Universität Leipzig
Institut für Informatik
Abteilung Datenbanken

UNIVERSITÄT LEIPZIG

Seminararbeit

Hadoop

Autor: Thomas Findling (Mat.-Nr. 1740842)
Studiengang: Master Informatik (3. Semester)

Betreuer: Lars Kolb
Gutachter: Lars Kolb

Eingereicht am: 31.03.2010

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Googles eigens gesetztes Ziel ist es, alles Wissen der Welt zu erfassen, zu ordnen und gezielt zugreifbar zu machen. Dafür müssen riesige Datenmengen gesammelt und zugreifbar gespeichert werden.

Hierbei ergibt sich jedoch das Problem, dass die Speicherkapazitäten von Plattenspeichern in den letzten Jahrzehnten zwar massiv gestiegen sind, im Gegensatz aber die Zugriffszeiten sich nicht im selben Maße verändert haben. Aus dem Verhältnis dieser beiden Faktoren ergibt sich, dass die Zeit, die benötigt wird, um einen Speicher komplett zu lesen, immer größer wird. [WHITE09, Seite 3]

Um dem Problem der Zugriffszeiten beizukommen, werden heutzutage mehrere Recheneinheiten mit eigenen Plattenspeichern eingesetzt, die zu Clustern zusammengeschlossen werden. Da die Daten verteilt gespeichert werden, ergibt sich daraus eine deutliche Zeitreduzierung bei Zugriffsoperationen. Auf Basis dieses Konzepts betreibt Google Rechenzentren in aller Welt. [WHITE09, Seite 3-4]

Damit solche Rechenzentren jedoch sinnvoll eingesetzt werden können, müssen Mechanismen geschaffen werden, die einen Transfer zwischen ihnen ermöglichen. Eine Rolle nimmt dabei das Google File System (GFS) ein, welches in dieser Arbeit aber nicht behandelt wird [GHEM03, Seite 1]. Ein weiterer wichtiger Faktor ist das von Google entwickelte MapReduce Verfahren, dessen Funktionsweise in der Arbeit „MapReduce - Konzept“ von Thomas König näher betrachtet wird [KOEN10]. Eine Implementation dieses Verfahrens wurde von der Apache Software Foundation im Open-Source-Java-Framework Hadoop realisiert und bildet das Kernthema dieser Arbeit.

Durch den Einsatz dieser Technologien wird der Gedanke des Konzeptes "Cloud Computing" aufgegriffen. Dieses beinhaltet u.A., dass abstrahierte IT-Infrastrukturen, fertige Programmpakete und Programmierumgebungen dynamisch an den Bedarf angepasst über Netzwerk zur Verfügung gestellt werden. [URL009]

1.2 Zielstellung der Arbeit

Zielstellung dieser Arbeit ist es, die Funktions- und Arbeitsweise des Open-Source-Java-Framework Hadoop zu untersuchen und an einem praktischen Beispiel nachzuvollziehen. Dabei wird wird auch auf die Fehlerbehandlung bei Hadoop und auf Ein- bzw. Ausgabetypen eingegangen.

1.3 Aufbau der Arbeit

Diese Seminararbeit teilt sich in zwei Bereiche auf, den theoretischen und den praktischen Teil.

Im theoretischen Teil wird auf die Grundlagen von MapReduce und Hadoop eingegangen. Anschließend folgt eine detaillierte Betrachtung der Funktionsweise von Hadoop im Bezug auf den Ablauf eines MapReduce-Jobs. Im nächsten Kapitel wird auf die Fehlerbehandlung in Hadoop eingegangen. Im letzten Abschnitt des theoretischen Teils werden die von Hadoop unterstützten Ein- und Ausgabetypen untersucht.

Der praktische Teil beschäftigt sich mit der Anwendung des Hadoop-Frameworks auf eine vorgegebene Aufgabensituation. Dieser Teil gliedert sich in die Darstellung der Aufgabensituation, die Umsetzung des konkreten MapReduce-Jobs und die daraus gewonnenen Erkenntnisse bzw. Ergebnisse.

Die Seminararbeit endet mit einer Zusammenfassung des untersuchten Themas, sowie dem Ausblick.

2 Hadoop

2.1 MapReduce

Das Hadoop zugrunde liegende Programmiermodell MapReduce ist besonders für die Verarbeitung und Generierung von großen Datensätzen geeignet. Um große Mengen an gesammelten Daten verarbeiten zu können, wird der Aufwand der dafür benötigen Berechnungen auf mehrere Recheneinheiten verteilt. Das MapReduce-Framework ermöglicht es, solche Berechnungen auf vielen Rechnern parallel auszuführen. Zudem hat MapReduce die Aufgabe, die Teilergebnisse der einzelnen parallelen Verarbeitungsschritte zu aggregieren und die Ergebnisse in das gemeinsame Dateisystem zu schreiben. [WHITE09, Seite 15]

Der Hauptzweck von MapReduce liegt dabei in der Bereitstellung und dem Management der dafür benötigten Berechnungsinfrastruktur. Bei der Entwicklung von verteilten Anwendungen kann so, durch die Nutzung der vom MapReduce-Framework bereitgestellten Mechanismen, der Aufwand für die Implementierung einer geeigneten Parallelisierungslösung reduziert werden. Verteilte Anwendungen können deshalb problemlos auch auf eine hohe Anzahl an Usern bzw. Clients hochskalieren, ohne dass Codeänderungen nötig werden. Zudem kann der MapReduce-Ansatz durch die Nutzung von handelsüblichen Computern realisiert werden, so dass Cluster auch ohne spezielle Server aufgebaut und betrieben werden können. [WHITE09, Seite 27]

Das MapReduce-Konzept basiert auf zwei separat ablaufenden Prozessen: Map und Reduce. Der Map-Vorgang ist hierbei für die Gruppierung der Daten zuständig. Dabei werden aus den Eingabedaten geeignete Key/Value-Paare gebildet. Im Anschluss werden diese zu gruppierten Key/Value-Paaren zusammengefasst, welche aus einem Key und einer Menge von zugeordneten Values bestehen. In der Reduce-Phase werden diese gruppierten Key/Value-Paare verarbeitet, so dass eine neue Liste von Key/Value-Paaren entsteht, bei der jedem Key nur noch eine Value zugeordnet ist. [WHITE09, Seite 18] [DEAN04, Seite 1-2]

2.2 Hadoop Framework

Das Open-Source-Projekt Hadoop der Apache Software Foundation ist eine Implementierung des MapReduce-Konzepts und wird zur Programmierung von MapReduce-Tasks genutzt. Ziel des Frameworks ist die parallele Verarbeitung großer Datenmengen. Dazu werden Computer eingesetzt, die zu Clustern verbunden sind. Ein weiterer Vorteil von Hadoop ist die hohe Fehlertoleranz, auf welche in Kapitel 2.4 gesondert eingegangen wird. [WHITE09, Seite 159] [URL008]

Hadoop ist für Unix und Linux verfügbar und benötigte eine installierte Java Virtual Machine (JVM) in der Version 1.6. Unter Windows kann Hadoop hingegen nur zusammen mit einer Kompatibilitäts-API wie etwa Cygwin eingesetzt werden. Bei der Entwicklung werden Programmierer nicht allein auf Java beschränkt, auch andere Programmiersprachen wie Phython und C++ können mit Hadoop genutzt werden. [URL008] [IX0307]

