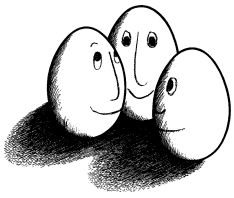
Bachelorarbeit

Speicherung und Analyse von BigData am Beispiel der Daten des FACT-Teleskops

Niklas Wulf



Bachelorarbeit
am Fachbereich Informatik
der Technischen Universität Dortmund

Dortmund, 16. Dezember 2013

Betreuer:

Prof. Dr. Katharina Morik
Dipl.-Inform. Christian Bockermann

Inhaltsverzeichnis

1. Einleitung

Die Menge an neu generierten Daten wächst in diversen Bereichen des alltäglichen Lebens immer weiter an, wodurch die sinnvolle Verarbeitung zunehmend erschwert wird. Das Internet mit von Benutzern erzeugten Inhalten, höher auflösende, billigere und vielfältigere Sensoren in vielen alltäglichen Produkten und in industriellen Anlagen sorgen für eine weiter wachsende Flut an Daten. Auch im Bereich der Wissenschaft führen genauere Resultate verschiedener Experimente zu größeren Datenmengen und damit zu Problemen bei ihrer Analyse.

Diese wachsenden Datenmengen und die Lösungen der daraus entstehenden Probleme werden heute meist unter dem Stichwort *Big Data* geeint. In dieser Arbeit sollen diese Probleme und Lösungen anhand eines physikalischen Experiments erarbeitet werden: Den Daten des *Ersten G-APD Cherenkov Teleskops* [4] (*First G-APD Cherenkov Telescope*, FACT, Abbildung 1.1).

Zunächst setzt sich diese Arbeit mit dem Big Data Begriff auseinander. Unklare Definitionen und ein großes allgemeines Interesse an Big Data in den letzten Jahren kann zu vorschnellem Zuordnen von Problemen zu diesem Bereich führen. Daher werden gleich zu Beginn einige Aspekte von Big Data betrachtet, die die gängige Definition des Begriffes bilden und es ermöglichen zu entscheiden, wann eine Datenmenge „groß“ im Sinne des Begriffs Big Data ist.

Hat man eine solche große Datenmenge, ergeben sich verschiedene Probleme. Noch bevor an eine Auswertung gedacht werden kann, müssen die Daten zunächst permanent gespeichert werden. Schon diese Aufgabe ist nicht trivial. Die vorliegende Ausarbeitung zeigt verschiedene Ansätze und die daraus entstehenden Vor- und Nachteile auf. Vor allem wird sie sich mit dem verteilten *Hadoop Distributed Filesystem* (HDFS) [6] auseinandersetzen und seine Einsatzzwecke für die Datenspeicherung und -analyse erörtern.

Ein weiteres Problem stellt der Transport der Daten dar. Nicht nur die Verbindung von verschiedenen Rechenzentren, sondern auch die Datenübertragung innerhalb eines einzelnen Rechnerclusters, also der Verbund vieler Rechner in einem Netzwerk, kann zum Engpass einer Berechnung werden. Es kann daher sinnvoll sein, die Daten überhaupt nicht von ihrem Speicherort zu den auswertenden Systemen zu transferieren. Eine modernere Vorgehensweise sieht vor, statt die Daten zu verschieben, die Analyseprogramme direkt auf den Computern auszuführen, auf denen die Daten ursprünglich gespeichert werden. Diese Vorgehensweise wird als *code-to-data* bezeichnet und wird in dieser Arbeit eine zentrale Rolle spielen.

Die Datenanalyse ist ein weiteres Problem, mit dem sich diese Arbeit auseinandersetzt. Oft ist die einzige Möglichkeit, große Datenmengen zu betrachten, auf einem Cluster verteilt ausgeführte Programme, das heißt Programme, die auf vielen Rechnern parallel Rechnen und jeweils einen Teil der Daten bearbeiten. Dies stellt

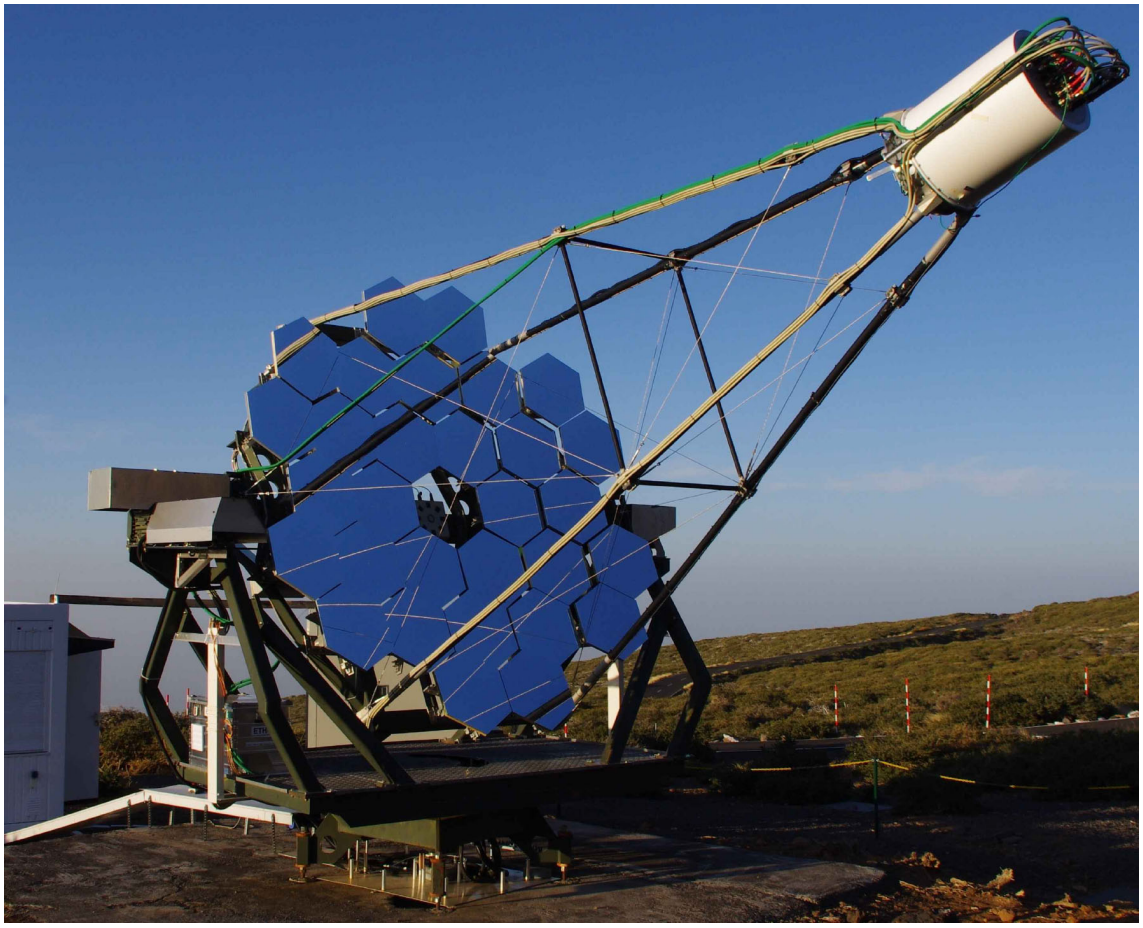


Abbildung 1.1.: Das Erste G-APD Cherenkov Teleskop auf Gran Canaria. [3]

besondere Herausforderungen an die Entwickler. Diese sind gerade im Bereich der Wissenschaft häufig keine ausgebildeten Programmierer und müssen sich die benötigten Kenntnisse neben ihrer eigentlichen Tätigkeit im Selbststudium aneignen.

In den letzten Jahren sind verschiedene Technologien entwickelt worden, die Analysten helfen sollen, mit Massen an Daten zu arbeiten. Diese Arbeit beachtet eine solche Technologie besonders: Das *Apache Hadoop*[1] Projekt. Es stellt mit dem bereits erwähnten HDFS eine gut geeignete Speichermethode bereit. Außerdem enthält es mit der eigentlichen *Hadoop* Java-Programmierbibliothek ein Werkzeug zur vereinfachten Programmierung von verteilten Programmen. Das Framework setzt auf verteilte Speicherung der Daten und dazu passend auf das *MapReduce* Programmiermodell, das sich gut für die Umsetzung des code-to-data-Prinzips eignet [7].

Eine andere Betrachtungsweise von Daten ist die kontinuierliche Abarbeitung neu eintreffender Daten. Das *streams* Java-Framework[2] verfolgt diesen Ansatz [5]. Darin werden Daten als kontinuierliche Ströme (*streams*) betrachtet, auf denen, in logische Pakete unterteilt, Berechnungen angestellt werden können. Das Framework ist bewusst simpel gehalten und damit einfach zu erlernen. Entscheidend für diese Aus-

[1] Apache Hadoop Homepage: http://hadoop.apache.org – Abgerufen am 15.12.2013 16:37
[2] Streams Projekt Homepage: http://www.jwall.org/streams – Abgerufen am 15.12.2013 16:37

arbeitung ist, dass viele Analyseschritte für FACT-Daten bereits im Rahmen dieses Frameworks implementiert worden sind.

Das FACT-Teleskop

Die Daten des *Ersten G-APD Cherenkov Teleskops* sollen in dieser Arbeit als Beispiel für Big Data dienen [3, 4]. Das Teleskop soll sehr energiereiche Quellen von Gammastrahlenblitzen (*very high enery gamma-ray*, VHE) im Weltraum beobachten. Diese kosmischen Strahlen induzieren in unserer Atmosphäre Kaskaden anderer Teilchen, die wiederum Cherenkov-Licht erzeugen, welches vom am Boden stehenden Teleskop aufgezeichnet werden kann.

Das FACT-Teleskop setzt zu diesem Zweck auf eine Kamera mit 1440 neuartigen *Geiger-mode Avalanche Photodiodes* (G-APD). Es wird versucht zu zeigen, dass diese neuen Dioden Vorteile für die VHE Gammastrahlen-Astronomie mit sich bringen. Im Herbst 2011 wurde die Kamera am Observatorio del Roque de los Muchachos auf Gran Canaria (Spanien) installiert und in Betrieb genommen.

Gammastrahlenblitze sind sehr kurze Ereignisse von wenigen Nanosekunden. Die Kamera soll solche Ereignisse anhand verschiedener Auslöser erkennen und aufzeichnen. Die Analyse dieser Ereignisse ist momentan eine der Aufgaben des Projekts. Da die Kamera eine komplette Neuentwicklung ist, muss zunächst erforscht werden, wie sie sich genau verhält. Aus den aufgezeichneten Daten müssen diejenigen Ereignisse extrahiert und analysiert werden, welche einem Gammastrahlenblitz entsprechen. Abbildung 1.2 zeigt als Beispiel zwei verschiedenartige Ereignisse.

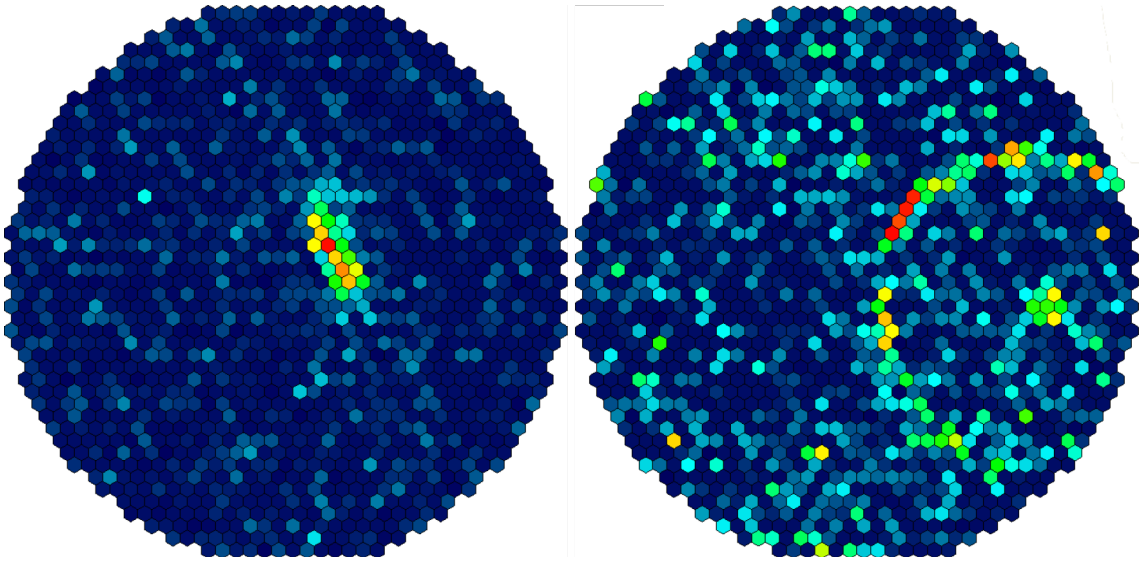


Abbildung 1.2.: Zwei Momentaufnahmen der FACT-Kamera. Je nach stattgefundenem Ereignis in unserer Atmosphäre fallen Photonen in unterschiedlicher Weise auf das Teleskop. Links ist ein Kandidat für einen Gammastrahlenblitz und rechts ein "Myonen Ereignis" zu sehen [3].

1.1. Fragestellungen und Ziele

Aus dieser kurzen Einleitung ergeben sich die folgenden Fragestellungen, mit denen sich diese Arbeit auseinandersetzt:

- Lassen sich Techniken moderner Big Data Systeme auf wissenschaftliche Aufgabenstellungen – insbesondere die Ereignisanalyse der FACT-Daten – anwenden?
- Wie kann es Datenanalysten ermöglicht werden, ohne zu großen Aufwand diese neuen Technologien für sich zu nutzen?
- Ist es möglich, die bereits verfassten Analyseprogramme weiter zu verwenden und dennoch die Vorteile des verteilten Rechnens ausnutzen?

Aus diesen Fragestellungen lässt sich bereits ablesen, dass es Ziel dieser Arbeit sein wird, eine Brücke zwischen den eher simplen streams und dem etwas komplexeren Programmiermodell von Hadoop zu schlagen. Zu diesem Zweck soll eine Erweiterung des streams Frameworks namens *streams-mapred* verfasst werden. Mit dieser Erweiterung soll ein Programmierer seine für streams geschriebene Datenanalyseprogramme ohne großen (Lern-)Aufwand in einer Hadoop Umgebung ausführen können.

Mit Hilfe von streams-mapred soll dann gezeigt werden, dass Umgebungen wie Hadoop Vorteile bei der Analyse großer Datenmengen auch im Bereich der Wissenschaft mit sich bringen können, ohne dass unverhältnismäßiger Aufwand betrieben werden muss. Dafür werden die FACT-Daten in verschiedenen Formaten in einem Hadoop Cluster abgelegt und ihre Analyse innerhalb des Clusters durchgeführt und untersucht.

1.2. Aufbau der Arbeit

In Kapitel 2 werden dafür zunächst die Grundlagen erörtert. Es wird sich zunächst mit den Definitionen von Big Data beschäftigen. Darauf aufbauend werden die angesprochenen Probleme und verschiedenen Lösungen der Speicherung und Analyse besprochen.

Im darauffolgenden dritten Kapitel werden die für diese Arbeit zentralen Technologien vorgestellt: Das Apache Hadoop Framework, sowie das streams Framework.

Kapitel 4 erläutert wie die streams Erweiterung streams-mapred geplant und implementiert worden ist. Außerdem wird eine kurze Einführung in seine Benutzung erfolgen.

Nachdem die technologischen Grundlagen eingeführt sind, setzt sich Kapitel 5 mit verschiedenen Speichermöglichkeiten von großen Datenmengen auseinander. Die FACT-Daten werden bei dieser Betrachtung besonders beachtet. Anschließend evaluiert das Kapitel außerdem, welche Ergebnisse die unterschiedlichen Methoden bei der Verwendung von streams-mapred erzielen.

Im abschließenden Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst und ein Ausblick auf mögliche Entwicklungen gegeben.

2. Grundlagen

Wie in Kapitel 1 dargelegt, werden heutzutage immer mehr Daten generiert. Das Stichwort Big Data fasst das Thema große Datenmengen und die daraus entstehenden Probleme sowie mögliche Lösungsansätze zusammen.

Im Abschnitt 2.1 dieses Kapitels wird dieser Begriff zunächst genauer definiert. Es sollen dann verschiedene Herangehensweisen an die Analyse großer Datenmengen betrachtet werden. Dabei wird besonders beachtet werden, welche Probleme der klassische Supercomputer und seine Programmierung mit sich bringen kann und wie moderne Big Data Systeme diese Probleme zu lösen versuchen. Eine genauere Betrachtung der Daten des FACT-Projekts soll aufzeigen, wie diese in den Big Data Bereich passen und wie die Ereignisanalyse mit den Big Data Techniken umgesetzt werden kann.

2.1. Big Data

Der Begriff Big Data lässt zunächst vermuten, dass er über eine bestimmte Datenmenge definiert ist, also beispielsweise alle Datensätze im Tera- oder Petabytebereich Big Data darstellen. Eine solche Definition wäre jedoch weder sinnvoll noch hilfreich.

Die anerkannteste Definition charakterisiert Big Data stattdessen anhand von drei Kriterien [12, 19]: Informationen mit großem Volumen, hoher Geschwindigkeit und großer Varietät sind Big Data. Aus den englischen Begriffen *volume*, *velocity* und *variety* werden „die 3Vs“ oder „V^3“, die beim Planen und Implementieren eines Systems im Hinterkopf behalten werden sollten.

- **Volumen** (*volume*)

  Der offensichtlichste Aspekt von Big Data ist, dass das anfallende Datenvolumen der zu verarbeitenden Informationen groß ist. Es ist dabei nicht wichtig zu definieren, wann eine Datenmenge groß ist. Wichtig ist es, zu erkennen, dass große Datenmengen schon beim Speichern Probleme bereiten können und ein geeignetes Speichersystem anzulegen ist.

  Lösungsansätze beginnen beim Aufrüsten der Speicherkapazität eines Einzelplatzrechners, gehen über die Lösung der Speichervirtualisierung in großen Rechenzentren und enden schließlich bei auf Big Data spezialisierten Lösungen wie verteilten Dateisystemen (zum Beispiel HDFS) oder verteilten Datenbanken (zum Beispiel *Apache HBase* [8]).

- **Geschwindigkeit** (*velocity*)

  In Big Data Systemen entstehen neue Informationen sehr schnell. Auch dieser Aspekt erhöht die Ansprüche an das Speichermedium. Es muss hinreichend

schnell an die Informationsquellen angebunden werden und die entstandenen Informationen auch genügend schnell ablegen können.

Ein weiterer Problempunkt entsteht dann, wenn die Informationsverarbeitung in Echtzeit geschehen soll, wenn also bestimmte Ergebnisse aus den Informationen innerhalb genau definierter Zeitgrenzen nach ihrem Eintreffen bereitstehen sollen. Das in der Einleitung bereits erwähnte streams Framework eignet sich für diese Aufgabe.

- **Varietät** (*variety*)

  Verschiedene Informationsquellen führen dazu, dass eine große Anzahl verschiedener Datenformate in einem Big Data System anfallen. Unterschieden wird dabei zwischen strukturierten Daten wie relationalen Datenbanken, semistrukturierten Daten wie in NoSQL Datenbanken, bis hin zu unstrukturierten Daten wie von Menschen verfassten Texten. Beim Entwurf eines Systems sollte in Betracht gezogen werden, dass sich Eingabeformate jederzeit ändern können und generell je nach Anwendung unterscheiden.

In den letzten Jahren haben sich einige weitere „Vs“ herausgebildet: Der Wert [2] (*value*), die Richtigkeit [18] (*veracity* oder *validity*) und die Flüchtigkeit [10] (*volatility*) der Daten:

- **Wert** (*value*)

  Häufig werden Daten nur zu archivarischen Zwecken oder für eine spätere Verwendung aufgezeichnet. Mit neuen Methoden des maschinellen Lernens und schnelleren Systemen kann es zu einem späteren Zeitpunkt möglich werden, neuen Wert aus zuvor gespeicherten Daten zu extrahieren. In der Wirtschaft kann mit diesem „neuen Wert“ eine Steigerung der Einnahmen gemeint sein. Im wissenschaftlichen Bereich können neue Erkenntnisse entstehen, ein Forschungsgebiet voran bringen und so neuen Wert erzeugen. Für ein informationsverarbeitendes System bedeutet dies, dass es möglich sein sollte, ohne großen Aufwand möglichst beliebige Fragen über die vorliegenden Informationen zu stellen, um eventuell neue Erkenntnisse daraus zu ziehen.

- **Richtigkeit** (*veracity*, *validity*)

  Bevor aus Daten Schlüsse und Antworten gezogen werden können, sollte dem Analysten ihre Korrektheit und Relevanz für das gestellte Problem bewusst sein. Im Falle der FACT-Daten haben sich mitunter einzelne Pixel als fehlerhaft herausgestellt. Solche Probleme können auch nach längerem fehlerfreien Betrieb auftreten und müssen erkannt werden.

- **Flüchtigkeit** (*volatility*)

  Es sind Daten vorstellbar, die nach einer gewissen Zeit ihre Korrektheit oder Relevanz verlieren. Sie sollten dann nicht mehr in Berechnungen mit einfließen und unter Umständen gelöscht werden. Für die hier betrachteten Daten

trifft dieser Aspekt nicht zu und fließt auch nicht in die weiteren Betrachtungen ein. Grundsätzlich ist das Thema jedoch wichtig und sollte beim Entwurf eines Systems im Auge behalten werden. Wenn Daten nach einiger Zeit gelöscht werden können, kann der Speicher im Vorhinein entsprechend kleiner dimensioniert werden und ist damit kostengünstiger.

2.2. Herangehensweisen

Um große Datenmengen zu speichern und zu verarbeiten, sind verschiedene Ansätze möglich. Da die Daten, wenn sie Big Data darstellen, wie auch bei FACT, die Kapazität eines einzelnen Rechners übersteigt, müssen andere Lösungen genutzt werden.

Die typische Vorgehensweise der letzten Jahre ist die Verwendung eines Clusters, also ein Verbund aus vielen Computern. Diese bestehen aus vielen Rechnerknoten und nutzen häufig eine heterogene Architektur. Das bedeutet, dass sie verschiedene Knotentypen innerhalb des Clusters einsetzen. Manche Knoten sind spezialisiert auf das Speichern von Daten, andere spezialisiert auf hohe Prozessorleistung, andere bieten besonders schnelle Netzwerkverbindungen untereinander. Der Supercomputer der Technischen Universität Dortmund *LiDOng*[1] hat beispielsweise 296 gewöhnliche Rechenknoten, 128 über Infiniband[1] stark vernetzte Knoten, 16 *Symmetric Multiprocessor* Knoten, 4 Knoten mit hoher GPU Leistung, und – für diese Arbeit entscheidend – 8 extra Knoten für die Datenhaltung mit 256TB Speicherplatz (siehe Abbildung 2.1).

Eine solche Architektur ist geeignet hohe Leistung zu vollbringen. Ist ein Programm gut auf das verwendete Rechencluster zugeschnitten und rechnet verteilt, kann das Cluster gleichmäßig ausgelastet und Overhead vermieden werden. Ein verteiltes Programm, welches viele Zwischenergebnisse austauschen muss, kann zum Beispiel extra an die stark vernetzten Knoten angepasst werden.

