Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Fachgebiet DIMA
Prof. Dr. Volker Markl

Datenbanksysteme und Informationsmanagement

Seminar

Large scale data mining mit Apache Mahout

Wintersemester 2009/2010

MapReduce: Simplified Data Processing on Large Clusters

Oleg Mayevskiy

Februar 2010

Betreuer: Isabel Drost

Oleg Mayevskiy
233197
Studiengang: Informatik (D)
E-Mail-Adresse olem@mailbox.tu-berlin.de

MANAGEMENT SUMMARY

MapReduce ist ein Programmiermodel, dass bereits in den meistbenutzten Programmiersprachen wie Java, C++, C# implementiert ist. Diese Arbeit konzentriert sich auf dem Erfinder Google und deren C++ Implementierung, die darauf ausgerichtet ist auf einem Cluster aus mehr als 1000 Maschinen zu laufen. Schon heute löst Google mit der Platform viele wichtige Probleme der Massendatenverarbeitung. Der prominenteste Vertretter ist Google Index, der für die allen bekannte Websuche benutzt wird. Die Vorteile von MapReduce gegenüber der traditionellen parallelen Programmierung liegen auf der Hand. Dank der starken Reduzierung des Progammiermodels können sich Entwickler auf das eigentliche Problem konzentrieren und brauchen sich nicht mit der Verteilung, Synchronisation und Ausführung auf einzelnen Rechnern und damit verbundenen Problemfällen zu beschäftigen. Dies führt auch zu stark verkürzten Entwicklungszeiten für neue oder bereits bestehende Aufgaben. Die Google Bibliothek ist so gebaut, dass eine Steigerung der Leistung des Clusters sehr einfach ist, in dem man weitere Maschinen hinzufügt. Die automatische Lastverteilung, das Auffinden von Knoten und die Ausfallsicherheit wird versteckt von MapReduce Anwender im System abgefangen.

INHALT

1 EINLEITUNG

Mit der Erfindung von Internet ist die Idee geboren, interessante Daten aus dem Internet herauszufiltern und dem Benutzer in einer anderen interessanten Form dazustellen und damit auch Geld zu verdienen. So entstand Google, aber auch eine Menge damit verbundener und zu lösenden Probleme. Die Extrahierung der Daten aus dem Internet bedeutet, dass man riesige Datenmengen herunterladen, speichern und verarbeiten muss. Diese Aufgabe ist unmöglich auf einzelnen Systemen zu bewältigen.

Man hat riesige Clustersysteme gebaut, die mächtig genug sind, die anfallenden Aufgaben in angebrachter Zeit zu lösen. Auf der Softwareebene hat man aber festgestellt, dass die Software für solche Systeme eine extrem kostenspielige und aufwendige Entwicklung benötigt. Es entstand die Idee von MapReduce.

Hierbei wird das Programmiermodel in einer funktionalen Art und Weise auf nur zwei Methoden map und reduce reduziert. Dadurch wird es möglich den kompletten Parallelisierungs- und Kommunikationsaufwand im Hintergrund abzufertigen und dem Entwickler des eigentlichen Algorithmus sich auf sein eigentliches Problemm konzentrieren zu lassen.

2 PROGRAMMIERMODEL

Das MapReduce Programmiermodel folgt dem Prinzip der funktionalen Programmierung. Wie der Name schon sagt wird das Model auf zwei Funktionen reduziert: map und reduce.

Die map Funktion bekommt als Input ein Paar aus Schlüssel und Wert und verarbeitet es zu eine Menge von erzeugten Schlüssel Wert Paaren.

Die reduce Funktion nimmt alle einem Schlüssel zugeordnete Werte und verarbeitet es zu einem aggregierten Ergebnis.

Die Vorgehensweise entspricht folgendem Pseudecode:

```
function map (String key1, String value1) {…}
output: Set<(key2; value2)>

function reduce (String key1, Iterator values2) {…}
output: Set
```

2.1 Beispiel WordCounter

Dieses Beispiel beschäftigt sich mit dem Problem der Ermittlung der Anzahl der Wörter in Textdokumenten.

