt in der im Vergleich zu Worker-Rechnern geringen Ausfallwahrscheinlichkeit des Master-Rechners, da es von diesem nur einen einzigen gibt.

2.5 Performanz und Optimierungen

Um die Performanz zu steigern entwickelte Google zusätzliche Komponenten zum bereits funktionierenden Grundgerüst des *MapReduce*-Frameworks, um vorhandene Flaschenhälse möglichst zu beseitigen. Diese Engpässe können durch den allgemeinen Aufbau und die Zusammensetzung des Clusters entstehen, oder genauso gut durch eine eingeschränkte Funktion einzelner Rechner. Die beiden größten Probleme in diesem Zusammenhang sind die verfügbare Netzwerkbandbreite sowie die Lese- und Schreibgeschwindigkeit der Festplatten um Ein- und Ausgabedaten und Zwischenergebnisse zu lesen und zu speichern. Hier hilft die enge Verzahnung von *MapReduce* mit dem verwendeten Speichersystem.

2.5.1 Zuteilung der Zwischenergebnisse

Aufgrund der Tatsache, dass die Anzahl der *reduce*-Worker von der Anzahl der *map*-Worker abweichen kann, müssen die, von der *map*-Funktion erstellten, temporären Schlüssel-Wert-Paare auf die verschiedenen *reduce*-Worker aufgeteilt werden. Im Normalfall geschieht dies, indem der Schlüssel jedes Paares in eine Hashfunktion gegeben und anschließend modulo der Anzahl der *reduce*-Worker genommen wird.

Allerdings kann es auch von Vorteil sein, bestimmte Paare vom gleichen *reduce*-Worker bearbeiten zu lassen, sodass die berechneten Daten aus diesen Paaren in die gleiche Ausgabedatei geschrieben werden. Für diesen Fall bietet das Google-Konzept dem Benutzer die Möglichkeit diese Partitionierungsfunktion selbst zu definieren.

Diese Partitionierung ist einer der Hauptgründe dafür, dass sich *MapReduce* derart stark parallelisieren lässt. Denn die *reduce*-Funktionen können nur dann parallel ablaufen, wenn sie auf voneinander unabhängigen Daten arbeiten. Um die Unabhängigkeit der Eingabedaten der *reduce*-Funktion zu garantieren, müssen lediglich Zwischenergebnisse mit den gleichen Schlüsseln vom gleichen *reduce*-Worker gelesen werden. Genau diese Tatsache stellt die Partitionierungsfunktion sicher.

2.5.2 Netzwerkbandbreite

Ein praktisches Problem ergibt sich durch die Verwendung handelsüblicher Hardware, denn durch die große Anzahl von