Was ein Vorteil sein kann, ist zugleich ein entscheidender Nachteil: Das Anpassen der Programme ist nicht immer trivial. LiDOng setzt auf eine *Distributed Memory* Architektur. Das heißt, dass jeder Prozessor seinen eigenen privaten Speicherraum erhält. Der Programmierer bekommt von einer solchen Architektur keine Unterstützung beim Planen und Ausführen seiner Programme. Er muss selbstständig eine geeignete Softwarearchitektur wählen und die Kommunikation zwischen den Knoten selbstständig programmieren.

Dabei kommen Protokolle wie das *Message Passing Interface* (MPI)[9] zum Einsatz. Solche Protokolle und zugehörige Bibliotheken erlauben dem Entwickler zwar viele Freiheiten, jedoch können eben diese Freiheiten zur Fehlerquelle werden und neben der Implementierung zu Mehraufwand führen. Dieser Mehraufwand kann schließlich darin resultieren, dass keine verteilte Anwendung verfasst wird. Stattdessen wird ein einzelner Knoten verwendet und ein klassisches (eventuell paralleles)

[1] Dokumentation von LiDOng: http://lidong.itmc.tu-dortmund.de/ldw/index.php/Main_Page – Abgerufen am 15.12.2013 20:20

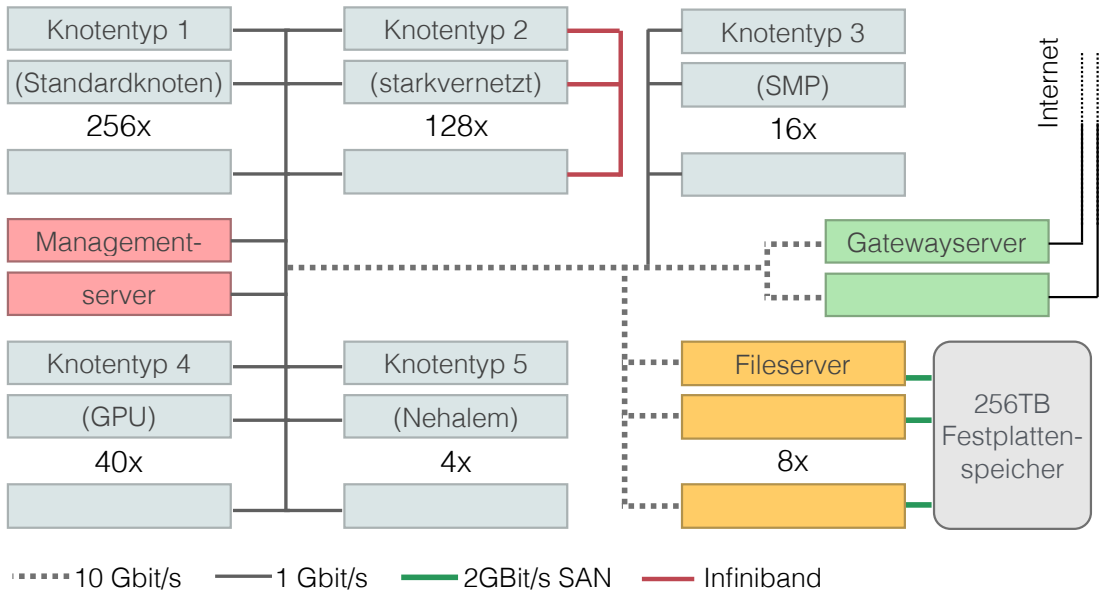


Abbildung 2.1.: Schematischer Aufbau von LiDOng mit seinen verschiedenen Knotentypen[2].

Programm darauf ausgeführt. Die Analyse der Daten dauert länger, als es bei einem verteilten Programm der Fall wäre.

Ein weiteres wichtiges Problem entsteht durch die klassische Trennung von Speicher- und Rechenknoten. Für jede Berechnung müssen die benötigten Daten erst vom Speicher auf die Rechner übertragen werden. Ist die Datenmenge zu groß, kann die Anbindung der beiden Knotentypen zum Flaschenhals der Analyse werden. Häufig ist diese Anbindung tatsächlich ein „physikalischer“ Engpass und alle Daten müssen gleichzeitig durch die selbe Netzwerkleitung.

Wenn schon die Datenübertragung vom Speicher auf die Rechner länger dauert als die eigentliche Analyse, hilft auch ein perfekt angepasstes Programm nicht, die Analyse in kurzer Zeit auszuführen. Steigende Datenmengen werden dieses Problem in Zukunft weiter verschlimmern. Die Geschwindigkeit der Dateneingabe wird dadurch wichtiger werden.

Der Big Data Ansatz

Bis hierhin wurden also zwei zentrale Probleme bestehender Systeme identifiziert: Die Übertragung der Daten vom Speicher zu den Rechnern kann die Anwendung ausbremsen und die Komplexität verteilter Programme kann zur Fehlerquelle beim Entwurf und bei der Implementierung werden.

[2]Grafik frei nach dem offiziellen LiDOng System Overview:
http://lidong.itmc.tu-dortmund.de/ldw/index.php/System_overview
– Abgerufen am 19.11.2013 10:42

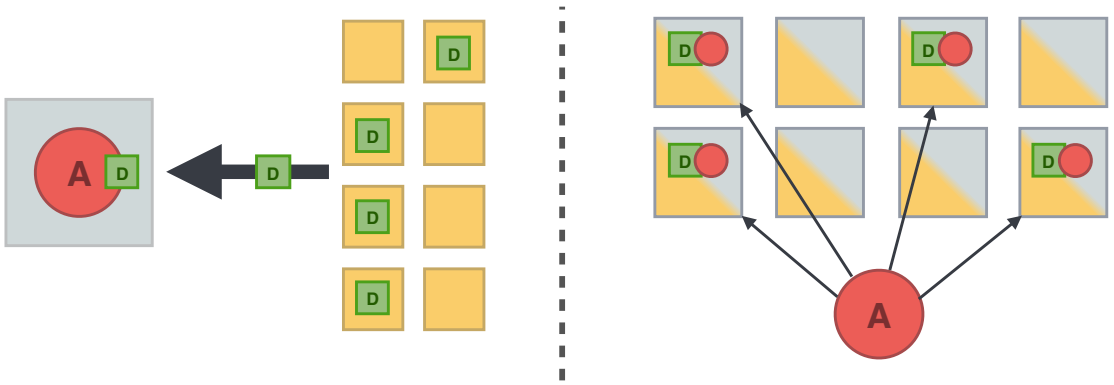


Abbildung 2.2.: Beim klassischen Rechnercluster (links) wird die Anwendung (A) von den Speicherknoten mit den Daten (D) versorgt. Beim *code-to-data*-Prinzip wird die Anwendung zu den Knoten mit den Daten gesendet und dort ausgeführt.

In modernen Big Data Systemen wird ein Paradigma verfolgt, das diese Probleme löst: Das eingangs erwähnte *code-to-data*-Prinzip. Abbildung 2.2 illustriert den Unterschied zum klassischen Vorgehen.

Der Big Data Ansatz setzt im Gegensatz zum klassischen Supercomputer auf ein homogenes Rechnercluster. Darin werden alle Knoten sowohl als Speicher- als auch als Rechnerknoten eingesetzt. Statt vor jeder Berechnung Daten auf einen Rechenknoten zu übertragen, werden sie von vornherein auf das komplette Cluster verteilt gespeichert. Soll eine Anwendung ausgeführt werden, wird diese zu den Knoten gesendet, die die benötigten Daten enthalten und dort ausgeführt.

Die Anwendung nutzt damit die Eingabe-Hardware des kompletten Clusters gleichzeitig. Statt des Engpasses zwischen Speicher- und Rechenknoten werden also direkt lokal gespeicherte Daten von Festplatten verwendet und so eine hohe Eingabegeschwindigkeit erzielt.

Der Big Data Ansatz löst damit die aufgezeigten Probleme. Die Übertragung der Daten zu anderen Knoten ist für eine Berechnung nicht mehr notwendig und mit Hilfe geeigneter Software wird das Implementieren von Software für eine solche Architektur wesentlich vereinfacht.

Ein weit verbreitetes Framework, das diese Idee umsetzt, ist das erwähnte Apache Hadoop Framework. Die Daten werden mit Hilfe des HDFS auf dem Cluster verteilt. Mit Hilfe des MapReduce Programmiermodells werden dann Anwendungen verfasst und auf dem Cluster bei den Daten ausgeführt. Das Framework spielt in dieser Arbeit eine zentrale Rolle und in Abschnitt 3.1 genauer vorgestellt.

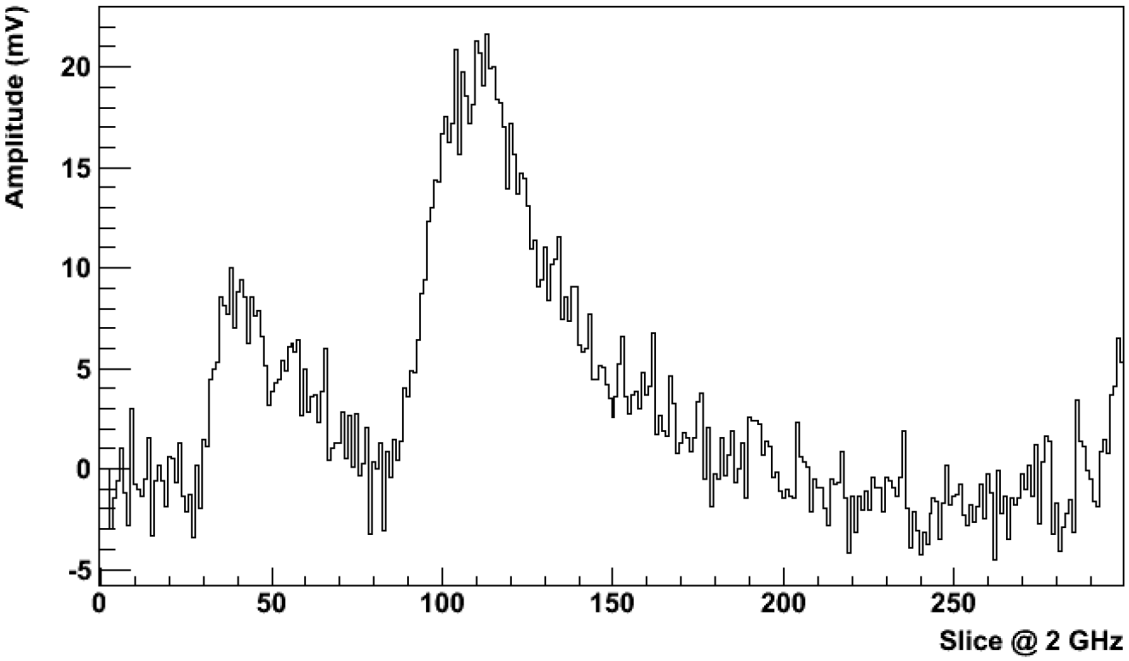


Abbildung 2.3.: Beispieldaten eines Ereignisses eines einzelnen Pixels. Die beiden Ausschläge sind das Ergebnis der Einschläge von einem Photon (etwa bei Wert 45) beziehungsweise zweien (Wert 120) [3]

2.3. FACT-Daten als Big Data

Wie in der Einleitung erwähnt, fallen im Rahmen des FACT-Projektes Daten in Form von Gammastrahlenblitz-Ereignissen an. Diese können als Big Data aufgefasst werden.

Für jedes der Ereignisse werden typischerweise pro Pixel 300 Werte mit 2 GHz (also 150 Nanosekunden lang) aufgezeichnet, wie in Abbildung 2.3 zu sehen. Diese hohe Taktrate ist für die großen anfallenden Datenmengen mitverantwortlich. Hier zeigt sich, warum das Teleskop eine Informationsquelle für ein Big Data System sein kann. Sowohl das große *Volumen*, als auch die große *Geschwindigkeit*, in der neue Daten entstehen, sind gegeben.

Die Daten werden im Flexible Image Transport System Format gespeichert [15] (FITS, zu deutsch „flexibles Bildtransportsystem“). Es ist ein von der NASA entwickeltes Binärformat für Bilder, kann aber auch allgemeine Daten in Tabellen speichern. Da man hier also von strukturierten Daten sprechen kann, ist der Aspekt der *Varietät* durch die Teleskopdaten alleine nicht gegeben. Bei den Überlegungen zu dieser Arbeit wird der Aspekt der Varietät jedoch nicht aus den Augen verloren. Auch andere wissenschaftliche Berechnungen sollen später profitieren können.

Im Anbetracht der Tatsache, dass das Teleskop seit Herbst 2011 Daten aufzeichnet, ist es eine typische Informationsquelle für bereits bestehende Daten mit weiterem potenziellen *Wert*. Die Daten werden im Moment zunächst auf Systemen direkt beim Teleskop gespeichert und von dort aus zu den Großrechnern verschiedener Universitäten transferiert. Dort lagern sie zum Beispiel an der Technischen Universität

Dortmund zu großen Teilen auf für Berechnungen ungeeigneten Systemen. Berechnungen auf allen Daten sind so nicht ohne weiteres möglich.

Code-To-FACT-Data

Ein häufiger Anwendungsfall im FACT Projekt ist die Analyse einer großen Menge an Ereignissen. Aus jedem der Ereignisse sollen dabei Informationen extrahiert und anschließend betrachtet werden. Es wird also über alle Ereignisse iteriert und aus jeder Iteration ein Ergebnis ausgegeben.

Im aktuellen System wird dafür zunächst eine Datei mit zehntausenden Ereignissen aus einem Speichersystem auf einen Rechenknoten kopiert. Dort werden alle darin enthaltenen Ereignisse analysiert und ihre Ergebnisse ausgegeben. Anschließend wird die Kopie der Eingabedatei gelöscht. Es handelt sich also um ein klassisches System mit getrennten Speicher- und Rechenknoten.

Eine Analyse, die jedes Ereignis einzeln betrachtet, lässt sich jedoch auch ohne große Probleme verteilt umsetzen. In einem moderneren Big Data System, das den code-to-data-Ansatz verfolgt, würden die Ereignisse möglichst gleichmäßig auf verschiedene Rechnerknoten verteilt werden. Eine Anwendung würde dann zu den Ereignissen geschickt werden, die untersucht werden sollen. Auf jedem Rechenknoten würde über die Teilmenge der Ereignisse iteriert, die dort abgelegt wurden. Die Ergebnisse jedes Ereignisses würden vom System gesammelt und zusammengefasst ausgegeben werden.

Ein solches modernes System soll durch diese Arbeit und ihre Ergebnisse ermöglicht werden.

2.4. Exkurs: Die Lambda Architektur

Die Lambda Architektur [14] beschreibt ein vollständiges System zur Verarbeitung von Big Data. Im Prinzip ist sie eine Sammlung von Regeln für den Entwurf eines solchen Systems. Die Architektur vereinigt dafür Technologien für die Echtzeitverarbeitung von Daten mit solchen, die geeigneter sind, große Datenmengen zu verarbeiten. Ziel dieser Vereinigung soll es sein, sowohl historische als auch sehr aktuelle Informationen aus kontinuierlichen Quellen in die Ergebnisse beliebiger Anfragen einfließen zu lassen. Sie betrachtet also die beiden Aspekte Volumen und Geschwindigkeit von Big Data.

Die Architektur ist dafür, wie in Abbildung 2.4 dargestellt, in drei Schichten unterteilt: *Batch*, *Speed* und *Serving* Schicht. Die Batch und Speed Schichten verarbeiten eintreffende Daten, zu so genannten vorberechneten Ansichten (*pre-computed view*), die die Serving Schicht dazu nutzt (beispielsweise durch das Zusammenfügen der Ansichten), um beliebige Anfragen zu beantworten.

Batch Schicht

Die Batch Schicht ist für die Haltung aller Daten verantwortlich. Alle neu eintreffenden Daten werden in einem geeigneten Speichersystem wie HDFS abgelegt und

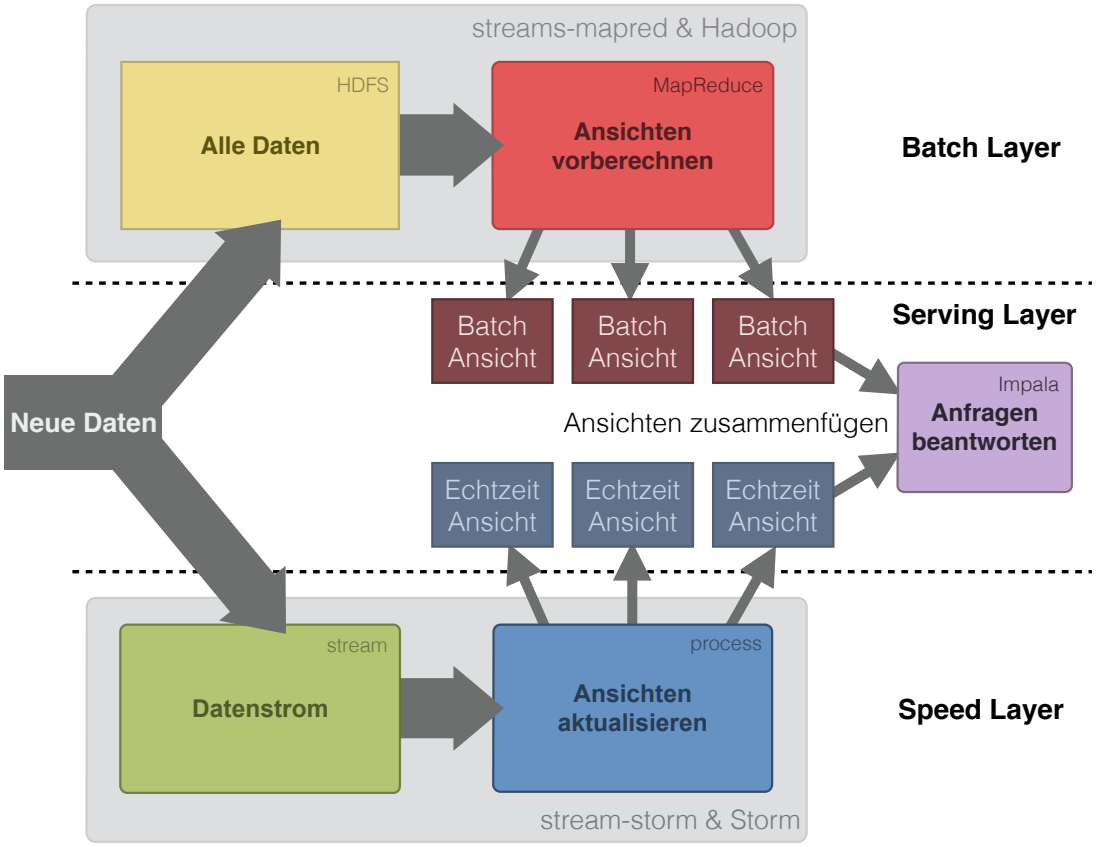


Abbildung 2.4.: Aufbau der Lambda Architektur[3], wie sie mit den streams-storm und streams-mapred Erweiterungen implementiert werden könnte.

permanent gespeichert. Die Schicht ist zudem dafür zuständig, aus diesen historischen Daten vorberechnete Ansichten zu generieren, die von der Serving Schicht verarbeitet werden können.

Entscheidend ist, dass diese Berechnungen sehr lange brauchen, da sie auf allen Daten basieren. Während eine Berechnung stattfindet, werden neu eintreffende Daten zwar abgelegt, fließen jedoch nicht in die Berechnung mit ein.

Speed Schicht

In der Speed Schicht werden alle neu eintreffenden Daten in Echtzeit verarbeitet und entsprechende vorberechnete Ansichten laufend aktualisiert. Ist eine Berechnung der Batch Schicht abgeschlossen, können, wie in Abbildung 2.5 dargestellt, die bis dahin inkrementell aktualisierten Ergebnisse der Speed Schicht verfallen, da sie nun in den Ansichten aus der Batch Schicht enthalten sind.

[3]Grafiken frei nach einem Blogeintrag über die Lambda Architektur: http://jameskinley.tumblr.com/post/37398560534/the-lambda-architecture-principles-for-architecting – Abgerufen am 25.11.2013 18:21

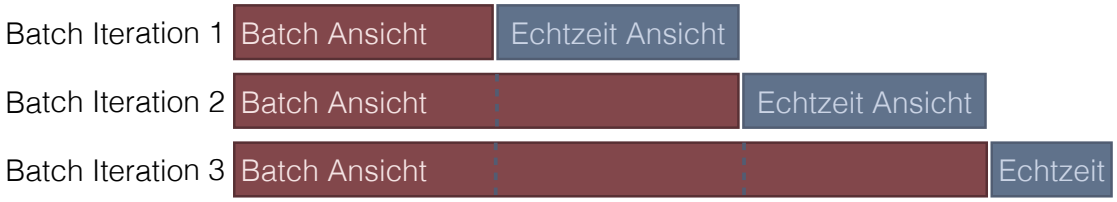


Abbildung 2.5.: Ist eine Berechnung in der Batch Schicht abgeschlossen, können Echtzeit-Ansichten, deren Daten dann in den Batch-Ansichten repräsentiert werden, verfallen.

Serving Schicht

Die Serving Schicht stellt eine geeignete Schnittstelle bereit, um Anfragen an das System zu stellen. Erfordert eine solche Anfrage Informationen aus dem kompletten Datensatz, werden die vorberechneten Ansichten aus Batch und Speed Schicht zu einer Antwort zusammengefasst.

Die gesamte Architektur ist damit besonders robust gegenüber Programmierfehlern. Es werden nur Informationen aus der ursprünglichen Quelle permanent gespeichert. Die daraus gewonnenen Ergebnisse verfallen mit jeder Iteration der Batch Schicht. War eine der Schichten fehlerhaft Implementiert, müssen nicht erst aufwändig ganze Datenbanken korrigiert oder konvertiert werden. Die Architektur enthält die Neuberechnung aller Ergebnisse schon im Entwurf.

2.4.1. Die Lambda Architektur im Kontext

Im folgenden Kapitel wird beschrieben, wie die Lambda Architektur im Rahmen dieser Arbeit Verwendung finden kann. Es zeigt, dass das streams Framework im Hinblick auf die Verarbeitung eintreffender Daten in Echtzeit entwickelt wurde. Damit eignet es sich gut für den Entwurf und die Implementierung einer Speed Schicht.

Außerdem wird gezeigt, dass sich Apache Hadoop gut eignet, um die Batch Schicht zu implementieren. Durch die Entwicklung der streams-mapred Erweiterung für streams in Kapitel 4 lässt sich das streams-Konzept auf die Batch Schicht übertragen.

Durch diese Weiterentwicklung können große Teile einer Lambda Architektur entworfen, implementiert und eingesetzt werden, unter der Verwendung eines einzigen, gut zusammenpassenden und einfach zu erlernenden Frameworks. Diese Idee wird in Abbildung 2.4 erläutert.

3. Verwendete Technologien

In Kapitel 2 werden grundlegende Eigenschaften von Big Data erläutert. Es zeigt, dass die klassische Verarbeitung von großen Datenmengen mit den typischen Supercomputern der letzten Jahre Nachteile mit sich bringen kann. Das code-to-data-Prinzip ist eine Möglichkeit, diese Probleme zu lösen. Im Rahmen dieser Arbeit soll das streams-mapred Framework entworfen und implementiert werden. Es soll die Möglichkeit bieten, mit dem streams Framework verfasste Anwendungen auf einem Apache Hadoop Cluster auszuführen.

