

Apache Hadoop

Hauptseminar "Cloud-Plattformen und Big Data"

Dozent Steffen Rupp

von

René Gentzen

`rene.gentzen@mni.thm.de`

im WS22/23

Inhaltsverzeichnis

1	Hadoop Grundlagen	1
1.1	Technischer und geschichtlicher Hintergrund	1
1.1.1	Anforderungen von Big Data	1
1.1.2	Vertikale Skalierung	2
1.1.3	Horizontale Skalierung	2
1.1.4	Historie	3
1.2	Hadoop Core	3
1.2.1	HDFS	4
1.2.2	MapReduce	4
1.2.3	YARN	7
1.2.4	Umgang mit HDFS und MapReduce	8
2	ETL mit Pig	21
2.1	Anwendungsfälle	21
2.2	Architektur	21
2.3	Pig Latin	21
2.4	Praxis	21
2.4.1	Hinzufügen zum Cluster	21
2.4.2	Anwendung auf dem Cluster	21
3	Data Ingestion	23
3.1	Sqoop	23
3.2	Flume	23
4	Datawarehousing mit Hive	25
4.1	Anwendungsfälle	25
4.1.1	Unterschiede zu Pig	25
4.2	Architektur	25
4.3	Interaktion	25
4.3.1	HiveQL	25
4.3.2	CLI	25
4.3.3	Java API	25
4.4	Praxis	25
4.4.1	Hinzufügen zum Cluster	25
4.4.2	Einrichtung einer Datenbank	25
4.4.3	Einlesen von Daten im CLI	25
4.4.4	Einlesen von Daten mit Sqoop	25

4.4.5	Absetzen einer Query	25
5	NoSQL mit HBase	27
5.1	Anwendungsfälle	27
5.1.1	CAP-Theorem	27
5.1.2	ACID und BASE	27
5.2	Architektur	27
5.3	Interaktion	27
5.3.1	HBase Shell	27
5.3.2	Java API	27
5.4	Praxis	27
5.4.1	Hinzufügen zum Cluster	27
5.4.2	Einrichtung einer Datenbank	27
5.4.3	Einlesen von Daten	27
5.4.4	Datenmigration aus einem RDBMS	27
5.4.5	Absetzen einer Query	27
6	Streaming mit Kafka	29
6.1	Anwendungsfälle	29
6.2	Architektur	29
6.3	Interaktion	29
6.3.1	Who knows	29
6.4	Praxis	29
6.4.1	Hinzufügen zum Cluster	29
6.4.2	Maybe, vielleicht kann man ja was zeigen	29
7	Hadoop heute	31
7.1	Aktuelle Anwendungsbeispiele zu Hadoop	31
7.1.1	AirBnB	31
7.2	Apache Spark als Gold Standard	31
7.2.1	Kann eh alles besser	31
Appendix		33
1	Mapper-Skript für Hadoop Streaming	33
2	Startskript für den NCDC Concatenation Job	35
Literatur		37

Abbildungsverzeichnis

1.1	Architektur des HDFS	5
1.2	Die Phasen von MapReduce	6
1.3	Der MapReduce Dataflow	6
1.4	Start aller Prozesse beim Cluster Startup	9
1.5	Hadoop Core Prozesse in der HDP Sandbox	10
1.6	Replication Factor Einstellung in Ambari	11
1.7	NCDC Archive für die Fallstudie	12
1.8	Erstellung des JARs mit Wetterdaten	12
1.9	Inputdatei des MapReduce Jobs	12
1.10	Dateitransfer mit scp	13
1.11	Dateiupload in das HDFS	13
1.12	Upload der NCDC Datensätze	15
1.13	Setzen von Dateiberechtigungen im Ambari File View	15
1.14	MapReduce Konsolenausgabe	18
1.15	MapReduce Ausgabedateien	18
1.16	Zusammengefügte, komprimierte Wetterdaten im HDFS	19
1.17	Fehlerhafte Anzeige der Daten im File View	19