Gegeben seien eine grosse Menge von Dokumenten. Es soll pro Wort die Gesamtanzahl der Vorkommnisse in allen Dokumenten berechnet werden. Dieses Problem kann man wie folgt lösen:

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


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

[DEA04 – S.2]

- Die map Funktion nimmt als Eingabe den Dokumentnamen (key) und Dokumentinhalt (value).
- Dann wird in der Funktion über alle Wörter itteriert und zu jedem Wort ein Objektenpaar aus dem Wort selbst (key2) zusammen mit einem Wert „1“ erzeugt.
- Akkumuliert dient diese Ausgabe als Eingabe für die reduce Funktion.

- Dabei werden alle Ergebnisse mit gleichem Schlüssel (key2 – aus der map-Function) zu der gleichen reduce Funktion gesendet.
- Die reduce Funktion hat als Eingabe demzufolge ein Wort(key) und einen Iterator über alle Ergebnisse, die zu diesem Wort gehören. In diesem Fall die Zahl „1“
- Als Ausgabe wird das aggregierte Ergebnis zurückgegeben. Dieses entspricht somit der Summe der Vorkomnisse der einzelnen Wörter.

Den kompletten C++ Code von dem WordCounter kann man im Anhang A betrachten.

3 VERARBEITUNGSABLAUF

Der Verarbeitungsablauf von einem MapReduce Aufruf kann mit sieben Schritten beschrieben werden, zur Veranschaulichung dient Abbildung 1.

3.1 Splitten des Inputs

Die Gesamtmenge der Eingabe, welche durchaus mehrere TB Daten erfassen kann, wird in kleinere Teile (16-64 MB) geteilt.

Jedes Teil wird einem der insgesamt M Workern zugeteilt. (M ist die Anzahl der Worker die jeweils eine map Funktion ausführen.)

3.2 Init Master worker

Der Gesamtprozess wird durch einen Master worker überwacht. Dieser kümmert sich um die Verteilung von allen map und reduce Workern auf verschiedene Rechner im Cluster.

Desweiteren besitzt der Master die Informationen darüber, wo sich die Daten befinden. Es gibt insgesamt M map Worker und R reduce Worker zu überwachen.

3.3 Start map worker

Jeder map Worker verarbeitet die ihm zugewiesene Dateisubmenge. Er liest die Daten und parst diese entsprechend. Dann werden die erzeugten key/value Paare an die vom Benutzer implementierte map Funktion übergeben. Die von der map Funktion berechnete Ergebnisse werden zuerst im Speicher zwischengespeichert.

3.4 Map worker Ergebnisse speichern

Die Zwischenergebnisse der map Funktion werden periodisch in R Regionen auf lokalen Festplatten geschrieben. Welches Ergebnis in welche Region geschrieben wird, kann dabei durch eine Partitionierungsfunktion extra bestimmt werden. Der map Worker benachrichtigt den Master über die Positionen der lokal geschriebenen Daten, damit dieser diese Information nach Ende der Map-Verarbeitung an die reduce Worker weiter delegieren kann.

3.5 Start reduce worker

Der Master worker startet reduce Worker und benachrichtigt diese darüber, wo sich die Zwischenergebnisse befinden. Die reduce Worker greifen entfernt auf die Zwischenergebnisse der map worker zu. Nachdem alle Daten ausgelesen wurden, werden diese nach Schlüsseln sortiert, um die zu einem gleichen Schlüssel gehörende Datenmengen zusammenzustellen.

3.6 Finale Ergebnisse speichern

Jeder reduce Worker itteriert über die sortierten Schlüsseln und übergibt das Schlüssel und das Iterator Paar an die benutzerdefinierte reduce Funktion. Die Ausgabe der reduce-Function wird in die endgültige Zieldatei für die jeweilige Partition ausgegeben.