Dieses Kapitel stellt daher zunächst diese beiden Technologien genauer vor. Im Abschnitt 3.1 dieses Kapitels erfolgt daher eine Einführung in das Hadoop Framework. Das MapReduce Programmiermodell wird allgemein erläutert und anschließend ein komplettes Hadoop Programm entworfen.

Das für diese Arbeit zentrale *streams* Framework eignet sich gut, um Anwendungen für die Echtzeitanalyse zu entwickeln. Im Abschnitt 3.2 werden die darin angewendeten Konzepte sowie deren Umsetzung im Java-Framework dargelegt.

3.1. Apache Hadoop

Apache Hadoop [17] ist Open Source und seit 2008 ein sogenanntes Top-Level-Projekt der *Apache Software Foundation*. Verschiedene große Firmen setzen auf Hadoop, darunter unter anderem Facebook, IBM und Yahoo [16]. Es ist der de facto Standard für verteiltes Rechnen mit MapReduce und damit eine gute Umgebung für Anwendungen nach dem code-to-data-Prinzip.

Wie in Kapitel 2.2 gezeigt, werden Anwendung dafür im Cluster verteilt bei ihren Eingabedaten gestartet. Anschließend ist es sinnvoll die Ergebnisse von den einzelnen Programminstanzen einzusammeln und zusammenzufassen. Genau diese Vorgehensweise wird vom MapReduce Programmiermodell verfolgt. Darin besteht ein Programm aus zwei separaten Schritten. Der *Map* Schritt wendet eine Berechnung auf alle Eingabedaten an. Die Ergebnisse jeder Berechnung werden dann im *Reduce* Schritt zum endgültigen Ergebnis zusammengefasst.

In Hadoop muss der Programmierer zunächst nur die beiden MapReduce Schritte implementieren. Das Verteilen der Anwendung zu den Knoten mit Eingabedaten sowie das Transferieren der Ergebnisse zum Reducer übernimmt das Framework. Abbildung 3.1 illustriert dieses Vorgehen.

Der Kern von Hadoop sind das *Hadoop Distributed File System* (HDFS) [6, 16], sowie ein Java Framework und eine Ausführungsumgebung für MapReduce Programme. Mit dem HDFS lassen sich Dateien auf ein Cluster verteilen. Das Java Framework unterstützt den Programmierer beim Verfassen von MapReduce Anwendungen. Die passende Ausführungsumgebung schließlich ist es, die die Anwendung

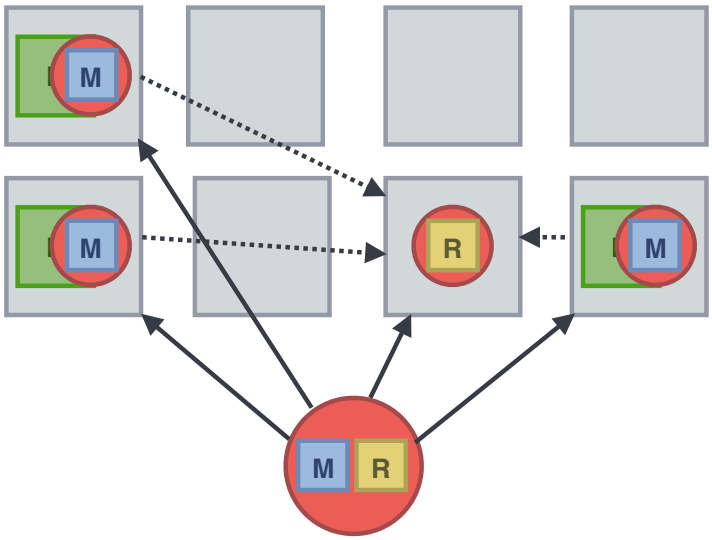


Abbildung 3.1.: Eine Anwendung in Hadoop besteht aus einem Map- (M) und Reduce-Programmteil (R). Der Map-Schritt wird automatisch bei den im Cluster verteilten Eingabedaten ausgeführt und ihre Ergebnisse zum Reducer gesendet und dort zusammengefasst.

im Cluster verteilt ausführt und die Kommunikation zwischen den einzelnen Programmteilen steuert.

Um Hadoop besser zu verstehen, wird zunächst das MapReduce Modell im Allgemeinen vorgestellt. Anschließend wird die Implementierung einer konkreten MapReduce Anwendung in Hadoop gezeigt.

3.1.1. MapReduce

MapReduce ist ein Programmiermodel, das in den letzten Jahren vor allem durch die Neuentdeckung und Verwendung bei Google Bekanntheit erlangt hat [7]. Es ist das vorherrschende Programmiermodell in Hadoop. Andere Programmiermodelle sind in Hadoop (insbesondere seit Version 2) möglich, sind aber kein Gegenstand dieser Ausarbeitung.

Eine komplette MapReduce Berechnung erhält als Eingabe eine Liste von Schlüssel- und Werte-Paaren und berechnet als Ergebnis ebenso eine Liste von Schlüssel- und Werte-Paaren:

$$MapReduce(list(k1, v1)) \rightarrow list(k3, v3)$$

MapReduce besteht aus zwei aufeinander folgende Schritte. Der Programmierer muss diese Schritte in Form einer *map* und einer *reduce* Funktion spezifizieren:

$$map(k1, v1) \rightarrow list(k2, v2)$$

$$reduce(k2, list(v2)) \rightarrow list(k3, v3)$$

Die *map* Funktion erhält als Eingabeparameter *eines* der Schlüssel-Werte-Paare der Eingabe $(k1, v1)$. Sie berechnet ein Zwischenergebnis, bestehend wiederum aus einer Liste von Schlüsseln und zugehörigen Werten $list(k2, v2)$. Die Ausführungsumgebung sortiert und gruppiert die Ausgaben aller *map* Aufrufe anhand ihrer Schlüssel.

Die *reduce* Funktion erhält als Eingabeparameter ein Paar aus Schlüssel und einer Liste der Ausgaben aller *map*-Aufrufe mit diesem Schlüssel $(k2, list(v2))$. Daraus berechnet sie einen Teil der Endergebnisse der gesamten Berechnung $list(k3, v3)$, welche schließlich von der Ausführungsumgebung zusammengefügt werden.

Entscheidend ist, dass die *map* und *reduce* Funktionen je einen Teil der Eingabe beziehungsweise der Zwischenergebnisse als Parameter übergeben bekommen und diese Teilberechnungen jeweils unabhängig voneinander sind. Erst dadurch lassen sie sich verteilt ausführen. Abbildung 3.2 zeigt eine schematische Darstellung des Ablaufs einer MapReduce Berechnung und illustriert zugleich das folgende Beispiel.

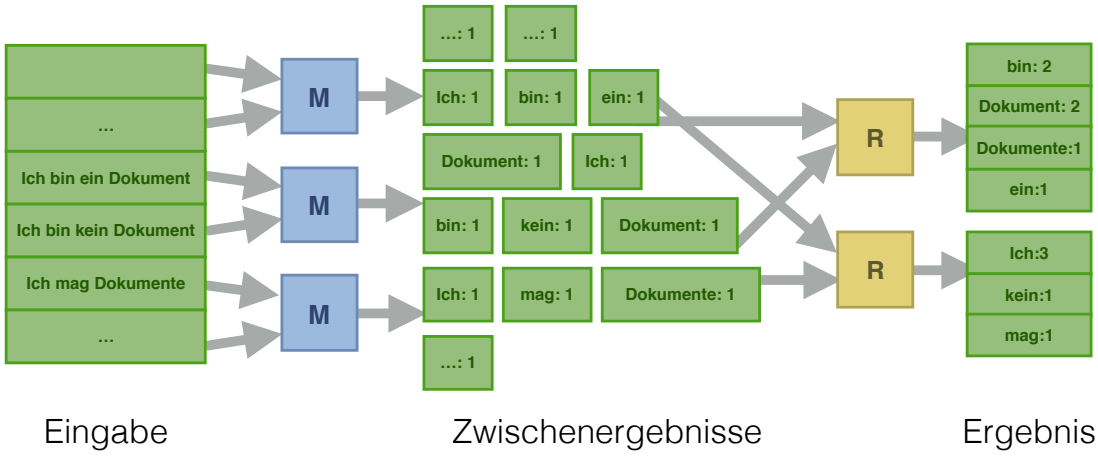


Abbildung 3.2.: Schematischer Ablauf von MapReduce am Beispiel von WordCount. Die Mapper (M) produzieren Zwischenergebnisse, die von den Reducern (R) zusammengefasst werden.

Das WordCount-Beispiel

Wenn MapReduce vorgestellt wird, ist das am häufigsten verwendete Beispiel das *WordCount* Problem. Dabei geht es darum, Wörter in mehreren Dokumenten zu zählen.

Als Eingabe erhält die Berechnung eine Liste von Paaren, bestehend aus Dokumentname und Dokumentinhalt. Für WordCount ist nur der Inhalt relevant, es gibt jedoch Aufgabenstellungen, für die Schlüssel-Werte-Paare notwendig sind, wie beispielsweise die Wortanzahl pro Dokument. Die Ausgabe der Berechnung ist eine Liste von Paaren, bestehend aus einem Wort und seiner Anzahl in allen Eingabedokumenten. Der Pseudocode im Quelltext 3.3 setzt diese Aufgabe als Map- und Reduce-Schritt um.

```
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));
```

Abbildung 3.3.: WordCount als Pseudocode [7].

Der Ablauf des Codes lässt sich am besten anhand eines konkreten Beispiels illustrieren. Als Eingabe sind drei Dokumente als Name-Inhalt-Paar gegeben:

```
("doc_1", "Ich␣bin␣ein␣Dokument")
("doc_2", "Ich␣bin␣kein␣Dokument")
("doc_3", "Ich␣mag␣Dokumente")
```

Die *map* Funktion wird für jedes der Eingabedokumente einmal aufgerufen. Für jedes der Wörter des Dokumentes produziert es ein Ausgabepaar bestehend aus dem Wort und einer „1“. Aus dem Aufrufen ergeben sich die folgenden Zwischenergebnisse:

```
map("doc_1", "Ich␣bin␣ein␣Dokument")
  = {("Ich", "1"), ("bin", "1"), ("ein", "1"), ("Dokument", "1")}
map("doc_2", "Ich␣bin␣kein␣Dokument")
  = {("Ich", "1"), ("bin", "1"), ("kein", "1"), ("Dokument", "1")}
map("doc_3", "Ich␣mag␣Dokumente")
  = {("Ich", "1"), ("mag", "1"), ("Dokumente", "1")}
```

Die Ausführungsumgebung fasst die Ergebnisse aller *map* Aufrufe zusammen. Sie erzeugt für jedes gefundene Wort eine Liste von Einsen. Die *reduce* Funktion wird dann für jedes Paar aus einem Wort und der dazu gehörigen „Liste von Einsen“ aufgerufen. Sie addiert die Einsen und produziert so alle Teile des Endergebnisses, die die Umgebung dann noch zu einer Liste zusammenfügt:

```
reduce("bin", {"1", "1"}) = ("bin", "2")
reduce("Dokument", {"1", "1"}) = ("Dokument", "2")
reduce("Dokumente", {"1"}) = ("Dokumente", "1")
reduce("ein", {"1"}) = ("ein", "1")
reduce("Ich", {"1", "1", "1"}) = ("Ich", "3")
reduce("kein", {"1"}) = ("kein", "1")
reduce("mag", {"1"}) = ("mag", "1")
```

Es ist erkennbar, dass alle Aufrufe von *map* und *reduce* unabhängig voneinander sind. Die Berechnungen können auf einem Cluster verteilt ausgeführt werden. Außerdem zeigt das Beispiel, wie viel Arbeit eine MapReduce Ausführungsumgebung – in dieser Arbeit Hadoop – dem Programmierer erspart. Es übernimmt das Aufteilen der Eingabemenge in kleine Teilberechnungen, das Zusammenfassen der Teilergebnisse und das Zusammenfügen des Endergebnisses.

Der folgende Abschnitt zeigt, wie diese Schritte mit Hadoop realisiert werden.

3.1.2. MapReduce in Hadoop

Um später die Implementierungsdetails von streams-mapred verstehen zu können, wird an dieser Stelle eine vollständige Hadoop Anwendung für das WordCount Beispiel verfasst.

In Hadoop wird eine MapReduce Berechnung als *Job* bezeichnet. Jedem Job muss eine Klasse zugeteilt werden, die das generische `Mapper` Interface aus dem `org.apache.hadoop.mapred` Paket implementiert – einen Mapper. Zusätzlich ist es in den meisten Fällen sinnvoll, eine Reducer-Klasse zu erstellen, die entsprechend das generische `Reducer` Interface implementiert. Ist kein Reducer konfiguriert, werden die Ergebnisse der Mapper als Endergebnisse interpretiert und ausgegeben. Auch das kann sinnvoll sein, wenn die *map*-Teilergebnisse nicht weiter zusammengefasst oder nachbearbeitet werden müssen.

Der Quelltext 3.4 zeigt die Konfiguration eines Jobs in Hadoop. Die konfigurierende Klasse (hier `class WordCount`) wird im Rahmen von Hadoop *Driver* genannt.

Zunächst konfiguriert dieser die Eingabe. Da Textdokumente betrachtet werden sollen, wird das `TextInputFormat` von Hadoop verwendet. Es ist eines von vielen mitgelieferten Eingabeformaten. Es ist jedoch auch möglich eigene Formate zu implementieren. `TextInputFormat` liest Textdokumente Zeilenweise ein und erzeugt Eingabepaare (wie $(k1, v1)$ aus Abschnitt 3.1.1) bestehend aus der Position im Dokument und Zeileninhalt.

Dann wird das Ausgabeformat festgelegt. Das simple `TextOutputFormat` schreibt je Ausgabepaar $(k3, v3)$ eine Zeile mit Textrepräsentationen von Schlüssel und Wert in eine Textdatei. Da Wörter gezählt werden sollen, wird für den Ausgabeschlüssel `Text` als Typ festgelegt und `IntWritable` für den Wert. Der Eingabepfad ist meist ein Verzeichnis mit vielen großen Dateien. Für WordCount lässt sich beispielsweise der Inhalt der Artikel von Wikipedia verwenden. Der Ausgabepfad wird nach der Ausführung eine Textdatei für jeden Reducer enthalten.

`Text` und `IntWritable` sind *Wrapper*-Klassen für `Integer` und `String`, die das `Writable` Interface aus `org.apache.hadoop.io` implementieren. Das `Writable` Interface ist eine von Hadoops Möglichkeiten Objekte zu serialisieren, um sie in Dateien abzulegen oder über das Netzwerk zu versenden. In Hadoop müssen alle Typen, mit denen das geschieht, `Writable` implementieren. Für sehr viele Typen enthält das Framework `Writable`s, aber auch eigene Typen können es implementieren und

```
class WordCount {
  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);

    conf.setInputFormat(TextInputFormat.class);
    FileInputFormat.setInputPaths(conf,
      new Path("/some/input/directory/in/the/hdfs/"));

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

    conf.setOutputFormat(TextOutputFormat.class);
    FileOutputFormat.setOutputPath(conf,
      new Path("/some/output/directory/in/the/hdfs/"));

    conf.setMapperClass(Map.class);
    conf.setReducerClass(Reduce.class);

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

Abbildung 3.4.: Konfiguration und Start eines Jobs in Hadoop[1]

werden dann automatisch von Hadoop zwischen den Rechenknoten transferiert oder im HDFS gespeichert.

Zum Abschluss der Konfiguration werden die Mapper und Reducer Klassen gesetzt. Diese Klassen sind eigene Implementierungen, die in den nächsten Listings folgen und besprochen werden.

Quelltext 3.5 zeigt die Implementierung eines Mappers für das WordCount Problem. Das generische `Mapper` Interface ist durch vier Typen parametrisiert. Die ersten beiden Parameter sind die Typen von Schlüssel und Wert der Eingabepaare der zu implementierenden *map*-Funktion. Sie müssen mit den Typen des gewählten `InputFormats` übereinstimmen. Der dritte und vierte Parameter sind die gewählten Ausgabetypen.

Außerdem schreibt das Interface die `map`-Methode vor. Sie wird von der Ausführungsumgebung für jedes Eingabepaar erneut aufgerufen. Die ersten beiden Parameter sind Schlüssel und Wert des Paares. Der zweite Parameter ist ein `OutputCollector`, der verwendet wird, um Ausgabepaare an Hadoop weiterzureichen. Der `Reporter` wird verwendet, um der Umgebung Statusmeldungen mitzuteilen. Die `map`-Methode für WordCount teilt die ihr übergebene Zeile in Wörter auf und erzeugt wie gehabt für jedes Wort ein Ausgabepaar aus Wort und einer „1“.

[1] Beispiel aus der offiziellen Hadoop Dokumentation:
https://hadoop.apache.org/docs/r1.1.2/mapred_tutorial.html – Abgerufen am 3.12.2013 13:37

```
public class Map
    implements Mapper<LongWritable, Text,
                      Text, IntWritable> {
  @Override
  public void map(LongWritable key, Text value,
      OutputCollector<Text, IntWritable> output,
      Reporter reporter) throws IOException {

    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      output.collect(
        new Text(tokenizer.nextToken()),
        new IntWritable(1));
    }
  }
}
```

Abbildung 3.5.: Mapper für WordCount in Hadoop

Quelltext 3.6 zeigt die Implementierung eines passenden Reducers. Das generische `Reducer` Interface ist wie das des Mappers durch vier Typen parametrisiert: Eingabeschlüssel und -wert und Ausgabeschlüssel und -wert.

Die `reduce`-Methode erhält als ersten Parameter einen der Ausgabeschlüssel der von Hadoop zusammengefassten Zwischenergebnisse aller Mapper und einen `Iterator` über alle Werte zu diesem Schlüssel $(k2, list(v2))$. Außerdem werden – wie beim Mapper – auch `reduce` ein `OutputCollector` und `Reporter` übergeben. Die `reduce`-Methode zu WordCount iteriert über und addiert die ihr übergebenen Einsen. Sie erzeugt ein Ausgabepaar $(k3, v3)$ mit dem Wort und der Anzahl seiner Vorkommen in allen Dokumenten der Eingabe.

Das aus den hier gezeigten drei Klassen bestehende Programm ist so bereits vollständig und kann auf einem Hadoop Cluster ausgeführt werden. Es muss zu diesem Zweck zunächst zu einer `jar` Datei kompiliert werden. Hierzu gibt es mehrere Möglichkeiten. Im Rahmen dieser Arbeit werden alle Projekte als *Apache Maven* Projekt angelegt und können so in einer Kommandozeile zu einem ausführbaren Programm zusammengefasst werden:

```
$ mvn package
```

Anschließend lässt sich das im Framework enthaltene `hadoop` Kommandozeilenwerkzeug verwenden, um den Job auf einem zuvor konfigurierten Cluster auszuführen (der dritte Parameter ist die auszuführende `main`-Klasse):

```
$ hadoop jar wordcount.jar somepackage.WordCount
```

Ist die Berechnung beendet, befindet sich im konfigurierten Ausgabeverzeichnis im HDFS eine Textdatei, die eine Zeile pro Wort mit der dazugehörigen Anzahl enthält.

```
public class Reduce
     implements Reducer<Text, IntWritable,
                        Text, IntWritable> {
  @Override
  public void reduce(Text key, Iterator<IntWritable> values,
      OutputCollector<Text, IntWritable> output,
      Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```

Abbildung 3.6.: Reducer für WordCount in Hadoop

Um verstehen zu können, was bei der Ausführung eines solchen Jobs auf dem Cluster passiert, muss zunächst das HDFS genauer betrachtet werden.

3.1.3. Hadoop Distributed File System

Das HDFS ist ein Dateisystem zum verteilten Speichern von Daten. Es wird besonders im Hinblick auf die Verwendung für MapReduce Programme entwickelt. Es bietet eine Verteilung von Dateien in einem Cluster, die vom Nutzer nur wenig beeinflusst und wahrgenommen wird. Legt man Dateien darin ab, merkt man im Allgemeinen nicht, dass sogar einzelne Dateien auf mehreren Rechnern gespeichert werden. Dieses implizite Verteilen der Daten macht das Dateisystem interessant, weil es den Verwaltungsaufwand der Datenhaltung eines Big Data Systems verringert.

Folgende Ziele und Annahmen sind in seine Architektur eingeflossen und werden darin widergespiegelt [6]:

- Hardware Ausfälle

  In sehr großen Rechnerclustern mit vielen Komponenten sind Ausfälle der Hardware nach einer gewissen Laufzeit wahrscheinlich. Ein verteiltes Dateisystem soll auftretende Fehler erkennen und effizient und automatisch beheben.

- Hoher Datendurchsatz

  Im Gegensatz zu allgemeineren, alltäglich verwendeten Dateisystemen soll HDFS vor allem einen hohen sequenziellen Datendurchsatz statt kurzen Latenzen beim Dateizugriff bieten. Das bedeutet unter anderem, dass sich HDFS gut dafür eignet, große Dateien am Stück einzulesen. Sollen jedoch viele kleine Dateien verarbeitet werden, wird die hohe Latenz der vielen einzelnen Zugriffe zum Engpass der Berechnung.

- Sehr große Datensätze

  Das Dateisystem soll darauf ausgelegt werden, sehr große Datensätze zu speichern. Dies gilt sowohl für die Größe einzelner Dateien im Bereich von Giga- bis Terabyte, als auch für die Anzahl an zu speichernder Dateien von bis zu Dutzenden Millionen. Der Datenzugriff auf große Dateien soll dabei auf das Cluster verteilt werden und der Durchsatz mit der Größe des Clusters skalieren.

- Hilfreiche Einschränkungen beim Schreiben

  Es wurde zunächst angenommen, dass eine Datei nur einmal geschrieben und häufig gelesen wird (*"write-once-read-many"*). Seit Version 0.19.0 unterstützt HDFS auch das Anhängen von Daten an bestehende Dateien [11]. Durch diese Einschränkung wird die verteilte Speicherung simpler und außerdem kann der Datendurchsatz verbessert werden.

  Das *write-once-read-many* Prinzip gilt sowohl für den eigentlichen Vorgang des Schreibens einer Datei, der nur einmal erfolgen kann (nur Anhängen ist erlaubt), als auch für die Anzahl der gleichzeitig stattfindenden Zugriffe. Eine Datei kann nur einmal zum Anhängen geöffnet, aber von vielen gleichzeitig gelesen werden.

- Anwendungen zu den Daten schicken

  Das code-to-data-Prinzip ist schon bei der Entwicklung des Dateisystems in die Überlegungen eingeflossen. HDFS stellt Schnittstellen bereit, um Anwendungen zu den gerade benötigten Daten zu transferieren.

- Portierbarkeit auf unterschiedliche Plattformen

  HDFS wurde so entworfen, dass es gut auf verschiedene Hardware und Software Plattformen portiert werden kann. Es kann so schneller verbreitet werden und kann zu einer guten Wahl für ein verteiltes Dateisystem werden.

Architektur

Das HDFS setzt auf eine *master-slave*-Architektur. In einem typischen HDFS-Cluster gibt es einen einzigen *NameNode* (*master*) und sehr viele *DataNodes* (*slaves*).

