Karlsruhe Institute of Technology

MapReduce und Hadoop

Seminararbeit von

Philipp Ruchser

am Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB)
der Fakultät für Wirtschaftswissenschaften des KIT

Erstgutachter:	Prof. Dr. A. Oberweis
Betreuender Mitarbeiter:	Dipl.-Wi.-Ing. D. Karlin

Bearbeitungszeit: 28. April 2014 – 03. Juli 2014

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

MapReduce ist ein Programmiermodell, welches von Google Inc. zunächst intern entwickelt, und schließlich 2004 der Öffentlichkeit zugänglich gemacht wurde [DeGh04]. Zusammen mit den zugehörigen Laufzeitumgebungen handelt es sich dabei um ein Framework für die verteilte, parallele Verarbeitung großer Datenmengen auf Rechnerverbünden. MapReduce ist dabei besonders für große, meist schemalose Datensätze ideal, welche problemlos gesplittet und getrennt verarbeitet werden können. Ein gängiger Anwendungsfall ist dadurch das Auswerten und Verarbeiten großer Log-Dateien, was zum Beispiel beim Erstellen eines inversen Index[1] für Suchmaschinen der Fall ist.

Historisch mangelte es vor der Entwicklung von MapReduce und zu den Anfängen großer datenbasierter Unternehmen wie Google Inc. an leistungsfähigen Lösungen für solche Anwendungsfälle. Klassische Software für die Verwaltung und Verarbeitung von Daten, wie die relationalen Datenbank Management Systeme von Oracle oder IBM, legte einen signifikanten Teil des Fokus auf Verlässlichkeit, was konsequenterweise die Performanz verringerte. Während die dabei garantierten ACID-Eigenschaften (Atomicity, Consistency, Isolation und Durability) für viele Anwendungen von großer Bedeutung waren, spielten sie für die verteilte Verarbeitung von Log-Dateien eine untergeordnete Rolle. Daher erstellten Entwickler innerhalb Googles zunächst individuelle Lösungen für die neuartigen Problemfälle, welche vor allem eine schnelle Verarbeitung großer Datenmengen auf möglichst vielen Knoten eines Computerclusters erforderten [DeGh04]. Die interne Entwicklung von MapReduce diente schließlich der Vereinheitlichung und Vereinfachung der Entwicklung solcher verteilten Anwendungen.

Neben den genannten Anforderungen von der Anwendungsseite standen die Entwickler zudem vor hardwareseitigen Herausforderungen. Google setzt in den Rechenzentren auf *commodity Hardware*, welche in erster Linie für handelsübliche Computer und nicht den dauerhaften, fehlerfreien Betrieb in Servern konzipiert ist. Während diese Komponenten ein gutes Verhältnis von Rechenleistung zu Kosten bieten, fallen durch ihren Einsatz einzelne Knoten eines Clusters regelmäßig aus [GhGL03]. Dies

[1] Ein inverser Index beinhaltet ein Mapping von Suchbegriff auf Speicherort, und wird benötigt um effizient relevante Dokument als Antwort auf eine Suchanfrage zu liefern

stellt besondere Anforderungen an die Ausfalltoleranz von verteilten Anwendungen auf solchen Clustern.

Die Ausführung von MapReduce Anwendungen erfolgt daher durch eine Laufzeitumgebung, welche das Programmiermodell implementiert und dem Nutzer die parallele Ausführung sowie den Umgang mit Hardwareausfällen abnimmt. Während Googles Laufzeitumgebung nicht frei verfügbar ist, entstand mit Apache Hadoop eine populäre und freie Distribution, welche MapReduce mittels Java implementiert und zudem ein Dateisystem für den Betrieb verteilter Rechnerverbünde beinhaltet.

Die Tatsache, dass das Programmiermodell MapReduce im Rahmen oben genannter anwendungsspezifischen Anforderungen und hardwareseitigen Gegebenheiten entwickelt wurde, lässt Schlüsse auf die resultierenden Eigenschaften und die optimalen Anwendungsfelder zu. Zu den wichtigsten Charakteristika des Modells gehören:

- *Einfache Entwicklung.* Der Nutzer gliedert sein Programm prinzipiell in zwei Teile: Map und Reduce. Die Serverumgebung skaliert und parallelisiert nahezu selbständig die Ausführung
- *Parallelisierbarkeit und Skalierbarkeit.* Die Kapselung und Unabhängigkeit der einzelnen Map-Tasks und Reduce-Tasks ermöglicht unkomplizierte Parallelisierung und Skalierung auf einem Cluster variabler Größe
- *Fehlertoleranz.* Der regelmäßige Ausfall einzelner Knoten im Cluster wird erwartet und stellt konzeptionell kein großes Performancehindernis dar

Zu den idealen Einsatzgebieten zählen Problemstellungen, die auf der einen Seite große, oft schemalose Datenmengen als Eingabe benötigen, welche zudem nur einmalig verarbeitet werden müssen (wie eine Log-Datei). Auf der anderen Seite sollten die zugehörigen Berechnungen konzeptionell gut parallelisierbar sein. Insbesondere heißt dies, dass die Datenverarbeitung zunächst in voneinander unabhängige Map-Tasks, und schließlich in voneinander unabhängige Reduce-Tasks zerlegt werden kann. Dies ist notwendig, da die einzelnen Map- beziehungsweise Reduce-Tasks von der Laufzeitumgebung automatisch skaliert und ausgeführt werden. Auch für die Fehlertoleranz ist es notwendig, einzelne Tasks neu starten zu können, sollten diese aufgrund eines Hardwareausfalls wiederholt werden müssen. Die vorausgesetzte Unabhängigkeit der einzelnen Prozesse schließt folglich auch viele klassische Eigenschaften relationaler Datenbanken, wie Transaktionalität zwischen Vorgängen in verschiedenen Tasks, konzeptionell aus. Sofern eine Problemstellung eine solche Zerlegung algorithmisch erlaubt, ist es möglich, sie mittels MapReduce zu lösen.

Diese Anforderungen an die Problemstellungen mögen zunächst einschränkend wirken. In der Praxis hat sich aber gezeigt, dass viele Probleme, inklusiver rechenintensiver Algorithmen aus dem Bereich des maschinellen Lernens, diese Kriterien erfüllen, und mittels des MapReduce Framework gelöst werden können [DeGh08, CKLY+07].

Nachfolgend wird in dieser Seminararbeit zunächst in Kapitel 2 erläutert, wie das Programmiermodell konzeptionell aufgebaut ist. Daraufhin wird in Kapitel 3 die Laufzeitumgebung Apache Hadoop beschrieben. Daran anschließend gibt Kapitel 4 einen kurzen Überblick über die Entwicklung des Programmiermodells sowie der Laufzeitumgebung, und analysiert einige moderne Alternativen. In Kapitel 5 wird ein

realer Datensatz mit MapReduce auf einem Cluster analysiert und die Performanz der Berechnungen evaluiert. Daraufhin evaluiert Kapitel 6 Stärken und Schwächen des Modells und hebt die idealen Einsatzgebiete hervor. Kapitel 7 zieht ein Fazit und gibt einen Ausblick auf anschließende Forschungsfragen.

2. Das Programmiermodell

Das Programmiermodell MapReduce spezifiziert den grundsätzlichen Aufbau von Anwendungen, die dieses Framework nutzen. Frei verfügbare Implementierungen dieses Programmiermodells gibt es unter anderem von der Apache Software Foundation als Bestandteil der Laufzeitumgebung Apache Hadoop (siehe Kapitel 3). Während das weit verbreitete und frei zugängliche Apache Hadoop nativ für Java-Anwendungen ausgelegt ist, gibt es auch die Möglichkeit, andere Programmiersprachen mittels *Hadoop Streaming*[1] zu verwenden.

2.1 Aufbau von MapReduce Anwendungen

Der Nutzer gliedert eine Anwendung grundsätzlich in einen *Map*- und einen *Reduce*-Teil, welche schließlich von der ausführenden Serverumgebung auf mehreren Knoten in einem Rechnerverbund parallel ausgeführt werden. Dabei wird im ersten Schritt die Eingabedatei (zum Beisiel eine Log-Datei) zerlegt, und von mehreren Map-Tasks parallel verarbeitet. Daraufhin werden die Zwischenergebnisse gespeichert, und anhand der Schlüsselmerkmale für die anschließende Reduce-Phase gruppiert. Während dieser werden erneut ein oder mehrere Reduce-Tasks parallel ausgeführt, um die temporären Zwischenergebnisse in das gewünschte Ausgabeformat zu transformieren. Dabei wird konzeptionell stets mit *key-value* Paaren gearbeitet, welche wie folgt als Parameter in den beiden Funktionen *map* und *reduce* verwendet werden [DeGh08]:

```
map(k1, v1) → list <k2, v2>
reduce(k2, list <v2>) → list <v3>
```

Dabei wird innerhalb der Map-Tasks die Eingabe verarbeitet und ein Wechsel in die Domäne, welche von Interesse für die spätere Auswertung ist, vorgenommen. Diese stellt meist den Schlüssel für die Zwischenergebnisse da, und kann zum Beispiel ein Begriff sein, welcher innerhalb eines Dokuments gezählt werden soll (siehe Beispiel in Kapitel 2.2). In Anwendungen, welche analog auch mit einer SQL Datenbankabfrage geschrieben werden könnten, würde der Map-Task somit die Aufgaben von `SELECT` und `WHERE` übernehmen. Anschließend werden die Zwischenergebnisse anhand

[1] http://hadoop.apache.org/docs/r1.2.1/streaming.html

der Schlüssel von der Laufzeitumgebung gruppiert und die temporären Ergebnisse im Netzwerk kommuniziert, sodass die anschließend ausgeführten Reduce-Tasks auf diese Zugriff haben. Dies entspricht Der `GROUP BY`-Klausel von SQL Abfragen. Die Reduce-Tasks führen meist eine eine erneute Reduktion bzw. Aggregation der zugehörigen *values* der Zwischenergebnisse durch. Im Kontext von SQL ist dies Vergleichbar mit der Berechnung von aggregierten Werten, wie `AVG, MIN, MAX` oder `SUM`, aber auch der Reduktion von zuvor selektierten Daten mittels `HAVING`. Die Vergleiche mit SQL dienen hierbei lediglich der Illustration. Die Möglichkeiten der Berechnung mit MapReduce sind durchaus flexibler als mit SQL, weshalb mit MapReduce auch relativ elegant Probleme lösbar sind, welche mit SQL nur sehr schwer berechenbar wären [DeGh10].