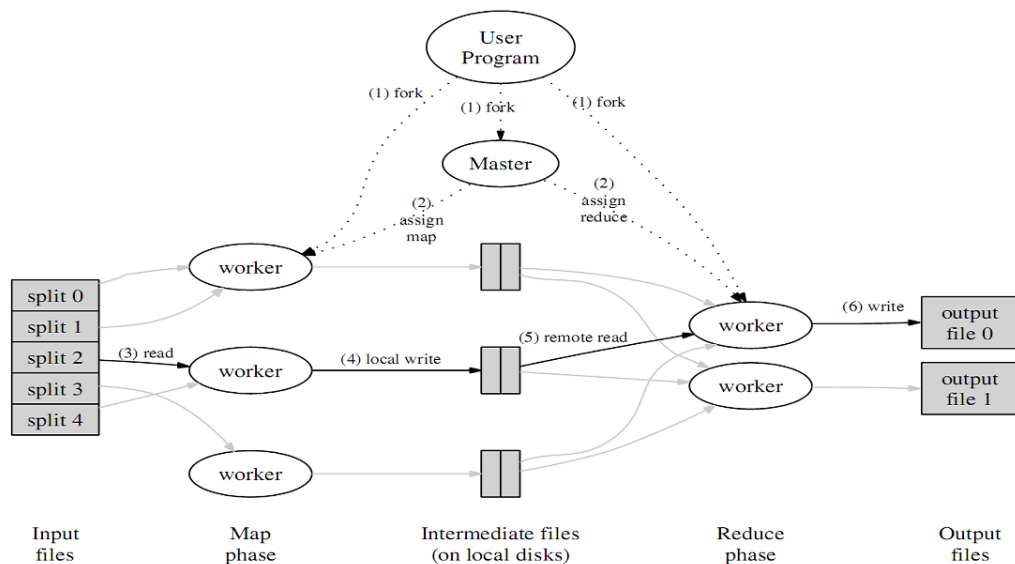


Abbildung 1: Schritte der Ausführung einer Google Map Reduce Operation (aus [6])

Rechnern ergibt sich ein hoher Kommunikationsaufwand, der das verwendete 1 Gbit/s-Netzwerk überlasten kann. Vor allem wenn eine Eingabedatei im Terabyte-Bereich geteilt und verteilt werden soll, stößt das Netzwerk schnell an seine Grenzen.

Allerdings hat Google bereits eine Lösung durch ein eigenes entwickeltes Speichersystem parat, das genau wie die MapReduce-Implementierung, auf ein großes Netzwerk aus Einzelrechnern zugeschnitten ist. Beim Google File System [9] werden Daten mehrfach repliziert und als *Chunks* auf verschiedenen Rechnern im Netzwerk gespeichert. So kann der Master beim Verteilen der *map*-Tasks versuchen, die Worker so zuzuweisen, dass möglichst jeder Rechner genau den Teil bearbeitet, der als Chunk bei ihm lokal gespeichert ist, oder zumindest im Netzwerk sehr nahe liegt.

Durch dieses Zusammenspiel des Google File Systems und MapReduce kann der Flaschenhals Netzwerk gut umgangen werden.

2.5.3 Ein-/Ausgabegeschwindigkeit

Neben der Netzwerkbandbreite gibt es als weiteren limitierenden Faktor die Lese-/Schreibgeschwindigkeit der verwendeten Festplatten. Eine Möglichkeit sowohl den Einfluss dieses Faktors zu verringern als auch das Netzwerk weiter zu entlasten bietet dabei die *combiner*-Funktion. Diese Funktion wird auf jeder Maschine ausgeführt die einen *map*-Task verarbeitet, und stellt im Großen und Ganzen eine vorgezogene *reduce*-Funktion auf dem *map*-Worker-Rechner dar.

Ein gutes Beispiel hierfür bietet die 'Wörter zählen' *MapReduce*-Operation, die in Kapitel 2.1 vorgestellt wurde: Eine normale *map*-Funktion würde aus einem Text eine große Menge von $\langle \text{wort}, 1 \rangle$ -Paaren erzeugen. Liegt zum Beispiel ein englischer Text vor, treten extrem viele $\langle \text{'the'}, 1 \rangle$ -Paare auf, die alle für die *reduce*-Funktion zwischengespeichert werden müssten. Mit der *combiner*-Funktion können diese Paare direkt im *map*-Worker aufaddiert werden, sodass es ausreicht ein einziges Paar $\langle \text{'the'}, \text{\#anzahl_ 'the'} \rangle$ zu speichern.

Hierdurch werden sowohl beim Speichern der Zwischenergebnisse, als auch bei späteren Lesen die Anzahl der nötigen Ein- und Ausgabeoperationen verringert.

2.5.4 Ausfall oder Verzögerung einzelner Rechner

Eine *MapReduce*-Operation ist erst beendet, wenn alle Worker die ihnen zugeteilten Aufgaben vollständig abgearbeitet haben. Im Umkehrschluss bedeutet dies, dass im Extremfall sogar ein einzelner langsamer oder ausgefallener Worker das Ende der gesamten Operation verzögern kann. In der Praxis hat sich gezeigt, dass solche Situationen keine Ausnahmen darstellen, sondern durchaus der Regelfall sind [6].