Das Dateisystem stellt eine für Dateisysteme typische Hierarchie aus Ordnern und Dateien bereit, die vom NameNode verwaltet wird. Zu jedem Ordner und jeder Datei werden dort Metadaten, wie beispielsweise Name, Zugriffsrechte oder Zugriffszeiten, gespeichert. Der Dateiinhalt wird vom Dateisystem in Blöcke unterteilt, welche typischerweise 64 oder 128 Megabytes groß sind. Diese Blöcke werden auf das Cluster verteilt, also auf verschiedenen Knoten abgelegt. Jede Datei liegt so auf verschiedenen Rechnern gleichzeitig.

Diese Blöcke werden außerdem im Cluster repliziert, das heißt jeder Block wird typischerweise auf insgesamt drei verschiedenen DataNodes abgelegt. Auf jedem DataNode entspricht ein Block zwei Dateien im nativen Dateisystem: Je eine Datei mit zusätzlichen Metadaten, wie der Cluster-globalen Block-ID und einer Datei mit

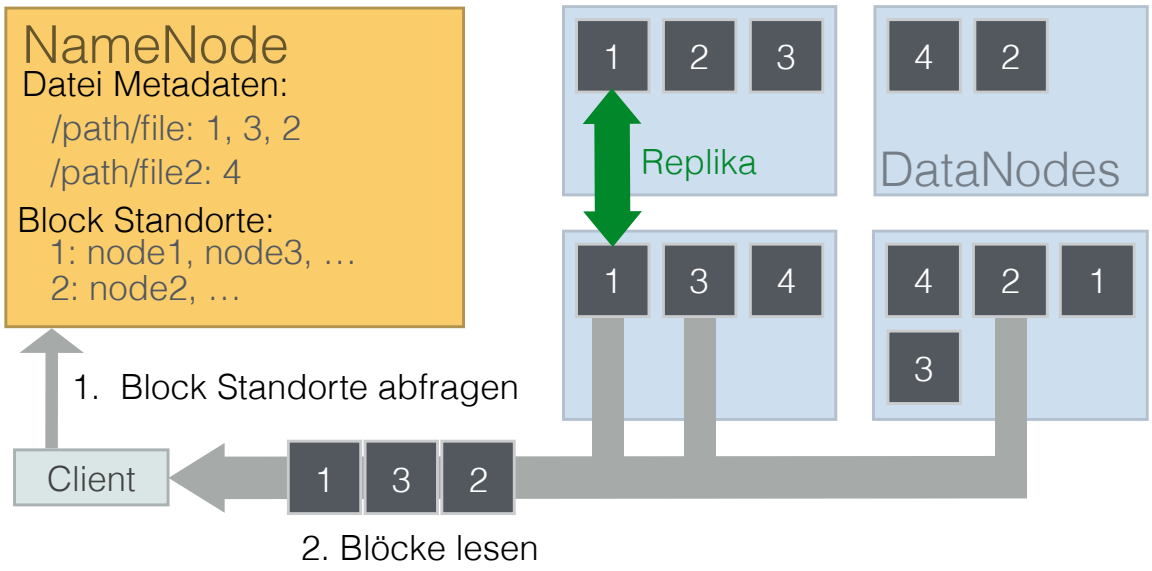


Abbildung 3.7.: Schematische Darstellung von HDFS. Es sind ein NameNode (gelb), vier DataNodes (hellblau) mit replizierten Blöcken (dunkelgrau) und ein Client zu sehen. Der Client ließt die Datei `/path/file` aus, indem er beim NameNode die Blöcke erfragt (1) und von den DataNodes ausliest (2).

dem eigentlichen Inhalt des Blockes. Fällt einer der DataNodes aus, erkennt das der NameNode. Dann werden die Blöcke, die nun nicht mehr genügend Replikate im Cluster haben, von den übrigen neu repliziert.

Die Verteilung der Blöcke ist eine der Eigenschaften von HDFS, auf die bei der Entwicklung viel Wert gelegt wurde. So werden Blöcke beispielsweise automatisch auf möglichst viele verschiedene *Racks* des Clusters verteilt, sodass der Ausfall eines kompletten Racks nicht zum Verlust der nur darin gespeicherten Blöcke führt. Außerdem sorgt der NameNode automatisch dafür, dass die Blöcke möglichst gleichmäßig auf den Cluster verteilt werden, sodass die Festplatten möglichst gleich ausgelastet werden. Dies führt zudem dazu, dass eine Datei selten "weit entfernt" (Entfernung im Netzwerk) von einem beliebigen Knoten ist.

Zugriff auf das HDFS erfolgt meist mit Hilfe der Hadoop Java Programmierschnittstelle oder dem `hadoop` Kommandozeilenwerkzeug. Diese und andere Schnittstellen bezeichnet man als HDFS-*Clients*. Sie stellen die für ein Dateisystem typischen Operationen *Schreiben*, *Lesen* und *Löschen* (und viele hilfreiche Kombinationen daraus) bereit. Beim Ablauf dieser Operationen bemerkt der Nutzer im Allgemeinen nichts davon, dass die Metadaten und der Inhalt einer Datei auf verschiedenen Rechnern verteilt liegen oder dass der Inhalt einer Datei repliziert wurde.

Alle im Folgenden beschriebenen Vorgänge werden vom jeweiligen Client und nicht seinem Benutzer ausgeführt. Der Quelltext 3.8 zeigt beispielhaft, wie Lesen und Schreiben in Java implementiert wird.

Der Schreibvorgang beginnt – wie alle Operationen – beim NameNode. Der Client fragt dort nach einem neuen Block und der NameNode sucht die geeignetsten

DataNodes, um den Block repliziert zu speichern. Diese DataNodes bilden eine Datenpipeline. Der Client öffnet eine direkte Verbindung zu nur einem der DataNodes, der die gesendeten Daten an den nächsten Knoten weiterleitet. Ist der Block mit Daten gefüllt, wird beim NameNode der nächste Block angefordert, so lange bis die Datei geschrieben ist.

Beginnt ein Client eine Datei zu lesen, fragt er den NameNode nach den zugehörigen Blöcken (in der Art sortiert, dass ihre Reihenfolge dem Inhalt der eigentlichen Datei entspricht) sowie der Positionen ihrer Replikate (sortiert, sodass die Netzwerkentfernung zum Client möglichst gering ist). Anschließend versucht er direkte Verbindungen zu den DataNodes aufzubauen und die Blöcke nacheinander zu lesen (siehe Abbildung 3.7). Ist einer der Knoten ausgefallen (der NameNode kennt nicht zu jedem Zeitpunkt den Status aller DataNodes), wird das nächste Replikat des Blockes angefordert.

```
import org.apache.hadoop.fs.*;
// [...]
FileSystem fs = FileSystem.get(new JobConf());
Path file = new Path("hdfs://dfsnamenode:8020/some/file");

void read() throws IOException {
        // FSDataInputStream implements java.io.InputStream
        FSDataInputStream inputStream = fs.open(file);

        // inputStream wie sonst auch einlesen und schliessen
}

void write() throws IOException {
        // FSDataOutputStream implements java.io.OutputStream
        FSDataOutputStream outputStream = fs.create(file);

        // in den outputStream wie immer schreiben und schliessen
}
```

Abbildung 3.8.: Lesen und Schreiben einer Datei aus dem HDFS in Java

Quelltext 3.8 zeigt, wie mit Hilfe der im Hadoop Framework enthaltenen Klassen `FSDataInputStream` und `FSDataOutputStream` Daten aus dem HDFS gelesen werden können. Wie im Abschnitt 3.1.2 herausgearbeitet wurde, ist das jedoch selten nötig. Im Rahmen eines Hadoop MapReduce Programms übernimmt ein geeignetes `InputFormat` dieses Detail. Dennoch zeigt der Code-Ausschnitt, dass ein HDFS sich für den Nutzer fast wie ein klassisches Dateisystem verhält.

3.1.4. Ausführung eines MapReduce Jobs

An dieser Stelle soll kurz Schritt für Schritt betrachtet werden, was passiert, wenn die in Abschnitt 3.1.2 vorgestellte WordCount Anwendung ausgeführt wird.

Dort wurde ein kompletter Ordner als Eingabe konfiguriert. Hadoop verwendet dann automatisch alle darin abgelegten Dateien als Eingabe. Jede dieser Dateien belegt im HDFS mindestens einen, vermutlich jedoch mehrere Blöcke, welche auf mehrere Rechner verteilt und repliziert wurden.

Im besten Falle ist es möglich, dass jeder der Blöcke von einem Aufruf des Mappers bearbeitet wird. Dies ist jedoch nur dann möglich, wenn das gegebene `InputFormat` *zerteilbar* (*splittable*) ist. Ansonsten wird für jede Datei ein Mapper auf einem Knoten, der den ersten Block der Datei enthält, aufgerufen. Die restlichen Blöcke müssen von anderen Knoten dorthin übertragen werden, was das Netzwerk zum Flaschenhals der Berechnung machen kann.

Textdateien sind Beispiele für zerteilbare Daten. Es muss beispielsweise nur nach dem nächsten Wort oder Zeilenumbruch gesucht werden. Binäre Daten können in einigen Fällen nicht so einfach zerteilt werden. Dies ist einer der Gründe, warum das FITS-Format der FACT-Daten im Abschnitt 5.2 extra betrachtet wird. Mehr zu zerteilbaren Formaten im Allgemeinen folgt außerdem in Abschnitt 5.1.4.

Jeder Hadoop-Job (und Cluster) ist für eine maximale Anzahl gleichzeitig laufender Map- und Reduce-Schritte konfiguriert. Häufig ist die Anzahl der parallelen Mapper etwas höher als die Anzahl der Knoten im Cluster, sodass Mehrkern- und Multiprozessorsysteme ausgenutzt werden. Währenddessen wird in vielen Fällen nur ein einziger Reducer eingesetzt, da sonst unter Umständen die Ausgabe der Reducer noch nachträglich zusammengefügt werden muss.

Zunächst werden also so viele Mapper wie möglich für jeweils einen Block – oder eine Datei, wenn sie nicht zerteilbar ist – der Eingabedateien gestartet. Hadoop sucht dazu die Knoten mit möglichst geringer Auslastung und überträgt die komplette Anwendung auf den jeweiligen Knoten. Dort startet Hadoop eine neue JVM und darin den Mapper, sowie das gegebene `InputFormat`.

Das `InputFormat` instanziiert einen `RecordReader`, welcher für das eigentliche Einlesen des Blockes verantwortlich ist. Der generisch parametrisierte RecordReader muss zum Typ des Eingabe-Schlüssel-Werte-Paar des Mappers passen. Er liest aus dem Eingabeblock nach und nach Daten (zum Beispiel wie im Quelltext 3.8) und gibt Schlüssel-Werte-Paare zurück. Diese werden von Hadoop an den Mapper übergeben, der darauf Berechnungen anstellen kann. Pro Eingabepaar wird die `map`-Methode einmal aufgerufen (Quelltext 3.5). Ist ein Block abgearbeitet, wird wie zuvor ein neuer Knoten gesucht und ein neuer Mapper gestartet bis keine Eingabedaten mehr bearbeitet werden müssen.

Die über den `Collector` gesammelten Ergebnisse (Schlüssel-Wert) werden immer sofort serialisiert. Das bereits erwähnte `Writable` Interface kann diesem Zweck dienen, da es Methoden zum (De-)Serialisieren vorschreibt. Die Ergebnisse werden so über das Netzwerk zu einem Reducer übertragen. Ein Reducer läuft auf einem beliebigen freien Knoten. Sind mehrere Reducer konfiguriert müssen die Schlüssel zuvor partitioniert und so auf die Reducer verteilt werden (siehe `Partitioner`-Interface). Beim Reducer werden die Paare deserialisiert und nach dem „Sortieren und Zu-

sammenfassen“-Schritt an den Reducer übergeben. Die `reduce` Methode wird pro Schlüssel einmal mit der Liste der damit verbundenen Werte aufgerufen (Quelltext 3.6).

Die vom Reducer gesammelten Ergebnisse werden dann vom konfigurierten `OutputFormat` geschrieben. Häufig sind die Ergebnisse eines MapReduce Programms direkt die Eingabedaten eines weiteren MapReduce Schrittes. Daher bietet sich für die Ausgabe wieder das HDFS an.

3.2. Das streams Framework

Das *streams* Framework [5] ist eine Java Bibliothek für die Datenverarbeitung. Seine Kernidee setzt auf das Prinzip von *Datenströmen*, die in logische *Datenpakete* unterteilt und nach und nach von sogenannten *Prozessoren* verarbeitet werden. Es ist auf Java-Ebene bewusst einfach gehalten und damit einfach zu erlernen.

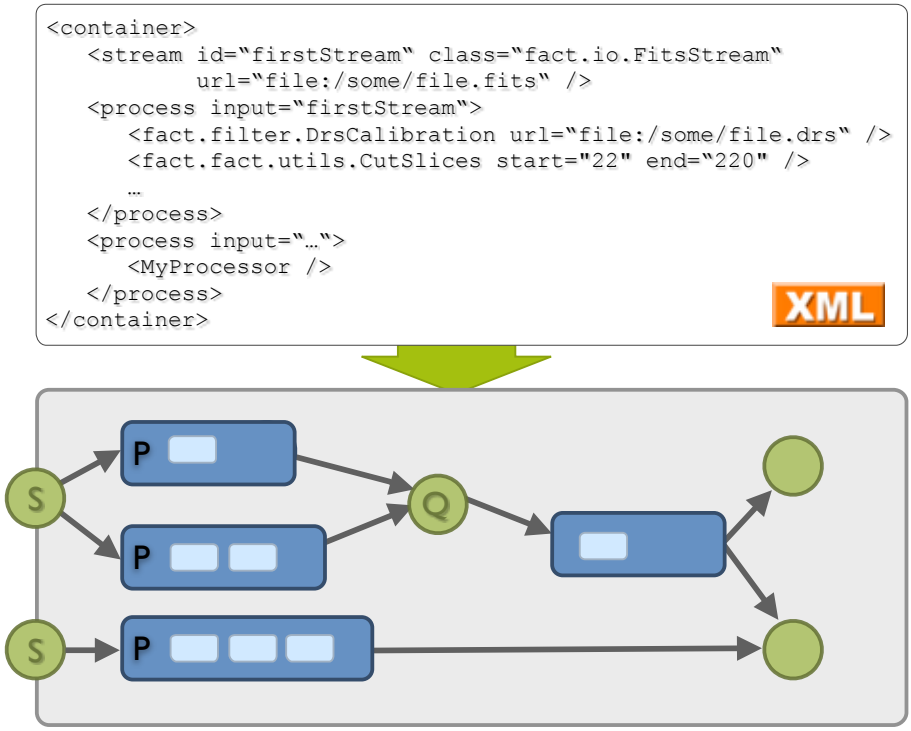


Abbildung 3.9.: Schematisches Vorgehen von streams [5]: Aus XML Definitionsdateien wird ein Datenflussgraph aus Strömen (S), Prozessen (P) mit enthaltenen Prozessoren und Warteschlagen (Q) zusammengesetzt und ausgeführt.

Streams ist besonders geeignet für die Echtzeitverarbeitung kontinuierlich eintreffender Daten. Beispiele sind Twitter-Feeds oder Sensordaten einer Industrieanlage. Es kann jedoch auch problemlos und sinnvoll für die Analyse von Bestandsdaten auf einer Festplatte eingesetzt werden – wie zum Beispiel der FACT-Daten. Für die

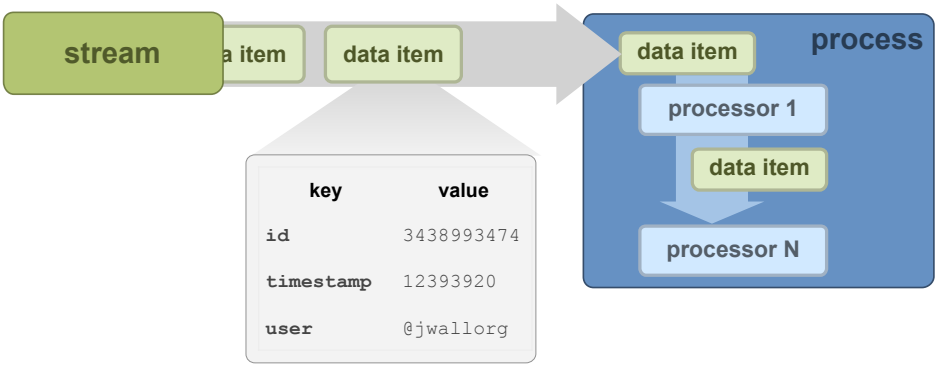


Abbildung 3.10.: Ablauf eines streams-Prozesses [5]: Aus einem Strom werden nach und nach Datenpakete abgefragt und von den enthaltenen Prozessoren abgearbeitet.

FACT-Analyse existiert bereits das fact-tools-Projekt, das das streams Framework für die Analyse verwendet.

Das Framework hat sich zum Ziel gesetzt, eine "adäquate Abstraktionsschicht bereitzustellen, die es dem Benutzer ermöglicht, streaming-Prozesse zu entwerfen, ohne Code zu schreiben". Zu diesem Zweck setzt es auf XML-Definitionsdateien, in denen in einem Top-Level *Container* aus *Strömen*, in *Prozesse* gruppierte *Prozessoren*, *Warteschlangen* (*Queue*) und *Services* ein „Datenflussgraph“ modelliert wird, wie in Abbildung 3.9 zu sehen. Eine solche XML-Datei kann mit Hilfe der *streams-runtime* direkt in einer Java VM ausgeführt werden. Erweiterungen wie *streams-storm* erlauben es jedoch auch, den modellierten Datenfluss in eine Storm[2]-Topologie [13] zu übersetzten und auf einem solchen Cluster auszuführen.

Die für diese Arbeit wichtigen streams-Elemente sind Datenpakete, Ströme und Prozesse mit ihren Prozessoren (siehe Abbildung 3.10).

3.2.1. Datenpakete

Ein Datenpaket (*data item*) ist eine logische Einheit eines Datenstroms. Es kann beispielsweise ein Tweet eines Twitter-Feeds, eine Zeile einer Textdatei oder aber ein FACT-Ereignis aus einer FITS-Datei sein. Jedes Paket wird durch eine Reihe an Schlüssel-Werte-Paare repräsentiert. Ein Datenpaket der fact-tools enthält beispielsweise unter anderem einen Schlüssel `"Event- Num"` mit inkrementell steigenden Nummern und unter dem Schlüssel `"data"` die Teleskop-Kameradaten als `float`-Array.

Auf Java-Ebene wird ein Datenpaket durch eine Instanz von `stream.Data` repräsentiert. `Data` ist eine Implementierung des aus Java bekannten `Map` Interfaces und kann innerhalb der Anwendung genauso verwendet werden.

[2]Storm Projekt Homepage: http://storm-project.net/ – Abgerufen am 25.11.2013 12:44

```
class LineReaderStream extends AbstractStream {
        private String inputFile;
        public String getInputFile() { return inputFile; }
        public void setInputFile(String inputFile)
        { this.inputFile = inputFile; }

        private BufferedReader in;

        public void init() throws Exception {
                super.init();
                in = new BufferedReader(
                        new FileReader(inputFile));
        }

        public void finish() {
                in.close();
        }

        public Data readNext() throws Exception {
                String line = in.readLine();
                Data data = DataFactory.create();
                data.put("line", line);
                return data;
        }
}
```

Abbildung 3.11.: Implementierung eines einfachen Datenstroms

3.2.2. Ströme

Ein Strom (*stream*) kapselt das Einlesen einer Datenquelle und das Interpretieren von dessen Format. So führt beispielsweise ein `SQLStream` eine SQL `SELECT`-Abfrage aus, iteriert zeilenweise über das Ergebnis und erzeugt Datenpakete, deren Schlüssel-Werte-Paare den Feldern des Ergebnisses entsprechen. Die fact-tools enthalten einen `FitsStream`, der aus einer FITS-Datei die erwähnten FACT-Ereignis-Datenpakete ausließt.

Ströme werden im XML als Eingabe eines oder mehrerer Prozesse konfiguriert. Das Framework sorgt dafür, dass die Datenpakete jedes Stroms von diesen Prozessen nach und nach abgefragt (*pull*-Prinzip) und verarbeitet werden. Jeder Strom ist eine Implementierung des `stream.io.Stream`-Interfaces. Verwendet man `AbstractStream` als Basisklasse, muss nur die `readNext`-Methode implementiert werden, die das nächste Datenpaket ausliest und zurückgibt.

Viele Elemente von streams können über die XML Datei konfiguriert werden. Das gilt auch für Ströme. Sie müssen dafür *Getter*- und *Setter*-Methoden für seine

```
public class LongestLineProcessor implements StatefulProcessor {
        private String key;
        public String getKey() { return key; }
        public void setKey(String key) { this.key = key; }

        private String longestLine;

        public void init(ProcessContext context)
                        throws Exception {
                longestLine = "";
        }

        public void resetState() throws Exception {
                longestLine = "";
        }

        public Data process(Data item) {
                String line = (String) item.get(key);
                if (line == null) {
                        return null; // item verwerfen
                }

                // laengste Zeile merken
                if (line.length() > longestLine.length()) {
                        longestLine = line;
                }

                // Speichere die bisher laengste Zeile
                // im aktuellen Datenpaket
                item.put("currentLongestLine", longestLine);

                return item;
        }

        public void finish() throws Exception {
                System.out.println("longestLine="+longestLine);
        }
}
```

Abbildung 3.12.: Implementierung eines Prozessors

Parameter nach Java-Konventionen bereitstellen. Diese können dann über XML-Attribute gesetzt werden.

Der Quelltext 3.11 zeigt die Implementierung eines einfachen Datenstroms für Textdateien. Das `inputFile` kann mit Hilfe der gezeigten Getter- und Setter-Methoden in der XML-Datei konfiguriert werden.

3.2.3. Prozesse und Prozessoren

Ein Prozess (*process*) ist im Kontext von streams eine Gruppe von Prozessoren (*processor*). Ein Prozess ist ein Programmbestandteil des Frameworks, wird also nicht vom Benutzer implementiert. Die Prozessoren können hingegen durch den Entwickler verfasst werden. Sie sind die rechnenden Bestandteile einer streams-Anwendung.

Die Prozessoren eines Prozessors bilden eine sequenzielle Liste. Der erste Prozessor bekommt ein vom Eingabestrom abgefragtes Datenpaket und kann es verändern, daraus ein neues Paket erzeugen oder es verwerfen. Das Paket wird dann vom Framework an den nächsten Prozessor weitergegeben und durchläuft so den Prozess, bis es über eine Warteschlange an einen anderen Prozess weitergegeben wird oder das Ende des Datenflusses erreicht hat.

Ein Prozessor ist eine Implementierung des `stream.Processor` Interfaces. Es definiert nur die `process`-Methode, die das aktuelle Paket übergeben bekommt und verändert wieder zurückgegeben werden kann. Jeder Prozessor kann wie ein Strom mit Hilfe von Gettern und Settern konfigurierbar gemacht werden.

Muss ein Prozessor einen Zustand verwalten, also Informationen unabhängig von den Verarbeitungsschritten speichern, muss das `StatefulProcessor` Interface implementiert werden. Es schreibt zusätzlich die `init`, `resetState` und `finish` Methoden vor, die vom Framework an den jeweiligen Zeitpunkten aufgerufen werden.