Abbildung 2.1 illustriert die Abfolge der einzelnen Phasen eines MapReduce Jobs. Jede der drei Ebenen (Eingabe, (gruppiertes) Zwischenergebnis, Ergebnis) kann dabei verschiedene Datentypen nutzen, wobei die Ausgaben der Map-Funktion vom gleichen Typ wie die erwarteten Eingabe-Parameter der Reduce-Funktion sind. Die Eingabedatei wird von der Laufzeitumgebung gesplittet und von mehreren Map-Tasks parallel bearbeitet. Die Zwischenergebnisse werden daraufhin anhand des temporären Schlüssels gruppiert, und von einem oder mehreren Reduce-Tasks weiterverarbeitet.

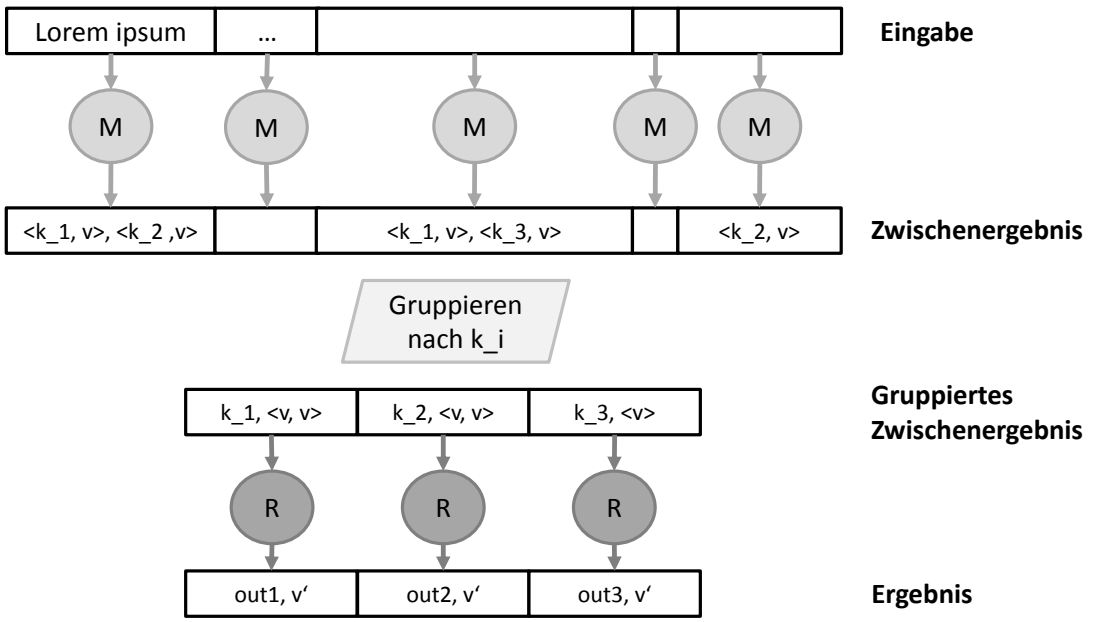

Abbildung 2.1: *Schemahafte Darstellung der einzelnen Ebenen bei der Ausführung eines MapReduce Programms. M: Map-Task, R: Reduce-Task.*

Hervorzuheben ist erneut die Tatsache, dass bei der Entwicklung keine charakteristische Besonderheiten paralleler und verteilter Berechnungen, wie Speicherverwaltung und Ausfallsicherheit, berücksichtigt werden müssen. Die einzige Voraussetzung ist die Unabhängigkeit der einzelnen Map- bzw. Reduce-Tasks untereinander, sodass diese problemlos skaliert und parallelisiert werden können. Alles weitere wird dem Nutzer von der ausführenden Serverumgebung abgenommen, welche in weiterem Detail in Kapitel 3 vorgestellt wird.

2.2 Beispiel: Worthäufigkeit in Dokumenten

In Analogie zu dem Beispiel in [DeGh04] wird in Programm 1 dargestellt, wie mittels MapReduce ein Programm zum Zählen von Worthäufigkeiten in Dokumenten erstellt werden kann. Dieser Anwendungsfall ist ebenfalls für die Erstellung eines inversen Index relevant, da hiermit Statistiken über die Verteilung eines Begriffes über verschiedene Dokumente erstellt werden können. Diese ermöglichen anschließend, die relevantesten Dokumente auf eine Suchanfrage zu liefern.

Der Nutzer implementiert lediglich die beiden Prozeduren `MAP` und `REDUCE`. Die Prozeduren zur Rückgabe der (Zwischen-)Ergebnisse, hier beispielhaft `EmitTemp` und `EmitFinal` genannt, sind Teil der MapReduce Implementierung und werden von der Laufzeitumgebung bereitgestellt. Insbesondere bedeutet dies für den Anwender, dass lediglich die Auswertungs- und Aggregationslogik der Daten implementiert werden muss. Die Gruppierung sowie die Ausgabe der finalen Ergebnisse werden von der Laufzeitumgebung übernommen.

Programm 1 Worthäufigkeit mit MapReduce

```
 1: procedure MAP(String input-key, String input-value)
 2:     // input-key: Name des Dokuments, oder Zeilennummer
 3:     // input-value: Text des Dokuments
 4:     for each word w in input-value do
 5:         EmitTemp(w, 1);
 6:     end for
 7: end procedure
 8:
 9: procedure REDUCE(String temp-key, List<Integer> temp-value)
10:     // temp-key: Begriff, der gezählt werden soll
11:     // temp-value: Liste mit den zu temp-key gehörigen Werten, hier nur Einsen
12:     int result = 0;
13:     for each integer i in temp-value do
14:         result = result + i;
15:     end for
16:     EmitFinal(result);
17: end procedure
```

3. Laufzeitumgebung

Eines der Hauptmerkmale des MapReduce Programmiermodells ist, dass die Anwendungen nahezu autonom von der Laufzeitumgebung ausgeführt werden. Der Nutzer muss sich daher nicht um die Kommunikation der Knoten innerhalb eines Clusters kümmern, und auch keine Mechanismen für den Umgang mit Hardwarefehlern implementieren. Während Google eine eigens entwickelte, nicht publizierte Laufzeitumgebung nutzt, greifen mittlerweile viele andere Unternehmen auf das frei zugängliche Open-Source Projekt Apache Hadoop zu. Zu den Nutzern von Hadoop zählt auch das Unternehmen Yahoo Inc., welches damit Cluster mit über 3500 Knoten betreibt und Datenmengen jenseits von 25 Petabyte verwaltet und verarbeitet [SKRC10]. Im weiteren Verlauf dieses Kapitels wird der Aufbau und die Funktionsweise von Hadoop näher erläutert.

Zu den wichtigsten Bestandteilen der Apache Hadoop Distribution gehören das *Hadoop Distributed File System (HDFS)* und die verteilte Laufzeitumgebung *Hadoop MapReduce*, welche auch Java-Bibliotheken für die Entwicklung von Anwendungen bereitstellt. Für die Durchführung von verteilten Berechnungen auf Rechnerverbünden spielt das Dateisystem eine zentrale Rolle, da es die Verlässlichkeit der Datenspeicherung auch im wahrscheinlichen Fall eines Hardwareausfalls sicherstellen muss [Whit12]. Daher wird zunächst der Aufbau und die Funktionsweise von HDFS vorgestellt, um anschließend die eigentliche Ausführung von MapReduce Anwendungen adäquat beschreiben zu können.

3.1 Hadoop Distributed File System

Bei HDFS handelt es sich um ein Dateisystem für die Verwaltung von Datenmengen, die zu groß für eine einzelne Maschine sind oder aus Performanzgründen auf mehreren Maschinen innerhalb eines Clusters gespeichert und verarbeitet werden sollen. Daher unterstützt HDFS die verteilte Verwaltung von Daten innerhalb eines Clusters und ist spezialisiert auf große Dateien. Dabei beträgt die typische Dateigröße in HDFS 128 Megabytes, was vom Nutzer allerdings konfiguriert werden kann. Diese, sowie die nachfolgenden Beschreibungen von HDFS basieren auf den detaillierten Ausführungen in [SKRC10].

Architektonisch unterscheidet HDFS zwischen zwei Knotentypen im Cluster, die untereinander alle vollständig im Netzwerk verbunden sind. Auf der einen Seite werden auf der *NameNode* die Metadaten zu den Dateien, darunter insbesondere ihre Adressen im Cluster, gespeichert. Davon gibt es typischerweise eine Instanz, die in der Regel für Ausfallsicherheit auf mindestens einem weiteren Knoten repliziert wird. Auf der anderen Seite befinden sich die eigentlichen Dateien auf mehreren sogenannten *DataNodes*. Beide Knotentypen bestehen in der Regel aus handelsüblicher *commodity* Hardware, wobei einzelne Knoten selten signifikant mehr Speicherkapazität als gängige Endnutzer-Geräte besitzen. Auf redundante Speicherung innerhalb eines Knotens mittels eines RAID-Verbundes wird ebenfalls verzichtet. Vielmehr wird eine Replikation der einzelnen Dateien auf mehreren *DataNodes* vorgenommen (typischerweise auf 3 Instanzen). Dieses von klassischer Datenhaltung abweichende Konzept hat den Vorteil, dass der Datendurchsatz im Idealfall linear mit der Anzahl der *DataNodes* skaliert. Zudem besteht durch das hohe Verhältnis von Rechenkapazität zu Speicherkapazität in den *DataNodes* die Möglichkeit, Berechnungen (zum Beispiel die Map-Phase einer MapReduce Anwendung) nahe an den Daten ohne zusätzlichen Kommunikationsaufwand durchzuführen.

Funktionell kommunizieren Anwendungen in HDFS sowohl mit der *NameNode*, als auch mit den *DataNodes* direkt. Dabei wird zunächst eine Anfrage an die *NameNode* gestellt, um den Speicherort einer Datei zu erfragen (Lesen) oder Speicherplatz für eine neue Datei zu beantragen (Schreiben). Der eigentliche Datenaustausch findet daraufhin direkt mit den betroffenen *DataNodes* statt, welche ihrerseits erneut mit der *NameNode* den Status der Transaktionen kommunizieren. Ebenfalls senden die *DataNodes* regelmäßig Meldungen über ihren Systemstatus (sog. *heartbeats*) an die *NameNode*. Sollte ein *DataNode* ausgefallen sein, kann die *NameNode* daher die Replikation der verlorenen Dateien von anderen *DataNodes* initiieren, sodass stets die gewünschte Anzahl an Replikationen im Cluster vorhanden ist. Dadurch wird schon vom Dateisystem ein Mechanismus zum Umgang mit Hardwareausfällen geschaffen, auf den Hadoop MapReduce konzeptionell aufbaut.