Die Ursachen für das Auftreten derartiger Ausreißer können sehr vielschichtig sein. Ob durch die Überlastung des Rechners durch andere Benutzer und ihre Anwendungen, fehlerhaften Festplatten oder eine falsche Konfiguration, dies alles verringert die Leistung des Workers. Als Lösung für dieses Problem wurden sogenannte *Backup-Tasks* eingeführt.

Sollte sich die *MapReduce*-Operation dem Ende nähern und nur noch wenige *map*- oder *reduce*-Tasks laufen, so werden diese von einem mittlerweile freien Worker zusätzlich ein zweites Mal gestartet. Sollte nun der Backup-Task vor dem ursprünglichen fertig werden, so wird dessen Ergebnis verwendet, anstatt auf das Ergebnis des eigentlichen Workers zu warten. Laut Google [6] wächst die Ausführungszeit von bestimmten Operationen ohne Verwendung der Backup-Tasks um bis zu 44%.

2.5.5 Performance-Vergleich

Leider gibt Google selbst keinen Vergleich zu Programmen an, die ähnliche Aufgabe wie MapReduce ausführen können, nicht einmal zu den vorher verwendeten Programmen, die bei Google durch MapReduce ersetzt wurden. Dies hat allerdings nicht verhindert, dass MapReduce sich mittlerweile als Möglichkeit zur parallelen Verarbeitung etabliert hat.

Allerdings gibt es auch Stimmen die MapReduce als einen „major step backwards“ [10] bezeichnen. David J. DeWitt veröffentlichte einen Artikel [10], in dem er darauf hinweist,

dass die Verarbeitung von großen Datenmengen durch ein paralleles Datenbank Management System (DBMS) deutlich effektiver sei und auch noch andere Vorteile wie Schemas für Daten oder Indizes besitzt.

In [11] vergleicht Andrew Pavlo in einer Reihe von Benchmarks zwei DBMS mit der Open-Source-MapReduce-Implementierung Hadoop und kommt zu dem Ergebnis, dass MapReduce zwar die Daten schneller lädt, aber die DBMS dafür bei der Ausführung zwei bis sechs mal so schnell sind wie MapReduce. Um auf die selbe Performanz wie ein DBMS mit 100 Rechnern zu kommen, müsste die MapReduce-Operation mit deutlich mehr Worker-Rechnern gestartet werden. Da durch eine höhere Anzahl Worker-Rechner auch die Ausfallwahrscheinlichkeit und damit die Verzögerung durch Ausfälle und deren Behandlung zunimmt, werden bis zu 1000 MapReduce-Worker-Rechner nötig um Leistung des DBMS zu erreichen. Auf der anderen Seite kommt Pavlo in [11] zu dem Schluss, dass es für den Benutzer doch sehr viel aufwendiger ist, ein DBMS aufzusetzen, als zwei kurze Funktionen für das *MapReduce*-Framework zu schreiben.

3. DISCO

Disco wurde vom Nokia Research Center [12] als Open-Source-Implementierung von Googles MapReduce-Konzept entworfen. Zusätzlich enthält Disco neben der MapReduce-Komponente auch noch ein eigenes Speichersystem, genannt *Disco Distributed File System*, sowie das Indexsystem *Discodex*.

Disco bietet neben all den Möglichkeiten von Googles MapReduce-Konzept auch noch ein Webinterface, mit dem der Status der einzelnen Rechner eingesehen werden kann und auch dynamisch neue Rechner zur aktuellen Berechnung hinzugezogen werden können.

Disco selbst wurde dabei in der Sprache Erlang geschrieben, deren Stärke in der Unterstützung für Nebenläufigkeit liegt [13].