3.7 Rückker zum Benutzerprogramm

Wenn alle map und reduce Worker abgearbeitet sind, kehrt das Frameworks an die Stelle des MapReduce Aufrufs zum Benutzerprogramm zurück.

Nach der erfolgreicher Verarbeitung stehen die ermittelten Ergebnisse in R Ausgabedateien zur Verfügung. Üblicherweise werden die Ausgabedateien nachher nicht manuell zusammgesetzt, sondern entweder an eine weitere MapReduce übergeben oder an eine andere Anwendung welche mit verteilt vorliegenden Daten umgehen kann.

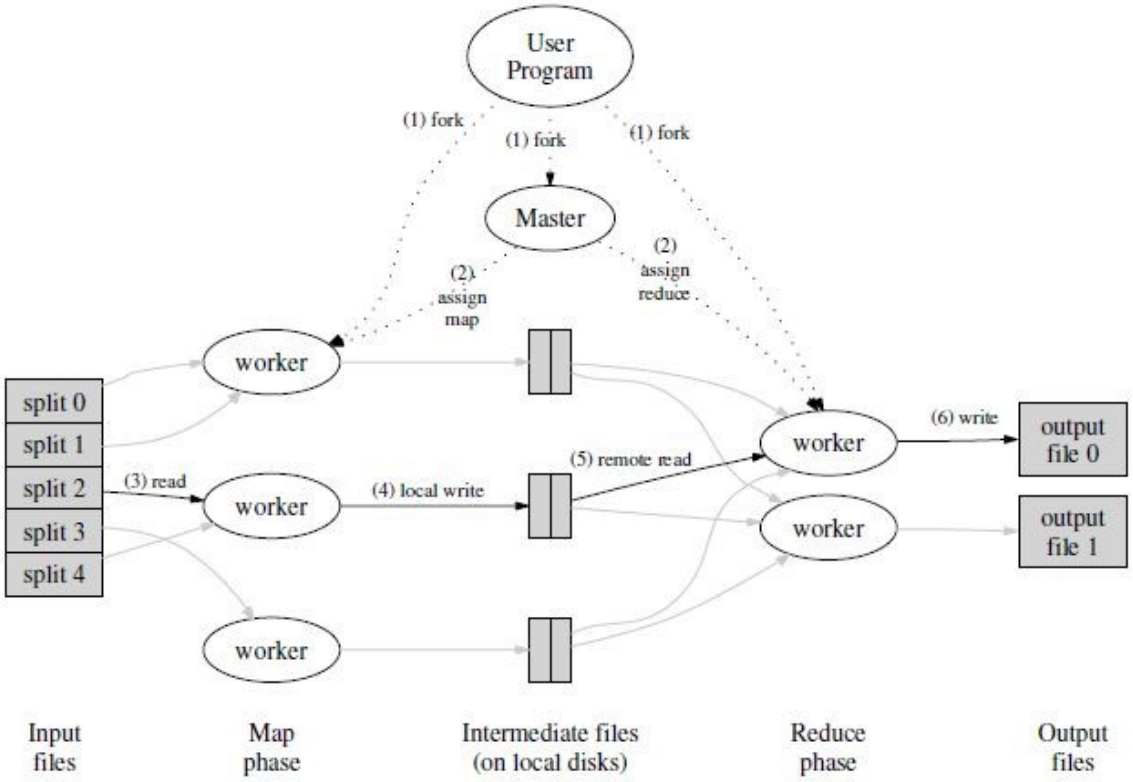

Abbildung 1 [DEA04 – S.3]

4 MASTER WORKER

Der Master Worker ist eine zentrale Komponente in der MapReduce Bibliothek. Er speichert alle anfallenden Metadaten die im Verlauf entstehen, steuert anhand dieser den Gesamtprozess und bietet ausserdem einen http Server an, mithilfe dessen ein Benutzer den Prozess überwachen kann.