Um Hadoop sinnvoll einsetzen zu können, müssen sämtliche Daten im gesamten Cluster zur Verfügung stehen. Dies beinhaltet, dass alle Eingabe-Dateien erst in ein gemeinsames Dateisystem kopiert werden müssen. Damit soll erzielt werden, dass sämtliche Daten von allen Knoten aus erreichbar sind. Dies wird auch Shared Nothing Architecture genannt, mit welcher eine logische Shared-Disk-Architektur umgesetzt wird. Hadoop setzt hierfür das Hadoop Distributed File System (HDFS) ein, welches dem Google File System (GFS) nachempfunden wurde [GHEM03, Seite 2]. Hadoop ist jedoch nicht ausschließlich auf das HDFS beschränkt, auch andere Dateisysteme wie beispielsweise CloudStore (ehemals Kosmos) oder Amazon S3 sind möglich. [WHITE09, Seite 48] [URL008]

2.3 Funktionsweise

Dieses Kapitel beschäftigt sich mit dem Ablauf eines MapReduce-Jobs und den dabei ablaufenden internen Prozessen.

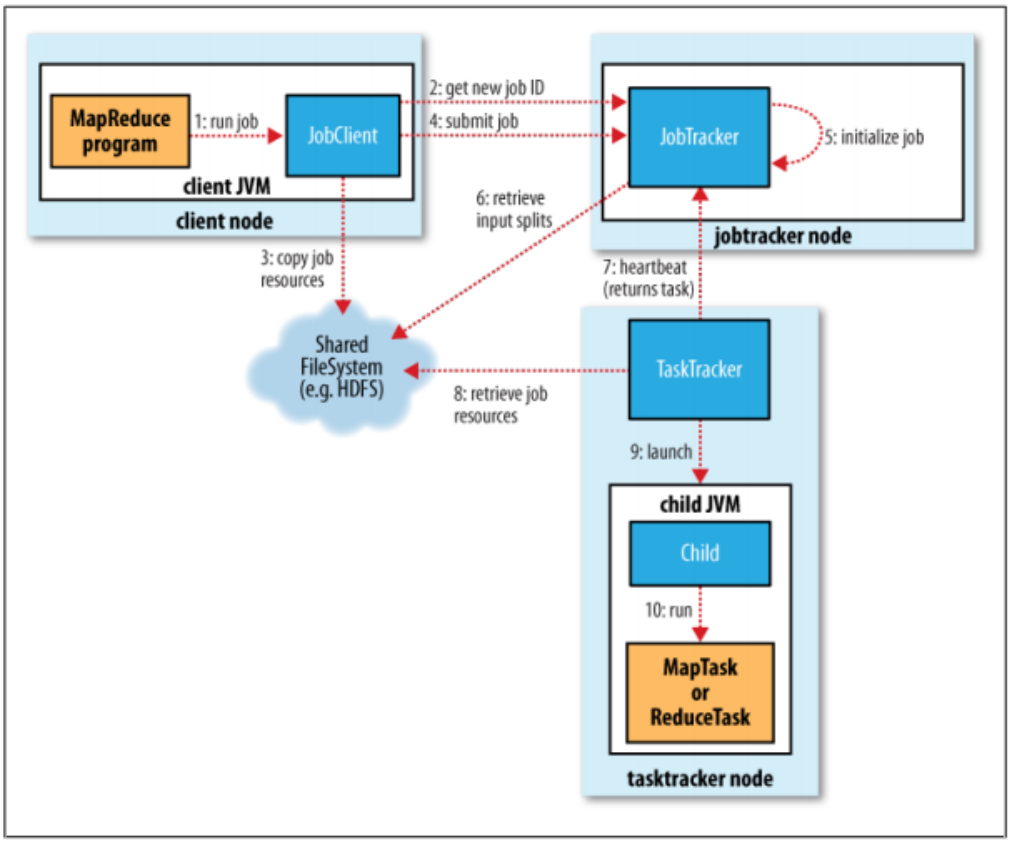


Abbildung 2.1: Ablauf eines MapReduce Jobs [WHITE09, Seite 154]

Bei der Betrachtung von Abbildung 2.1 ergibt sich eine Sicht auf vier zentrale Komponeten [WHITE09, Seite 153]:

- Client: Der Client startet die Jobinitialisierung. Dies beinhaltet das Programm welches Hadoop ausführen soll, sowie Konfigurationsdaten und Parameter.

- Jobtracker: Der Jobtracker übernimmt die Koordinierung der einzelnen Jobs. Im Rahmen dieser Aufgabe ist er auch dafür zuständig einen Job in einzelne Tasks zu unterteilen.

- Tasktracker: Ein Tasktracker führt einzelne Map- oder Reduce-Tasks aus. Hierbei sei zu Beachten, dass sich die Taskanzahl nach den verfügbaren Prozessorkernen und dem Hauptspeicher richtet. Normalerweise wird Hadoop

auf einen Task je Prozessorkern beschränkt, unter Umständen kann aber auch höhere Anzahl an Threads sinnvoll sein.

- Dateisystem: Das gemeinsame Dateisystem stellt die Job-Ressourcen bereit. Dabei werden die Daten in Blöcke gesplittet und redundant gespeichert.

2.3.1 Job-Vorbereitung

Beim Aufruf der Methode `runJob` wird eine neue `JobClient`-Instanz auf dem Client erzeugt (Abbildung 2.1, Schritt 1). Anschließend fragt der `JobClient` mit der Methode `getNewJobId` beim Jobtracker eine neue Job-ID an (Abbildung 2.1, Schritt 2). Daraufhin kopiert der `JobClient` alle benötigten Job-Daten in das gemeinsame Dateisystem. Dies umfasst das auszuführende Programm und die dafür nötigen Bibliotheken, sowie individuelle Konfigurationsdaten (Abbildung 2.1, Schritt 3). Nachdem dieser Prozess abgeschlossen ist, wird der Job durch den Aufruf von `submitJob` an den Jobtracker übergeben. Dabei werden die festgelegten Spezifikationen für Input und Output überprüft. Falls diese nicht den Vorgaben entsprechen wird der Job abgebrochen und ein entsprechender Fehler zurückgegeben (Abbildung 2.1, Schritt 4). [WHITE09, Seite 153-155]

2.3.2 Job-Initialisierung

Nach dem Aufruf von `submitJob` muss der Job auf dem Jobtracker initialisiert werden. Der Jobtracker verwaltet dabei alle übergebenen Jobs mit einer Job-Warteschlange. Der Jobtracker ist ebenfalls für die Einteilung eines Jobs in mehrere Tasks zuständig, welche jeweils eine eigene Task-ID erhalten. Die eigentliche Initialisierung beinhaltet die Instanzierung eines Job-Objekts, dem nachfolgend die einzelnen Tasks eines Jobs sowie deren Status- und Verlaufsmeldungen zugeordnet sind (Abbildung 2.1, Schritt 5). [WHITE09, Seite 155]

Anschließend greift der Jobtracker auf die vom `JobClient` in den Konfigurationsdaten festgelegten Input Splits zu und übergibt sie dem jeweiligen Job. Input Splits sind logische Referenzen auf einen Block im gemeinsamen Dateisystem und repräsentieren die vom Job zu verarbeitenden Daten. Jedem (Map-)Task wird dabei genau ein Input Split zugeordnet (Abbildung 2.1, Schritt 6). [WHITE09, Seite 155]