3.2 Ausführung mittels Hadoop MapReduce

Der Ablauf einer MapReduce Anwendung innerhalb eines verteilten Dateisystems ist in Abbildung 3.1 dargestellt, welche zusammen mit den nachfolgenden Beschreibungen [DeGh04] und [DeGh08] entnommen wurde. Dabei beginnt die Ausführung damit, dass die Hadoop MapReduce Umgebung anhand der Konfiguration bestimmt, wie viele Map-Tasks (M) und Reduce-Tasks (R) benötigt werden, um die anstehende Datenmenge zu verarbeiten. Daraufhin wird das Programm auf einen Knoten kopiert, der fortan als Master fungiert und den weiteren Ablauf koordiniert. Dieser bestimmt in Absprache mit dem *NameNode* des Dateisystems die Knoten, auf denen die zu verarbeitenden Dateien liegen, und kopiert auf diese die MapReduce Anwendung. Im Idealfall ist es dadurch möglich, die Map-Phase ohne Austausch unverarbeiteter Daten über das Netzwerk zu starten. Sollte dies nicht möglich sein, werden Knoten in der unmittelbaren Umgebung der *DataNode*, welche die Eingabedaten speichert, genutzt. Aufgrund des typischen Aufbaus von Datenzentren gibt es Knoten, welche zusammen in einem sogenannten Server-Rack eingebaut sind, und untereinander schneller kommunizieren können als mit Knoten außerhalb des Racks. Diese Tatsache wird dadurch im Sinne eines höheren Durchsatzes von MapReduce ausgenutzt.

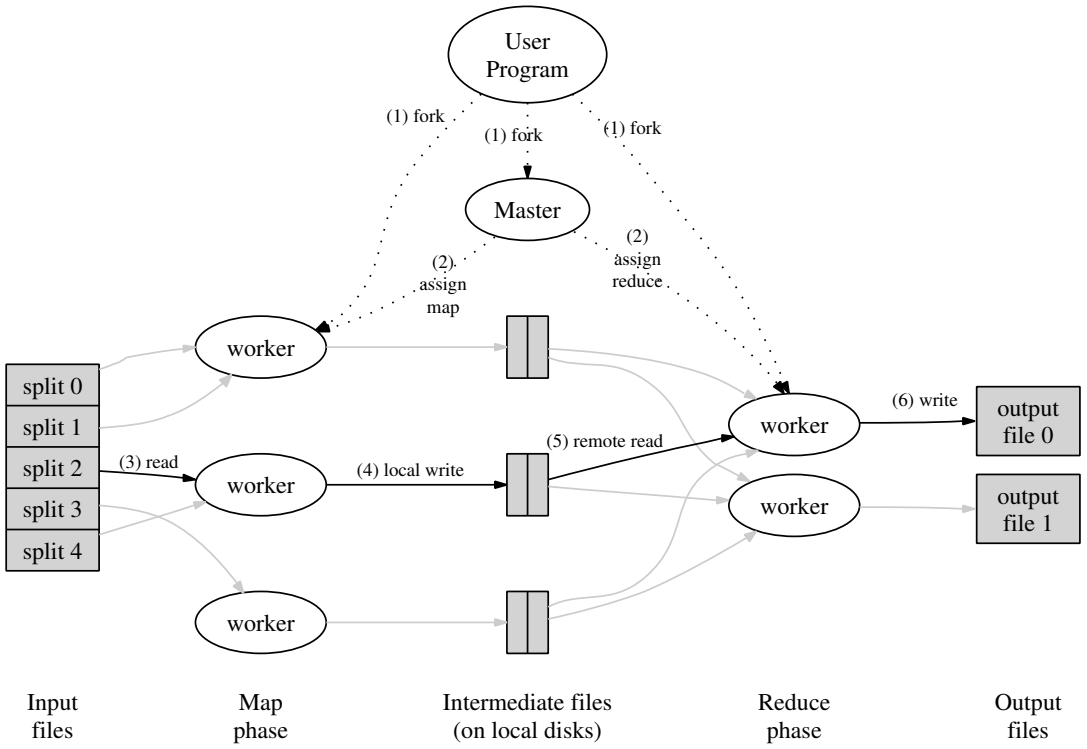

Abbildung 3.1: *Ausführung eines MapReduce Programms in der Laufzeitumgebung. Grafik entnommen von [DeGh04].*

Sobald die Map-Tasks vom Master den einzelnen Arbeitsknoten zugewiesen wurde, führen diese die Umwandlung der Eingabedateien gemäß der spezifizierten Map-Prozedur in die temporären *key-value* Paare durch. Die Zwischenergebnisse werden auf dem lokalen Dateisystem der Arbeitsknoten gespeichert, wobei eine Partitionsfunktion sie anhand der *keys* in R Partitionen unterteilt. Nach Beendigung der Map-Phase teilen die Arbeitsknoten dem Master die Lage und Partitionen der gespeicherten Zwischenergebnisse mit. Dieser leitet diese Informationen an die R Reduce-Knoten weiter, welche jeweils ihre zugehörig Partition über das Netzwerk beziehen und die Reduce-Phase durchführen. Jeder Reduce-Knoten speichert das Ergebnis seiner Berechnungen schließlich in einer Ausgabe-Datei, und teilt dem Master die Lage dieser Datei sowie seinen Status mit. Im Gegensatz zur Map-Phase werden diese Dateien aber nicht nur auf der lokalen Maschine gespeichert, sondern in das HDFS eingespeist. Somit sind diese redundant und ausfallsicher vorhanden. Nach Beendigung aller Berechnungen benachrichtigt der Master das ursprünglich aufrufende Nutzerprogramm über den Status der Berechnungen. In der Praxis werden die heterogenen Ausgabedateien oft noch mittels eines separaten Reduce-Tasks zu einer finalen Ausgabedatei zusammengefasst.

Vergleichbar zu dem *NameNode* in HDFS steht der Master-Knoten bei der Ausführung eines MapReduce Auftrags in ständigem Kontakt mit den Arbeitsknoten, um Ausfälle aufzuspüren. Sollte ein Knoten beim Ausführen eines Map-Tasks ausfallen, wird dieser Task lediglich auf einem anderen Arbeitsknoten neu gestartet. Dies ist auch notwendig wenn der Knoten nach Beendigung der Map-Prozedur, aber vor Ende aller Berechnungen ausfällt, da die temporären Zwischenergebnisse nur lokal und nicht redundant gespeichert werden. Dies ist bei der Reduce-Phase nicht notwen-

dig, da alle Ergebnisse global im HDFS gespeichert werden. Lediglich ein Ausfall während der Reduce-Phase erfordert eine Neuberechnung dieser Partition auf einem anderen Reduce-Knoten.

4. Entwicklung und Trends

4.1 Rückblick auf die Entwicklung

Obwohl MapReduce vor über einer Dekade von Google konzipiert und vorgestellt wurde, ist das Programmiermodell auch heute noch beliebt und von großer Relevanz für die Analyse großer Datenmengen. Den nahezu linearen Wachstum an Suchanfragen zu dem Begriff *MapReduce* kann man einer Google Trend Analyse entnehmen, welche in Abbildung 4.1 dargestellt ist. Während MapReduce zum ersten Mal im März 2005 in der Trend-Analyse mit positivem Gewicht erscheint, dauerte es drei weitere Jahre bis dies für *Apache Hadoop* der Fall ist. Letzteres ging aus dem zunächst von Doug Cutting und Mike Cafarella unter dem Namen *Nutch* vorgestellten Projekt hervor, welches ab 2006 von ersterem in einer Projektgruppe bei Yahoo weiter entwickelt wurde. Die erste Version von Hadoop als Teil der Apache Software Foundation wurde daraufhin im Spätjahr 2007 veröffentlicht. Ab 2008 wurde Hadoop zudem ein Top-Level Projekt bei Apache, was auf die hohe Beliebtheit und die aktive Entwicklergemeinde zurückzuführen ist [Whit12, Foun14]. Seit der ersten Veröffentlichung unter dem Namen *Apache Hadoop* stiegen die Suchanfragen zu diesem Begriff ebenfalls nahezu linear ohne merkbaren Einbrüche an.

Neben den konzeptionellen Vorteilen des Programmiermodells MapReduce und der kostenlosen Verfügbarkeit der Open-Source Distribution Apache Hadoop haben zwei weitere Entwicklungen zur stetig wachsenden Beliebtheit des Frameworks beigetragen. Zum einen entwickelte sich mit dem Unternehmen *Cloudera*[1] ein Anbieter einer kommerziellen Apache Hadoop Distribution mit professionellem Support und zusätzlichem Cluster-Manager. Dadurch wurde das Framework für eine große Zahl weiterer Unternehmen interessant, da die Administration und Verwaltung vereinfacht wurden. Heute ermöglicht das Unternehmen, das unter anderem vom Erfinder von Apache Hadoop, Doug Cutting, geleitet wird, einer großen Zahl von Unternehmen (darunter *ebay*, *Groupon* und *AMD*) den einfachen Zugang zu Apache Hadoop und Berechnungen mittels MapReduce[2]. Auf der anderen Seite entstand ein Angebot an cloud-basierten Distributionen von Hadoop, mittels welcher jederzeit ein

[1] http://www.cloudera.com/

[2] http://www.cloudera.com/content/cloudera/en/our-customers.html

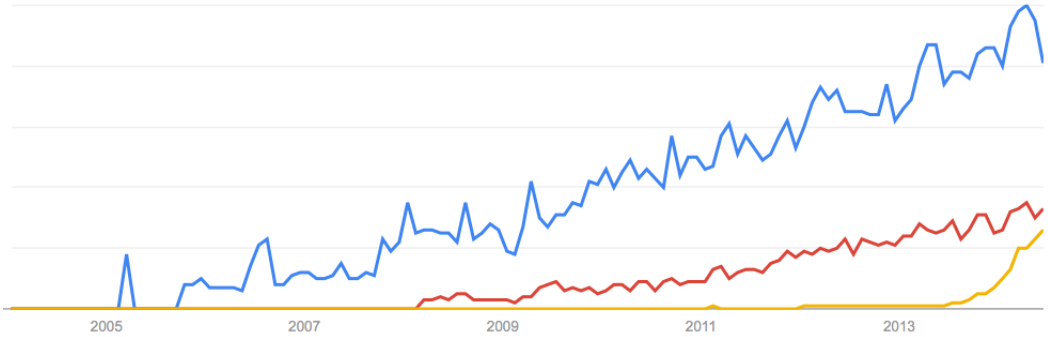

Abbildung 4.1: *Google Trend Analyse zu den Suchbegriffen MapReduce (blau), Apache Hadoop (rot) und Apache Spark (gelb). Der Wertebereich der vertikalen Achse wurde auf 100 normiert und die restlichen Werte wurden proportional an diesen angepasst. Zeitraum der Analyse: Januar 2004 bis Juni 2014. Quelle: google.de/trends*