3.1 Disco MapReduce

Wie bereits bei Googles MapReduce, werden auch bei Disco die *map*- und *reduce*-Tasks auf viele verschiedene Worker verteilt, die auch hier von einem Master überwacht und koordiniert werden. Im Gegensatz zum Google-Konzept werden bei Disco der Master und die einzelnen Worker nicht vom Benutzerprogramm selbst gestartet, sondern es läuft auf einem Rechner im Cluster ein Prozess, bei dem alle, für MapReduce-Operationen freigegebenen Rechner angegeben sind. An diesen Server werden von den Benutzerprogrammen Aufträge für MapReduce-Operationen gesendet, woraufhin dieser eine bestimmte Anzahl freier Rechner im Cluster auswählt. Einer der freien Rechner wird zum Master, der auf den anderen die verschiedenen Worker startet.

Auch bei Disco werden die beiden Funktionen *map* und *reduce* vom Benutzer geschrieben, allerdings steht es dem Benutzer frei, auf eine der beiden Funktionen zu verzichten. Disco unterstützt zudem die Möglichkeit, die Art der Partitionierung der Ausgabe der *map*-Funktionen festzulegen oder ganz darauf zu verzichten und bietet auch an, bereits partitionierte Eingabedaten der *reduce*-Funktion wieder zusammenzuführen [14], um diese von einem einzigen *reduce*-Task bearbeiten zu lassen.

Diese Möglichkeiten den Datenfluss genauer zu regeln, erlauben eine freiere Verkettung von *map*- und *reduce*-Tasks.

So kann die Ausgabe eines *map*-Tasks fest abgespeichert werden, anstatt sie direkt weiter zu verarbeiten, sodass verschiedene *reduce*-Tasks auf diesen, einmal berechneten Daten arbeiten können.

3.2 Disco Distributed Filesystem

Der allgemeine Aufbau des *Disco Distributed Filesystems (DDFS)* [15] orientiert sich stark am Google File System [9], denn wie dieses besteht das DDFS aus einem den Zugriff koordinierenden Master und einer bestimmten Anzahl an Rechnern zur Speicherung der Daten. Diese Daten werden wie bei Google mehrfach repliziert und auf verschiedenen Rechnern gespeichert, um einem Verlust bei einem eventuellem Ausfall eines Rechners vorzubeugen.

Wie beim Google File System hilft diese verteilte Speicherung der Daten dabei, die Netzwerklast und die Zugriffszeit bei auf diesen Daten arbeitenden MapReduce-Operationen zu verringern, indem die Worker derart eingeplant werden, dass sie hauptsächlich auf ihren lokalen Daten arbeiten, anstatt sich die Daten erst durchs Netzwerk holen zu müssen. Wie auch das Google File System ist das DDFS zur Speicherung und zum Lesen von sehr großen Daten ausgelegt, wie sie in der Regel von MapReduce-Operationen verwendet werden.

Allerdings bietet DDFS auch Neues: Zusätzlich zu den normalen Datensätzen (in DDFS „Blobs“ genannt) werden auch noch sogenannte „Tags“ gespeichert. Diese enthalten Metadaten über andere Tags oder Blobs, wie Zugriffs-URL, oder sogar benutzerdefinierte Metadaten. So wird im Speichersystem ein Graph aus Tags und Blobs aufgebaut, der traversiert werden kann und damit einen selektiven Zugriff auf Daten mit bestimmten Eigenschaften erlaubt.

3.3 Discodex

Ein Problem, dass beim Zugriff auf bestimmte Datensätze innerhalb einer sehr großen Datenmenge immer wieder auftritt, ist das Auffinden des gewünschten Datensatzes in akzeptabler Zeit. Das Disco-Projekt löst dieses Problem durch die Komponente *Discodex*, die aus einem Datenbestand einen Index erstellt und durch das Hashing eines Schlüsselwertes schnell auf die Daten zugreifen kann.