1 Hadoop Grundlagen

“Die Apache Hadoop Softwarebibliothek ist ein Framework, das die über Computercluster verteilte Verarbeitung großer Datensätze mit einfachen Programmiermodellen ermöglicht. Es ist so konzipiert, dass es von einzelnen Servern bis hin zu Tausenden von Rechnern skaliert werden kann, von denen jeder lokale Rechenleistung und Speicherplatz bietet. Anstatt sich auf Hardware zu verlassen, um eine hohe Verfügbarkeit zu gewährleisten, ist die Bibliothek selbst so konzipiert, dass sie Ausfälle auf der Anwendungsebene erkennt und bewältigt, so dass ein hochverfügbarer Dienst auf einem Cluster von Computern bereitgestellt wird, von denen jeder für sich für Ausfälle anfällig sein kann.”[1]

So beschreibt (übersetzt aus dem Englischen) die Apache Software Foundation ihr Top Level Projekt **Apache Hadoop**. Diese Arbeit wird eine Einführung in Hadoop und die Komponenten im Hadoop Ecosystem geben. Dabei wird die Benutzung im Vordergrund stehen. Theoretische Hintergründe werden nur so weit vermittelt, dass dem Leser die Einordnung der vorgestellten Technologien in den größeren Kontext von Big Data-Technologien möglich wird. Auch auf eine detaillierte Beschreibung der Installation und Konfiguration von Hadoop wird verzichtet (siehe dazu die offizielle Dokumentation¹). Es soll anhand von Anwendungsfällen demonstriert werden, wie die einzelnen Hadoop Komponenten zur Lösung bestimmter Problemstellungen eingesetzt werden können.

1.1 Technischer und geschichtlicher Hintergrund

1.1.1 Anforderungen von Big Data

“Der Begriff „Big Data“ bezieht sich auf Datenbestände, die so groß, schnelllebig oder komplex sind, dass sie sich mit herkömmlichen Methoden nicht oder nur schwer verarbeiten lassen.”[2]

Schon Anfang der Neunziger war es nicht mehr praktikabel, Webseiten händisch, zum Beispiel in “Web Directories”, zu katalogisieren. Man wollte Nutzern trotzdem die Möglichkeit geben, Informationen durch das Durchsuchen zentraler Anlaufstellen ausfindig zu machen. Automatisierte Tools, die sogenannten “Web Crawler” wurden erfunden, um diese Arbeit zu übernehmen.[3]

Das Internet erlebte in den letzten Jahren des 20. Jahrhunderts ein explosionsartiges Wachstum an Nutzern und Webseiten, und damit auch an Informationen, die katalogisiert werden

¹<https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html>

mussten.[4] Um eine immer größer werdende Menge an Informationen verarbeiten zu können, gibt es zwei Ansätze der Skalierung: Vertikale und horizontale Skalierung. Diese sollen in den folgenden Abschnitten erläutert werden, um die Designphilosophie hinter Hadoop zu verstehen.

1.1.2 Vertikale Skalierung

Bei der vertikalen Skalierung ("scaling up") werden *einem* System mehr Ressourcen wie zum Beispiel größerer Speicher, oder eine schnellere CPU hinzugefügt. Dadurch bekommt man einen Performance-Gewinn: Man kann mehr Daten speichern, oder Berechnungen werden schneller fertig gestellt. Ein großer Vorteil der vertikalen Skalierung ist, dass Anwendungsprogramme in der Regel nicht angepasst werden müssen, um vom diesem Performance-Wachstum zu profitieren. Wenn man eine 5TB große Festplatte gegen eine 10TB Festplatte austauscht, dann hat man den Speicherplatz eines Servers vertikal skaliert. Die darauf laufenden Programme müssen nicht angepasst werden, sondern man kann einfach doppelt so viele Daten speichern.[5]