2.3.3 Task-Zuordnung

Die einzelne Tasks werden den Tasktrackern gemäß dem Job Scheduling zugeteilt. Die einzelnen Tasktracker kommunizieren über periodische gesendete "heartbeats" mit dem Job Tracker. Der sogenannte „heartbeat“ ist dafür zuständig Statusmeldungen an den Jobtracker zu melden (Abbildung 2.2). Dies umfasst beispielsweise den aktuellen Fortschritt der Task-Bearbeitung, ob der Tasktracker überhaupt zur Verfügung steht, aber auch jede andere Art von Nachrichten. Wenn ein Tasktracker dem Jobtracker meldet, dass er zur Verfügung steht kann ihm ein neuer Task zugeteilt werden. Dabei gilt zu beachten, dass Tasktracker nur eine begrenzte Anzahl an Tasks gleichzeitig bearbeiten können. Standardmäßig gilt, dass nur ein Task pro Prozessorkern erlaubt ist, jedoch kann es unter Umständen auch sinnvoll sein eine höhere Anzahl an Threads zu definieren. Die Anzahl der einzelnen Map- und Reducetasks lässt sich in Hadoop unabhängig von der Anzahl der Prozessorkerne konfigurieren (Abbildung 2.1, Schritt 7). [WHITE09, Seite 155-156]

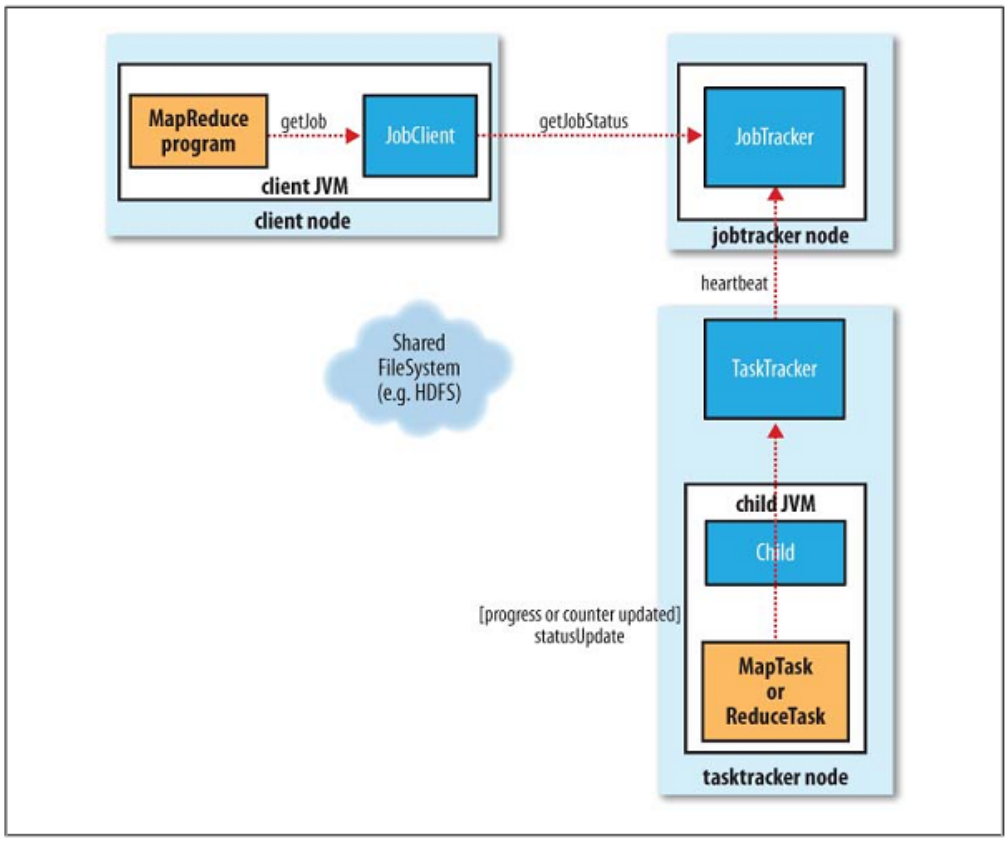


Abbildung 2.2: Kommunikation zwischen den Komponenten [WHITE09, Seite 159]

2.3.4 Task-Ausführung

Wenn einem Tasktracker ein Task zugeordnet wurde, werden zuerst die für den Task benötigten Daten und Bibliotheken, sowie das eigentliche Programm aus dem gemeinsamen Dateisystem abgeholt. Hierbei gilt, dass Tasks nur lokal vorhandene Input Splits bearbeiten und jeder (Map-)Task nur einen Input Split verarbeitet. Die aus dem gemeinsamen Dateisystem überführten Dateien werden in einem lokalen Arbeitsverzeichnis gespeichert (Abbildung 2.1, Schritt 8). [WHITE09, Seite 156]

Daraufhin startet die `TaskRunner` Instanz eine neue Java Virtual Machine (JVM) zur Ausführung eines Tasks. Für jeden Task wird dabei eine eigene JVM gestartet (Abbildung 2.1, Schritt 9). [WHITE09, Seite 156]

Im Anschluss wird der eigentliche Task ausgeführt. Die JVM des Child-Prozesses kommuniziert dabei periodisch mit dem Parent-Tasktracker und tauscht Statusmeldungen aus (Abbildung 2.2). Ein Task wird anschließende als „finished“ gemeldet, falls der Task erfolgreich beendet wurde, oder als „failed“ gemeldet, falls der Task abgebrochen werden musste. Sobald alle Tasks eines Jobs erfolgreich beendet wurden, wird der Job vom Jobtracker als „successful“ gemeldet (Abbildung 2.1, Schritt 10). [WHITE09, Seite 156]

2.4 Fehlerbehandlung

In diesem Kapitel soll es um die Fehlerbehandlung innerhalb von Hadoop gehen. Um die verschiedenen Fehler besser nachvollziehen zu können wird an dieser Stelle noch einmal auf die verschiedenen Komponenten am Beispiel eines einfachen Master/Slave-Clusters eingegangen (Abbildung 2.3).

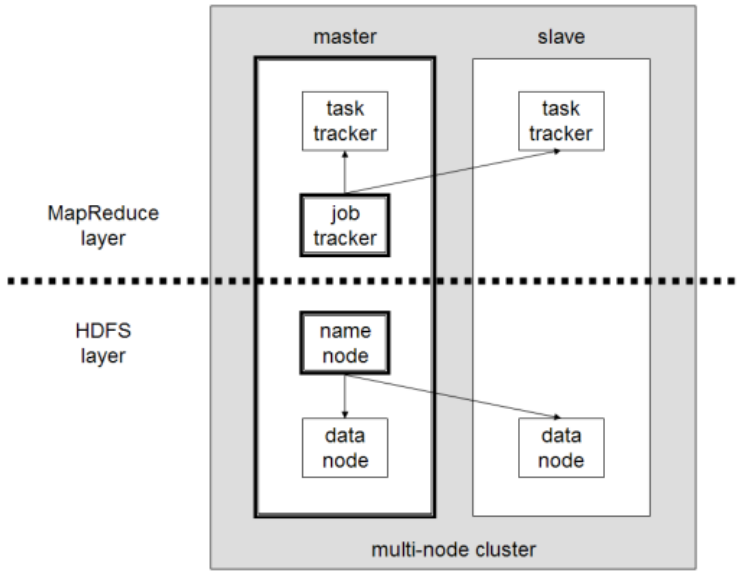


Abbildung 2.3: Multi-Node-Cluster [URL007]