Hadoop-Cluster nahezu beliebiger Größe bei Bedarf gemietet werden kann. Somit ist es möglich, große Berechnungen mit geringem Zeitaufwand und ohne den Besitz leistungsfähiger Hardware durchzuführen. Ein großer Anbieter ist hierbei Amazon mit dem Dienst *Elastic MapReduce (EMR)*[3], welcher auch im Anwendungsbeispiel in Kapitel 5 genutzt wird. Der Nutzer kann dabei die Größe und Leistungsfähigkeit seines Clusters definieren und MapReduce Anwendungen auf der vorkonfigurierten Hadoop Umgebung starten. Grundsätzlich kosten bei Amazon N Stunden einer einzelnen Recheneinheit soviel wie N solcher Recheneinheiten für eine Stunde. Die Möglichkeiten, die dieses Preismodell in Kombination mit MapReduce mit sich bringt, erlangten erstmals 2007 durch einen Blog-Eintrag der New York Times großes öffentliches Interesse [Gott07]. Durch die Nutzung eines Clusters von Amazon mit 100 Knoten war es der New York Times möglich, über 11 Millionen gescannte Zeitungsartikel in weniger als 24 Stunden zu digitalisieren.

Wie zuvor erläutert lässt Grafik 4.1 durch ein stetig steigendes Interesse an Suchergebnissen zu MapReduce auf eine nach wie vor große Relevanz und Verbreitung schließen. Dies ist zunächst mit hoher Wahrscheinlichkeit auch korrekt, da viele Unternehmen große Hadoop-Cluster betreiben und diese Infrastruktur nicht spontan umstellen werden [Metz13]. Allerdings gibt es mittlerweile modernere Konzepte, welche das Potential besitzen, MapReduce zukünftig als Programmiermodell zu ersetzen bzw. dies schon getan haben. Darauf hat Apache in der Version 2.x von Hadoop reagiert und die Architektur der Distribution grundlegend geändert. In früheren Versionen waren das Dateisystem HDFS und die Ausführung von Programmen via Hadoop MapReduce (siehe auch Kapitel 3) architektonisch aneinander gekoppelt. In der zweiten Version von Apache Hadoop wurde hierbei allerdings eine konzeptionelle Trennung durchgeführt, und mit YARN (Yet Another Resource Negotiator) ein generischer Ressource-Manager eingeführt [Foun13]. Dadurch wurde das Hadoop Framework offener für andere Programmiermodelle, die neben Hadoop MapReduce auf dem Dateisystem HDFS, und damit auf bestehenden Infrastrukturen, aufsetzen können.

[3] http://aws.amazon.com/de/elasticmapreduce/

4.2 Moderne Alternativen

Ein solches Programmiermodell, das ebenfalls von der Apache Software Foundation verwaltet wird, ist Apache Spark. Spark kann mittels YARN auf existierende Hadoop-Cluster zugreifen, alternativ aber auch eigenständig davon oder in Kombination mit alternativen Cluster-Umgebungen wie Apache Mesos[4] betrieben werden. Dabei ist Spark besonders geeignet für wiederholte Iterationen auf einem Datensatz, was insbesondere im Bereich des maschinellen Lernens für eine Verkürzung der Berechnungsdauer um das zehnfache im Vergleich zu MapReduce führt. Durch die zusätzliche Unterstützung von *in-memory Computing* kann mittels Spark diese Berechnung bis auf ein hundertstel der ursprünglichen Berechnungsdauer mit MapReduce gesenkt werden [ZCFS+10]. Gemäß der Analyse mit Google Trend in Grafik 4.1 tauchte *Apache Spark* (eingezeichnet in gelb) als Suchbegriff erstmals im Februar 2011 auf, wobei die Nachfrage nach dieser Lösung seit Juli 2013 nahezu exponentiell angestiegen ist.

Zudem hat Google mitgeteilt, selbst intern nicht mehr auf MapReduce zu setzen, da mittlerweile effizientere Alternativen entwickelt und in Betrieb genommen wurden. Dabei handelt es sich insbesondere um *Flume* und *MillWheel*, welche zusammengefasst in dem Dienst *Google Cloud Dataflow* der breiten Öffentlichkeit zugänglich gemacht werden. Hierbei wurden zwei Schwachstellen adressiert, welche das MapReduce Framework hinsichtlich der Verarbeitung von Daten besitzt. Zum einen bietet Flume die Möglichkeit, den Lebenszyklus von Daten bei der Analyse zu vereinfachen. Während mit MapReduce manche Berechnungen mehrere, aufeinander folgende Map- und Reduce-Phasen erfordern, können diese Probleme mittels *Flume* konzeptionell deutlich leichter gelöst werden. Zum anderen können dank *MillWheel* Daten nicht nur im *Batch*-Modus (d.h. vorliegend im Dateisystem und als geschlossene Einheit), sondern direkt in Echtzeit als *Stream* verarbeitet werden [Metz14]. Darüber hinaus hat Google seit der Publikation von MapReduce 2004 in [DeGh04] noch zwei weitere Ansätze vorgestellt, welche in der Lage sind, gewisse Problemstellungen effizienter zu lösen als MapReduce. Dazu gehört *Percolator*, welches im Gegensatz zu MapReduce nicht eine Verarbeitung eines gesamten Datensatzes (im *Batch*-Modus) erfordert, sondern auch nach inkrementellen Änderung eines Datensatzes effizient die Analysen updaten kann, ohne den gesamten Datensatz erneut lesen zu müssen [PeDa10]. Zum anderen wurde *Pregel* entwickelt, um effizient verteilte Berechnungen in Graphen durchzuführen. Google benötigt dies für die Bestimmung des PageRanks, welcher für die Relevanz von Suchergebnissen maßgebend ist. Gemäß den Entwicklern ist *Pregel* dabei konzeptionell ähnlich zu MapReduce, jedoch spezialisiert auf die Besonderheiten von Graphen und die dabei notwendigen wiederholten Iterationen [MABD+10].

Neben diesen ausschließlich bei Google entwickelten Konzepten und Lösungen gibt es noch weitere Ansätze anderer Unternehmen. Eine umfangreichere Auflistung dieser, sowie eine detailliertere Ausführung der oben erwähnten Konzepte liegt dabei allerdings außerhalb des Rahmens dieser Arbeit. Zusammenfassend ist aber hervorzuheben, dass MapReduce zwar auf der einen Seite über zehn Jahre nach der Entwicklung durchaus leistungsfähigere Konkurrenz bekommen hat. Dazu gehören optimierte Lösungen für verschiedene Anwendungsfälle, die gewisse Problemstellungen effizienter lösen können als MapReduce und Hadoop. Auf der anderen Seite

[4] http://mesos.apache.org/

erfreut sich MapReduce als Programmiermodell aufgrund seiner Flexibilität und der weiten Verbreitung von Apache Hadoop weiterhin großer Beliebtheit. Apache Hadoop wurde zudem konzeptionell mittels YARN von MapReduce abgegrenzt und besitzt damit auch zukünftig Relevanz als Betriebssystem von Clustern, welche Berechnungen mit anderen Programmiermodellen (z.B. Apache Spark) durchführen. Es ist zudem, im Gegensatz zu den oben vorgestellten Konzepten von Google, als Open-Source Software frei zugänglich und kann dadurch auf privaten Clustern betrieben werden.

5. Anwendungsbeispiel

Für die Illustration der Funktionsweise von MapReduce in der Praxis wurde im Rahmen dieser Arbeit eine Analyse auf einem realen Datensatz durchgeführt. Dabei wurden umfangreiche Informationen zu Taxi-Fahrten aus New York City im Jahr 2013 analysiert und ausgewertet. Das Ziel war es dabei, durch ein MapReduce Programm heraus zu finden, welche Bezirke in New York City den meisten Umsatz im Taxigeschäft generieren. Das implementierte Programm wurde anschließend in Amazons Cloud-basierter Variante von Apache Hadoop auf einem Cluster variierender Größe ausgeführt. Zudem wurde dabei die Performanz in Abhängigkeit der Clustergröße evaluiert und mit einer klassischen Java-Anwendung ohne Verwendung des MapReduce Frameworks verglichen.

5.1 Datensatz und Problemstellung

Die verwendeten Daten zu dem Taxibetrieb in New York City wurden im Rahmen des *Freedom of Information Law* von der *NYC Taxi & Limousine Commision (TLC)* veröffentlicht und sind frei zugänglich[1]. Der verfügbare Datensatz von 2013 besteht aus 24 Dateien mit einer Gesamtgröße von über 50 Gigabyte. Für jeden Monat liegen zwei Dateien im CSV-Format vor. Eine davon enthält Trip-Informationen zu den Fahrten, wie Dauer, Fahrtstrecke und Anfangs- und Endkoordinaten. Die zweite Datei enthält für die einzelnen Fahrten Zahlungsinformationen, darunter insbesondere den fälligen Taxibetrag. Eine Fahrt wird dabei in beiden Dateien über die Kombination aus Medallion-Nummer (der fahrzeugspezifischen Taxi-Lizenz), der Lizenznummer des Fahrers und der Startzeit der Fahrt spezifiziert. Da beide Dateien diese drei Spalten enthalten, können die Informationen zusammengeführt werden. Dafür wurde auf ein bereits existierendes Skript[2] zurück gegriffen, welches die beiden Dateien nicht nur zusammen führt, sondern zudem ein Mapping von den Geokoordinaten der Abfahrt und Ankunft auf New Yorker Stadtteile vornimmt. Damit ist es unter anderem möglich, die Taxifahrten abhängig von Abfahrtsort zu analysieren, was für eine Optimierung der Taxirouten für Fahrer von Interesse sein könnte. Da bei der Kombination der Daten zudem die Redundanz doppelt vorhandener Attribute entfernt

[1] http://chriswhong.com/open-data/foil_nyc_taxi/
[2] https://github.com/tswanson/TaxiNYC2013