Vertikale Skalierung hat drei große Nachteile: Erstens kann man nicht unbegrenzt vertikal skalieren. Ein Server kann physisch nur eine begrenzte Anzahl an Hardware aufnehmen. Zweitens wächst die Performance eines Systems bei vertikaler Skalierung höchstens linear[6], die Kosten allerdings nicht[7]. Heutzutage kann man gerade bei Cloud-Anbietern sehr leistungsfähige Systeme bei linearem Preisanstieg mieten.[8] Sucht man aber noch mehr Performance in *einem* System, dann steigen die Kosten exponentiell[9]. Drittens skalieren nicht alle Faktoren in einem System gleich gut vertikal. Die Speicherkapazität von SSDs ist zum Beispiel seit 1978 von 45MB auf 100TB gestiegen (Faktor $2222,22 \cdot 10^3$), während sich die Datenrate nur von 1.5MB/s auf 500/460MB/s (Sequential Read/Write) erhöht hat (Faktor $0,333 \cdot 10^3$).[9][10]

1.1.3 Horizontale Skalierung

Ein Cluster ist ein Verbund aus Computern (Nodes), die wie ein einziger, deutlich leistungsfähigerer Computer arbeiten. Aufgaben und Daten werden in kleinere Teile zerlegt und auf alle Nodes im Cluster aufgeteilt, welche dann parallel Teilaufgaben lösen. Ergebnisse werden zusammengefügt und zurückgegeben. Anders als bei der vertikalen Skalierung kann man gerade in Zeiten des Cloud Computings praktisch unendlich horizontal skalieren. Allerdings muss man dafür kompliziertere Anwendungslogik verwenden, die mit der parallelen Ressourcenverteilung eines Clusters funktioniert.[11]

Bei der horizontalen Skalierung ("scaling out") werden einem Cluster zur Leistungssteigerung zusätzliche Nodes hinzugefügt. So kann ein Rechner zum Beispiel 500MB/s von seiner Festplatte lesen und verarbeiten, zehn Rechner lesen und verarbeiten in dieser Zeit allerdings 5000MB/s und können ihre Teilergebnisse anschließend zu einer Antwort zusammenfügen. Hierbei entsteht zwar zusätzlicher Netzwerk- und Verwaltungsaufwand ("Overhead"), aber die Leistungsfähigkeit des Clusters wächst mit jedem hinzugefügten Node. Ein horizontal

skalierbares System ist mit höheren anfänglichen Kosten verbunden, kann dann aber bei linearem Kostenaufwand praktisch unendlich skaliert werden.^[7]

Wie im eingänglichen Zitat erwähnt, setzt Hadoop auf eben dieses Prinzip der Skalierbarkeit, um “die über Computercluster verteilte Verarbeitung großer Datensätze mit einfachen Programmiermodellen”^[1] als Dienst mit hoher Verfügbarkeit anzubieten. Die Technologien, die konkret dahinter stecken, werden im nächsten Abschnitt behandelt.

1.1.4 Historie

2002 begannen Doug Cutting und Mike Cafarella ihre Arbeiten an Apache Nutch², einer Open Source Web Search Engine als Teil des Apache Lucene Projekts³. Die beiden mussten Wege finden, um ihr Projekt auf die Milliarden Webseiten des Internets zu skalieren. 2003 veröffentlichte Google ein Whitepaper zur Architektur des Google File System (GFS), Googles eigenem verteilten Dateisystem.⁴ Als Google 2004 dann ein weiteres Whitepaper zum MapReduce Programmiermodell veröffentlichte⁵, sahen Cutting und Cafarella darin die Lösung für Nutch’s Skalierungsproblem. Sie implementierten eigene Versionen von MapReduce als Processing Engine und des GFS zur Datenhaltung (NDFS, Nutch Distributed File System) als Basis für Nutch. Da diese beiden Komponenten mannigfaltige Anwendungsfälle außerhalb der Web-Suche bedienen konnten, wurden sie 2006 als eigenes Projekt Apache Lucene unterstellt und erhielten den Namen **Hadoop**. Ungefähr zur gleichen Zeit wurde Doug Cutting von Yahoo! rekrutiert, um Hadoop dort mit zusätzlichen Ressourcen weiterzuentwickeln. 2008 wurde Hadoop schließlich zu einem Top Level Projekt der Apache Software Foundation.⁶^[14]

1.2 Hadoop Core

Der Kern von Hadoop (Hadoop Core) besteht seit Hadoop 2.x aus vier Modulen, welche im offiziellen Download zusammengefasst sind^[1]:

- Hadoop Distributed File System (HDFSTM): Hadoops verteiltes Dateisystem
- Hadoop MapReduce: Hadoops Parallel Processing Engine für große Datenmengen
- Hadoop YARN: Ein Framework für Job Scheduling und Ressourcenverwaltung im Cluster
- Hadoop Common: Unterstützende Programme für die anderen Hadoop-Module

²<https://nutch.apache.org/>

³<https://lucene.apache.org/>

⁴^[12], The Google File System.