In der MapReduce-Schicht befinden sich die Tasktracker. Sie haben die Aufgabe die einzelnen Tasks auszuführen. Dabei kann es beliebig viele Tasktracker geben. Hier unterscheidet man zwischen zwei Fehlerarten – dem Taskfehler und dem Tasktrackerfehler. [WHITE09, Seite 159-161]

Ein Taskfehler tritt auf, wenn ein Task wegen eines Fehlers abgebrochen wurde oder bei seiner Ausführung ein Timeout überschreitet. Standardmäßig ist in Hadoop das Tasktimeout auf 10 Minuten beschränkt. Im Falle eines Taskfehlers wird versucht den Task neu zu vergeben. Dabei werden andere Tasktracker gegenüber dem Tasktracker bei dem der Taskfehler aufgetreten ist bevorzugt. Dieser Vorgang wird jedoch von Hadoop begrenzt, sobald die Zahl der Taskneustarts ein Limit erreicht gilt der Job als gescheitert. Standardmäßig tritt dies in Hadoop nach vier Fehlversuchen auf. Ob wirklich der ganze Job als gescheitert angesehen wird kann ebenfalls konfiguriert werden. [WHITE09, Seite 159-161]

Ein Tasktrackerfehler tritt auf wenn ein Tasktracker abstürzt oder ein Timeout verursacht. In diesem Fall wird der betroffene Tasktracker aus dem Tasktracker-Pool entfernt. Ein Tasktracker kann ebenfalls aus dem Pool entfernt werden, wenn er zu viele Taskfehler verursacht – dies wird als Blacklisting bezeichnet. Auch kann ein Tasktracker entfernt werden, wenn er bei der Ausführung von Tasks durchgehend signifikant langsamer ist als andere Tasktracker – dieser Umstand wird Slowdown genannt. Wenn ein Tasktracker einen Tasktrackerfehler verursacht werden die einzelnen Tasks des unvollständigen Jobs auf andere Tasktracker verteilt. [WHITE09, Seite 161]

In der MapReduce-Schicht befindet sich ebenfalls der sogenannte Jobtracker. Der Jobtracker ist für die Verwaltung der Jobs zuständig und innerhalb von Hadoop nur einmal vorhanden. Ein Jobtrackerfehler tritt auf wenn der Jobtracker abstürzt oder ein Timeout verursacht. Dabei handelt es sich um einen schwerwiegenden Fehler, da durch die Einzigartigkeit des Jobtrackers auf keine zweite Instanz ausgewichen werden kann. Nach [WHITE09, Seite 161] gibt es für diesen Umstand noch keine Lösungsmöglichkeit.

In der Schicht des Dateisystems befindet sich das Name-Node und die Data-Nodes. Das Name-Node verwaltet dabei das Dateisystem und die Zugriffe darauf. Das Name-Node ist, wie auch das Jobtracker-Node, nur einmal vorhanden und verursacht dementsprechend ähnliche Probleme. Das Data-Node enthält die einzelnen Datenblöcke mit den zu verarbeitenden Daten und erlaubt somit einen direkten Zugriff auf die Ressourcen. Es können beliebig viele Data-Nodes existieren. Ein fehlerhaftes Data-Node schlägt sich innerhalb der Fehlerbehandlung in einem Taskfehler nieder, führt aber aufgrund der redundanten Speicherung von Daten nicht automatisch zu einem Tasktrackerfehler bzw. Job-Abbruch. [URL006]

2.5 Ein- und Ausgabetypen

In diesem Abschnitt wird auf die Formatierung der Ein- und Ausgabe innerhalb von Hadoop eingegangen. Um sich dieser Thematik zu nähern wird in Kapitel 2.5.1 noch einmal die zugrunde liegende Datenstruktur betrachtet.

2.5.1 Input Splits

Die Daten, welche die einzelnen Map- und Reducetasks verarbeiten sollen, liegen redundant gespeichert als Block im gemeinsamen Dateisystem vor. Auf diese wird mittels sogenannter Input Splits zugegriffen, welche logische Referenzen auf diese Daten darstellen. Ein Block enthält in Hadoop standardmässig 64 Megabyte Daten. Die Blockgröße kann jedoch manuell konfiguriert werden. [WHITE09, Seite 184-186]

```
public interface InputSplit extends Writable {
    long getLength() throws IOException;

    String[] getLocations() throws IOException;
}
```

Listing 2.1: Java Interface von InputSplit [WHITE09, Seite 185]

Jeder Input Split hat dabei optional Informationen über seine Lage im gemeinsamen Dateisystem bzw. darüber welche Knoten den größten Abschnitt eines Blocks verwalten. Damit Hadoop die Daten verarbeiten kann werden die Methoden `getRecordReader` und `getSplits` der Klasse `InputFormat` eingesetzt. Dabei erzeugt der `RecordReader` aus den Daten Key/Value-Paare, sogenannte Records. [WHITE09, Seite 184-186]

```
public interface InputFormat<K, V> {
    InputSplit[] getSplits(JobConf job, int numSplits) throws IOException;

    RecordReader<K, V> getRecordReader(InputSplit split, JobConf job,
                                       Reporter reporter) throws IOException;
}
```

Listing 2.2: Java Interface von InputFormat [WHITE09, Seite 185]

Die vorgegebenen Eingabetypen bestimmen dabei welche Werte diese Paare annehmen können und damit auch die Struktur der einzelnen Records. Die eingelesenen Records werden als `InputFormat` verarbeitet (Abbildung 2.4) und

anschließend an den Mapper übergeben. Ein Input Split wird dabei als Chunk seperat und unabhängig von anderen Input Splits von einem Mapper parallel bearbeitet [WHITE09, Seite 184-186]. Die einzelnen Vorgänge bei der Bildung von Key/Values-Paaren werden in [KOEN10] noch einmal genauer beschrieben.

```
K key = reader.createKey();
V value = reader.createValue();
while (reader.next(key, value)) {
    mapper.map(key, value, output, reporter);
}
```

Listing 2.3: Quellcode Mapper-Aufruf [WHITE09, Seite 186]

Ein- und Ausgabetypen eines MapReduce-Jobs können sich prinzipiell unterscheiden, jedoch muss zumindest der Ausgabetyp des Map-Vorgangs immer mit dem Eingabetyp des Reduce-Vorgangs übereinstimmen, damit ein Job ausgeführt werden kann. Zur Ausgabe und Weiterverarbeitung der Ergebnisse wird das `OutputFormat` verwendet (Abbildung 2.5). [WHITE09, Seite 184-186]

2.5.2 Textdaten

Das `TextInputFormat` wird zur Verarbeitung von unstrukturiertem Text genutzt und ist das Standard-Eingabeformat von Hadoop. Ein Record, welcher vom `RecordReader` gebildet wird, setzt sich dabei aus dem Inhalt einer Zeile (Value) und ihrer relativen Position zu den restlichen Daten zusammen (Key) und wird als Text-Objekt abgelegt. Der Key wird entweder durch eine Zeilennummer bzw. -identifier (`KeyValueTextInputFormat`) oder das Byteoffset des jeweiligen Zeilenbeginns repräsentiert (`TextInputFormat`). [WHITE09, Seite 196-199]