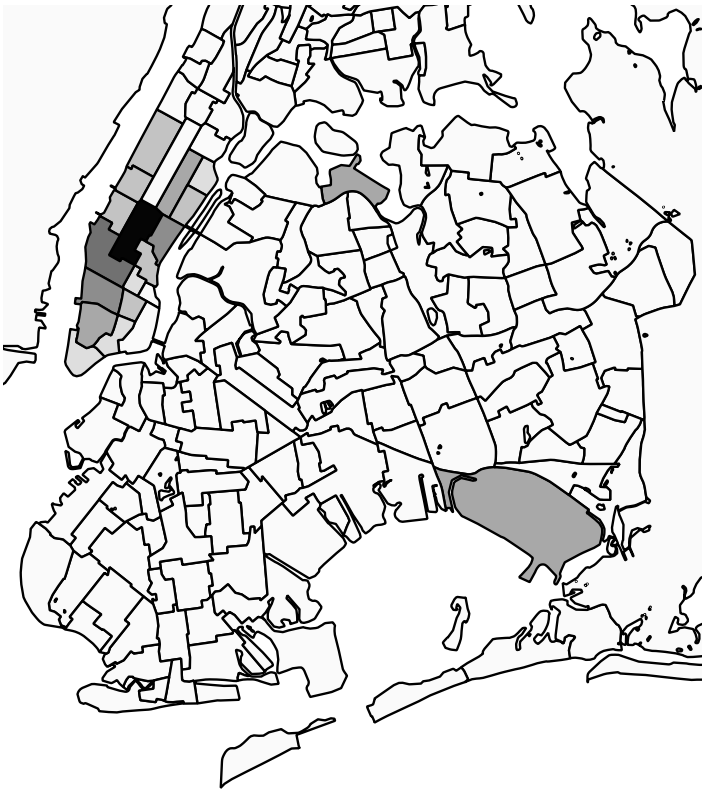

Abbildung 5.1: *Verteilung des im Januar 2013 generierten Umsatzes der New Yorker Taxi & Limousine Commision mit Fokus auf Manhattan, Brooklyn und Queens. Der Umsatz ist dabei den Stadtteilen zugeordnet, in denen einen Fahrt begonnen wurde. Dunklere Stadtteile symbolisieren in der Darstellung einen höheren Umsatz.*

wurde und speicherintensive Attribute auf eine kompaktere ID gemappt wurden, bestand der resultierende Datensatz aus etwa 1,3 Gigabyte pro Monat. Für die weitere Analyse wurde diese kombinierte Datei für Januar 2013 genutzt, da diese zum einen groß genug ist um die Analysen realistisch zu gestalten, aber auch mehrmalige Berechnungen und den Upload zu Cloud-basierten Diensten mit angemessenem Zeitaufwand erlaubt. Insgesamt fanden in diesem Monat über 14 Millionen Fahrten mit Taxis der TLC statt. Mittels MapReduce soll herausgefunden werden, welche der New Yorker Stadtteile dabei den größten Umsatz für das Taxiunternehmen generieren. Dabei wurde eine Fahrt anhand des Abfahrtsorts kategorisiert, wobei das Ziel der Fahrt nicht beachtet wurde. Dies lässt für das Taxiunternehmen Schlüsse auf Gebiete mit besonders hoher finanzieller Nachfrage zu, in welchen leere Taxis bevorzugt nach potentiellen Fahrgästen patrouillieren sollten.

Die mittels MapReduce berechneten Umsatzzahlen wurde mit der Software QGis[3] auf eine Karte mit den New Yorker Bezirken gemappt (Abbildung 5.1). Dunkle Gebiete repräsentieren dabei Zonen in denen durch beginnende Taxifahrten ein hoher Umsatz generiert wurde. Die beiden dunkleren Zonen außerhalb Manhattans sind die Flughäfen LaGuardia im Norden und John F. Kennedy im Süden. Der Gesamtumsatz im Januar 2013 belief sich dabei auf über $166 Millionen, wovon nahezu die Hälfte auf die fünf umsatzstärksten Zonen Manhattans und den John F. Kennedy Flughafen zurückzuführen ist.

[3] http://www.qgis.org/de/site/

5.2 Entwicklung der Anwendung

Konzeptionell wird in einem MapReduce Programm für diese Problemstellung während der Map-Phase die Eingabedatei eingelesen und als Zwischenergebnis für jede Fahrt der Stadtteil des Abfahrtorts als *key*, sowie der in der Fahrt generierte Umsatz als *value* zurück gegeben. Daraufhin wird von der Laufzeitumgebung eine Gruppierung der Zwischenergebnisse anhand des Abfahrtsortes vorgenommen, wobei die zugehörigen Umsatzzahlen in jeweils einer Liste zusammen gefasst werden. Dadurch können die über 14 Millionen Fahrten auf 192 verschiedene Paare aus Abfahrtsort und Umsatzliste reduziert werden. Diese werden daraufhin in der Reduce-Phase weiter verarbeitet, wobei die Umsatzdaten für jeden Abfahrtsort aggregiert werden, und schließlich als Ergebnis im Kontext des Abfahrtsortes zurück gegeben werden.

Technisch erfolgte die Umsetzung mit Java und unter Einbezug der Apache Hadoop Bibliotheken. Dabei sind drei Klassen notwendig, um ein MapReduce Programm zu spezifizieren. Der Quellcode zu diesen drei Klassen befindet sich in Anhang A.1. Die Mapper-Klasse (NYCTaxiMapper) erbt von `Mapper`[4] und verarbeitet die Eingabe, die von der Laufzeitumgebung weitergeben wird. Ergebnisse werden sowohl in der Map- als auch in der Reduce-Phase über ein `Context`-Objekt ausgegeben, welches direkt mit der Laufzeitumgebung kommuniziert. Die Reduce-Klasse (NYCTaxiReducer) erbt von `Reducer`[4] und bekommt eine iterierbare Liste mit dem zugehörigen *key* von der Laufzeitumgebung übergeben. Schließlich existiert eine weitere Klasse mit zugehöriger `main`-Methode (NYCTaxi), welche den MapReduce-Ablauf über ein Objekt der Klasse `Job`[4] spezifiziert und anschließend startet. Alle Klassen greifen dabei auf Datentypen der Laufzeitumgebung zurück, welche in dem Paket `org.apache.hadoop.io` zu finden sind. Es ist auch möglich, eigens erstellte komplexe Datentypen zu erstellen und nutzen. Diese müssen das Interface `Writable` aus diesem Paket implementieren, welches gewisse Methoden zum Handling der Daten beim Lesen und Schreiben erfordert.

5.3 Ausführung mittels Apache Hadoop

Wie in Kapitel 3 erläutert, erfordert die Ausführung von MapReduce Programmen eine Laufzeitumgebung. Für dieses Anwendungsbeispiel wurde zunächst eine lokale Installation von Apache Hadoop für das Debuggen während der Entwicklung verwendet. Diese lokale Installation erlaubt das Ausführen von Anwendungen auf einem Einzelcomputer in einem pseudo-verteilten Modus. Hierbei werden je ein Prozess für eine *NameNode* und eine *DataNode* auf dem lokalen Rechner gestartet, und HDFS sowie MapReduce darauf ausgeführt. Somit kann lokal eine Anwendung entwickelt und getestet werden.

Für die Evaluation der Performanz von MapReduce ist allerdings die Ausführung auf einem Cluster wünschenswert. Dafür wurde auf das *Elastic MapReduce (EMR)* Angebot von Amazon zurück gegriffen, welches entgeltlich das Konfigurieren und Betreiben einer maßgeschneiderten Laufzeitumgebung zulässt. Dazu werden die Daten und das kompilierte Programm als **.jar*-Verzeichnis zunächst in einem *Bucket* von Amazons Speicherdienst S3 abgelegt, welcher auch für die Speicherung der Logs und der Ergebnisse zur Laufzeit verwendet wird. Daraufhin kann ein Cluster nach Bedarf

[4] enthalten in dem Paket `org.apache.hadoop.mapreduce`

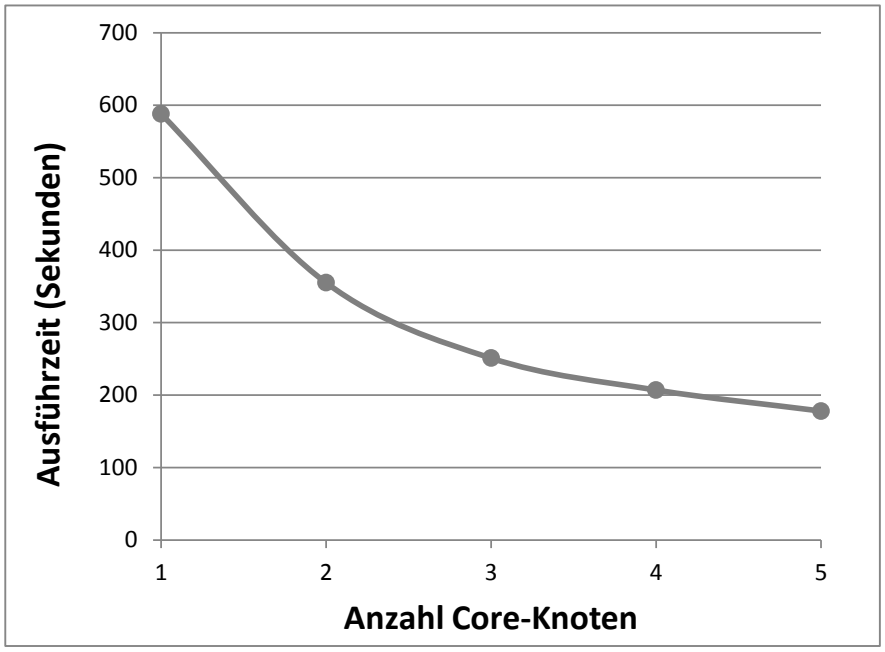

Abbildung 5.2: *Laufzeit der MapReduce Anwendung NYCTaxi in Abhängigkeit der Clustergröße unter Verwendung von Amazon Elastic MapReduce.*

zusammengestellt werden, wobei der Nutzer auf beliebig viele virtuelle Instanzen zurück greifen kann. Deren Preis variiert mit der Leistungsfähigkeit und Ausstattung. Für die im Rahmen dieser Arbeit durchgeführten Berechnungen wurden Instanzen des Typs *m1.medium* genutzt, welche eine virtuelle CPU und 3,75 GB Arbeitsspeicher besitzen[5]. In Kombination mit der Laufzeitumgebung für MapReduce kosten diese Instanzen jeweils $0,109 pro Stunde und rangieren damit am unteren Ende der Kosten- und Leistungsfähigkeitsskala. Ist ein Cluster einmal gestartet, können zur Laufzeit weitere Knoten hinzugefügt und Anwendungen neu gestartet werden. Auf diese Art wurde für die Evaluation der Performanz zunächst ein Cluster mit einem *Master*-Knoten (für die Verwaltung der Metadaten und der Koordination des Ablaufs) und einem *Core*-Knoten (für die eigentlichen Berechnungen) gestartet und die Analyse der Taxidaten darauf durchgeführt. Daraufhin wurden dem Cluster sukzessive weitere *Core*-Knoten hinzugefügt und die Berechnungen erneut durchgeführt.