Zu den Metadaten gehören Informationen über Prozesszustand der map und reduce Worker (idle, in-progress, complete), Addressen der entsprechenden Maschinen, der Positionen der Daten für map sowie reduce Worker. Der Master Worker befindet sich mit allen anderen Workern im ständigen Kontakt. Werden Daten von einem map Worker aktualisiert, so wird das unverzüglich an Master Worker gesendet, dieser teilt es dann den laufenden reduce Workern mit.

Das Debuggen im cluster ist schwierig, trotzdem bietet hier der interne http Server eine gute Hilfe. Hier kann durch einen Menschen beobachtet werden, wie weit der Prozess ist, wieviele Worker in-progress oder fertig sind, welche Worker und mit welchen Daten fehlgeschlagen sind, wieviel Input / Output (in GB) beansprucht wurde. Mit Hilfe dieser Informationen kann abgeschätzt werden, wie lange die Ausführung noch andauern wird und ob zusätzliche Resourcen angefordert werden sollten. So kann ein Benutzer die Anzahl der map und reduce Worker anpassen. Ausserdem kann man anhand von fehlgeschlagenen Workern erkennen, ob es Fehler in Code von map oder reduce Funktionen gibt und kann anhand der verarbeiteten Daten einschätzen, wo der Fehler liegen kann.

Der Master Worker kann ausfallen. Um den Prozess trotzdem weiterführen zu können, wird überlegt save points einzubauen. Man kann periodisch den Zustand des Master Workers absichern und ab dem letzten gesicherten Zustand neustarten, falls der Master Worker fehlschlägt. Die MapReduce Entwickler von Google haben sich allerdings dazu entschieden den Prozess in einem solchen Fall abzubrechen. Der Benutzer muss danach sein Programm neustarten und MapReduce erneuet aufrufen. Dies erwies sich als praktikabel, da ein Ausfall des Master workers unwahrscheinlich und selten ist.

5 FEATURES

5.1 Lokalität

In einem Cluster sind die einzelnen Maschinen über Netzwerk miteinander verbunden. Typischerweise handelt es sich um 100Mbit oder 1 Gbit Netzwerkadapter. Somit ist das Netzwerk der Flaschenhals für sehr grosse Datenmengen. Aus diesem Grund achtet Google sehr darauf, die Netzwerkauslastung so gering wie möglich zu halten. Es ist besonders wichtig, dass Daten möglichst von lokalen Platten gelesen werden. Der Master worker gibt Acht bei dieser Angelegenheit. Er besitzt Meta Informationen darüber, wo sich die Daten befinden. Es wird versucht die Map Worker auf den Maschinen zu starten, auf dennen sich auch die zugewiesenen Daten befinden. Sollte es nicht möglich sein, wird die am nächst liegende Maschine gesucht (z.B. eine Maschine im gleichen Switch).

5.2 Backup tasks

Durch Beobachtungen hat das Team von Google-Entwicklern festgestellt, dass es bei einer solch massiven Rechneranzahl öfters so genannte straggler gibt. Es handelt sich um Maschinen, die für eine map oder reduce Ausführung ungewöhnlich lange brauchen. Es kam vor, dass der Festplattencache oder der Second level Cache der CPU durch Fehlkonfiguration oder Hardwareausfall abgeschaltet wurde. Dies führte dazu, dass sich die Ausführung um 30 bis 100 Fach verlangsamt hat. Für diesen Fall wurden die Backup tasks entwickelt. Ist der Gesamtprozess nah am Ausführungsende, so werden für bereits laufende Prozesse backup tasks gestartet, die den gleichen Job zugewiesen bekommen wie die laufenden Prozesse. Durch besonderes Tuning führt dies bei Operationen ohne einen einzigen straggler zu nur ein paar Prozent Leistungsnachteilen. Dagegen verkürztete sich die Gesamtausführungszeit bei vielen Fällen bis zu 44%, denn nicht richtig laufende Maschinen in einem sehr grossen Cluster sind vorprogrammiert.