Anschließend wird jeder Record an einen Mapper übergeben, außer bei Nutzung des `NLineInputFormat`, welches auch die Verarbeitung mehrerer Zeilen bzw. Records durch einen Mapper erlaubt. Ein Sonderfall für die Nutzung von `TextInput` bildet XML, hier werden Records nicht zeilenweise eingelesen sondern mit Hilfe des `StreamXmlRecordReader` an die start- und end-Tags der einzelnen XML-Elemente angepasst. [WHITE09, Seite 199]

Das Standardausgabeformat von Hadoop ist `TextOutputFormat`. Es nutzt die Methode `toString` um Records in Strings umzuwandeln. Die Ausgabe der Strings

erfolgt dabei zeilenweise. Key und Value der einzelnen Records werden durch Tabs oder andere konfigurierbare Trennzeichen voneinander abgegrenzt. Durch die Nutzung von `NullOutputFormat` kann eine Ausgabe auch unterdrückt bzw. durch die Nutzung des Typs `NullWritable` auf eine Komponente des Records begrenzt werden. [WHITE09, Seite 202-203]

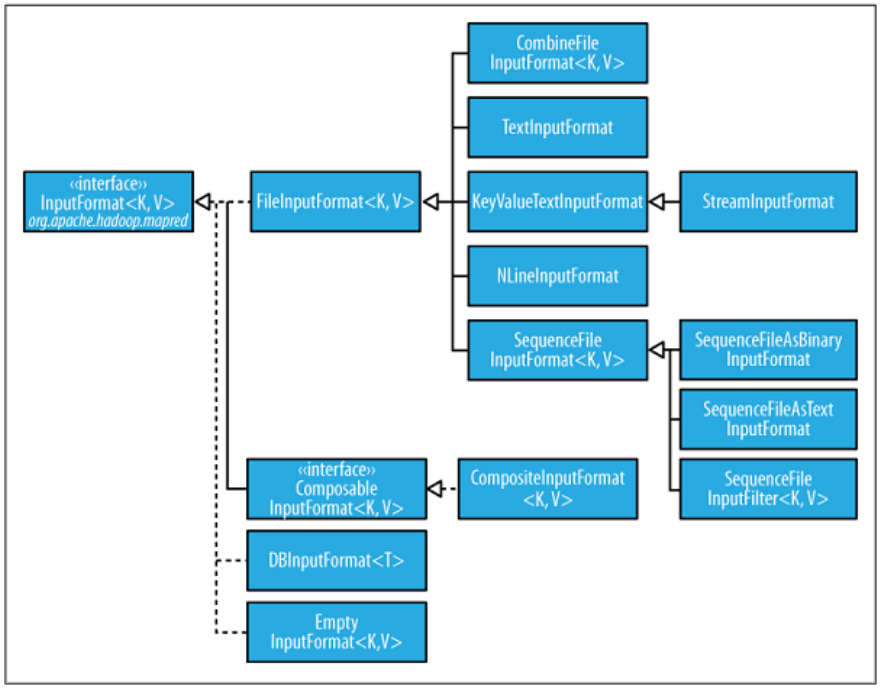


Abbildung 2.4: Klassenstruktur von `InputFormat` [WHITE09, Seite 187]

2.5.3 Binärdaten

Zur Verarbeitung von Binärdaten wird das `SequenceFileInputFormat` genutzt. Hadoop speichert dabei alle Daten als Sequenzen von Key/Value-Paaren. Zusätzlich können solche Eingabedaten auch in komprimierter Form abgelegt werden. Die Sequenzen können entweder mit Hilfe des `SequenceFileAsTextInputFormat` in Text-Objekte konvertiert werden, oder werden unter Verwendung des `SequenceFileAsBinaryInputFormat` als Binär-Objekte interpretiert. [WHITE09, Seite 199-200]

Binärdaten können unter Nutzung des `SequenceFileOutputFormat` als Sequenzen ausgegeben werden. Hierbei ist auch eine Kompression der Daten möglich, welche durch statische Methoden des `SequenceFileOutputFormats` definiert wird. Durch die Verwendung des `SequenceFileAsBinaryOutputFormat`

können Daten auch direkt als Binär-Objekte in einen `SequenceFile`-Container ausgegeben werden. Das Format `MapFileOutputFormat` dient der sortierten Ausgabe von Daten in ein `MapFile`. [WHITE09, Seite 203]

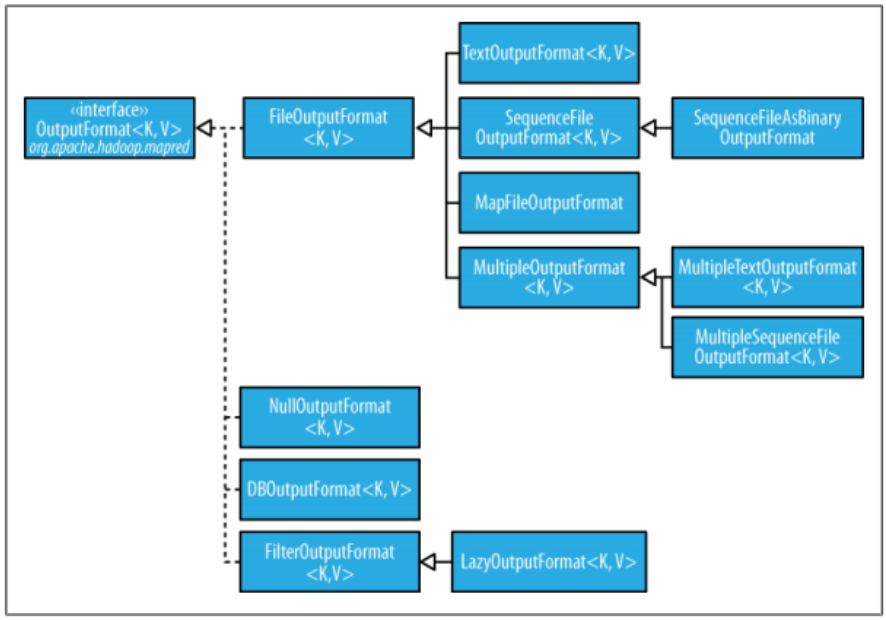


Abbildung 2.5: Klassenstruktur von `OutputFormat` [WHITE09, Seite 202]

2.5.4 Datenbank-Zugriff

Zum Ansprechen relationaler Datenbanken wird das `DBInputFormat` genutzt. Der Zugriff erfolgt dabei über die Datenbankschnittstelle Java Database Connectivity (JDBC). `DBInputFormat` unterstützt MySQL, PostgreSQL und einige andere Datenbanksysteme, welche über einen JDBC-Treiber zusammen mit Hadoop genutzt werden können. Die Datenbank kann mit Hilfe von `configureDB` konfiguriert werden und `SELECT`-Queries werden mit der Hadoop-Methode `setInput` gegen die Datenbank ausgeführt. Ein Chunk ist dabei die Menge aller Zeilen einer Tabelle, einer View oder eines `SELECT`-Statement, so dass diese einzelnen Zeilen als Record verarbeitet werden können. Um das Ergebnis eines MapReduce-Jobs in eine Datenbank zu schreiben wird das `DBOutputFormat` genutzt. [WHITE09, Seite 201-202] [URL001]