5.4 Performanz

Abbildung 5.2 zeigt den aus den Log-Dateien bestimmten Verlauf der Ausführungszeit in Abhängigkeit von der konfigurierten Clustergröße. Während zunächst das Hinzufügen eines zweiten und dritten *Core*-Knotens noch eine deutliche Reduktion der Laufzeit bewirkt, nimmt dieser Effekt mit steigender Anzahl an Knoten ab. Eine mögliche Erklärung ist eine gewisse Menge an Berechnungsaufwand, die unabhängig von der eigentlichen Ausführung eines MapReduce Programmes ist. Dazu gehört das Aufteilen der Eingabedatei und eventuell der Transfer dieser Teile zu den *Core*-Knoten. Zudem muss das Zwischenergebnis nach der Map-Phase in das lokale Dateisystem geschrieben werden, und dem Reduce-Task zugänglich gemacht

[5] http://aws.amazon.com/de/ec2/

werden. Bei der verwendeten Implementierung betrug der Umfang des Zwischenergebnis mit nahezu 130 Megabyte noch circa 10% der Eingabedatei. Bei einer größeren Anzahl von Knoten im Cluster entsteht ein höherer Kommunikationsaufwand über das Netzwerk, der bei relativ kurzen absoluten Rechendauern (wie im vorliegenden Anwendungsfall) der verkürzten Berechnungszeit der eigentlichen MapReduce Anwendung entgegen wirkt. Dies erklärt, warum die Berechnungsdauer mit steigender Anzahl an Knoten zu konvergieren scheint. Während eine Analyse mit deutlich größeren Datenmengen außerhalb des Rahmens dieser Arbeit liegen, sollten dabei aber die oben erwähnten Faktoren relativ betrachtet weniger ins Gewicht fallen, und die Laufzeit besser mit der Anzahl der Knoten skalieren. Diese Hypothese wird von der Ergebnissen in [RRPB+07] unterstützt. Hier konnten die Autoren durch diverse Tests zeigen, dass eine erhöhte Anzahl Prozessoren bei großen Datensätzen und komplexen Berechnungen signifikantes Einsparpotenzial mit sich bringt. Auf der anderen Seite werden einfache Berechnungen auf kleinen Datensätzen bei einer einer erhöhten Anzahl an Prozessoren von dem zusätzlichen Overhead des MapReduce Frameworks (u.a. Scheduling und Kommunikation) ausgebremst. Dies kann teilweise auch die Laufzeit im Vergleich zu kleineren Rechnerverbünden verschlechtern.

Zusätzlich zu der MapReduce Anwendung wurde eine klassische Java Anwendung angefertigt, die funktional die gleiche Analyse durchführt und ein identisches Ergebnis ausgibt. Hierbei wird eine `ArrayList` verwendet, welche eigens erstellte Datentypen verwaltet (der Quellcode befindet sich im Anhang A.2). Diese Struktur erlaubt eine prozedurale Berechnung ohne die Notwendigkeit, Zwischenergebnisse in das Dateisystem zu schreiben. Diese Anwednung wurde als Referenz auf einer einzelnen *m1.medium* Instanz in Amazons Elastic Compute Cloud (EC2) ausgeführt, um vergleichbare hardwareseitige Bedingungen zu gewährleisten. Dabei dauerte die komplette Berechnung 196 Sekunden, und liegt damit zwischen dem Zeitaufwand mit Elastic MapReduce unter Verwendung von 4 bzw. 5 solcher Instanzen in einem Cluster. Dies verdeutlicht nochmals, dass das MapReduce Framework einen signifikanten Overhead mit sich bringt, welcher unter anderem auf das Schreiben der Zwischenergebnisse und den Datenaustausch zwischen den einzelnen Knoten fällt. Allerdings profitiert der Nutzer durch den Einsatz von MapReduce von der einfachen Skalierbarkeit der Programme und der Fehlertoleranz. Sollte ein Datensatz aus Größen- und Zeitgründen den Betrieb auf mehreren Hundert Instanzen erfordern, bringt eine konventionelle Anwendung einen deutlichen höheren Entwicklungsaufwand mit sich, um besagte Vorteile des MapReduce Frameworks zu replizieren.

Zusammengefasst ist die Performanz des MapReduce Framework bei dem analysierten, relativ kleinen Datensatz einer konventionellen Anwendung signifikant unterlegen. Dies liegt an dem zusätzlichen Zeitaufwand, der bei einer MapReduce Anwendung unter anderem auf die Koordination der Knoten sowie das Schreiben und die Kommunikation der Zwischenergebnisse fällt. Bei komplexeren Berechnungen und größeren Datensätzen kommen allerdings die Vorteile von MapReduce stärker zum Tragen, da hier der oben genannte Overhead relativ zur Gesamtlaufzeit weniger ins Gewicht fällt. Dadurch skaliert die Laufzeit besser (im Idealfall nahezu linear) mit der Größe des Clusters (siehe dazu auch [RRPB+07]). Eine detaillierte Analyse der Performanz in dieser Größenordnung liegt allerdings außerhalb des Rahmens dieser Arbeit.

6. Evaluation

Aufbauend auf die vorangehenden Ausführungen wird in diesem Kapitel das Programmiermodell MapReduce kritisch evaluiert. Dabei wird zunächst das Konzept unter verschiedenen Gesichtspunkten mit relationalen Datenbanken verglichen. Anschließend werden die daraus gezogenen Erkenntnisse mit den Schlussfolgerungen aus den Kapiteln 4 und 5 kombiniert und dadurch die idealen Einsatzgebiete von MapReduce heraus gearbeitet.

6.1 Vergleich zu verteilten Datenbanksystemen

Nachdem MapReduce von Google Inc. veröffentlicht wurde und allgemein an Popularität gewann, wurde es zunehmend als Ersatz zu parallelen Datenbankmanagementsystemen zu nutzen. Obwohl solche Datenbanksysteme und MapReduce auf den ersten Blick verschiedene Anwendungsfälle bedienen und MapReduce historisch explizit für andere Anwendungsfälle als die Datenbanksysteme entwickelt wurde, lassen sich dennoch theoretisch fast alle Probleme mit beiden Systemen lösen (eventuell unter Verwendung von eigens definierten Funktionen in SQL) [PPRA+09]. Zudem nahm mit Facebook zumindest ein großes Unternehmen ein Datawarehouse in Betrieb, welches technologisch komplett auf MapReduce und Hadoop setzte [Mona09]. Auf der anderen Seite konnten parallele Datenbanken zunehmend auf größeren Clustern betrieben werden, wodurch sie sich ebenfalls verstärkt für die Auswertung großer, verteilt gespeicherter Datenmengen eigneten [PPRA+09] und damit eine prädestinierte Domäne von MapReduce betraten.

Federführend in der diesbezüglich geäußerten Kritik an MapReduce und der Open-Source Laufzeitumgebung Apache Hadoop ist ein Beitrag in einem der Spalten-Datenbank Vertica zugehörigen Magazin [DeSt08]. Die Autoren, darunter auch der Mitgründer von Vertica, Michael Stonebraker, bemängeln vor allem, dass das Programmiermodell MapReduce ein „gigantischer Schritt rückwärts“ in der Verarbeitung von großen Datenmengen sei. Konkreter wird diese Kritik in einer späteren, umfangreicheren Publikation, welche unter anderem auch von Michael Stonebraker verfasst wurde [PPRA+09]. Hierbei werden verschiedene Anwendungsfälle auf drei Systemen verarbeitet und deren Performanz verglichen. Bei den Systemen handelt

es sich um MapReduce mit Apache Hadoop als Laufzeitumgebung, die spaltenorientierte parallele Datenbank Vertica, sowie eine nicht identifizierte, kommerzielle zeilenorientierte Datenbank[1]. Alle Systeme wurden verteilt auf einem Cluster mit 100 Knoten betrieben. Auf die Kritik, welche mittels dieses Artikels publiziert wurde, gab es daraufhin eine Antwort via eines Artikels von den Erfindern von MapReduce, welcher versucht einige der Punkte zu relativieren [DeGh10]. Basierend auf diesen beiden Artikeln lässt sich die wichtigste negative Kritik an MapReduce in drei Punkten zusammen fassen.

- *Mangelnde Performanz wiederholter Berechnungen.* Sofern Daten einmal eingelesen sind, führen die parallelen Datenbanken Berechnungen wie Aggregationen und Selektionen um ein vielfaches schneller durch als die Installation von Hadoop auf einem Cluster gleicher Größe (bei den beiden genannten Anwendungsfällen war Vertica in den Testfällen von [PPRA+09] bis zu 12 mal schneller als Hadoop). Dies liegt daran, dass die Datenbanken die Daten zuvor eingelesen haben und mit zugehörigem Schema in einem separaten Dateisystem gespeichert haben. Vor allem eine spaltenorientierte Datenbank kann dadurch sehr effizient einzelne Attribute aggregieren. Eine MapReduce Anwendung muss hingegen jedes mal die gesamten Daten, inklusiver der für die Berechnung irrelevanter Teile, von der Festplatte einlesen. Die Entwickler von MapReduce halten in [DeGh10] entgegen, dass MapReduce durchaus mit gewissen Datenbanksystemen, wie BigTable von Google [CDGH+08], kombiniert werden kann, wodurch dieser Kritikpunkt zu gewissen Teilen entkräftet wird.

- *Schemalose Datenhaltung.* Klassische Datenbanksysteme setzen bei der Datenhaltung auch Schemata, was die Durchführung von Berechnungen vorhersehbar und Anfragen ex-ante evaluierbar macht. Zudem ist eine Kapselung der Datenschicht von der Anwendungsschicht mittels *views* in SQL möglich, sodass Anwendung ohne explizite Kenntnis der Daten verfasst werden können. Dies ermöglicht eine breite und fehlertolerante Nutzung der Daten, auch von Anwendern ohne Kenntnisse von höheren Programmiersprachen. MapReduce nutzt hingegen klassischerweise eine Datenhaltung mittels Textdateien. Sollte sich der Aufbau dieser ändern, ist eine Anpassung der Anwendung notwendig (zumindest in der Verarbeitungsebene des Map-Teils). Allerdings wurde auch hier erneut die Option der Verwendung einer Datenbank, wie BigTable, in [DeGh10] genannt. Dadurch können mit MapReduce durchaus die Datenhaltung mit einem Schema kombiniert werden.