5.3 Partitionierungsfunktion

Die Partitionierungsfunktion bestimmt, an welchem Ort die Ergebnisse der reduce worker gespeichert werden. Die Standart Funktion lautet hash(key) mod R. Es bietet eine gute Standartverteilung an. Die Daten werden auf lokalen Platten der reduce worker geschrieben, was auch aus Performance Gründen erwünscht ist. Es ist aber oft sinnvoll, selbst zu bestimmten, wo die Daten gespeichert werden sollen. Die Partitionierungsfunktion kann durch den Anwender bestimmt werden. Es macht z.B. Sinn es auf hash(Hostname(urlkey)) festzulegen. Dadurch werden alle Ergebnisse die zu einem Host gehören in die gleiche Datei geschrieben.

5.4 Bad records

Software wird von Menschen geschrieben. Auch die besten Softwareexperten schaffen es nicht Software mit mehreren Tausend Zeilen an Code bugfrei zu entwickeln. Bei MapReduce Algorithmen kann es vorkommen, dass die worker bei bestimmten records fehlschlagen. Der Master Worker registriert, welche Worker an welchen Daten fehlschlagen. Bricht ein Worker mehr als ein Mal an der gleichen Stelle ab, so wird ein solcher record in eine black list eingetragen und nicht mehr zur Verarbeitung übergeben.

Eine solche Vorgehensweise ist sinnvoll. Bei riesigen Datenmengen kommt es auf einzelne Tupeln nicht an, meistens möchte man eine statistische Auswertung ausführen, bei der die genauen Zahlen unwichtig sind. Man möchte nicht, dass eine lange Ausführung wegen einpaar Datensätzen vollständig abbricht.

5.5 Lokales Debuggen

Das Debuggen im Cluster ist schwierig. Eine Hilfestellung ist der interne http Server des Master workers. Dies kann nicht mit dem traditionellen Profiling und Debuggen vergliechen werden. Hierfür wurde von Google eine spezielle MapReduce Bibliothek entwickelt. Diese kann lokal gestartet und wie üblich debuggt werden. Die Ausführung findet nur auf einer Maschine statt, man führt es aber wie auf einem Cluster aus und hat ein annehmbar gleiches Verhalten. Es hilft Bugs in der eigenen Software zu entdecken. Vor Ausfällen auf dem live System trotz dem feinem Debuggen schützt es allerdings nicht.

6 ZAHLEN UND FAKTEN

MapReduce wurde im Februar 2003 von Google entwickelt.

MapReduce Bibliotheken sind auch heute im Einsatz auf sehr grossen Clustersystem.

Der Google Websuche Index wird im MapReduce Still erstellt.

Bei Google werden viele andere Probleme über riesige Datenmengen mit Hilfe von MapReduce gelöst, z.B. das Extrahieren von Eigenschaften der Webseiten.

Abbildung 2 veranschaulicht , wie schnell und stark MapReduce Jobs an Bedeutung zugenommen haben.

Abbildung 3 zeigt die Ausmasse der Datenmenge, die nur im August 2004 verarbeitet wurden und andere Interessante statistische Angaben.

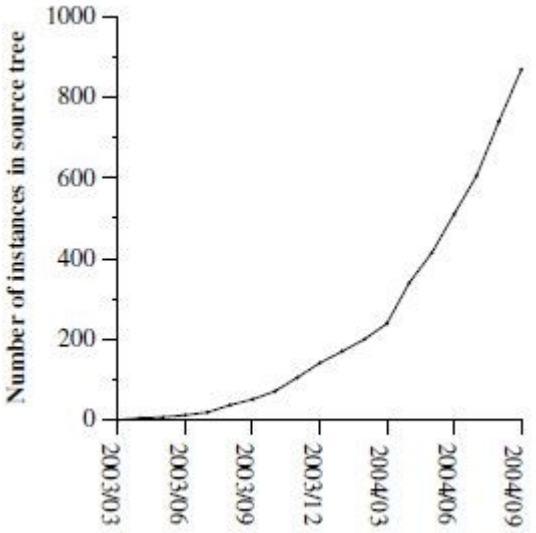

Abbildung 2 [DEA04 S.10]

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Abbildung 3 [DEA04 S.10]