Um dies zu erreichen, wird aus den Eingabedaten durch eine Disco MapReduce-Operation, die allerdings in diesem Fall keine *reduce*-Funktion enthält, eine Menge an Schlüssel-Wert-Paaren erstellt. Diese Paare werden in mehreren *discodb*-Objekten abgelegt, die im Disco Distributed Filesystem gespeichert werden und dieser Speicherort der Discodex-Komponente bekannt gegeben wird. Diese *discodb*-Objekte erlauben den Zugriff auf die gespeicherten Werte durch *minimales perfektes Hashing* [16] wodurch ein Zugriff in konstanter Zeit möglich ist.

4. BOOM

Das an der Berkeley-Universität gestartete BOOM-Projekt hat das Ziel, festzustellen, in wie weit datenzentriertes Programmieren die Möglichkeit bietet, verteilte, stark skalierbare Programme mit vergleichsweise wenig Code zu realisieren und zu erweitern. Die BOOM-Entwickler gehen dabei von einem Ansatz aus, der Benutzern von MapReduce-Implementierungen bereits bekannt ist. Anstatt sich um die Koordination von Prozessen und Rechnern zu kümmern, stehen beim datenzentrischen Programmieren die Daten selbst im Mittelpunkt. Während MapReduce dabei lediglich dem

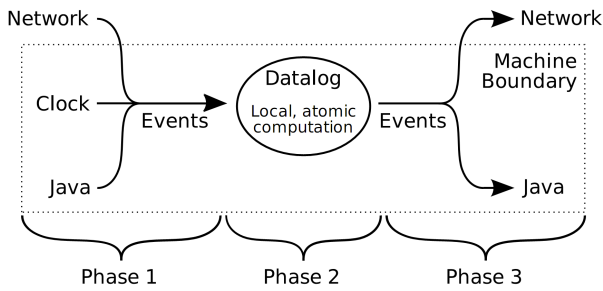


Abbildung 2: Ablauf einer Zustandsmodifikation in Overlog (aus [17])

Benutzer die Möglichkeit gibt in einer datenzentrischen Weise zu programmieren, so versuchen die BOOM-Entwickler festzustellen in wie weit datenzentrisches Programmieren das Entwickeln von komplexen nebenläufigen Programmen vereinfachen kann.

Dabei sehen sie den Ablauf eines Programmes nicht als eine Folge von festgelegten Schritten, sondern als einen Zustand und dessen Modifikation. Oder anders ausgedrückt: „Distributed computing is all about state“ [8] und „Computing = Creating, updating, and communicating that state“ [8]. Dieser Zustand wird in Form von Relationen gespeichert.

Um die Realisierbarkeit dieses Ansatzes zu testen wurde beschlossen, den Kern des Hadoop-Projekts [3] nach Overlog [18] zu portieren, einer Erweiterung der logischen Anfragesprache Datalog um die Möglichkeiten der Modifikation von Daten, der Möglichkeit den Ort der Daten anzugeben, sowie SQL-artige Erweiterungen wie Primärschlüssel und Aggregate. Diese Ortsangabe erlaubt es auch mit Overlog Relationen zu unterstützen, bei denen die einzelnen Tupel auf verschiedenen Maschinen liegen [17]. Trotz dieser Portierung sollte allerdings nach außen die alte Java-Schnittstelle erhalten bleiben.

Damit die Overlog-Logik auf die Teile des Systems, die noch in Java-Code vorliegen, sowie auf das Netzwerk und Zeitgeber Zugriff hat, wurde ein *Event*-System eingeführt. Die Bearbeitung des Zustandes erfolgt dabei immer während eines atomaren lokalen Overlog-Zeitschritts. Diese Zeitschritte stellen Abschnitte dar, in denen jeder Rechner nur seinen eigenen Zustand sieht. Durch diese Zeitschritte wird sichergestellt, dass keine Operation auf den Zuständen unterbrochen werden kann, und damit am Ende des Zeitschritts immer ein konsistenter Zustand vorliegt. Ein einzelner Zeitschritt lässt sich, wie Abbildung 2 zeigt, in drei verschiedene Phasen zerlegen.