- *Fehlende Indizierung.* Oft wiederholte Berechnungen können in Datenbanken durch die Erstellung eines Index beschleunigt werden. Dieser speichert für ein Attribut, an welcher Stelle im Sekundärspeicher die jeweiligen Einträge gespeichert sind. Dies erfolgt meist in Form eines flachen, breiten Baums (B+Tree Index), wodurch einzelne Einträge, aber auch zusammenhängende Wertebereiche, mit wenigen Lesevorgängen des in der Regel langsamen Sekundärspeichermediums gefunden werden können. MapReduce erfordert hierfür hingegen in einer klassischen Implementierung einen Scan über das gesamte Dateisystem.

[1] Zahlreiche Hersteller verbieten in den 'End User License Agreements' ihrer Produkte das Publizieren von Benchmarks, sofern dies nicht vorher ausdrücklich gestattet wurde. Siehe dazu auch: http://sqlmag.com/sql-server/devils-dewitt-clause

Die Erfinder von MapReduce halten diesem Kritikpunkt entgegen, dass durchaus in einer eigenen Implementierung einer MapReduce Anwendung die Funktionalität eines Index nachgebildet werden kann. Hierfür kann der Nutzer zum Beispiel die Textdateien sinnvoll aufteilen und anhand einer Hash-Funktion die relevante Quelle aufspüren. Dies erfordert aber im Gegenteil zu den Indizes von Datenbanksystemen eigenen Entwicklungsaufwand des Nutzers und unterstützt zunächst nur ein Attribut, während bei Datenbanken mehrere[2] Indizes gebildet werden können.

Auf der anderen Seite räumen die Autoren von [PPRA+09] auch gewisse Stärken von MapReduce und Hadoop im Vergleich zu den beiden Konkurrenzprodukten ein. Diese lassen sich, unter Berücksichtigung der Anmerkungen in [DeGh10], zusammenfassend in vier Kategorien gliedern.

- *Einfache, günstige Installation.* Im direkten Vergleich mit den beiden Datenbanksystemen lässt sich Hadoop sehr leicht konfigurieren und auf einer großen Anzahl Knoten betreiben, was laut den Autoren von [PPRA+09] unter anderem die große Beliebtheit der Plattform erklärt. Für die Installation der zeilenorientierten Datenbank musste hingegen bei den Testläufen in [PPRA+09] der Support des Herstellers in Anspruch genommen werden. Zudem ist Hadoop eine open-source Anwendung und damit kostenlos nutzbar, während die beiden parallelen Datenbanksysteme in [PPRA+09] kommerzielle Produkte sind.

- *Performanz im Fehlerfall.* Im Gegenteil zu den Datenbanksystemen muss im MapReduce Framework bei Ausfall eines Knotens lediglich der darauf ausführte Prozess neu gestartet werden. Dies ist mit weniger Aufwand verbunden als bei den anderen Vergleichsprodukten, welche den kompletten Neustart der gesamten Berechnungen erfordern [PPRA+09].

- *Performanz einmaliger Berechnungen.* Falls Eingabedaten einmalig verarbeitet werden sollen, ist MapReduce aufgrund der geringen Ladezeiten den Datenbanken signifikant überlegen. Diese müssen zunächst die Daten einlesen und in ihren eigenem Dateisystem mit zugehörigem Schema speichern. Bei den in [PPRA+09] durchgeführten Benchmarks dauern dabei die Ladephasen circa fünf bis 50-mal so lange wie die eigentlichen Berechnungen mit MapReduce. Falls daher lediglich Log-Dateien gelesen und verarbeitet werden sollen, ist MapReduce den Datenbanken überlegen. Wie schon zu Beginn in Kapitel 1 erläutert, wurde MapReduce ursprünglich insbesondere für solche Anwendungsfälle entwickelt.

- *Flexibilität der Berechnungen.* Im Gegensatz zu den Datenbanken ist MapReduce nicht auf SQL angewiesen, was flexibel komplexe Berechnungen mit höheren Programmiersprachen erlaubt. Dies ermöglicht es Google zum Beispiel, Straßeninformationen zu aggregieren und damit Kartenausschnitte für

[2]Eine Relation in einer klassischen zeilenorientierten Datenbank kann mehrere Hash- oder B+Tree-Indizes haben, solange diese *unclustered* sind. Es kann allerdings pro Relation maximal einen *clustered* Index geben, da dieser Index die Anordnung der einzelnen Daten auf dem Hauptspeicher diktiert. Letzteres Format hat den Vorteil, dass zusammenhängende Einträge sequenziell gelesen werden können, was bei häufigen Ungleichheits-Abfragen die Performanz positiv beeinflusst.

den Dienst Google Maps zu generieren [DeGh10]. Dies ist zwar konzeptionell auch mithilfe von sogenannten 'user defined functions' (UDF) in Verbindung mit SQL möglich. In der Praxis verursachen diese aber oft Probleme bzw. sind bei manchen Datenbanken nicht unterstützt, wie die Autoren von [PPRA+09] eingestehen.

Zusammenfassend haben beide Konzepte, MapReduce und die parallelen Datenbanksysteme, ihre Vorteile und damit prädestinierte Einsatzgebiete. Diesen Schluss haben auch die Autoren von [PPRA+09] in einer späteren Publikation gezogen [SADM+10]. In diesem Artikel sind sie von einem kritischen Vergleich der verschiedenen Systeme mittels Anwendungsfällen, für die die Systeme teils nicht geschaffen wurden, abgewichen. Stattdessen halten sie eine Kombination beider Konzepte für gewinnbringend, wodurch die jeweiligen Stärken kombiniert werden können. Dies ist mittels eines Interfaces für MapReduce denkbar, sodass komplexe Analysen in MapReduce durchgeführt werden, für gewisse wiederkehrende Queries aber trotzdem eine ebenfalls vorhandene Datenbank genutzt werden kann. Solche hybride Lösungen gibt es mittlerweile unter anderem schon in Form der HadoopDB und einer Erweiterung für Vertica [SADM+10]. Wie in den ersten beiden Kritikpunkten schon erwähnt, besteht zudem die Möglichkeit, MapReduce mit einer textuell-orientierten Datenbank, wie Googles BigTable zu kombinieren, wodurch ebenfalls in gewissem Maße Vorteile beider Welten kombiniert werden können [DeGh10, CDGH+08].

6.2 Ideale Einsatzgebiete

Die vorab geschilderte Analyse legt nahe, insbesondere bei komplexen Berechnungen, die selten wiederholt werden müssen dem Programmiermodell MapReduce den Vorzug im Vergleich zu verteilten Datenbankanwendungen zu geben. Wie in Kapitel 4.2 erläutert gibt es aber auch für solche Szenarien zum Teil schon modernere Alternativen, die maßgeschneidert gewisse Anwendungsfälle effizienter lösen. Diese sind vor allem dann empfehlenswert, wenn eine Anwendung mehrmalige Iterationen über den Datensatz erfordert, inkrementelle Anpassungen der Daten effizient zu einer Adjustierung der Berechnungen führen sollen oder Eingabedaten mittels eines Streams und nicht als vorliegende Datei verarbeitet werden sollen. Zudem gibt es Lösungen wie Google Pregel (siehe Kapitel 4.2), die Paradigmen aus dem MapReduce Programmiermodell auf Graphen anwenden. Zudem wurde durch praktische Erfahrungen in Kapitel 5 geschlossen, dass der Einsatz von MapReduce sich insbesondere bei großen Datensätzen und komplexen Berechnungen lohnt. Hierbei entfällt relativ betrachtet ein geringerer Zeitaufwand auf den Mehraufwand an Berechnungen durch das MapReduce Framework. Zudem macht nur bei solchen Szenarien der Einsatz eines großen Clusters Sinn, wofür MapReduce prädestiniert ist und Eigenschaften wie Ausfalltoleranz und Skalierbarkeit relevant sind.

Zusammenfassend eignet sich das MapReduce Framework daher insbesondere für die verteilte Analyse großer Datensätze auf Rechnerverbünden mit einer großen Anzahl Knoten, wobei die Daten im Dateisystem vorliegen und als gesamtes analysiert werden (*batch*-Modus). Funktional sollten bei der Berechnung nur wenige Iterationen über den gesamten Datensatz notwendig sein und eine Unabhängigkeit der einzelnen Datenfragmente untereinander gewährleistet sein. Dafür können die Berechnungen

beliebig komplex sein, und sind nicht auf das Vokabular von Datenmanipulationssprachen, wie SQL, beschränkt. Des weiteren eignet sich MapReduce für Datensätze, die nur einmal analysiert werden müssen und nicht regelmäßig geändert werden, da MapReduce inkrementelle Anpassungen der Analyse nicht effizient unterstützt. Auch wenn es zehn Jahre nach Vorstellung des Frameworks moderne Alternativen mit diversen Vorteilen gibt, erfreut sich MapReduce und die Laufzeitumgebung nach wie vor einer großen Beliebtheit und Verbreitung. Daher kann auf eine umfangreiche Dokumentation, und bei Bedarf auch auf leistungsfähige Laufzeitumgebungen in der Cloud zurück gegriffen werden, was MapReduce nach wie vor zur einem attraktiven Framework für viele Anwendungsfälle macht.

7. Zusammenfassung und Ausblick

In dieser Arbeit wurde das Programmiermodell MapReduce und eine weit verbreitete Laufzeitumgebung für MapReduce Anwendungen, Apache Hadoop, vorgestellt. Das MapReduce Framework erlaubt es, mit geringem Entwicklungsaufwand verteilte Anwendungen für die Ausführung auf großen Rechnerverbünden zu erstellen. Dabei wird dem Anwender bei der Ausführung der Programme die Parallelisierung, die Skalierung und der Umgang mit Hardwarefehlern von der Laufzeitumgebung abgenommen.