2.5.5 Nutzung mehrerer Formate

Da ein MapReduce-Job auch mehrere Dateien verarbeiten kann, ist es möglich, dass die Daten in den einzelnen Dateien in jeweils verschiedenen Formatierungen

vorliegen. Dieser Fall liegt meist vor, wenn ähnliche Daten aus verschiedenen Quellen stammen oder wenn die Daten über einen längeren Zeitraum erhoben wurden und sich dabei das genutzte Speicherformat zwischenzeitlich geändert hat. Um diese Daten korrekt verarbeiten zu können, bietet Hadoop die Möglichkeit im Rahmen eines MapReduce-Jobs durch Nutzung von `MultipleInputs` für jedes der sich unterscheidenden Input-Formate einen eigenen Mapper zu definieren. Der Map-Output muss jedoch bei allen genutzten Mappern übereinstimmen, damit ein Job mit dem Reduce-Vorgang fortgesetzt werden kann. [WHITE09, Seite 200-201]

Nachdem ein MapReduce-Job abgeschlossen wurde können mittels `MultipleOutputs` mehrere Output-Formate festgelegt werden. Durch die Nutzung von `MultipleOutputFormat` können die Ergebnisse auch in mehrere Output-Dateien ausgegeben werden. [WHITE09, Seite 203-210]

2.5.6 LazyOutput

Bei `LazyOutput` handelt es sich um einen Output-Wrapper, das heißt jedes Output-Format kann in Verbindung mit `LazyOutput` genutzt werden. Normalerweise bewirkt das `FileOutputFormat`, dass bei einem MapReduce-Job immer eine Ausgabedatei erzeugt wird, selbst wenn keine Records erzeugt werden. `LazyOutput` stellt sicher, dass nur dann eine Ausgabedatei erzeugt wird, wenn das Ergebnis des MapReduce-Jobs aus mindestens einem Record besteht. [WHITE09, Seite 210]

3 Praxisbeispiel

In diesem Kapitel wird die Funktionsweise von Hadoop bzw. der Ablauf eines MapReduce-Jobs an einem praktischen Beispiel demonstriert.

3.1 Aufgabe

In dieser Beispiel-Implementation soll es darum gehen einen Job auszuführen, welcher aus einer gegebenen Menge an Preisvergleichsdaten für jedes Produkt den günstigsten Preis bestimmt und anschließend ausgibt.

Als Basis dienen ca. 3,5 Millionen Datensätze mit Produktdaten eines Online-Preisvergleichportales, welche der Abteilung Datenbanken der Universität Leipzig zur Verfügung gestellt wurden. Es handelt sich dabei um 3,8 Gigabyte semi-strukturierte Textdaten, welche in 44 Dateien nach Kategorien gespeichert sind. Die einzelnen Datensätze bestehen dabei aus Attributen wie etwa Produktname, Preis, Händler oder Zustand, welche jeweils durch Tabs getrennt vorliegen.

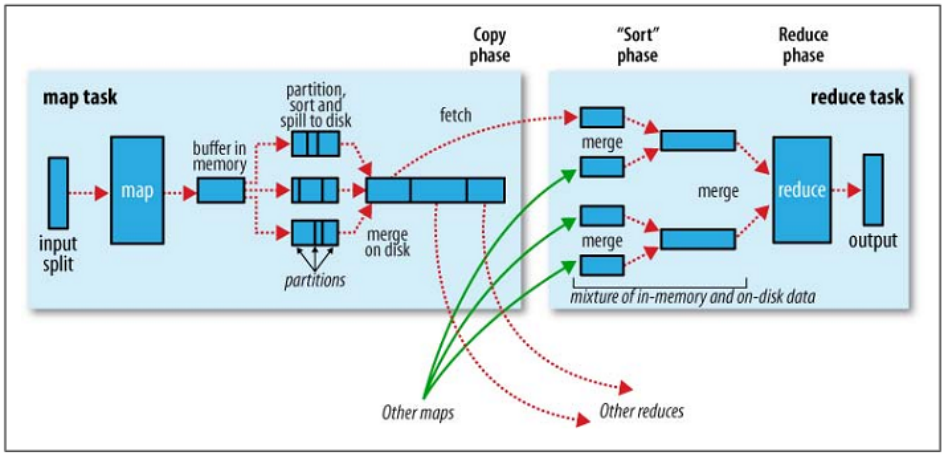


Abbildung 3.1: Ablauf eines MapReduce-Jobs [WHITE09, Seite 163]

3.2 Umsetzung

Zunächst muss der Job konfiguriert werden. Dabei wird ein Job-Name vergeben, sowie jeweils der Pfad für die zu verarbeitenden Daten, welche schon vorher ins gemeinsame Dateisystem geladen wurden, und für die Ausgabedaten festgelegt. Anschließend werden die Klassen des Mappers bzw. Reducers definiert und das Speicherformat der auszugebenden Records wird vorgegeben. Nach Abschluss der Konfiguration wird der Job ausgeführt.

```
public class MinPrice
{
    public static void main(String[] args) throws IOException
    {
        JobConf conf = new JobConf(MinPrice.class);
        conf.setJobName("Min Price");

        Path input = new Path("hdfs://master:54310/home/hadoop/input/")
        FileInputFormat.setInputPaths(conf, input);

        Path output = new Path("hdfs://master:54310/home/hadoop/output/")
        FileOutputFormat.setOutputPath(conf, output);

        conf.setMapperClass(MinPriceMapper.class);
        conf.setReducerClass(MinPriceReducer.class);

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);

        JobClient.runJob(conf);
    }
}
```

Listing 3.1: MapReduce-Beispiel: Quellcode

In der Map-Phase verarbeiten die einzelnen Tasks die Input Splits, welche in diesem Beispiel durch die Preisvergleichsdaten repräsentiert werden. Der Mapper fungiert dabei als Parser und ermittelt die nötigen Wertepaare für die einzelnen Records. Konkret werden die Daten zeilenweise abgearbeitet und für jede Zeile die Attribute Produktname und Preis ermittelt. Im Rahmen der Key/Value-Zuordnung wird anschließend jedem Produktnamen ein Preis zugeordnet und als Record abgelegt.

```
public class MinPriceMapper extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable>
{
    public void map(LongWritable key, Text value, OutputCollector<Text,
IntWritable> output, Reporter reporter) throws IOException
    {
        String line  = value.toString();
        String title = gettitle(line);
        int    price = getprice(line);

        try
        {
            output.collect(new Text(title), new IntWritable(price));
        }
        catch (Exception ex)
        {
            System.out.println("Exception: " + line);
        }
    }
}
```

Listing 3.2: MapReduce-Beispiel: Quellcode des Mappers

Im Rahmen der Map-Phase wurde nun für jeden Datensatz der Preisvergleichsdaten ein Record bestehend aus Produktname und Preis angelegt. Dabei können einem Produktnamen-Key mehrere Preis-Values zugeordnet sein. Aufgabe des Reducers ist es nun die Key/Value-Zuordnung auf eine Value pro Key zu reduzieren. Im Rahmen der Reduce-Phase wird nun also für jeden Produktnamen der kleinste Preis ermittelt und anschließend als Ergebnis ausgegeben.