Um einen Zustand zu modifizieren, wird ein Ereignis ausgelöst, das in eine Einfüge- oder Löschoption von Tupeln auf einer Relation konvertiert wird. Dies geschieht in der ersten Phase eines Overlog-Zeitschritts. In der zweiten Phase werden diese Tupeloperationen ausgeführt und die festgelegten Overlog-Regeln auf den neuen Daten ausgeführt bis sich keine Daten mehr ändern. In der dritten Phase werden die neuen Daten fest gespeichert und durch die Overlog-Regeln bestimmte Ereignisse als Antwort abgeschickt.

4.1 BOOM-MapReduce

Das Ziel, das die Entwickler des BOOM-Projekts bei der Portierung der MapReduce-Implementierung des Open-Source-Projekts Hadoop [3] nach Overlog hatten, war es zu zeigen,

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: Relationen zum speichern des Zustands des BOOM MapReduce Masters (aus [17])

dass es möglich ist einen komplexeren Teil des Systems relativ einfach zu erweitern, falls eine datenzentrische Programmierung verwendet wird. Dazu haben sich die BOOM-Entwickler entschieden, die MapReduce-Portierung um einen LATE¹-Scheduler [19] zu ergänzen.

Um den Zustand mit Hilfe von Overlog-Regeln modifizieren zu können, musste dieser in Form von Relationen gebracht werden, wie sie in Abbildung 3 zu sehen sind. Einplanungsentscheidungen werden nun getroffen, indem die *taskAttempt*-Relation mit der *taskTracker*-Relation, d. h. die Relation, die die Daten zu den MapReduce-Workern enthält, kombiniert wird, um freie Worker zu finden. Wie [17] zeigt, ist der Patch, der die LATE-Implementierung zum bestehenden MapReduce-Scheduler hinzufügt mit 82 Zeilen bei BOOM deutlich kleiner gegenüber den 2102 Zeilen bei Hadoop.

Wie bereits Disco oder Google-MapReduce profitiert auch BOOM-MapReduce vom darunterliegenden verteiltem Speichersystem.

4.2 BOOM File System

Nach dem Erfolg der Portierung der MapReduce-Komponente, versuchen die BOOM-Entwickler nun auch die Filesystem-Komponente in datenzentrischer Programmierung zu schreiben. Die neue Herausforderung bei dieser Komponente im Vergleich zur MapReduce-Portierung ist die verteilte Speicherung des Zustands. Während bei BOOM-MapReduce der komplette Zustand in Relationen auf dem Master gespeichert werden konnte, so sind beim Dateisystem lediglich die Metadaten auf dem Master gespeichert, während die eigentlichen Daten auf die verschiedenen Speicherknoten verteilt sind.

Unter der Verwendung von Overlog für das Modifizieren der Metadaten und das Initiieren des Datentransfers, sowie Java für das eigentliche Versenden der Daten, wurde die Codelänge im Vergleich zu Hadoop auf unter 10% gesenkt [17].

Wie bereits bei der MapReduce-Portierung soll auch hier bewiesen werden, wie einfach sich datenzentrisch programmierte Komponenten um komplexe verteilte Zusatzfunktionen ergänzen lassen. Um dies zu zeigen, wurde für den Master eine *hot standby*-Funktion eingebaut, die ein Replikat des Masters in einem konsistentem Zustand hält, sodass dieses im Falle eines Ausfalls des Primär-Masters dessen Aufgaben sofort übernehmen kann. Nach Angaben der BOOM-Entwickler war die erste Version dabei lediglich 53 Zeilen lang [17], ein weiterer Beleg, dass die datenzentrische Programmierung dem Anspruch des BOOM-Projekts, um ein vielfaches größere Systeme in einem Bruchteil an Code zu schreiben, gerecht wird [17].