Quelltext 3.12 zeigt die Implementierung eines Prozessors, der die vorgestellten Aspekte des Frameworks verwendet. Der Prozessor betrachtet alle Datenpakete und berechnet zu jedem Zeitpunkt die längste Zeichenkette, die in den Paketen unter dem Schlüssel `key` gespeichert ist.

Zu diesem Zweck wurde der `key`-Parameter mit Hilfe geeigneter Getter und Setter konfigurierbar gemacht. Weil der Prozessor sich die aktuelle längste Zeichenkette merken muss, ist der Prozessor zustandsbehaftet und implementiert daher das entsprechende Interface.

3.2.4. Konfiguration und Ausführung

Eine XML Konfiguration, die den `LineReaderStream` sowie den `LongestLine- Processor` verwendet, könnte wie im Quelltext 3.13 aussehen.

Der Strom spezifiziert seine `id`, die benötigt wird, um ihn mit Prozessen zu verbinden, und seine Klasse. Außerdem bekommt er den Eingabeparameter `inputFile` übergeben. Die dabei gezeigte ${property}-Syntax wird vom streams Framework automatisch eingesetzt. Eine derartige Variable kann später auch mit einem Kommandozeilenparamter übergeben werden.

Der damit verbundene Prozess enthält zwei Prozessoren. Zunächst wird jedes Datenpaket vom `LongestLineProcessor` verarbeitet. Außerdem wird der im Framework enthaltene `Message` Prozessor verwendet um eine Ausgabe in der Kommando-

zeile zu erzeugen. Im `message`-Parameter dieses Prozessors ist eine weitere streams-eigene Sytax zu sehen. Alle Prozessorparameter, die dieser %{data.key}-Syntax folgen, werden während der Laufzeit automatisch durch die Werte des gerade verarbeiteten Datenpakets ersetzt.

```
<container>
  <property name="infile" value="/some/file.txt" />
  <stream id="lineStream"
    class="myPackage.LineReaderStream"
    inputFile="${infile}" />
 <process input="lineStream">
    <myPackage.LongestLineProcessor key="line" />
    <stream.logger.Message
      message="currentLongestLine=%{data.currentLongestLine}" />
  </process>
</container>
```

Abbildung 3.13.: Beispielhafte XML Konfiguration einer streams Anwendung

Hat man die Anwendung mit Hilfe von Apache Maven inklusive aller Prozessoren und Ströme zu einer `jar` Datei gepackt (wie in Abschnitt 3.1.2), kann sie ausgeführt werden. Am einfachsten ist das, wenn die streams-runtime in das Projekt eingebunden ist. Sie startet die Anwendung lokal in einer Java VM:

```
$ java -jar myProject.jar stream.run Konfiguration.xml
```

Diese kurze Einleitung lässt bereits erahnen, wie simpel es sein kann, Daten verarbeitende Anwendungen mit Hilfe von streams zu entwickeln. Das Framework bietet noch vielfältigere Möglichkeiten als das hier dargestellte, die wichtigen Aspekte wurden jedoch herausgestellt.

- Durch XML-Konfigurationen bietet streams eine weitere Abstraktionsebene gegenüber reinem Java.
- Es ist einfach zu erlernen. Es lassen sich schnell neue Analyseprozesse entwickeln.
- Die hohe Wiederverwendbarkeit von Strömen und Prozessen lädt zum experimentieren ein.
- Viele Ströme und Prozesse sind bereits im Framework oder seinen Erweiterungen enthalten. Insbesondere die fact-tools sind wichtig für diese Arbeit.

4. Streams und Hadoop: streams-mapred

In Kapitel 2 wird der Big Data Begriff genauer vorgestellt. Es zeigt, dass die Verarbeitung von Big Data neue Herausforderungen an Systeme und Programmierer mit sich bringt. Abschnitt 3.1 geht darauf ein, wie diese Herausforderungen mit Hilfe von Hadoop und MapReduce angegangen werden können.

Abschnitt 3.2 stellt das streams-Framework vor, welches eine Abstraktionsebene zum einfachen Modellieren von Daten verarbeitenden Anwendungen bietet. Es zeigt, wie simpel es sein kann, neue Analysen zu implementieren und dass viele Elemente einer solchen Analyse bereits in streams und dazugehörigen Projekten implementiert wurden. Das fact-tools Projekt enthält viele Programmteile, die die Analyse von FACT-Daten ermöglichen.

Im Rahmen dieser Arbeit soll mit streams-mapred eine Erweiterung für das streams Framework geschaffen werden, die es dem Nutzer ermöglicht, seine Ströme- und Prozessoren ohne großen Aufwand in einer Hadoop-Umgebung verteilt auszuführen. Dieses Kapitel legt dar, wie streams-mapred geplant und implementiert wurde und wie es verwendet werden kann.

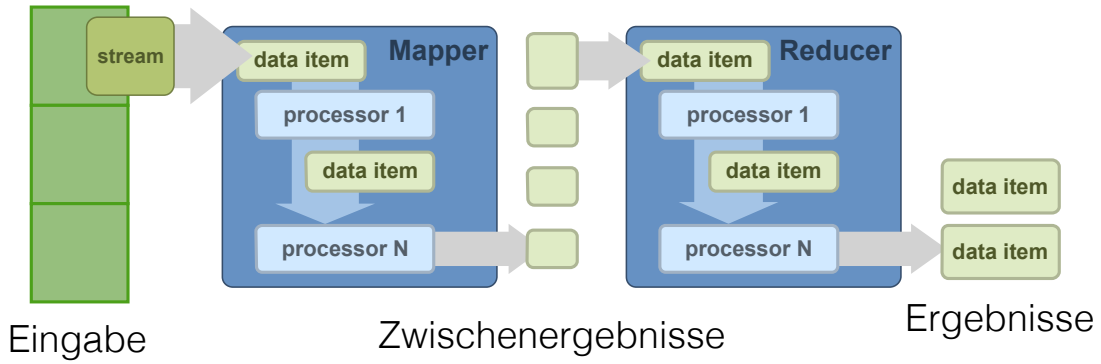


Abbildung 4.1.: Das Konzept von streams-mapred: Mapper und Reducer eines MapReduce Programms sind streams Prozesse. Sie verarbeiten Datenpakete, die von Strömen erzeugt und von der Umgebung verwaltet werden.

Idee

Die grundlegende Idee von streams-mapred ist, jeden Mapper und jeden Reducer als streams-Prozess – also eine Ansammlung von Prozessen – zu betrachten. Diese

müssen von Hadoop mit Hilfe von Strömen mit Datenpaketen versorgt werden, wie in Abbildung 4.1 zu sehen ist.

Ist es möglich, alle Programmbestandteile eines MapReduce Programms aus diesen Bestandteilen zusammenzusetzen, wird dem Programmierer ermöglicht, ohne neuen Code zu verfassen, bestehende streams Programme auf einem Cluster verteilt ausführen zu lassen.

4.1. Vorüberlegungen

Eine zentrale Idee von streams ist, dass in XML-Konfigurationen ein Datenflussgraph entworfen und anschließend ausgeführt werden kann. Dieses Konzept soll auch in streams-mapred beibehalten werden. In Anbetracht der Tatsache, dass das MapReduce Programmiermodell umgesetzt wird, ist eine direkte Wiederverwendung des streams XML-Formats jedoch nicht sinnvoll. Es soll nicht möglich sein, beliebige Datenflüsse zu definieren. Stattdessen soll dem Programmierer das MapReduce Modell direkt auferlegt werden.

Im streams-mapred XML-Format soll sowohl das streams-Framework, aber auch MapReduce wiedererkennbar sein. Aus diesem Grund wurde die Struktur aus einem alles enthaltendem Container, der die Eingabedaten und verschiedene Prozesse mit ihren Prozessoren enthält, beibehalten. Die Elemente spiegeln jedoch die des Hadoop MapReduce Modells wider. Der Container enthält `jobs`. Diese haben ein Eingabe- und Ausgabeverzeichnis, die im HDFS liegen sollten. Die Eingabe soll weiter über Ströme erfolgen. Die Prozessoren werden in zwei speziellen Prozessen gruppiert: Einem `mapper`- und einem `reducer`-Prozess.

Entsprechend dieser Überlegungen hat eine streams-mapred XML-Konfiguration das in Listing 4.2 gezeigte Format.

```
<mapred>
  <job id="meinProzess">
    <input paths="/path/in/hdfs" class="myStreamClass" />
    <output path="/another/hdfs/path" class="writerClass" />

    <mapper outputKeyClass="...">
      <MyProcessor />
    </mapper>

    <reducer outputKeyClass="...">
      <AnotherProcessor />
    </reducer>
  </job>
</mapred>
```

Abbildung 4.2.: Übersicht über das streams-mapred XML Format

Diese Konfiguration soll ähnlich wie in streams ohne Aufwand auf einem Hadoop Cluster gestartet werden können. Das Projekt wird daher mit Hilfe von Apache Maven konfiguriert und verwendet. Bindet man streams-mapred in ein anderes Maven Projekt ein – beispielsweise den fact-tools – kann die daraus entstehende `jar`-Datei direkt wie gewohnt auf einem Cluster ausgeführt werden:

```
$ hadoop jar fact-tools.jar streams.mapred.run Konfiguration.xml
```

Die in fact-tools (und anderen streams-Projekten) gebündelten Prozessoren und Ströme können so direkt als Map- oder Reduce-Schritt einer Berechnung eines Hadoop Clusters verwendet werden.

4.1.1. Einsammeln der Ergebnisse

Wie in den Abschnitten 3.1.1 und 3.1.2 gezeigt, fehlt dem ganzen Konzept noch ein entscheidendes Element: Das Einsammeln der (Zwischen-)Ergebnisse im Map- und Reduce-Schritt durch einen `Collector`. Es wurde entschieden, dass es sinnvoll ist, die Sammlung der Daten explizit durch den Nutzer implementieren zu lassen. Dies geschieht im Rahmen so genannter `CollectorProcessor`en, wie im nächsten Abschnitt dargelegt.

4.2. Implementierungsdetails

Nachdem feststeht, welche Ziele streams-mapred erreichen will, sollen einige der nötigen Implementierungsschritte erläutert werden, die zum Erreichen dieser Ziele führen.

Dieser Abschnitt wird sich zunächst mit dem Übersetzen der XML Konfiguration in einen Hadoop Job auseinandersetzen. Anschließend wird aufgezeigt, was unternommen werden musste, um Ströme und Datenpakete aus dem streams Framework in Hadoop zu verwenden. Es folgt ein Abschnitt über die Implementierung der beiden Berechnungsschritte Map und Reduce, welcher auch weitere Details zu `CollectorProcessor`en enthält.

4.2.1. Konfiguration und Start

In den Vorüberlegungen wurde festgestellt, dass eine streams-mapred Anwendung aus der Kommandozeile heraus ausgeführt werden kann, ohne weiteren Code zu verfassen. Zu diesem Zweck muss eine Klasse mit einer geeigneten `main`-Methode mitgeliefert werden. In streams-mapred ist dies die Klasse `streams.mapred.run`.

Im Prinzip sorgt diese Methode dafür, dass aus dem XML eine passende Hadoop `JobConf` zusammengesetzt wird, wie sie in Listing 3.4 vorgestellt wurde. Die folgenden Schritte werden dafür von `run` angestoßen und lokal auf dem Rechner des Entwicklers ausgeführt.

Im Sinne einer besseren Aufteilung der Verantwortlichkeiten, initialisiert die `run` Klasse nur einige streams-spezifische Dinge und führt anschließend einen für Hadoop typischen `Driver` aus. Dort wird aus den gegebenen Kommandozeilenargumenten eine Basis-`JobConf` gebaut. So wird dem es Nutzer ermöglicht, einzelne Parameter

der Jobs per Kommandozeile zu variieren, was die Verwendung von Skripten, wie beispielsweise `bash`-Skripten, ermöglicht.

Anschließend wird mit Hilfe der `Configuration` Klasse das XML eingelesen und interpretiert. Auf der Basis-`JobConf` aufbauend, werden die nötigen Parameter einer Hadoop Konfiguration aus der XML Datei ausgelesen, sodass eine Liste an gültigen Job Konfigurationen entsteht (Instanzen von `JobConf`). Dabei werden – wie in streams üblich – etwaige Variablen expandiert, sodass auch darüber ein Variieren der Parameter ermöglicht wird.

Zu diesem Zeitpunkt werden zunächst die Dinge konfiguriert, die Hadoop bereits an dieser Stelle voraussetzt. Dazu zählen Eingabe- und Ausgabepfade, Typen der MapReduce Schlüssel- und Werte und alle von streams-mapred mitgelieferten Helfer-Klassen wie `streams.mapred.Map` und `streams.mapred.Reduce` als `Mapper` und `Reducer`, sowie die Eingabe- und Ausgabeformate `DataStreamInputFormat` beziehungsweise `DataWriterOutputFormat`, die im Laufe dieser Arbeit noch genauer betrachtet werden.

In den fertig zusammengesetzten Konfigurationen ist auch das komplette XML Dokument enthalten. Das Mitsenden des XML ist nötig, da das Instanziieren der Ströme und Prozessoren erst während des Ausführungsschrittes auf dem jeweiligen Rechenknoten erfolgen kann.

Schlussendlich kann der `Driver` die Liste von `JobConf`s sequenziell zur Ausführung an den Cluster senden. Schlägt einer der Jobs fehl, werden die folgenden Jobs nicht ausgeführt.

4.2.2. Ausführung

In Abschnitt 3.1.4 wurde die Anatomie eines einfachen MapReduce Jobs in Hadoop gezeigt. Hier soll nun erörtert werden, wie diese Ausführungsschritte in streams-mapred implementiert wurden, sodass der Programmierer seine streams-spezifischen Klassen weiter verwenden kann.

Einlesen der Daten

Zunächst soll das Einlesen der Daten mit Hilfe von Strömen betrachtet werden. In Hadoop müssen zu diesem Zweck zwei Klassen verfasst werden: Ein `InputFormat`, welcher einen passenden `RecordReader` instanziiert. In streams-mapred sind das die Klassen `DataStreamInputFormat` und`DataStreamRecordReader`.

In dieser ersten Version von streams-mapred sind diese Klassen recht simpel gehalten. Das `InputFormat` erledigt keine weiteren Aufgaben als das Instanziieren des `RecordReaders`. An dieser Stelle sollte später eine Möglichkeit ergänzt werden, zerteilbare Ströme zu ermöglichen (siehe Abschnitt 5.1.4).

Der `RecordReader` ist nicht wesentlich komplexer. Bei seiner Instanziierung bekommt er die `JobConf` und den aktuell zu bearbeitenden Block übergeben. Diese Information nutzt er, um den gegebenen Strom zu instanziieren und auf einen entsprechenden Java `InputStream` anzusetzen. Hier wird auch eine genutzte Komprimierung der Daten erkannt und entpackt.

Idealerweise verwenden sowohl streams als auch Hadoop das *pull*-Prinzip, fragen also nach und nach Daten bei ihren jeweiligen Eingabeklassen ab, sodass die `RecordReader.next` Methode genau der `stream.io.Source.read` Methode entspricht.

Da in streams die Prozessoren im Gegensatz zu Hadoops Mappern und Reducern keine Schlüssel-Werte-Paare verarbeiten, sondern nur einfache Datenpakete, wurde an dieser Stelle entschieden, dass beim Einlesen der Daten der erzeugte Schlüssel für die weitere Berechnung irrelevant sein wird und das Datenpaket als Wert an Hadoop weitergereicht wird.

Der Schlüssel ist aus zwei Gründen irrelevant. Zusatzinformationen können genauso gut direkt im Datenpaket, welches ja eine `Map` ist, abgelegt werden. Außerdem wird der Schlüssel an dieser Stelle noch nicht zum Gruppieren der Daten verwendet. Dies ist erst später zwischen Map- und Reduce-Schritt der Fall, wo ein Nutzen des Schlüssels erst wirklich sinnvoll und von streams-mapred unterstützt wird.

Verarbeitung

Die beiden Datenverarbeitungschritte sind in den Helferklassen `Map` und `Recude` im Paket `streams.mapred` implementiert. Diese Schritte haben große Gemeinsamkeiten und dementsprechend eine gemeinsame Basisklasse `MapReduceBase`. Diese Basis implementiert die von Hadoops `Configurable` Interface spezifizierte `configure()` Methode so, dass die Prozessoren des aktuellen Schrittes instanziiert und konfiguriert werden.

Es wird dazu eine mit Hilfe der in streams implementierten `ProcessorFactory` eine`ProcessorList` erstellt. Dazu müssen nur die in der `JobConf` eingetragenen und von der Kommandozeile übergebenen Variablen und das XML-Element des aktuellen Jobs übergeben werden. Die eigentliche Arbeit läuft dann in streams ab, sodass der Nutzer exakt gleiches Verhalten zwischen dem Original streams und der Erweiterung streams-mapred erwarten kann.

Eine weitere Gemeinsamkeit der beiden Verarbeitungsschritte ist die Art und Weise, wie Hadoop-spezifische Informationen der MapReduce-Schicht an die Prozessoren übergeben werden. Im Rahmen von streams werden zu diesen Zweck allen `StatefulProcessor`en ein `Context` an ihre `init` Methode übergeben. In streams-mapred wird ein extra `MapReduceContext` erstellt, der Zusatzinformationen enthält. Im Moment sind dies der `Collector`, der `Reporter`, die `JobConf` sowie der aktuelle Schlüssel der Datenpakete.

Unterschiede der beiden Klassen kommen bei der eigentlichen Datenverarbeitung zum Tragen. Im Mapper wird die `map` Methode je Datenpaket genau einmal aufgerufen. Die in ihm enthaltenen Prozessoren werden im Mapper bei seinem Start initialisiert (`init(context)`) und am Ende geschlossen (`finish()`).

Im Gegensatz dazu muss im Reducer beachtet werden, dass die gegebenen Datenpakete zuvor nach ihren jeweiligen Schlüsseln gruppiert wurden. Dies wird den streams Prozessoren dadurch kommuniziert, dass ihre `init` und `finish` Methoden für jede Gruppe erneut aufgerufen werden. Der aktuelle Schlüssel der Gruppe kann vom `MapReduceContext` erfragt werden, falls dies nötig sein sollte.

Einsammeln der (Zwischen-)Ergebnisse

Es ist nicht sinnvoll, einfach alle Datenpakete, die den letzten Prozessor der beiden Berechnungsschritte verlassen, als Ergebnis zu interpretieren und von Hadoop weiterreichen oder ausgeben zu lassen. Zunächst wären die Daten viel zu groß, um über das Netzwerk transferiert zu werden. Am wichtigsten ist jedoch, dass der Nutzer das MapReduce Modell dann nicht vollständig ausnutzten könnte oder müsste.

Aus diesem Grund führt streams-mapred einen weiteren speziellen Typ von Prozessoren ein: Den `CollectorProcessor`. Jeder Map- und Reduce-Schritt muss mindestens einen dieser Prozessoren enthalten, sonst hat der Schritt keine Ausgabe.

Diese Prozessoren sind gewöhnliche `StatefulProcessor`en und enthalten eigentlich keine zusätzlichen Funktionen, außer dass direkter Zugriff auf den `MapReduceContext` gegeben ist. Ihre Existenz unterstützt den Programmierer jedoch beim Entwerfen einer MapReduce Anwendung, weil dadurch beim Entwurf der Verarbeitungsschritte das Modell direkt beachtet werden muss.

Im Verarbeitungsschritt eines `CollectorProcessor`s muss der darin enthaltene Hadoop `Collector` verwendet werden, um Paare aus Schlüsseln und Datenpaketen zu sammeln. Dies können selbstverständlich sowohl die originalen Datenpakete sein, aber auch neu erstellte.

Der in streams-mapred enthaltene `streams.mapred.processors.Collect` ist ein sehr simpler `CollectorProcessor`. Er erhält zwei Parameter `key` und `values`. Er sammelt dann Datenpakete, unter dem Schlüssel `key`. Das können mit Hilfe der aus streams bekannten `%{data.key}`-Syntax (Abschnitt 3.2.4) auch Werte aus dem jeweiligen Datenpaket sein. In `values` sind die Schlüssel enthalten, die aus jedem Datenpaket gesammelt werden sollen.

Datentransfer

Der Datentransfer zwischen den verschiedenen Programmteilen in Hadoop wird in streams-mapred mit Hilfe des von Hadoop vorgeschriebenen generischen `Writable` Interfaces implementiert. Sowohl die Weitergabe der Daten innerhalb eines Berechnungsschrittes – beispielsweise vom `RecordReader` zum Mapper – als auch der Transfer der Daten über das Netzwerk im Cluster – von Mapper zu Reducer –, wird in `Writable`-Implementierungen gekapselt.

Diese Tatsache soll vor dem Nutzer von streams-mapred verborgen werden. Zu diesem Zweck wurde die `DataWritable`-Klasse verfasst. Sie implementiert das `Writable`-Interface und ist zugleich eine Wrapper-Klasse für `stream.Data` auf Basis von Hadoops `ObjectWritable`. Dadurch ist es an allen Stellen in streams-mapred einfach, Datenpakete ein- oder auszupacken. Diese Aufgabe wird von streams-mapred übernommen, sodass Prozessoren und Ströme nur die bekannten streams-Datenpakete zu Gesicht bekommen.

Das `Writable` Interface schreibt die Methoden `readFields` und `write` vor, die von Hadoop zum (de-)serialisieren der Daten verwendet werden. In `DataWritable` wurden diese mit Hilfe von streams' `Serializer` implementiert. Es ist ein Interface, dass spezialisierten (De-)Serialisierung von Datenpaketen erlaubt. `DataWritable`

verwendet im Moment nur die `JavaSerializer` Implementierung, ist jedoch auf diese Art und Weise sehr gut auf andere Serialisierungslösungen vorbereitet.

Datenausgabe

In Hadoop bekommt jeder Reducer (oder wenn keine konfiguriert sind, jeder Mapper) eine einzelne Ausgabedatei im konfigurierten Ausgabeordner zugewiesen. Diese werden von einem `OutputFormat` befüllt, welches jedes Ausgabepaar des Schrittes einzeln übergeben bekommt und in die Datei schreibt. In streams-mapred wurde dafür das `DataWriterOutputFormat` implementiert. Es bietet zweierlei Möglichkeiten. um Daten auszugeben.

Eine Möglichkeit ist, einen gewöhnlichen streams-Prozessor als Ausgabe zu konfigurieren. Dieser muss einen Konstruktor mit einem `OutputStream` als einziges Argument besitzen und hat anschließend die Aufgabe, alle ihm übergebenen Datenpakete in diesen zu schreiben. Der Prozessor darf `StatefulProcessor` implementieren, um sich zu initialisieren und zu schließen. Der Prozessor hat jedoch keinen Zugriff auf die Schlüssel der Ausgabepaare, es sei denn, sie sind zusätzlich in den Datenpaketen enthalten.

Zugriff auf diese Schlüssel erhält man, indem eine Klasse als Ausgabe konfiguriert wird, die das `DataWriter`-Interface aus `streams.mapred.io.writer` implementiert. Hier können verschiedene Methoden überschrieben werden, die zum Beispiel auf den Schlüssel als `WritableComparable` und das Datenpaket als `DataWritable` Zugriff haben. Eine spezialisierte Ausgabe ist so möglich.