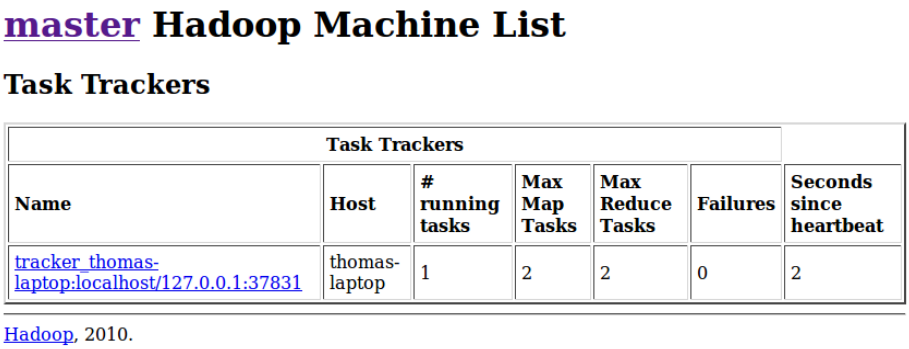
```
public class MinPriceReducer extends MapReduceBase implements
Reducer<Text, IntWritable, Text, IntWritable>
{
    public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter) throws
IOException
    {
        int minValue = Integer.MAX_VALUE;

        while (values.hasNext())
        {
            minValue = Math.min(minValue, values.next().get());
        }

        output.collect(key, new IntWritable(minValue));
    }
}
```

Listing 3.3: MapReduce-Beispiel: Quellcode des Reducers

Der Verlauf des Jobs kann über ein Webinterface verfolgt werden. Man erhält Einsicht in das Dateisystem, die Statusmeldungen der Tasks (Abbildung 3.3) bzw. Tasktracker (Abbildung 3.2), sowie eine Übersicht über das Name-Node.



master Hadoop Machine List

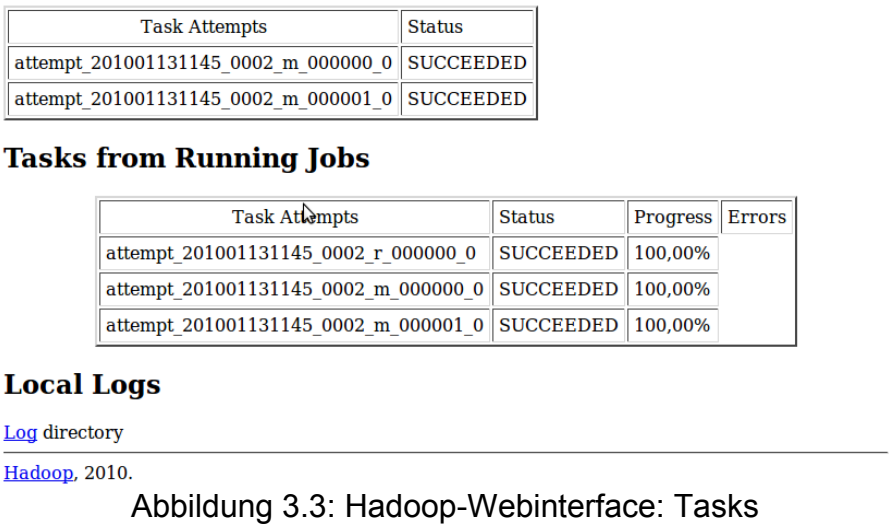
Task Trackers

Task Trackers						
Name	Host	# running tasks	Max Map Tasks	Max Reduce Tasks	Failures	Seconds since heartbeat
tracker_thomas-laptop:localhost/127.0.0.1:37831	thomas-laptop	1	2	2	0	2

Hadoop, 2010.

Abbildung 3.2: Hadoop-Webinterface: Tasktracker

Zusätzlich können Logs Aufschluss über den Verlauf eines Tasks liefern. Diese können nach Abschluss eines Tasks oder auch während dieser noch läuft eingesehen werden und liefern mögliche Hinweise auf Fehlerquellen.



Task Attempts	Status
attempt_201001131145_0002_m_000000_0	SUCCEEDED
attempt_201001131145_0002_m_000001_0	SUCCEEDED

Tasks from Running Jobs

Task Attempts	Status	Progress	Errors
attempt_201001131145_0002_r_000000_0	SUCCEEDED	100,00%	
attempt_201001131145_0002_m_000000_0	SUCCEEDED	100,00%	
attempt_201001131145_0002_m_000001_0	SUCCEEDED	100,00%	

Local Logs

Log directory

Hadoop, 2010.

Abbildung 3.3: Hadoop-Webinterface: Tasks

3.3 Ergebnisse

Bei diesem MapReduce-Job zur Bestimmung des günstigsten Preises eines Produktes wurden die vorliegenden 3,5 Millionen Datensätze auf 2,5 Millionen Ergebnis-Records reduziert. Die geringe Datensatz-Reduktion ist damit zu erklären, dass der Produktname aufgrund von Modulationen bei der Namenszusammensetzung nicht als eindeutiger Bezeichner geeignet ist.

Es folgt nun ein Vergleich bezüglich der Ausführungszeit auf verschiedenen Testumgebungen.

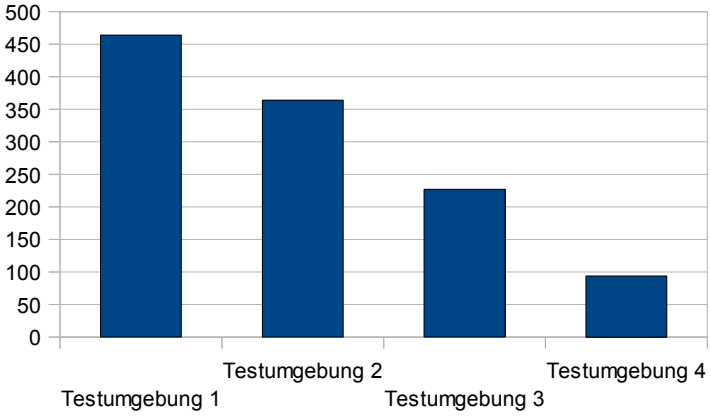


Abbildung 3.4: Ausführungszeit in Sekunden

- Testumgebung 1
  - Hardware: Notebook mit 1 x 2,0 GHz (Single Core CPU)
  - Ergebnis: Ausführungszeit von 464 Sekunden (7:42 Minuten)
- Testumgebung 2
  - Hardware: Notebook mit 2 x 2,0 GHz (Dual Core CPU)
  - Ergebnis: Ausführungszeit von 364 Sekunden (6:04 Minuten)
- Testumgebung 3
  - Hardware: Cluster aus Testumgebung 1 und Testumgebung 2
  - Ergebnis: Ausführungszeit von 227 Sekunden (3:47 Minuten)

- Testumgebung 4
    - Hardware: Cluster aus Rechnern mit 8 x 2,66 GHz und 2 x 2,66 GHz
    - Ergebnis: Ausführungszeit von 94 Sekunden (1:34 Minuten)

Wie aus den Ergebnissen zu entnehmen ist, konnte keine lineare Skalierung erreicht werden. Dies ist damit zu erklären, dass die verwendeten Einheiten nicht in vergleichbaren Leistungsklassen liegen, so verhält sich bei Betrachtung aller Testumgebungen etwa die Anzahl der CPUs nicht proportional zur Anzahl der Plattenspeicher. Jedoch kann trotzdem eine deutliche Reduzierung bei der Ausführungszeit des Jobs festgestellt werden.