Es gibt neben MapReduce eine große Anzahl Alternativen, die ebenfalls für die Verarbeitung großer Datenmengen entwickelt wurden. In dieser Arbeit wurde dabei vor allem zwischen parallelisiert betrieben Datenbank Management Systemen und alternativen Programmierparadigmen für die Entwicklung verteilter Anwendungen unterschieden. Durch eine Analyse der Stärken und Schwächen der einzelnen Systeme sowie der Implementierung eines Anwendungsbeispiels wurden die idealen Einsatzgebiete für MapReduce herausgearbeitet. Grundsätzlich eignet sich MapReduce für die Analyse großer Datensätze, die einmalig verarbeitet werden sollen, funktional aufteilbar sind und bei der Verarbeitung keine wiederholten Iterationen erfordern (die idealen Einsatzgebiete werden in weiterem Detail in Kapitel 6.2 erläutert).

Obwohl das Konzept schon vor über einer Dekade von Google öffentlich vorgestellt wurde und es für viele Einsatzzwecke effizientere und maßgeschneiderte Lösungen (z.B. für die Analysen von Graphen) gibt, erfreut sich MapReduce nach wie vor großer Beliebtheit und Verbreitung. Viele Unternehmen betreiben Cluster mit der Laufzeitumgebung Apache Hadoop, und es gibt Cloud-basierte Angebote, die verteilte Berechnungen auf skalierbaren Clustern bei Bedarf kostengünstig zur Verfügung stellen. Zudem ist die Entwicklung von MapReduce Anwendungen dank ausführlichen Dokumentationen und des einfachen und flexiblen Programmiermodells schnell erlernbar. Daher ist das Programmiermodell auch heute noch für gewisse Anwendungsfälle von Relevanz. Die Laufzeitumgebung Apache Hadoop wurde darüber hinaus in der aktuellen Version dank YARN architektonisch unabhängig von MapReduce, was eine Investition in einen Hadoop Cluster attraktiv gestaltet. Somit können ebenfalls modernere Konzepte, wie Apache Spark, auf einer bestehenden Infrastruktur als Alternative zu MapReduce eingesetzt werden.

Abschließend lässt diese Seminararbeit aufgrund des begrenzten Umfangs Raum für mindestens zwei weitere Analysen. Zum einen kann unter Einsatz größerer Ressourcen eine umfangreichere Analyse der Performanz des Frameworks in der virtuellen Cluster-Umgebung von Amazon Elastic MapReduce (EMR) vorgenommen werden. Hierbei wäre von Interesse wie sich die Laufzeit mit der Größe des Clusters verändert wenn umfangreichere Datensätze analysiert werden. EMR bietet hierzu die Möglichkeit, den laufenden Cluster leicht zu vergrößern. Zudem wäre eine Unterscheidung nach Komplexitätsgrad der Anwendung interessant. Die in Kapitel 5 referenzierte Arbeit [RRPB+07] hat eine ähnliche Testmethodik, allerdings wurden die Tests lediglich auf einem *shared-memory* System mit mehreren Prozessorkernen durchgeführt. Auf der anderen Seite wäre eine detailliertere Analyse von alternativen Programmierkonzepten (wie in Kapitel 4.2 knapp beschrieben) interessant. Hierbei wäre auch bemerkenswert, ob es ein System gibt, das ähnlich flexible Anwendungen wie MapReduce erlaubt, diesem aber hinsichtlich der Performanz überlegen ist.

A. Quellcode

A.1 MapReduce Anwendung NYCTaxi

```
//NYCTaxiMapper
package bda;
import java.io.IOException;
import java.util.Arrays;
import java.util.List;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class NYCTaxiMapper
  extends Mapper<LongWritable, Text, Text, DoubleWritable> {

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    String line = value.toString();
    String pickup = parseCSV(line, 15);
    String trip = pickup + ",";

    // pure fare, no tolls or other surcharges
    double fareAmount = Double.parseDouble(parseCSV(line, 11));
    context.write(new Text(trip), new DoubleWritable(fareAmount));
  }
  public String parseCSV(String line, int pos){
         List<String> items = Arrays.asList(line.split("\\s*,\\s*"));
         return items.get(pos);
  }
}
```

```
// NYCTaxiReducer
package bda;
import java.io.IOException;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class NYCTaxiReducer
  extends Reducer<Text, DoubleWritable, Text, DoubleWritable> {

  @Override
  public void reduce(Text key, Iterable<DoubleWritable> values,
      Context context)
      throws IOException, InterruptedException {
          // Sum up the number trips and the dollarPerHour-ratio for one
              connection
          double aggregatedFare = 0.0;
          for (DoubleWritable v : values){
                  aggregatedFare += v.get();
          }
          context.write(key, new DoubleWritable(aggregatedFare));
  }
}
// NYCTaxi - main
package bda;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class NYCTaxi {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: NYCTaxi <input path> <output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(NYCTaxi.class);
    job.setJobName("NYCTaxi Example");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.setMapperClass(NYCTaxiMapper.class);
    job.setReducerClass(NYCTaxiReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(DoubleWritable.class);
    job.waitForCompletion(true);
  }
}
```

A.2 Konventionelles Äquivalent zu NYCTaxi

```
// SingleProcess
package classic;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Collections;

public class SingleProcess {

/**
 * @param input output
 */
 public static void main(String[] args) {
       if (args.length != 2) {
              System.err.println("Usage: SingleProcess <input path>
                  <output path>");
          System.exit(-1);
       }
       long startTime = System.currentTimeMillis();
       BufferedReader br;
       String line, pickup;
       double fareAmount;
       StorageType temp, temp2;
       ArrayList<StorageType> list = new ArrayList<StorageType>();
       try {
              br = new BufferedReader(new FileReader(args[0]));
              while ((line = br.readLine()) != null) {
                pickup = parseCSV(line, 15);
                fareAmount = Double.parseDouble(parseCSV(line, 11));
                temp = new StorageType(pickup, fareAmount);
                if (!list.contains(temp))
                {
                  // Pickup-Code does not exist yet, add temp to list
                  list.add(temp);
                } else {
                  // Pickup-Code does exist in list, add fare of temp to
                      existing entry
                  int idx = list.indexOf(temp);
                  temp2 = list.remove(idx);
                  temp.add(temp2);
                  list.add(temp);
                }
              }
              br.close();
       }
       catch (Exception e)
```

```
        {
                e.printStackTrace();
        }

        // in order to provide a fair comparison, sort list since MR does
           the same
        Collections.sort(list);
        // Write to file
        try {
                FileWriter writer = new FileWriter(args[1]);
                for(StorageType st: list) {
                        writer.write(st.getPickup() + ", " + st.getFare() +
                           "\n");
                }
                writer.close();
        } catch (IOException e) {
                e.printStackTrace();
        }
        long endTime  = System.currentTimeMillis();
        long totalTime = endTime - startTime;
        System.out.println("Runtime: " + totalTime/1000 + " seconds");
 }

 public static String parseCSV(String line, int pos){
        List<String> items = Arrays.asList(line.split("\\s*,\\s*"));
        return items.get(pos);
 }
}

// StorageType
package classic;
public class StorageType implements Comparable{
        private String pickup;
        private double fare;

        @Override
        public String toString() {
                return "StorageType [pickup=" + pickup + ", fare=" + fare
                   + "]";
        }
        public StorageType(String pickup, double fare) {
                super();
                this.pickup = pickup;
                this.fare = fare;
        }
        public String getPickup() {
                return pickup;
        }
        public void setPickup(String pickup) {
                this.pickup = pickup;
        }
        public double getFare() {
```

```
            return fare;
        }
        public void setFare(double fare) {
            this.fare = fare;
        }
        public boolean add(StorageType addMe)
        {
            if (!this.pickup.equals(addMe.getPickup())) return false;
            this.fare += addMe.getFare();
            return true;
        }
        // Compare only according to pickup location
        @Override
        public int hashCode()
        {
            return pickup.hashCode();
        }
        @Override
        public boolean equals(Object o)
        {
            if (o == null) return false;
            if (!(o instanceof StorageType)) return false;
            StorageType st = (StorageType) o;
            return this.pickup.equals(st.getPickup());
         }
        @Override
        public int compareTo(Object o) {
           if (o == null) return 0;
           if (!(o instanceof StorageType)) return 0;
           StorageType st = (StorageType) o;
           return this.pickup.compareTo(st.getPickup());
        }
}
```

Literaturverzeichnis

[CDGH+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes und Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2), 2008, S. 4.

[CKLY+07] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng und Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, Band 19, 2007, S. 281.

[DeGh04] Jeffrey Dean und Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, USA*, 2004, S. 137–150.

[DeGh08] Jeffrey Dean und Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008, S. 107–113.

[DeGh10] Jeffrey Dean und Sanjay Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53(1), 2010, S. 72–77.

[DeSt08] David DeWitt und Michael Stonebraker. MapReduce: A major step backwards. *The Database Column*, 2008. Nicht mehr im Original verfügbar. Kopie unter http://craig-henderson.blogspot.de/2009/11/dewitt-and-stonebrakers-mapreduce-major.html (Stand 07/14).

[Foun13] The Apache Software Foundation. The Apache Software Foundation Announces Apache Hadoop 2, Oktober 2013. blogs.apache.org/foundation/entry/the_apache_software_foundation_announces48.

[Foun14] The Apache Software Foundation. Apache Hadoop Releases, Juni 2014. http://hadoop.apache.org/releases.html.

[GhGL03] Sanjay Ghemawat, Howard Gobioff und Shun-Tak Leung. The Google file system. In *ACM SIGOPS Operating Systems Review*, Band 37. ACM, 2003, S. 29–43.

[Gott07] Derek Gottfrid. Self-Service, Prorated Supercomputing Fun, November 2007. http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun.

[MABD+10] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser und Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, S. 135–146.

[Metz13] Cade Metz. Spark: Open Source Superstar Rewrites Future of Big Data, Juni 2013. http://www.wired.com/2013/06/yahoo-amazon-amplab-spark/all/.

[Metz14] Cade Metz. Google Unleashes More Big-Data Genius With a New Cloud Service, Juni 2014. http://www.wired.com/2014/06/google-cloud-data-flow/.

[Mona09] Curt Monash. Cloudera presents the MapReduce Bull case. *In DBMS.com*, 2009. http://www.dbms2.com/2009/04/15/cloudera-presents-the-mapreduce-bull-case/.

[PeDa10] Daniel Peng und Frank Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[PPRA+09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden und Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, S. 165–178.

[RRPB+07] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski und Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. IEEE, 2007, S. 13–24.

[SADM+10] Michael Stonebraker, Daniel Abadi, David J DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo und Alexander Rasin. MapReduce and parallel DBMSs: friends or foes? *Communications of the ACM*, 53(1), 2010, S. 64–71.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia und Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, S. 1–10.

[Whit12] Tom White. *Hadoop: The definitive guide. 3rd Edition.* O'Reilly Media, Inc. 2012.

[ZCFS+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker und Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, S. 10–10.