4.3. Verwendung

In diesem Abschnitt wird Schritt für Schritt erläutert, wie streams-mapred eingesetzt werden kann, ein bestehendes streams-Projekt um die Möglichkeit zu erweitern, die darin implementierten Analysen auf einem Hadoop Cluster auszuführen. Alle Schritte werden im Rahmen der erwähnten fact-tools ausgeführt werden, welche dann als Beispiel dienen.

Genau wie das streams Framework verwendet streams-mapred Apache Maven für sein Abhängigkeitsmanagement und seine Verteilung. Um streams-mapred in einem solchen Projekt zu verwenden, muss die streams-mapred Abhängigkeit wie in Listing 4.3 zum Projekt hinzugefügt werden.

4.3.1. Konfiguration

Anschließend kann – dem Gedanken folgend keinen Code schreiben zu müssen – bereits mit dem Verfassen einer streams-mapred XML begonnen werden. Listing 4.4 zeigt mögliche Elemente einer solchen Konfiguration. Die gezeigte Konfiguration enthält alle möglichen Elemente. Sie eignet sich damit für eine Kurzbesprechung des Formates. Es wird ein einzelner Job konfiguriert mit der ID "fact-calculation".

Innerhalb des `hadoop`-XML-Tags kann Hadoop konfiguriert werden. In diesem Beispiel wird die maximale Java-Heap-Größe jedes Kind-Tasks (also zum Beispiel jedes

```
<dependencies>
        <!-- [...] -->
        <dependency>
                <groupId>org.jwall</groupId>
                <artifactId>streams-mapred</artifactId>
                <version>0.0.2-SNAPSHOT</version>
        </dependency>
</dependencies>
```

Abbildung 4.3.: Wird die streams-mapred Abhängigkeit zum Maven Projekt hinzugefügt, können die enthaltenen Prozesse auf einem Hadoop Cluster ausgeführt werden.

Mappers und Reducers) auf 2 Gigabyte gesetzt. Die aus dem streams Framework bekannten `property`-Tags können weiterhin verwendet werden. Ihre Werte werden innerhalb jedes Jobs bei Verwendung der `${property}`-Syntax eingesetzt.

Das `input`-Tag spezifiziert durch Kommata getrennte Eingabepfade, sowie den zugehörigen Eingabestrom. Jeder verwendete Strom muss einen Konstruktor definieren, dem nur ein `InputStream` als Parameter übergeben wird. Der gegebene Strom muss dann zwingend verwendet werden. Ein Öffnen einer Datei, die beispielsweise über einen Parameter übergeben wird, ist in streams-mapred nicht sinnvoll, da der `DataStreamRecordReader` den `InputStream` öffnet.

Das `output`-Tag spezifiziert einen Ausgabepfad und eine Ausgabeklasse. Die Ausgabeklasse kann entweder eine Implementierung des `DataWriter`-Interfaces aus streams-mapred sein, oder ein gewöhnlicher Prozessor, der einen Konstruktor mit einem `OutputStream` als einziges Argument akzeptiert.

Anschließend werden die Mapper und Reducer konfiguriert. Beide enthalten eine Liste von Prozessoren, die wie auch in streams sequenziell abgearbeitet werden. In streams-mapred muss zusätzlich mindestens ein `CollectorProzessor`, wie in diesem Beispiel der `Collect`-Prozessor, aus streams-mapred eingesetzt werden.

Auffallend ist dass sowohl bei Mapper und Reducer das `outputKeyClass` Attribut gesetzt wird. Die eingetragene Klasse wird als Ausgabeklasse des jeweiligen MapReduce-Schrittes gesetzt und muss daher Hadoops `WritableComparable`-Interface implementieren. Der Standardwert ist `Text`.

Die Einführung der Möglichkeit einer Änderung der Ausgabeschlüsselklasse ist sinnvoll, weil spezialisierte Lösungen sowohl für die Serialisierung (über das `Writable`-Interface), als auch der Vergleich der Schlüssel (über das `Comparable`-Interface) in Hadoop große Leistungssteigerungen des Jobs erlauben. Gerade die Sortierung und Gruppierung der Schlüssel-Werte-Paare vor dem Reduce-Schritt geht so wesentlicher schneller vonstatten.

4.3.2. Ausführung

Die Ausführung erfolgt wie zu Beginn dieses Kapitels erwähnt, genau wie man es von einer Verbindung aus Hadoop und streams erwarten kann. Zunächst wird das

Projekt kompiliert, in einer `jar`-Datei zusammengefasst und anschließend mit dem Hadoops Kommandozeilenwerkzeug ausgeführt.

In einigen Maven Projekten wird das Maven *Shade* Plugin[1] in Verbindung mit Maven *Profiles* verwendet, um `jar`-Dateien zu bauen, die ihre `main`-Klasse bereits gesetzt haben. Ein solches Projekt stellen auch die fact-tools dar.

In einem solchen Fall, sollte man ein neues Profil erstellen, welches dann `streams.mapred.run` als Startklasse verwendet. Eine streams-mapred Konfiguration der fact-tools lässt sich dann wie folgt auf einem Hadoop Cluster ausführen:

```
$ mvn package -Dmapred
$ hadoop jar target/fact-tools-mapred-VERSION.jar Konfiguration.xml
```

[1] http://maven.apache.org/plugins/maven-shade-plugin/

```
<mapred>
<job id="fact-calculation">
  <hadoop>
    <property>
      <name>mapred.child.java.opts</name>
      <value>-Xmx2G</value>
    </property>
  </hadoop>

  <property name="infile"
    value="/FACT/input/20130102_060.fits.gz.split/" />
  <property name="drsfile"
    value="/FACT/input/20130104_076.drs.fits.gz" />
  <property name="outfile" value="/FACT/output/" />

  <input paths="${infile}" class="fits.io.FitsReader" />
  <output path="${outfile}" class="stream.io.writer.CsvWriter"
    deleteExisting="false" />
  <mapper outputKeyClass="org.apache.hadoop.io.Text">
    <streams.mapred.processors.AddTimestamp
      key="timestamp.start" />
    <streams.mapred.processors.AddTaskAttemptId
      key="mapred.taskAttemptId" />
    <fact.filter.DrsCalibration url="${drsfile}"
      key="Data" outputKey="DataCalibrated" />
    <streams.mapred.processors.AddTimestamp
      key="timestamp.end" />
    <streams.mapred.processors.Collect
      key="%{data.mapred.taskAttemptId}"
      values="mapred.taskAttemptId,timestamp.start,timestamp.end" />
  </mapper>

  <reducer outputKeyClass="org.apache.hadoop.io.Text">
    <streams.mapred.processors.Maximum
      key="timestamp.end" output="max.timestamp.end" />
    <streams.mapred.processors.Minimum
      key="timestamp.start" output="min.timestamp.start" />
    <streams.mapred.processors.Collect
      collectEvery="-1"
      key="%{data.mapred.taskAttemptId}"
      values="min.timestamp.start,max.timestamp.end" />
  </reducer>
</job>
</mapred>
```

Abbildung 4.4.: Eine vollständige XML Konfiguration für streams-mapred

5. Speicherung von Daten in Hadoop

Dieses Kapitel erläutert genauer, welche Aspekte beim Speichern von Big Data beachtet werden sollten, wenn man sie mit dem in Kapitel 3 vorgestellten Apache Hadoop verarbeiten will. Dabei wird gesondert auf die Verarbeitung mit Hilfe des in Kapitel 4 vorgestellten streams-mapred Frameworks eingegangen.

Kurz wurde bereits erwähnt, dass die Verarbeitung bestimmter Daten mit dem Hadoop Framework problematisch sein kann. Das FITS-Format der FACT-Daten ist ein solcher Problemfall. Dieses Kapitel wird daher genauer auf für Hadoop problematische Formate eingehen.

5.1. Mögliche Parameter

An dieser Stelle werden zunächst – ohne Blick auf die FACT-Daten – die Faktoren erörtert, die beachtet werden müssen, will man Daten mit Hadoop verarbeiten. Es wird erläutert, wie sich diese auf die Planung, Implementierung und Performanz eines Systems und Anwendungen auswirken können. Im Verlaufe des Kapitels werden alle beeinflussbaren Parameter anhand der FACT-Daten getestet und überprüft.

5.1.1. Blockgröße

In Abschnitt 3.1.3 wurde erläutert, dass Daten im HDFS in Blöcken abgelegt werden. Diese Blöcke sind immer pro Datei und in den meisten Fällen im ganzen Cluster gleich groß. Die Wahl einer geeigneten Blockgröße ist entscheidend für die Performanz eines Hadoop Jobs und teilweise auch des gesamten Clusters. Das hängt damit zusammen, dass jeder Block von einem Mapper bearbeitet wird (solange sie zerteilbar ist). Die Größe – und damit die Anzahl – der Blöcke hängt also direkt mit der Anzahl und Laufzeit der Mapper zusammen. Jeder Mapper sollte eine Laufzeit vorweisen, die weder zu lang noch zu kurz ist.

Zu lange Laufzeiten können zu zweierlei Problemen führen. Wenn ein sehr lange laufender Job fehlschlägt, zum Beispiel wegen Hardwarefehlern, wird er von Hadoop automatisch auf einem anderen Knoten neu gestartet und seine Ergebnisse verworfen. Längere Laufzeiten erhöhen also unter Umständen die Menge der doppelt ausgeführten Berechnungen. Lange Laufzeiten führen außerdem zu gröberer zeitlicher Granularität. Das kann dazu führen, dass ein stark ausgelastetes Cluster Schwierigkeiten hat, die Auslastung gleichmäßig auf die Rechenknoten zu verteilen.

Kurze Laufzeiten hingegen führen dazu, dass der Overhead der Java-VM und des Hadoop Frameworks die generelle Performanz des Clusters beeinträchtigt. Alleine

das Starten und Beenden einer JVM für jeden Mapper kostet einige Sekunden. Dieses Problem kann zum Teil – aber nur bis zu einem gewissen Grad – durch die Wiederverwendung von JVM Instanzen angegangen werden.

In kleinen Clustern mit einigen Dutzenden Rechnern fällt das Argument der Hardwareausfälle weniger ins Gewicht, da sie sehr viel seltener sind. In wenig ausgelasteten Clustern ist zudem die zeitliche Granularität weniger entscheidend. Dies trifft beides auf das Cluster zu, auf dem im Rahmen dieser Arbeit getestet wird. Hat man einen solchen Fall, sollte ein Job dann am schnellsten ablaufen, wenn jeder Rechenknoten genau einen Block bearbeiten muss. Ob diese Vermutung zutrifft, wird im Laufe dieses Kapitels untersucht.

5.1.2. Dateigröße

Sehr große Dateien stellen für Hadoop keinerlei Problem dar und sind sehr gut zum Speichern im HDFS und Verarbeiten mit Hadoops MapReduce. Dies gilt solange sie zerteilbar sind. Mehr dazu in Abschnitt 5.1.4.

Dateien, die im Vergleich zur Blockgröße klein sind, können hingegen sowohl für das HDFS als auch Hadoop zum Problem werden[1]. Sind die Dateien selber sehr klein und dennoch "Big Data", ist davon auszugehen, dass es sich um sehr viele Dateien handelt.

Das entscheidende Problem für MapReduce ist damit bereits im Abschnitt über Blockgrößen erörtert worden: Jede Datei wird von einem einzelnen Mapper bearbeitet, was gerade bei extrem vielen Dateien zu viel Overhead führt. HDFS ist jedoch auch unabhängig von MapReduce nicht für kleine Dateien geeignet.

Zunächst stellen diese für den NameNode ein Problem dar, da die Metadaten der Dateien im Betrieb im Arbeitsspeicher gehalten werden. Der Arbeitsspeicher muss dann einige Gigabytes an Metadaten aufnehmen, und das System wird überlastet. Ein weiteres Problem ist eine der Designentscheidungen beim Entwurf von HDFS: Es wurde für hohen Durchsatz, aber nicht niedrige Latenzen ausgelegt. Müssen viele kleine Dateien nacheinander ausgelesen werden, wird diese Latenz zum limitierenden Faktor der Laufzeit der Anwendung.

Die FACT-Daten sind in großen zusammenhängenden FITS-Dateien gespeichert, sodass diese Problematik für dieses Beispiel unerheblich ist.

Häufig ist es jedoch unumgänglich, kleine Dateien zu verarbeiten. Die Datenquelle könnte es erforderlich gemacht haben, viele kleine Daten zu schreiben (zum Beispiel Log-Dateien), oder die Daten sind inhärent in viele kleine Dateien zu unterteilen (wie zum Beispiel Bilddateien). In einem solchen Fall bieten sich verschiedene Lösungsansätze, die hier nur kurz erwähnt werden sollen, da sie für die FACT-Daten nicht relevant sind.

Eine einfache Möglichkeit ist es, die Daten zu *Hadoop Archives* (HAR) zusammenzufassen. Dies sind extra für diesen Fall entworfene Dateicontainer, ähnlich wie `tar`-Dateien.

[1] Cloudera Blogeintrag über "Das kleine Dateien Problem":
http://blog.cloudera.com/blog/2009/02/the-small-files-problem/
– Abgerufen am 3.12.2013 17:18

Ein anderes spezielles Dateiformat ist ein Hadoop *SequenceFile*. Dies sind Schlüssel-Werte-Dateien, die mit einem geeigneten `SequenceFileOutputFormat` erzeugt und dann mit dem zugehörigen `SequenceFileInputFormat` gelesen werden können. Darin speichert man für kleine Dateien den Namen als Schlüssel und Dateiinhalt als Wert.

Will man die Dateien im Originalzustand im HDFS ablegen sollte man das `MultiFileInputFormat` verwenden. Dadurch wird zumindest die Anzahl der Mapper reduziert, die Probleme von HDFS jedoch nicht angegangen.

Manchmal kann es sinnvoll sein, kleine Dateien komplett anders zu speichern und stattdessen in verteilten (No-)SQL Datenbanken abzulegen. Als Beispiele seien hier Apache Cassandra und Apache HBase genannt.

Alle hier aufgelisteten Lösungsansätze werden bis jetzt nicht von streams-mapred unterstützt, da sie die zugrunde liegenden Daten in einem anderen Format abspeichern und so nicht mehr von normalen Strömen ausgelesen werden können. Soll streams-mapred in Zukunft für einen solchen Datenbestand zum Einsatz kommen, müssten neue spezialisierte `DataStreamInputFormat`e geschrieben werden: `DataStreamSequenceFileInputFormat` oder ähnliches.

5.1.3. Kompression

Datenkompression ist ein Thema, dass bei der Verarbeitung von größeren Datenmengen nicht außer Acht gelassen werden kann. Das Komprimieren von Dateien kann zweierlei Vorteile haben. Offensichtlich ist, dass die Daten – sofern sie komprimierbar sind – weniger Speicherplatz verbrauchen, also Hardwarekosten gespart werden.

Wichtig ist jedoch auch, dass bei vielen Berechnungen die Lesegeschwindigkeit der Eingabedaten der Engpass ist. Der Prozessor wird nicht ausgelastet. Sind die Daten komprimiert, „verschiebt“ sich die Auslastung von der Eingabehardware auf die CPU, was zu einer Steigerung der Performanz führen kann. Verschiedene Kompressionsarten haben also grundsätzlich die zwei Eigenschaften Kompressionsrate – wie sehr werden Daten komprimiert – und Kompressionsgeschwindigkeit – wie schnell werden die Daten entpackt. Der zweite Aspekt kann auch anders betrachtet werden: Wie sehr wird die CPU zusätzlich belastet.

Mit den bekannten Kompressionsalgorithmen stellen diese beiden Eigenschaften einen Trade-Off dar. Den perfekten Kompressionsalgorithmus – schnell und gleichzeitig klein – gibt es nicht. Alle Verfahren lassen sich in eine Skala von langsam-und-klein bis schnell-und-groß einteilen. Dem Nutzer bieten sich damit verschiedene Abstufungen, die je nach Einsatzzweck zu verschiedenen Ergebnissen führen können.

Die Daten sollten im Idealfall so komprimiert werden, dass weder CPU noch die Eingabehardware überlastet werden. Dies ist jedoch schwierig, da auf den gleichen Daten verschiedene Berechnungen ausgeführt werden, die die CPU verschieden auslasten. Außerdem würde eine Änderung der Hardwareleistung zu neuen Anforderungen führen. Insofern ist es in den meisten Fällen schwierig die „richtige“ Komprimierung zu wählen. Eine allgemeingültige Aussage kann auf keinen Fall getroffen werden.

Zwei Eigenschaften, die eher praktische Argumente einbeziehen, können damit ausschlaggebend für die Wahl einer Kompression sein: Unterstützung und Zerteilbarkeit. Einige Kompressionsverfahren werden von nahezu jedem Betriebssystem unterstützt oder können zumindest durch weit verbreitete Zusatzprogramme entpackt werden. Das kann entscheidend sein, will man die Datei zum Beispiel während der Entwicklung auf dem lokalen Rechner betrachten. Ein Entpacken einer Datei kann schwierig werden, wenn sie zuvor in einem wenig verbreiteten Format komprimiert wurde.

Das Verfahren sollte außerdem von allen verwendeten Programmiersprachen unterstützt werden. Es nützt nichts, wenn streams-mapred die Datei betrachten kann, das Python-Skript auf dem Entwicklerrechner jedoch nicht.

Die Eigenschaft der Zerteilbarkeit ist hauptsächlich für die Verarbeitung mit Hadoop interessant. Der nächste Abschnitt wird darauf genauer eingehen. Entscheidend ist hier, dass manche Kompressionsverfahren zerteilbar sind und andere nicht.

Die folgende Tabelle listet einige typische Kompressionsverfahren und ihre Eigenschaften auf. Später soll untersucht werden, welches davon sich für die FACT-Daten bei dem gegebenen Cluster am besten eignet.

Verfahren	**Dateierweiterung**	**Geschwindigkeit**	**Größe**	**Zerteilbar**
BZIP2	bz2	langsam	klein	Ja
GZIP	gz	mittel	mittel	Nein
Snappy	snappy	schnell	groß	Nein

Tabelle 5.1.: Kompressionsverfahren und ihre Eigenschaften

5.1.4. Zerteilbarkeit

Die Verteilung einer Datei auf mehrere Blöcke im HDFS missachtet den Inhalt der Datei komplett. Jeder Block ist gleich groß, völlig unabhängig davon, ob das letzte Byte des Blockes eine logische Grenze innerhalb der Daten darstellt, wie beispielsweise ein Zeilenumbruch.

Ein `InputFormat` und der zugehörige `RecordReader` müssen diese Tatsache beachten. Tun sie das, wird das `InputFormat` *zerteilbar* (*splittable*) genannt. Jedoch erlauben nicht alle Dateiformate, ein `InputFormat` für sie zu implementieren.

Ein Dateiformat kann dann zerteilbar gemacht werden, wenn es möglich ist, an einer von einer beliebigen Stelle innerhalb der Datei aus den Beginn des nächsten logischen Pakets zu finden. Eine Datei in ihrer Mitte beginnend einzulesen, ist wie gesagt bei komprimierten Daten nicht immer möglich, da manche Kompressionsverfahren nur komplett, also von Anfang an, gelesen werden können.

Zur Erinnerung: Ist ein Format nicht zerteilbar, wird für die gesamte Datei nur ein einziger Mapper gestartet, obwohl nicht alle Blöcke zwangsläufig auf einem einzelnen Rechenknoten liegen. Der Mapper wird dann auf einem Knoten gestartet, der den ersten Block speichert. Die nicht auf diesem Knoten liegenden Blöcke werden dann

(durch einen `FSDataInputStream` implizit) über das Netzwerk auf diesen Knoten übertragen. Dann kann das Netzwerk zu einem Engpass der Berechnung werden.

(Un-)zerteilbare Formate

Einige Dateiformate sind nur schwer oder gar nicht zerteilbar. Verschiedene Techniken machen Formate zerteilbar. Möglich wird Zerteilbarkeit von Formaten, wenn sie eine Grenze zwischen Paketen definieren, also der Reader nur eine gewisse Bitfolge suchen muss (nichts anderes ist schließlich auch ein Zeilenumbruch einer Textdatei). Eine solche Technik bieten beispielsweise `SequenceFile`s. Darin werden sogenannte *sync marker* in die Datei eingefügt, an denen das dazugehörige Eingabeformat eine Paketgrenze erkennt.

Eine andere Möglichkeit für zerteilbare Formate sind festgelegte Paketgrößen. Dann kann der `RecordReader` an das nächste Vielfache dieser Größe springen und davon ausgehen, dass sich dort das nächste Paket befindet.

Die Zerteilbarkeit von Formaten mit Kopfdaten (*header*) ist schwierig. Alle `RecordReader` bis auf den ersten, müssten sich den Header extra aus dem ersten Block lesen.

Auch Textformate können schwer zerteilbar sein. Das gilt zum Beispiel, wenn sie geschachtelt sind wie XML oder JSON. Ein hypothetischer `XMLRecordReader`, der alle Top-Level Elemente als Datenpaket betrachten will, kann das nächste Top-Level Element nicht allein an Hand der XML-Struktur finden, wenn er mitten in der Datei beginnt, danach zu suchen.

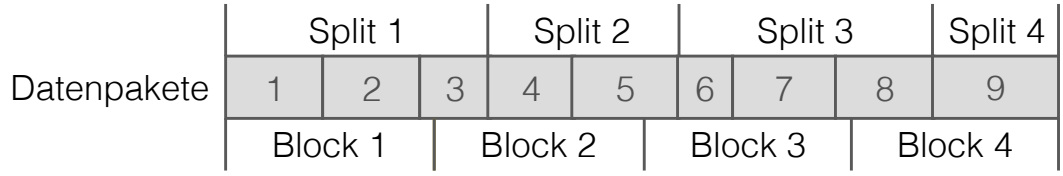


Abbildung 5.1.: Eine auf vier Blöcke (unten) verteilte Datei. Die Blöcke beachten die Grenzen der neun logischen Datenpakete (mitte) nicht. Eine logische Zerteilung einer Datei (oben) muss explizit implementiert werden.

Implementierung eines zerteilbaren Formats

Ein `InputFormat` wird durch zwei Schritte zerteilbar gemacht. Zunächst muss die `isSplittable` Methode des `InputFormat`s unter Beachtung der verwendeten Komprimierung signalisieren, dass der zugehörige `RecordReader` in der Lage ist, mit zerteilten Dateien umzugehen.

Der `RecordReader` ist es schließlich, der die Aufgabe hat, die zerteilten Daten korrekt auszulesen. Zwei Situationen müssen dabei betrachtet werden.

Der einfache Fall tritt am Ende eines Blockes auf. In Abbildung 5.1 ist dies zum Beispiel das Datenpaket 3 des ersten Splits. Das Datenpaket beginnt im Block 1 und setzt sich anschließend im Block 2 fort. Der `RecordReader` muss die Daten im Block

2 über das Netzwerk übertragen. Durch die Verwendung von `FSDataInputStream` geschieht dies jedoch implizit und ohne Zutun des Programmierers. Hier muss der Programmierer also nur innerhalb der `readNext` Methode die gegebenen Blockgrenzen beachten, statt den kompletten `InputStream` zu verarbeiten.

Komplizierter ist der Fall, dass das Paket irgendwo innerhalb des aktuellen Blockes beginnt. Der `RecordReader` des Mappers, der auf Block 2 angesetzt wurde, muss zunächst den Anfang des vierten Pakets suchen und die Daten davor ignorieren (sie werden vom ersten Mapper verarbeitet).