¹Longest Approximate Time to End

4.3 Bloom

Bloom ist eine logische Sprache, die auf Daedalus aufgebaut ist, und es Benutzern einmal ermöglichen soll, mit BOOM Programme für Clouds zu schreiben. Daedalus ist dabei eine „temporal logic language“, d. h. eine Sprache, die auf Grund der Einsicht entworfen wurde, dass es beim verteilten Programmieren nicht um den Ort der Daten geht, sondern um die Veränderung von Daten über die Zeit. Die Entwickler rechnen mit einem Release von Bloom gegen Ende 2010 [4].

5. FAZIT

Auch wenn es teilweise Vorbehalte bezüglich der Effizienz von MapReduce im Vergleich mit anderen Systemen wie parallelen Datenbank-Management-Systemen gibt, so lässt es sich doch nicht leugnen, dass es viele Probleme beim verteilten Verarbeiten von Daten gibt, die mit MapReduce einfach und in zufriedenstellender Zeit gelöst werden können.

An allererster Stelle steht dabei natürlich, wie simpel es durch dieses Framework wird, sich die Rechenleistung einer Cloud zunutze zu machen. Denn letztendlich muss wenig mehr als Code für zwei, je nach Berechnung sogar teilweise sehr kurze, Funktionen geschrieben werden und der Entwickler hat ein komplett parallelisiertes Programm.

Und da es mittlerweile sogar Cloud-Anbieter gibt, die MapReduce anbieten, ohne dass der Benutzer eines der Frameworks installieren muss, wird die Verwendung noch einmal vereinfacht. Ein anderer Aspekt, der MapReduce dabei half, seinen Siegeszug fortzusetzen bestand natürlich in den Open-Source-Implementierungen, mit denen sich jeder diese Funktionalität zu Nutze machen kann.

Beim Betrachten der verschiedenen MapReduce-Implementierungen fällt außerdem auf, wie stark das Konzept von einer verteilten Speicherlösung abhängt. Das liegt vor allem an zwei Problemen, die in einer Cloud oder einem Cluster aus handelsüblicher Hardware auftreten. Zum einen wird in derartigen Clouds oder Clustern oft auch ein handelsübliches Netzwerk verwendet, welches durch das Verteilen von Daten im Terabyte-Bereich sofort überlastet wäre, weshalb es daher nötig ist, die Berechnung möglichst auf lokalen Daten ausführen zu können. Zum anderen sind die MapReduce-Operationen oftmals wenig prozessorlastig, sondern eher einfache Berechnungen. Aus diesem Grund hängt die Leistung eines Workers vor allem von der Lese- und Schreibgeschwindigkeit der Festplatten ab. Durch eine höhere Verteilung der Daten können diese von mehreren Festplatten gleichzeitig gelesen werden, sodass sowohl das Laden der Daten, als auch das Schreiben der Ergebnisse schneller abgeschlossen ist. Zusätzlich muss neben der verteilten Speicherung der Daten auch die Verteilung der Worker-Prozesse betrachtet werden. Nur wenn die Worker-Prozesse direkt auf den Rechnern arbeiten auf denen ihre Eingabedaten liegen oder zumindest sehr nah an diesen, kann eine Überlastung des Netzwerkes vermieden werden. Das Speichersystem muss zusätzlich in der Lage sein, mit den sehr großen Dateien arbeiten zu können, die oft als Ein- und Ausgabe von MapReduce-Operationen dienen. Diese Punkte zeigen auf, dass ein performantes MapReduce-System nur mit einem zugehörigem Speichersystem zu erreichen ist, da andernfalls durch einen oder mehrere der angesprochenen Flaschenhälse die Leistungssteigerung negiert wird, die durch die Ausführung der Berechnung in einem Cluster angestrebt wird.