4 Zusammenfassung und Ausblick

Das MapReduce-Verfahren dient der parallelen Verarbeitung großer Datenmengen und erzielt so eine deutliche Zeitreduzierung bei Zugriffsoperationen. Das eingesetzte MapReduce-Framework ist dabei für das Management der eingesetzten Berechnungsinfrastruktur zuständig, so dass bei der Entwicklung von verteilten Anwendungen der Aufwand für die Parallelisierung fast komplett ausgeblendet werden kann. Das MapReduce-Konzept basiert auf zwei separat ablaufenden Prozessen: Map und Reduce. Der Map-Vorgang ist hierbei für die Gruppierung und Zusammenfassung der Daten als Key/Value-Paare zuständig. In der Reducephase werden diese gruppierten Key/Value-Paare verarbeitet, so dass eine neue Liste von Key/Value-Paaren entsteht, bei der jedem Key nur noch eine Value zugeordnet ist.

Hadoop ist eine Implementierung des MapReduce-Konzeptes und ist für Unix/Linux verfügbar. Hadoop nutzt das Hadoop Distributed File System (HDFS) als gemeinsames Dateisystem, kann aber auch mit anderen Dateisystemen zusammenarbeiten. Die Ausführung eines MapReduce-Jobs in Hadoop beinhaltet vier zentrale Komponenten: Client, Jobtracker, Tasktracker und Dateisystem. Der Client ist dabei für die Jobinitialisierung zuständig und beinhaltet das Programm, welches Hadoop ausführen soll, sowie Konfigurationsdaten und Parameter. Der Jobtracker übernimmt die Koordinierung der einzelnen Jobs. Im Rahmen dieser Aufgabe ist er auch dafür zuständig einen Job in einzelne Tasks zu unterteilen. Ein Tasktracker führt einzelne Map- oder Reduce-Tasks aus. Das gemeinsame Dateisystem stellt die Job-Ressourcen bereit. Dabei werden die Daten in Blöcke gesplittet und redundant gespeichert. Im Bezug auf die Fehlerbehandlung unterscheidet man beim Tasktracker in Taskfehler und Tasktrackerfehler. Ein Taskfehler tritt auf wenn ein Task wegen eines Fehlers abgebrochen wurde oder bei seiner Ausführung ein Timeout überschreitet. Im Falle eines Taskfehlers wird versucht den Task neu zu vergeben, dabei werden andere Tasktracker gegenüber dem Tasktracker bei dem der Taskfehler aufgetreten ist bevorzugt. Ein Tasktrackerfehler tritt auf wenn ein Tasktracker abstürzt, ein Timeout verursacht, zu viele Taskfehler verursacht (Blacklisting) oder eine signifikant langsamere Taskausführung hat (Slowdown). In diesem Fall wird der betroffene Tasktracker aus

dem Tasktracker-Pool entfernt und die einzelnen Tasks des unvollständigen Jobs werden auf andere Tasktracker verteilt. Jobtrackerfehler können hingegen nicht behoben werden, da nur ein Jobtracker existiert und somit keine Ausweichmöglichkeit vorhanden ist.

Damit Hadoop die vorliegenden Daten verarbeiten kann werden Ein- und Ausgabetypen benötigt. Die Daten liegen redundant gespeichert als Block im gemeinsamen Dateisystem vor. Auf diese wird mittels sogenannter Input Splits zugegriffen, welche logische Referenzen auf diese Daten darstellen. Damit Hadoop diese Daten verarbeiten kann werden Key/Value-Paare erzeugt, sogenannte Records. Die Ein- und Ausgabetypen bestimmen dabei welche Werte diese Paare annehmen können. Man unterscheidet bei den Ein- und Ausgabetypen zwischen Textdaten, Binärdaten und Datenbanken. Textdaten werden mit dem TextInputFormat verarbeitet. Ein Record setzt sich dabei aus dem Inhalt einer Zeile und ihrer relativen Position zu den restlichen Daten zusammen. Das SequenceFileInputFormat wird für Binärdaten verwendet. Die Daten werden dabei als Sequenzen von Key/Value-Paaren verarbeitet und können auch komprimiert vorliegen. DBInputFormat ermöglicht den Zugriff auf relationale Datenbanken. Dies ist in Hadoop über die Datenbankschnittstelle Java Database Connectivity (JDBC) möglich. Damit ein Job auch verschiedene Formate verarbeiten kann ist es möglich mehrere Mapper zu definieren. Bei der Ausgabe können auch unterschiedliche Formate und Ausgabedateien angesprochen werden. Das LazyOutput ist ein Output-Wrapper und stellt sicher, dass nur dann eine Ausgabedatei erzeugt wird, wenn das Ergebnis des MapReduce-Jobs aus mindestens einem Record besteht.

Hadoop wird aufgrund seiner Leistungsfähigkeit inzwischen schon von ca. 100 Firmen oder Institutionen eingesetzt und etabliert sich immer mehr [URL002]. So hat Hadoop etwa in den Jahren 2008 und 2009 den Terabyte Sort Benchmark gewonnen. Es ist damit aktuell die schnellste Implementierung um sehr große Datenmengen verteilt zu sortieren und somit das erste Java- bzw. Open-Source-Programm welches einen Performancebenchmark gewonnen hat [URL003] [URL004]. Ob Hadoop hingegen auch weiterhin frei eingesetzt werden kann ist noch ungeklärt, da die Google Cooperation im Januar 2010 das Patent für die Map/Reduce-Technik zugesprochen bekommen hat [URL005].

5 Literaturverzeichnis

5.1 Literatur

WHITE09 Tom White: Hadoop – The Definite Guide, O'Reilly Media, 2009

KOEN10 Thomas König: MapReduce – Konzept, Universität Leipzig Abteilung Datenbanken, 2010

IX0307 Michael Nebel: Faltplan, iX 3/2007 Seite 84-86, Heise Zeitschriften Verlag, 2007

GHEM03 Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: The Google File System, Google Inc., 2003

DEAN04 Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters, Google Inc., 2004

5.2 Web

URL001 http://www.cloudera.com/blog/2009/03/database-access-with-hadoop/ (abgerufen 13.02.2010)

URL002 http://wiki.apache.org/hadoop/PoweredBy (abgerufen 13.02.2010)

URL003 http://sortbenchmark.org/ (abgerufen 13.02.2010)

URL004 http://developer.yahoo.net/blogs/hadoop/2008/07/apache_hadoop_wins_terabyte_sort_benchmark.html (abgerufen 13.02.2010)

URL005 http://www.heise.de/newsticker/meldung/Google-laesst-Map-Reduce-patentieren-908531.html (abgerufen 13.02.2010)

URL006 http://www.michael-noll.com/wiki/Running_Hadoop_On_Ubuntu_Linux_%28Single-Node_Cluster%29 (abgerufen 13.02.2010)

URL007 http://www.michael-noll.com/wiki/Running_Hadoop_On_Ubuntu_Linux_%28Multi-Node_Cluster%29 (abgerufen 13.02.2010)

URL008 http://hadoop.apache.org/ (abgerufen 13.02.2010)

URL009 http://de.wikipedia.org/wiki/Cloud_Computing (abgerufen 30.03.2010)

6 Abbildungsverzeichnis

7 Listing-Verzeichnis