Dieses Verhalten wird am besten in der `initialize` Methode implementiert. Dort sucht der `RecordReader` vom Anfang seines Blocks aus den Anfang des ersten Pakets und verschiebt seine Startposition und die des verwendeten `InputStream`s auf die entsprechende Position. Anschließend kann die `readNext` Methode wie zuvor die Pakete lesen.

Verarbeitung unzerteilbarer Daten

Ist ein Format schwer zu zerteilen – im Sinne der Unterstützung durch ein `InputFormat` – bieten sich zwei Möglichkeiten. Entweder man konvertiert die Daten in ein zerteilbares Format oder man zerlegt die Daten schon vor der Übertragung in das HDFS auf Dateiebene.

Dabei werden die Daten in neue Dateien des gleichen Formats unterteilt, die dann genauso groß wie (oder kleiner als) ihre späteren HDFS-Blöcke sein sollten. So wird es unerheblich, ob das Format selber zerteilbar ist, da ein Block genau einer Datei entspricht. Jeder Mapper bearbeitet wie gewohnt eine dieser Dateien und muss keine Blöcke von anderen Rechenknoten übertragen. Gerade bei Formaten, die extrem schwer oder gar nicht zu zerteilen sind, führt diese Technik schneller zum Erfolg, als dies sonst der Fall wäre.

Sowohl das Konvertieren, als auch das Vorzerlegen haben ihre Daseinsberechtigung. Schlussendlich müssen viele verschiedene Faktoren in die Entscheidung für eine der Möglichkeiten beachtet werden. Dazu zählen bestehende Eingabe- und Konvertierungsmöglichkeiten der Formate und die prinzipielle Eignung anderer Formate für die Daten.

Zerteilbarkeit in streams-mapred

Das `DataStreamInputFormat` aus streams-mapred ist momentan nicht zerteilbar. Das hat zwei Gründe: Zunächst einmal müssten bestehende Ströme angepasst werden. Außerdem ist das FITS-Format nur mit unverhältnismäßig großem Aufwand zerteilbar. Es wird sich zeigen, dass eine Vorzerlegung für das Format besser geeignet ist. Für die einzigen bis hier hin bearbeiteten Daten wurde ein zerteilbares `InputFormat` also nicht benötigt.

An dieser Stelle wird kurz gezeigt, wie streams-mapred um zerteilbare Ströme ergänzt werden könnte. Zwei Techniken könnten – je nach Anwendung – zum Ziel führen: Die Erweiterungen von Strömen oder weitere Eingabeformate. Die Erweiterung von bestehenden Strömen ließe sich beispielsweise mit einem `SplittableStream` Interface realisieren, wie es Listing 5.2 zeigt.

```
interface SplittableStream {
/**
 * Sucht im gegebenen inputStream nach dem Beginn des naechsten
 * Datenpakets zwischen inklusive start und exklusive end.
 * Der Stream ist zuvor bereits auf start positioniert.
 * Setzt den InputStream auf die Position des Paketbeginns
 * (siehe InputStream#seek) und gibt diese Position zurueck.
 */
long seekToNextItemStart(InputStream inputStream,
                         long start,
                         long end);
}
```

Abbildung 5.2.: Das SplittableStream Interface ist eine denkbare Möglichkeit um Zerteilbarkeit in streams-mapred einzuführen

`DataStreamInputFormat` könnte dann signalisieren, dass es zerteilbar ist, wenn der gegebene Strom das Interface implementiert. Der `RecordReader` würde dann in `initialize` die `seekToNextItemStart` Methode verwenden, um das nächste Paket vom Strom finden zu lassen.

Zusätzlich dazu wäre es denkbar, weitere `DataStreamXYZInputFormat`e zu implementieren. Diese würden dann statt dem allgemeinen `FileInputFormat` spezialisierte Formate wie `TextInputFormat` oder `SequenceFileInputFormat` erweitern und deren Zerteilbarkeit erben.

5.2. Die FACT-Daten und Hadoop

Nachdem erläutert wurde, was zu beachten ist, wenn man Daten mit Hadoop verarbeiten will, sollen die erläuterten Möglichkeiten und Techniken auf die Daten des FACT-Teleskops angewendet und erprobt werden. Um zu verstehen, welche Möglichkeiten und Techniken auf das verwendete FITS Format [15] anwendbar sind, ist es zunächst notwendig das Format genauer zu betrachten.

5.2.1. Das (FACT-)FITS-Format

Das FITS-Format wurde in den späten siebziger Jahren entwickelt und 1982 von der International Astronomical Union formal bestätigt. Es liegt seit 2008 nach einigen Erweiterungen in der Version 3.0 vor. Es wurde als Format für den Austausch und die Archivierung von astronomischen Bilddaten entwickelt.

Die lange Verwendungszeit und relativ große Verbreitung hat dazu geführt, dass eine Vielzahl an Werkzeugen für eine große Menge verschiedener (Programmier-) Umgebungen geschaffen wurde.

Die Daten des FACT-Teleskops werden in einem Format gespeichert, das nur einen Teil der Möglichkeiten des FITS-Formats nutzt. Für diese Arbeit ist nur diese Teilmenge des Formats relevant. Der Einfachheit halber ist in den folgenden Betrachtun-

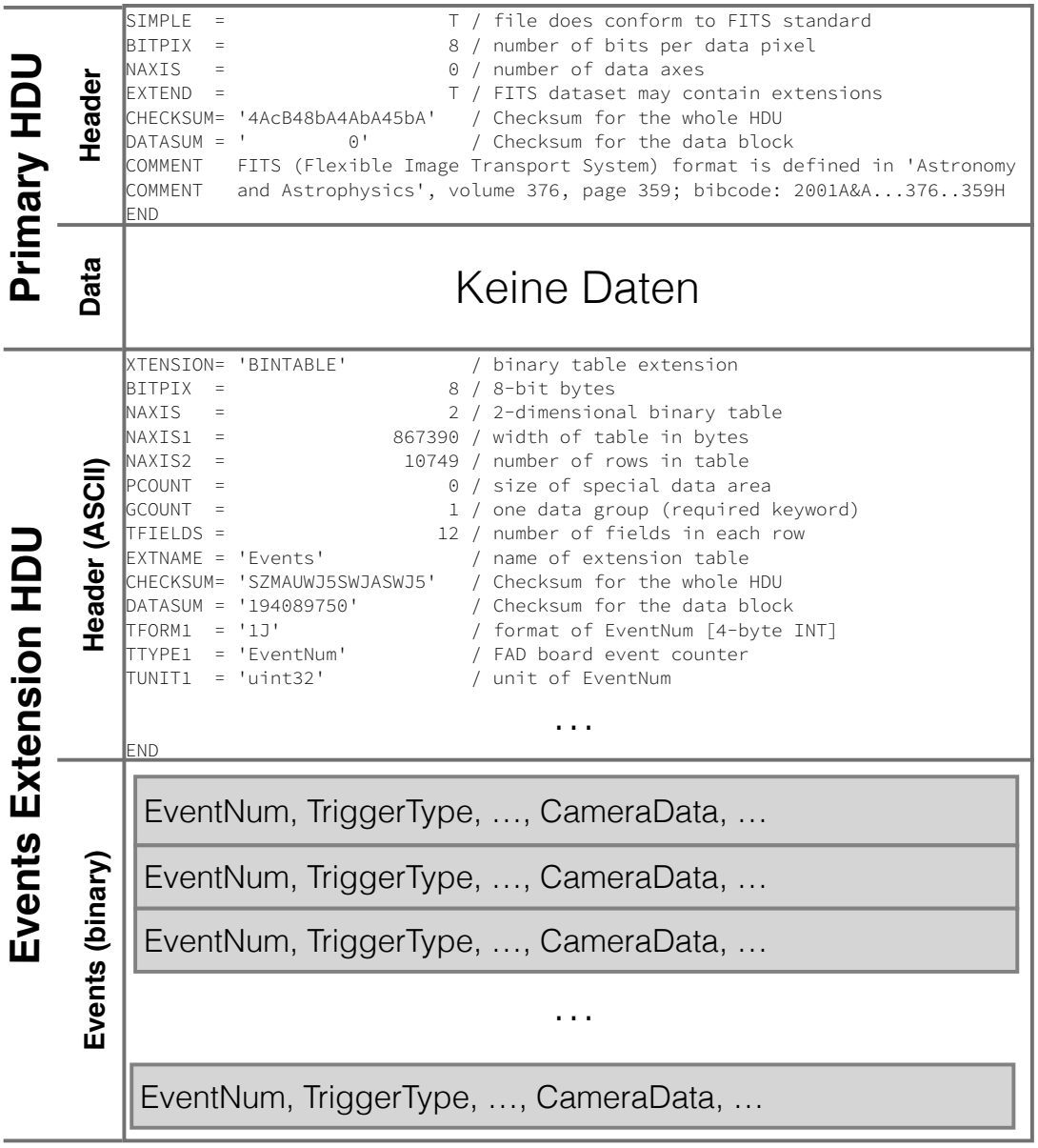


Abbildung 5.3.: Schematischer Aufbau einer FACT-FITS-Datei.

gen häufig vom FITS-Format die Rede, während das FACT-FITS-Format gemeint ist (siehe Abbildung 5.3). Eine FITS Datei besteht aus so genannten „Kopfdaten- und Dateneinheiten“ (*header and data unit*, HDU). Jede HDU besteht aus einem ASCII-Header, der die enthaltenen Daten beschreibt, und einem Datenblock.

Jede FITS-Datei muss mit einer primären HDU beginnen und darf beliebig viele so genannter Erweiterungen (*Extension*) in Form weiterer HDU enthalten. Eine FACT-FITS-Datei enthält eine primäre HDU, die keine relevanten Informationen speichert und ignoriert werden kann. Die Kameradaten des Teleskops werden in einer einzelnen Binärtabellen-Erweiterung gespeichert (*binary table extension*). Sie speichert zeilenweise die in Abschnitt 2.3 angesprochenen Ereignisse und wird daher *Events*-Erweiterung genannt.

Events Erweiterung

Die Binärtabelle der FACT-Daten ist zweidimensional und wird vom Header der HDU beschrieben (siehe Abbildung 5.3). Der Header spezifiziert zunächst die Anzahl der Zeilen (NAXIS2) und die Breite der Tabelle (also die Anzahl der Bytes pro Zeile, NAXIS1). Die Zeilen sind nicht voneinander getrennt (zum Beispiel durch Zeilenumbrüche) und sind nur durch diese Information voneinander zu unterscheiden.

Anschließend wird definiert, wie jede Zeile zu interpretieren ist. Jede Spalte hat einen Namen (TTYPEN, N ist Spaltennummer), ein Format (TFORMN) und eine Einheit (TUNITN). Beispielsweise sind die ersten vier Bytes jeder Zeile als einzelner 32 Bit *unsigned int* zu interpretieren und enthalten die Event Nummer (TFORM1, TTYPE1 und TUNIT1). Die Events Erweiterung hat zwölf Spalten, die außerdem unter anderem die Uhrzeit, den Auslöser des Events, etwaige Fehler oder – am wichtigsten – die Kameradaten enthalten.

Der FitsStream

Im Rahmen des fact-tools Projekts wurde, um die FITS-Dateien des Teleskops mit Hilfe des streams Frameworks analysieren zu können, ein `FitsStream` implementiert. Er befindet sich im `fact.io` Paket.

Der Strom ist im Moment nur in der Lage, eine Datei von vorne beginnend zu lesen. Dabei wird zunächst die primäre HDU übersprungen. Dann wird das Einlesen der Events Erweiterung vorbereitet, indem ihr Header ausgelesen und interpretiert wird. Die Instanz des Stroms speichert die Spaltendefinitionen, merkt sich also, wie jede Zeile zu interpretieren ist.

Anschließend werden mit Hilfe dieser Informationen die Ereignisse ausgelesen und in streams-Datenpaketen gespeichert. Dabei ist wichtig, dass ohne die Information über die Byte-Anzahl einer Zeile und wie diese zu interpretieren sind, das Auslesen nicht möglich ist. Jedes Datenpaket enthält schließlich Schlüssel-Werte-Paare die jeweils einem FITS-Spaltennamen-Zeileninhalt-Paar entsprechen.

5.2.2. FITS-Dateien in streams-mapred

Wie sich gezeigt hat, ist die Art und Weise wie Daten im HDFS gespeichert werden entscheidend für die Performanz einer Hadoop MapReduce Anwendung. Viele verschiedene Aspekte müssen dabei beachtet werden, insbesondere dann wenn streams-mapred zum Einsatz kommen soll. Diese Überlegungen sollen hier am Beispiel des FITS-Formats angestellt werden. Dabei soll konkret auf das verwendete Hadoop Cluster Eingegangen werden.

- **Block- und Dateigröße**

  Die Daten des FACT-Teleskops liegen bereits in großen zusammenhängenden Dateien vor. Sie sind damit nicht vom Problem kleiner Dateien betroffen. Würde man beispielsweise jedes Event in einer einzigen Datei oder zeilenweise in einer verteilten (No-)SQL Datenbank ablegen, müssten andere Entscheidungen getroffen werden.

  Die Speicherung einzelner Events in einer verteilten Datenbank wie Apache HBase ist eine interessante Idee für zukünftige Entwicklungen und Untersuchungen.

  Die sehr großen Dateien bieten eine gute Grundlage, um verschiedene Blockgrößen zu testen. Es ist zu erwarten, dass sich größere Blöcke schneller verarbeiten lassen, solange mehr Blöcke als Rechenknoten vorliegen. Sonst liegen Ressourcen brach. Das zur Verfügung stehende Cluster ist vergleichsweise klein, daher ist es wahrscheinlicher als bei großen Clustern, dass eine kleine Veränderung der Blockgröße große Auswirkungen hat.

  Hat man beispielsweise zehn Rechenknoten und will elf Blöcke bearbeiten, wird es dazu kommen, dass zunächst nur zehn Blöcke verarbeitet werden. Der eine übrige Block wird dann anschließend von einem einzigen Rechenknoten alleine bearbeitet, während die neun übrigen warten. Die Berechnung wird dann fast doppelt so lange brauchen, als hätte man die Blockgröße ein wenig größer gewählt und zehn Blöcke erhalten.

  Diese Überlegung ist eher für einzelne Dateien relevant. Der gesamte Datensatz ist so groß, dass solche Probleme unerheblich sein sollten. Bei den folgenden Messungen muss diese Tatsache jedoch im Hinterkopf behalten werden.

- **Kompression**

  Die Daten des Teleskops lassen sich sehr gut komprimieren. Sie liegen aus diesem Grund bereits in komprimierter Form vor. Jede FITS-Datei wird vor der Abspeicherung mit dem GZIP Verfahren komprimiert und liegt dann als `fits.gz`- Datei vor.

  Der `FitsStream` der fact-tools kann dieses Format bereits während des Einlesens entpacken. In streams-mapred ist das Entpacken komprimierter Daten mit Hilfe von Hadoops `CompressionCodec`-Klassen eingebaut, sodass verschiedene andere Verfahren zum Einsatz kommen können.

Da jede Datenkompression einen anderen Trade-Off zwischen Größe und CPU-Belastung bietet, müssten unterschiedliche Kompressionsverfahren je nach Aufgabe besser geeignet sein. Ist eine Berechnung sehr fordernd für Prozessoren, dürfte eine schnelle (oder keine) Kompression besser geeignet sein. Gleichzeitig ist eine starke Kompression (im Sinne von klein) gut geeignet für Berechnungen, die eher schnell ablaufen und auf die Eingabehardware warten müssen.

Auf den FACT-Daten werden verschiedene Arten von Berechnungen angestellt, die einen Prozesser unterschiedlich stark belasten. Andere erst in Zukunft entwickelte Analysen könnten zudem andere Anforderungen stellen. Wegen dieser Ungewissheiten, ist ein Kompressionsverfahren sinnvoll, das den Kompromiss aus Größe und Geschwindigkeit ausgewogen löst. Die Zerteilbarkeit der Kompressionsverfahren ist für die FACT-Daten unerheblich, da, wie der nächste Abschnitt zeigt, das FITS Format aus Sicht von Hadoop nicht sinnvoll zerteilbar gemacht werden kann.

- **Zerteilbarkeit**

  Das FITS-Format macht es schwer bis unmöglich ein zerteilbares `InputFormat` dafür zu implementieren. Erstens enthält es Kopfdaten, die benötigt werden um die eigentlichen Daten zu interpretieren. Außerdem enthalten die Kopfdaten die Größe der Zeilen, sodass der Anfang des nächsten Events nicht gefunden werden kann. Aus diesen Gründen ist der `FitsStream` lediglich in der Lage eine FITS-Datei vom Anfang an einzulesen.

  Damit bleiben zwei Möglichkeiten, die Daten dennoch performant mit Hadoop zu verarbeiten. Zunächst können die Daten in ein anderes Format übertragen werden. Für die FACT-Daten ist es jedoch schwierig ein geeignetes Format zu finden. Dies gilt vor allem dann, wenn man nur Formate in die Überlegung mit einbeziehen will, für die bereits ein Strom implementiert wurde.

  Die zweite Möglichkeit ist das Vorzerteilen der Daten. Der existierende `FitsStream` ist extra für Geschwindigkeit optimiert und kann daher die Daten vergleichsweise schnell einlesen, sodass diese Möglichkeit die geeignetste ist.

Im Rahmen dieser Arbeit wurde für die Vorbearbeitung der Daten das Paket `fact.io.splitter` im fact-tools Projekt angelegt. Es enthält Klassen zur Zerteilung von FITS-Dateien, verschiedene Kompressionsverfahren sowie eine Anbindung an HDFS. Mit Hilfe der darin enthaltenen `Splitter`-Klasse wurde für die Messungen dieser Arbeit eine etwa 5,8 Gigabyte große FITS-Datei in verschieden große Teildateien zerteilt, mit verschiedenen Kompressionsverfahren komprimiert und mit verschiedenen Blockgrößen im HDFS gespeichert.

5.3. Messungen

Die Messungen wurden auf einem vergleichsweise kleinen Hadoop Cluster des Lehrstuhls 8 der Technischen Universität Dortmund ausgeführt. Es besteht aus elf Knoten, denen jeweils 1 Terabyte an Festplattenplatz zur Verfügung steht. Das Cluster

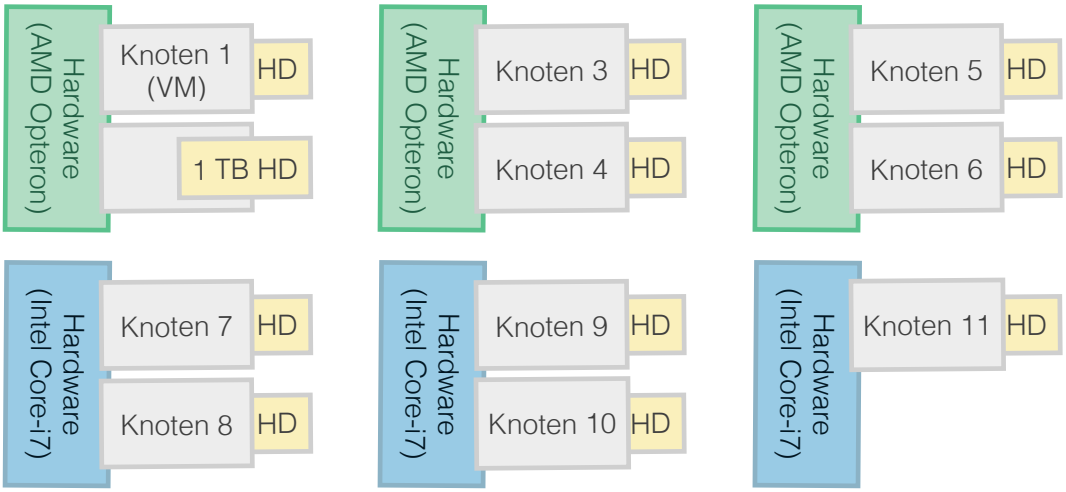


Abbildung 5.4.: Aufbau des verwendeten Clusters. Sechs Knoten laufen auf älterem AMD, fünf auf schnelleren Intel Prozessoren. Jeder virtuellen Maschine wurde eine eigene physikalische 1 TB Festplatte zugeordnet.

ist entgegen des gewünschten Aufbaus nicht homogen und besteht aus verschieden leistungsstarken Knoten.

Der langsamere Knotentyp stellt sechs der Knoten. Es sind virtuelle Maschinen, die jeweils zu zweit auf einem physikalischen System mit AMD Opteron Prozessoren aus den Jahren 2008 und 2009 ausgeführt werden. Die fünf schnelleren Knoten wurden 2013 angeschafft und verfügen über Intel Core-i7 Prozessoren, laufen aber auch jeweils zu zweit virtualisiert auf einem Rechner. Um das Cluster mit einem Einzelplatzrechner vergleichen zu können, wurde der Benchmark auf einem modernen System mit Core-i7 Architektur ausgeführt.

Auf jedem der Systeme wird nur ein einzelner Thread gestartet. Die virtualisierten Systeme (gerade die Opteron Systeme) würden von mehr Mappern pro Knoten nicht profitieren. Der Einzelplatzrechner müsste jedoch bessere Werte erzielen kön-

```
<fact.filter.DrsCalibration url="${drsfile}"
        key="Data" outputKey="DataCalibrated" />
<fact.utils.CutSlices start="22" end="220"
        outputKey="DataCalibrated" />
<fact.filter.RemoveSpikesMars key="DataCalibrated"
        topSlope="8.0" />
<fact.filter.AverageJumpRemoval key="DataCalibrated"
        outputKey="DataCalibrated" />
<fact.features.MaxAmplitudePosition key="DataCalibrated"
        outputKey="arrivalTime" />
```

Abbildung 5.5.: Prozess, der für alle Benchmarks verwendet wurde. Die `arrivalTime` wurde im `RootASCII` Format als Ergebnis gespeichert.

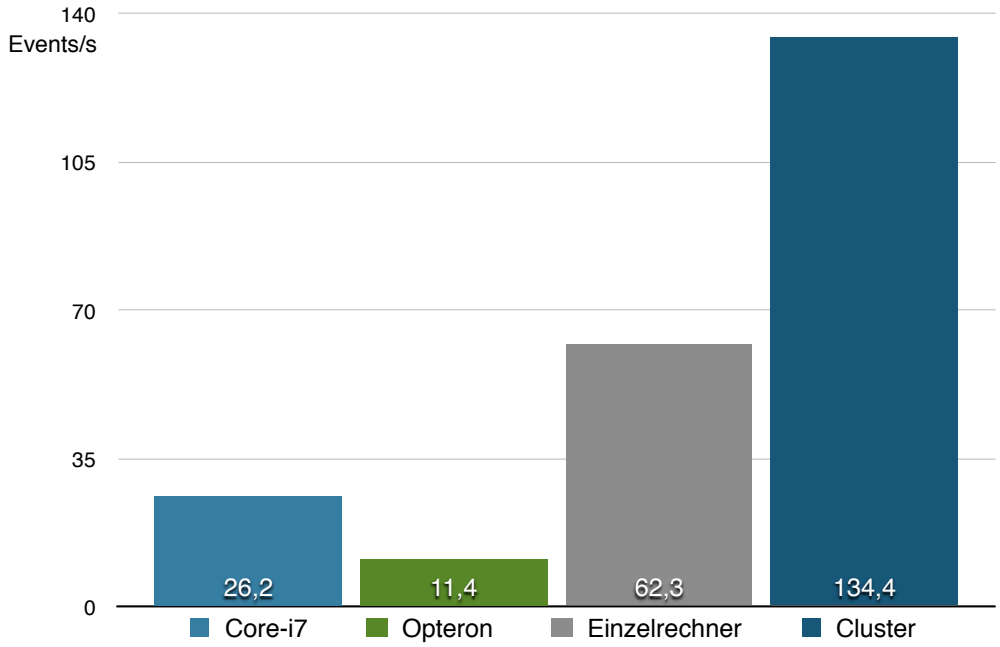


Abbildung 5.6.: Die verschiedenen Knotentypen, der Einzelrechner, sowie das Cluster im Vergleich. Die Knoten und Einzelrechner haben die Originaldatei (FITS und GZIP) verarbeitet. Auf dem Cluster wurden unkomprimierte 64 MB Dateien gewählt.

nen als hier abgebildet. Die Verwendung von nur einem Thread ist der besseren Vergleichbarkeit der Systeme geschuldet.