Allerdings ist MapReduce nicht das Ende der Fahnenstange bei der Programmierung für Clouds und Cluster. Wie bereits das BOOM-Projekt zeigt, werden Versuche unternommen, das Entwickeln von Programmen für verteilte Systeme auf eine höhere Abstraktionsebene zu heben.

Wie einfach es letztendlich in der Zukunft sein wird, Programme zur nebenläufigen Datenverarbeitung zu schreiben, wird sich beim BOOM-Projekt erst noch zeigen, wenn Bloom fertiggestellt ist, doch die Erfahrungen der BOOM-Entwickler haben gezeigt, dass datenzentrisches Programmieren komplexe Problemstellungen mit relativ wenig Code zu lösen vermag. Auch andere Projekte wie Pig [20], das eine Sprache zur Beschreibung von Datenflüssen zur Verfügung stellt, die anschließend in mehrere Hadoop-MapReduce-Operationen übersetzt wird, zeigen, dass es Möglichkeiten gibt, das Erstellen von verteilten Programmen noch weiter zu vereinfachen.

6. LITERATUR

- [1] Google. US Patent Nr 7650331. <http://patft.uspto.gov> (Stand 02.06.2010).
- [2] Disco. Disco Project. <http://discoproject.org> (Stand 02.06.2010).
- [3] Apache. Apache Hadoop Project. <http://hadoop.apache.org> (Stand 03.06.2010).
- [4] BOOM. Berkeley Orders Of Magnitude. <http://boom.cs.berkeley.edu> (Stand 03.06.2010).
- [5] Amazon. Amazon Elastic MapReduce. <http://aws.amazon.com/elasticmapreduce> (Stand 02.06.2010).
- [6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters, 2004.
- [7] Disco. Setting up Disco in Amazon EC2. <http://discoproject.org/doc/start/ec2setup.html> (Stand 03.06.2010).
- [8] Neil Conway. From Doom and Gloom to BOOM and Bloom. http://neilconway.org/talks/boom_bloom_oxford.pdf (Stand 02.06.2010).
- [9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *19th ACM Symposium on Operation Systems Principles*, 2003.
- [10] David J. DeWitt. MapReduce: A major step backwards. <http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards> (Stand 02.06.2010).
- [11] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. Dewitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD '09: Proceedings of the 2009 ACM SIGMOD International Conference*. ACM, June 2009.
- [12] Nokia. Nokia Research Center. <http://research.nokia.com> (Stand 03.06.2010).
- [13] Wikipedia. Erlang (programming language). [http://en.wikipedia.org/wiki/Erlang_\(programming_language\)](http://en.wikipedia.org/wiki/Erlang_(programming_language)) (Stand 03.06.2010).
- [14] Disco. Data Flow in Disco Jobs. <http://discoproject.org/doc/howto/dataflow.html> (Stand 08.06.2010).

- [15] Disco. Disco Distributed Filesystem. <http://discoproject.org/doc/start/ddfs.html> (Stand 02.06.2010).
- [16] Wikipedia. Perfekt Hash Function. http://en.wikipedia.org/wiki/Perfect_hash_function (Stand 08.06.2010).
- [17] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell C Sears. BOOM: Data-Centric Programming in the Datacenter. Technical Report UCB/EECS-2009-113, EECS Department, University of California, Berkeley, August 2009.
- [18] Boon Thau Loo et al. Implementing Declarative Overlays. <http://db.cs.berkeley.edu/papers/sosp05-p2.pdf> (Stand 09.06.2010).
- [19] Anthony D. Joseph Randy Katz Ion Stoica Matei Zaharia, Andi Konwinski. Improving MapReduce Performance in Heterogeneous Environments. http://www.usenix.org/events/osdi08/tech/full_papers/zaharia/zaharia_html/index.html (Stand 09.06.2010).
- [20] Apache. Apache Hadoop Pig. <http://hadoop.apache.org/pig> (Stand 03.06.2010).