7 FAZIT

Die Ausarbeitung hat den Begriff MapReduce erklärt. Bei näherer Erläutung wurde Bezug auf die Erfinder Implementierung von Google genommen. MapReduce bietet einem Entwickler eine stark reduzierte funktionale Sicht, um sein algorithmisches Problem zu lösen. Dies ermöglicht die Parallelisierung in die Bibliothek zu verlagern. Desweiteren werden viele andere Probleme wie Lokalität, backup tasks, straggler und Maschinenausfälle im Hintergrund gelöst. Dank dem Programmiermodel können sehr komplexe Algorithmen schneller und effektiver als mit den traditionellen parallelen Methoden entwickelt werden. Ein weiterer positiver Nebeneffekt ist, dass die Entwickler kein Wissen über die parallelisierte Software Entwicklung benötigen und sich ganz auf ihr eigentliches Problem konzentrieren können.

MapReduce Algorithmen sind, wie die Zahlen und Fakten belegen, auch heute stark im Einsatz und werden es weiterhin sein. Sei es in grossen Clustern, Shared Memory Systemen oder NUMA Multiprozessorsystemen.

Die wichtigste Erkenntnis ist die Vereinfachung der Entwicklung von parallelisierter Software durch das sehr schlanke Programmiermodel.

LITERATUR- UND QUELLENVERZEICHNIS

[WHI09] (Tom White März 2009) Hadoop: The Definitive Guide / von Tom White . O'Reilly Media, 2009

[DEA04] (Jeffrey Dean, Sanjay Ghemawat 2004) http://labs.google.com/papers/mapreduce-osdi04.pdf. Google, Inc.

[RAN07] (Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis 2007) http://csl.stanford.edu/~christos/publications/2007.cmp_mapreduce.hpca.pdf. Computer Systems Laboratory, Stanford University.

ANHANG A – GOOGLE BEISPIEL FÜR WORDCOUNTER IN C++

```
#include "mapreduce/mapreduce.h"
// User's map function
class WordCounter : public Mapper {
    public:
    virtual void Map(const MapInput& input) {
        const string& text = input.value();
        const int n = text.size();
        for (int i = 0; i < n; ) {
            // Skip past leading whitespace
            while ((i < n) && isspace(text[i]))
            i++;
            // Find word end
            int start = i;
            while ((i < n) && !isspace(text[i]))
            i++;
            if (start < i)
            Emit(text.substr(start,i-start),"1");
        }
    }
};
REGISTER_MAPPER(WordCounter);
// User's reduce function
class Adder : public Reducer {
    virtual void Reduce(ReduceInput* input) {
        // Iterate over all entries with the
        // same key and add the values
        int64 value = 0;
        while (!input->done()) {
            value += StringToInt(input->value());
            input->NextValue();
        }
        // Emit sum for input->key()
        Emit(IntToString(value));
    }
};
REGISTER_REDUCER(Adder);
int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);
    MapReduceSpecification spec;
    // Store list of input files into "spec"
    for (int i = 1; i < argc; i++) {
        MapReduceInput* input = spec.add_input();
        input->set_format("text");
        input->set_filepattern(argv[i]);
        input->set_mapper_class("WordCounter");
    }
    // Specify the output files:
    // /gfs/test/freq-00000-of-00100
    // /gfs/test/freq-00001-of-00100
    // ...
    MapReduceOutput* out = spec.output();
    out->set_filebase("/gfs/test/freq");
    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Adder");
    // Optional: do partial sums within map
```

```
    // tasks to save network bandwidth
    out->set_combiner_class("Adder");
    // Tuning parameters: use at most 2000
    // machines and 100 MB of memory per task
    spec.set_machines(2000);
    spec.set_map_megabytes(100);
    spec.set_reduce_megabytes(100);
    // Now run it
    MapReduceResult result;
    if (!MapReduce(spec, &result)) abort();
    // Done: 'result' structure contains info
    // about counters, time taken, number of
    // machines used, etc.
    return 0;
}
```