Da in der FACT Analyse häufig über viele Ereignisse iteriert und pro Ereignis je ein Ergebnis erzeugt wird, wird die Geschwindigkeit eines solchen Prozesses in Ereignissen pro Sekunde (Event/s) angegeben. Diese Einheit werden auch die folgenden Benchmarks verwenden. Alle Messungen wurden mindestens zehn mal durchgeführt. Die hier gezeigten Werte stellen den Mittelwert aller Messungen dar.

Die wichtigste Messung ist in Abbildung 5.6 zu sehen. Sie zeigt, wie schnell die Knotentypen und der Einzelrechner im Vergleich zum gesamten Cluster sind (dabei wurde auf dem Cluster die beste Speichermethode gewählt). Einige Aspekte dieses Vergleichs sind interessant. Zunächst zeigt sich, dass die Knotentypen erwartungsgemäß unterschiedlich schnell sind. Die neueren Knoten sind etwa 2,2 mal schneller. Außerdem zeigt sich, dass trotz ähnlicher Hardware der Einzelrechner wesentlich schneller ist, als die schnellsten Knoten des Clusters. Diese Tatsache lässt sich am besten durch die Virtualisierung des Clusters erklären.

Die wichtigste Erkenntnis ist, dass das Cluster zwar nur ungefähr 2,1 mal schneller als ein einzelner Kern des Einzelrechners ist, jedoch etwa 7,4 ($\approx 134,4/((6*11,4+5*26,2)/11)$) mal schneller, als seine 11 Knoten im Durchschnitt sind. Dass der theoretisch mögliche Wert von Faktor 11 (also 200 Events/s $\approx 6*11,4+5*26,2$) nicht erreicht wird, hängt zum Teil mit dem Overhead der Hadoop Umgebung zusammen. Wichtiger ist jedoch, dass die Berechnung häufig auf die langsamen Knoten warten muss.

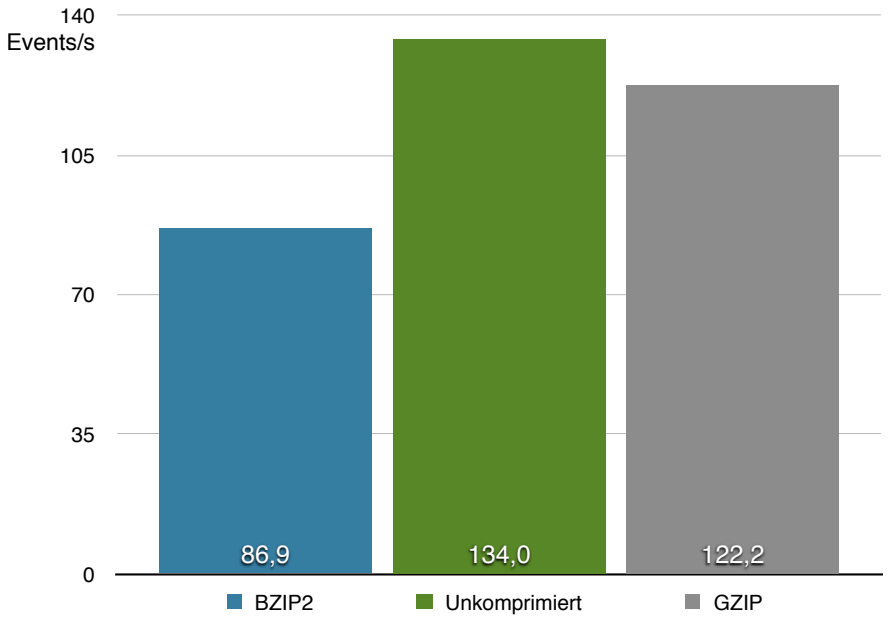


Abbildung 5.7.: Unterschiedliche Kompressionsverfahren im Vergleich. Alle Daten wurden in 128 MB großen Dateien gespeichert (nach dem Komprimieren) und in 128 MB Blöcken im HDFS abgelegt.

In einigen Fällen wurden von der Hadoop Umgebung sogar Ersatzmapper gestartet, die die selben Daten, wie die langsamen Knoten bearbeiten. Diese Maßnahme ergreift Hadoop, für den Fall, dass der zuvor gestartete Mapper sich zum Beispiel wegen Hard- oder Softwareschäden aufgehängt hat. Diese Ersatzmapper überholten teilweise die langsameren Knoten, sodass deren Ergebnisse verworfen wurden.

Die durchschnittliche Laufzeit eines Mappers auf den schnellen Knoten liegt beispielsweise bei 128 MB großen BZIP2 Dateien bei etwa 30 Sekunden, während sie auf den langsamen bei fast 90 Sekunden liegt. Die Knoten sind damit sogar langsamer als der ermittelte Faktor 2,2, da sie auch den Overhead von Hadoop (JVM initialisieren, Serialisieren der Ergebnisse, usw.) langsamer ausführen. Eine wichtige Erkenntnis hieraus ist, dass ein homogenes Cluster wesentlich gleichmäßiger und damit besser ausgenutzt werden könnte.

Kompressionsverfahren

Abbildung 5.7 vergleicht die Verarbeitung von Dateien, die mit den beiden wichtigsten Kompressionsverfahren GZIP und BZIP2 komprimiert wurden, sowie die von nicht komprimierten Dateien miteinander. Alle Daten wurden in 128 MB Dateien zerteilt gespeichert. Das Cluster verarbeitet – anders als vorhergesagt – die unkomprimierten Dateien schneller. Weniger überraschend ist dann, dass die langsamer zu entpackenden BZIP2-komprimierten Dateien auch langsamer verarbeitet werden als bei der Verwendung von GZIP.

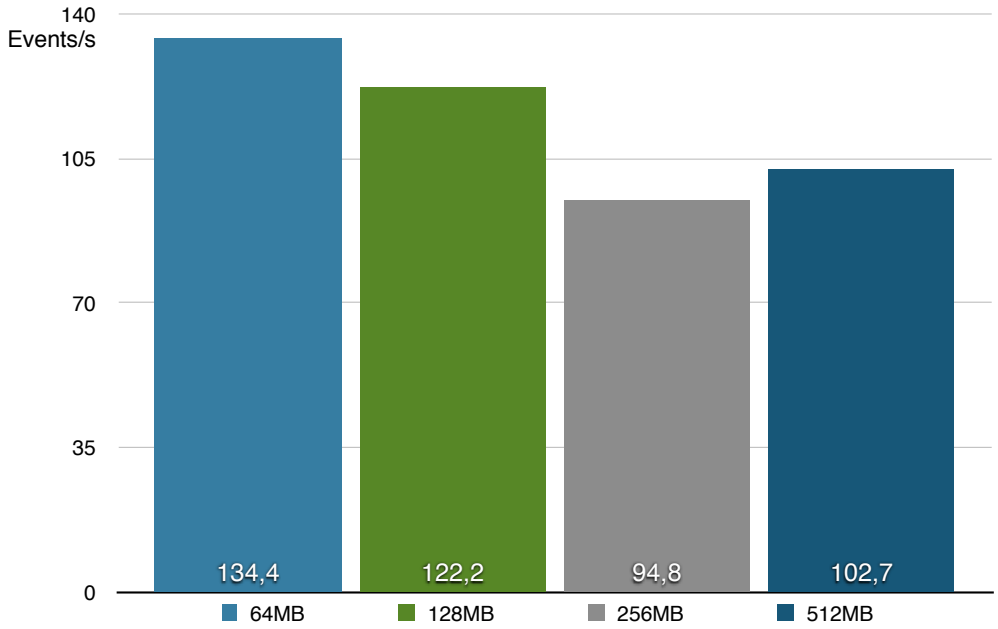


Abbildung 5.8.: Unterschiedliche Dateigrößen im Vergleich. Alle Daten wurden als GZIP gespeichert und mit ihrer Dateigröße entsprechenden Blockgröße im HDFS abgelegt.

Hier zeigt sich, dass das Cluster für die gewünschten Berechnungen etwas zu schwache Prozessoren verwendet, beziehungsweise die Eingabehardware im Vergleich dazu schnell ist. Nicht die Eingabe (I/O-Controller, Festplatte) limitiert die Geschwindigkeit, sondern die Leistung des Prozessors. Damit bleibt keine überschüssige CPU-Leistung für das Entpacken der Dateien übrig und die Geschwindigkeit sinkt. In Anbetracht der Tatsache, dass die unkomprimierten Dateien um den Faktor 1,6 größer sind als ihre GZIP Äquivalente, lohnt sich die Kompression jedoch um Hardwarekosten zu sparen. Ein weiterer Punkt ist, dass in einem schnelleren Cluster der CPU/Eingabehardware-Trade-Off stärker zu Gunsten der CPU ausfallen sollte, sodass eine Kompression mehr Zukunftssicherheit mit sich bringt.

Dateigröße

Für diese Messungen wurden die Daten auf verschieden große Dateien von 64 MB bis 512 MB aufgeteilt. Die Dateien wurden so im GZIP-Format komprimiert, dass die komprimierten Daten der gewünschten Größe entsprechen. Abbildung 5.8 zeigt die erreichten Ereignisse pro Sekunde für die verschiedenen Größen. Auch hier entsprechen die Ergebnisse nicht den Vorhersagen. Größere Dateien sollten eigentlich zu weniger Overhead führen und damit zu besserer Leistung. Die erzielten Ergebnisse lassen sich jedoch wieder durch das Cluster erklären.

Die insgesamt erzielte Leistung hängt bei genauerer Betrachtung stark vom schwächsten Mapper ab. Sogar bei den sehr kleinen Dateigrößen wartet die gesamte Berechnung auf diesen langsamstem Mapper. Größere Dateien führen zwar zu we-

niger Overhead, jedoch zu höherer Belastung pro Mapper. Die größere Belastung führt dabei zu größeren Leistungsunterschieden zwischen den beiden Knotentypen. Je größer die Dateien werden, desto langsamer werden also die langsamen Knoten im Vergleich zu den schnellen. Die Gesamtleistung sinkt damit, weil noch länger gewartet wird.

Die 512 MB großen Dateien lassen sich wieder schneller verarbeiten. Auch dieses auf den ersten Blick nicht intuitiv zu erfassende Ergebnis lässt sich anhand des Clusters erklären. Wie bereits erwähnt, startet Hadoop, sobald die schnellen Knoten beschäftigungslos sind, weitere Ersatzmapper, die bei so großen Dateien die früher gestarteten langsamen Mapper überholen. Dieses Verhalten war durch die 2,2-fache Geschwindigkeit der Knoten zu erwarten: Ab einer gewissen Dateigröße wird der Overhead, den eine nachträglich neu gestartete Berechnung mit sich bringt, wieder eingeholt.

Auch für dieses Ergebnis gilt: Wäre das verwendete Cluster homogener, leistungsstärker und generell größer, sollten die Ergebnisse anders ausfallen. Es ist eher zu erwarten, dass die mittelgroßen Dateien den größten Vorteile für die Gesamtleistung mit sich bringen.

Blockgröße

In dieser Messung wurden die Daten auf 512 MB Dateien aufgeteilt im GZIP Format gespeichert. Sie wurden dann im HDFS mit verschiedenen Blockgrößen von 64 MB bis 512 MB abgelegt. Abbildung 5.9 zeigt die Messwerte. Wieder entsprechen die Ergebnisse nicht den Erwartungen. Eigentlich sollte es dazu kommen, dass die Verarbeitung jeder Datei auf einem der Knoten gestartet wird, auf dem der erste

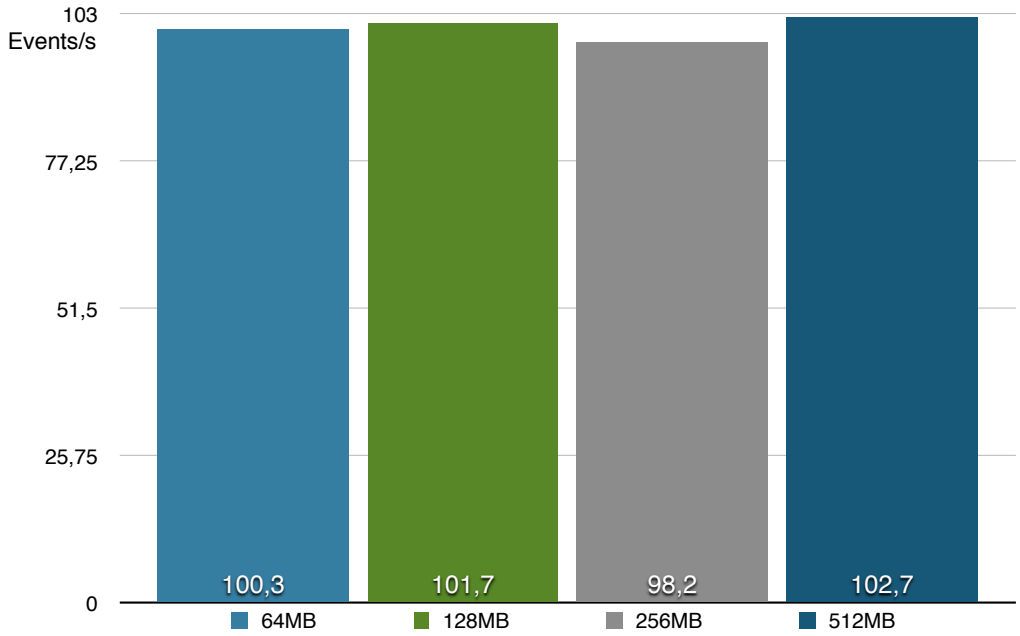


Abbildung 5.9.: Unterschiedliche Blockgrößen im Vergleich.

Block der Datei liegt. Die restlichen Blöcke müssen dann von anderen Knoten über das Netzwerk übertragen werden.

Mit Hilfe des Kommandozeilenwerkzeugs `iftop` wurde ermittelt, dass genau dieser Fall eintritt. Das Netzwerk zwischen den Knoten ist jedoch so schnell, dass es zu keinen Leistungseinbrüchen kommt. Tatsächlich ist der Zugriff über das Netzwerk sogar schneller als das Auslesen aus den Festplatten der langsameren Knoten. Da der Prozess schon zuvor durch die Prozessorleistung limitiert war, kommt es auch auf den schnelleren Knoten zu keinen Verzögerungen durch die Netzwerkübertragung.

Für diese Messung gilt wieder, dass die Ergebnisse nur im Sonderfall eines sehr kleinen Clusters auftreten. Selbst bei anderen Prozessen, die die CPU nicht so sehr fordern, würde es in größeren Clustern zu Engpässen im Netzwerk kommen. Insofern ist dieses Ergebnis an dieser Stelle nur ein weiteres Indiz dafür, dass dieses kleine unhomogene Cluster eine Sonderstellung einnimmt.

6. Zusammenfassung und Ausblick

Die Herausforderungen, die aus immer größer werdenden Datenmengen folgen, haben unter dem Stichwort Big Data neue Technologien und Prinzipien hervorgebracht oder neues Interesse an ihnen entfacht. In dieser Arbeit wurde gezeigt, dass diese modernen Methoden Daten zu betrachten auch im wissenschaftlichen Bereich zu Fortschritten bei der Datenanalyse führen kann.

Die in dieser Ausarbeitung grundlegenden Technologien beginnen mit dem Apache Hadoop Framework, das mit dem MapReduce Programmiermodell und dem HDFS eine geeignete Plattform für Anwendungen nach dem code-to-data-Prinzip bietet. Es ist damit zum de facto Standard für Analyse von Big Data geworden.

Die zweite wichtige und für diese Arbeit zentrale Technologie ist das streams Framework. Als dessen Erweiterung wurde im Rahmen dieser Arbeit streams-mapred entwickelt, dass Apache Hadoop nutzt, um damit die vorgestellten modernen Prinzipien umzusetzen. Durch die Erweiterung des streams Frameworks um die Fähigkeit auf Hadoop Clustern ausgeführt zu werden, wurde eine neue Möglichkeit geschaffen, bestehende Analyseprogramme mit Hilfe verteilten Rechnens zu beschleunigen.

Am Beispiel der Daten des Ersten G-APD Cherenkov Teleskops wurde überprüft, ob das Framework diese Ziele erreicht. Es wurde zunächst gezeigt, dass die Entwicklung verteilter Hadoop Programme mit Hilfe von streams-mapred einfach durchzuführen ist, wenn das streams Framework bereits eingesetzt wird. Die anschließend durchgeführten Laufzeitanalysen lassen erkennen, dass die verteilte Ausführung bestehender Prozesse schneller ablaufen kann, als das mit einem einzelnen Rechner möglich ist.

Es stellte sich jedoch auch heraus, dass ein Cluster, das für die Verwendung mit Hadoop entworfen wurde, eine entscheidende Komponente ist, um hohe Leistungen zu erzielen. Durch unhomogene Knotenleistung, Virtualisierung und generell geringer Knotenanzahl, hat das verwendete Cluster sogar Probleme mit einem einzelnen Rechner zu konkurrieren, der hier jedoch über eine modernere Architektur verfügt. Ohne diese Handicaps dürfte die erzielte Leistungssteigerung gegenüber Einzelrechnern nochmals um einiges höher ausfallen.

6.1. Ausblick

Streams-mapred kann nach Abschluss dieser Arbeit als solider Grundstein für weitere Entwicklungen gesehen werden. Ein wichtiger nächster Schritt wird es sein, weitere Datenanalyseprogramme mit Hilfe des Frameworks zu entwickeln und zu testen. Zur Zeit befindet sich beispielsweise eine Bachelorarbeit in Vorbereitung, die eine

Analyse der Daten des ViSTA-TV Projekts[1] mit Hilfe von streams-mapred zum Ziel hat.

Für die Weiterentwicklung von streams-mapred wäre es außerdem interessant, Datenanalysen auf größeren und besser für Hadoop geeigneten Clustern auszuführen. Es dürfte sich zeigen, dass das Framework schon jetzt gut geeignet ist, um Anwendungen für sehr große Cluster zu entwickeln. Bessere Konfigurationsmöglichkeiten und allgemeine Leistungsverbesserungen werden jedoch vermutlich nötig sein, um in einem solchen System die bestmöglichen Ergebnisse zu erzielen. Insbesondere die Implementierung und Untersuchung anderer Speichermethoden wie Apache HBase und anderen Eingabeformaten könnten zu interessanten Entwicklungen führen. Die Einbindung weiterer Hadoop Bestandteile, wie beispielsweise *Combinern* oder *Partitionern* [17], würden je nach Anwendung zu großen Leistungssteigerungen führen.

Auch für das FACT Projekt sind die erreichten Ergebnisse nur ein Zwischenschritt, der neue Entwicklungen ermöglichen wird. Nachdem der Nutzen der Technologie gezeigt wurde, kann die Zusammenarbeit mit dem Projekt ausgeweitet und größere Analysen in Angriff genommen werden.

Die Entwicklung von Software für ein größeres und produktiv einzusetzendes Big Data System wurde bereits zu Anfang dieser Ausarbeitung in Kapitel 2.4 in Aussicht gestellt. Mit dem streams Framework und seinen jetzt noch vielfältigeren Erweiterungen könnte ein System nach der Lambda Architektur aufgebaut werden. Die darin vereinten verschiedenen Aspekte der Datenanalyse – Batchverarbeitung, Echtzeitverarbeitung und das Beantworten beliebiger Fragen – könnten mit dem selben Code und einer einzigen gemeinsamen „Sprache“ umgesetzt werden.

Diese Menge an möglichen Weiterentwicklungen zeigt, dass diese Arbeit eine Grundlage für weitere interessante Forschung schafft.

[1] ViSTA-TV Homepage: http://vista-tv.eu/ – Abgerufen am 15.12.2013 22:53

Literaturverzeichnis

[1] *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[2] *Big Data - A New World of Opportunities*. Networked European Software and Services Initiative, 12 2012.

[3] Anderhub, H, M Backes, A Biland, V Boccone, I Braun, T Bretz, J Buss, F Cadoux, V Commichau, L Djambazov et al.: *Design and Operation of FACT – The First G-APD Cherenkov Telescope*. 04 2013.

[4] Anderhub, H, M Backes, A Biland, A Boller, I Braun, T Bretz, S Commichau, V Commichau, M Domke, D Dorner et al.: *FACT – the First Cherenkov Telescope using a G-APD Camera for TeV Gamma-ray Astronomy (HEAD 2010)*. 10 2010.

[5] Bockermann, Christian und Hendrik Blom: *The streams Framework*. Technischer Bericht 5, TU Dortmund, 12 2012.

[6] Borthakur, Dhruba: *The hadoop distributed file system: Architecture and design*. Hadoop Project Website, 2007.

[7] Dean, Jeffrey und Sanjay Ghemawat: *MapReduce: simplified data processing on large clusters*. Commun. ACM, 51(1):107–113, Januar 2008.

[8] George, Lars: *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[9] Gropp, William D, Ewing L Lusk und Anthony Skjellum: *Using MPI: portable parallel programming with the message-passing interface*, Band 1. the MIT Press, 1999.

[10] Hurwitz, Judith, Alan Nugent, Fern Dr. Halper und Marcia Kaufman: *Big Data for Dummies*. John Wiley & Sons, Inc, 2013.

[11] Kuang, Hairong, Konstantin Shvachko, Nicholas Sze, Sanjay Radia und Robert Chansler: *Append/Hflush/Read Append/Hflush/Read Design*. Technischer Bericht, Yahoo! HDFS team, 08 2009.

[12] Laney, Doug: *3D Data Management: Controlling Data Volume, Velocity, and Variety*. META Delta, 2001.

[13] Marz, Nathan: *Storm: Distributed and fault-tolerant realtime computation*, 2012.

[14] Marz, Nathan und James Warren: *Big Data - Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., 2013.

[15] Pence, W. D., L. Chiappetti, C. G. Page, R. A. Shaw und E. Stobie: *Definition of the Flexible Image Transport System (FITS), version 3.0*. Astronomy & Astrophysics, 2010.

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

[17] White, Tom: *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[18] Zicari, Roberto V: *Big Data: Challenges and Opportunities*. 2012.

[19] Zikopoulos, I.B.M.P., C. Eaton und P. Zikopoulos: *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. Mcgraw-hill, 2011.