⁵^[13], MapReduce: Simplified Data Processing on Large Clusters.

⁶<https://hadoop.apache.org/>

Diese Komponenten bringen alles mit, was man zur verteilten Verarbeitung und Speicherung großer Datenmengen benötigt. Dazu schreibt man in der Regel Java-Applikationen, die bestimmte Klassen aus den Bibliotheken von Hadoop ableiten. Beispiele zur Java API und zur Streaming API von MapReduce werden in den Abschnitten 1.2.4 und 1.2.4 gezeigt.

1.2.1 HDFS

Das HDFS ist ein Dateisystem, welches dem Anwender eine Abstraktionsschicht über verteilt gespeicherte Daten bietet. Dateien lassen sich ganz normal über einen Dateipfad im HDFS ansprechen, auch wenn sie im Hintergrund in Einzelteilen über viele Nodes verteilt gespeichert sind. Das HDFS ist für den Betrieb auf Clustern aus sogenannter *Commodity Hardware* konzipiert. Commodity Hardware ist günstige, leicht zu ersetzende Hardware. Bei Commodity-Hardware-Clustern wird nicht etwa versucht, Ausfälle einzelner Nodes durch den Einsatz von besonders ausfallsicherer (und somit teurer) Hardware zu verhindern. Fällt ein Node aus, was in einem Cluster von hunderten Maschinen kein Sonderfall ist, übernimmt ein anderer Node dessen Arbeit, ohne dass dadurch die Verfügbarkeit des Clusters beeinträchtigt wird. Das HDFS setzt dafür auf die Konzepte von Blöcken, Replikation und Redundanz.[15]

Ein vollwertiger Hadoop Cluster (Hadoop im *fully-distributed Mode*) besteht aus mindestens einem Master, dem **NameNode**, und einem oder mehr Workern, den **DataNodes** (vgl. Abb. 1.1). Um Dateien im HDFS zu speichern (Beispiel siehe 1.2.4), teilt ein *Client*-Prozess die Dateien in Blöcke von standardmäßig 128MB auf und kontaktiert den NameNode. Der NameNode hat einen Überblick über den verfügbaren Speicherplatz aller DataNodes und designiert manche davon, um einige der Blöcke aufzunehmen. Der NameNode achtet außerdem darauf, dass jeder einzelne Block repliziert und auf unterschiedlichen DataNodes gespeichert wird. Standardmäßig verteilt Hadoop drei Kopien eines jeden Blocks im Cluster, was durch den **Replication Factor** konfiguriert werden kann. Dadurch verbraucht man zwar drei mal so viel Speicher wie bei herkömmlichen, nicht redundanten Dateisystemen, erreicht dafür aber eine sehr hohe Verfügbarkeit. Der Einsatz von Commodity Hardware hält trotz des erhöhten Speicherbedarfs die Kosten niedrig.[15]

Die DataNodes senden in regelmäßigen Abständen sogenannte *Block Reports* an den NameNode. Dieser gleicht die Block Reports mit dem Soll-Zustand des Dateisystems ab. Ist zum Beispiel in einem Node eine Festplatte ausgefallen, so sind manche Blöcke unterrepliziert. Der NameNode veranlasst DataNodes, die Kopien der betroffenen Blöcke besitzen dazu, diese an andere DataNodes zu senden, bis der Soll-Zustand des Clusters wieder hergestellt ist.

1.2.2 MapReduce

MapReduce heißt sowohl ein Programmiermodell zur parallelisierten Verarbeitung von Datensätzen, als auch die konkrete Implementierung eben dieses Modells als Komponente des Hadoop Frameworks. MapReduce macht sich mehrere Prinzipien zu Nutze, um effizient mit

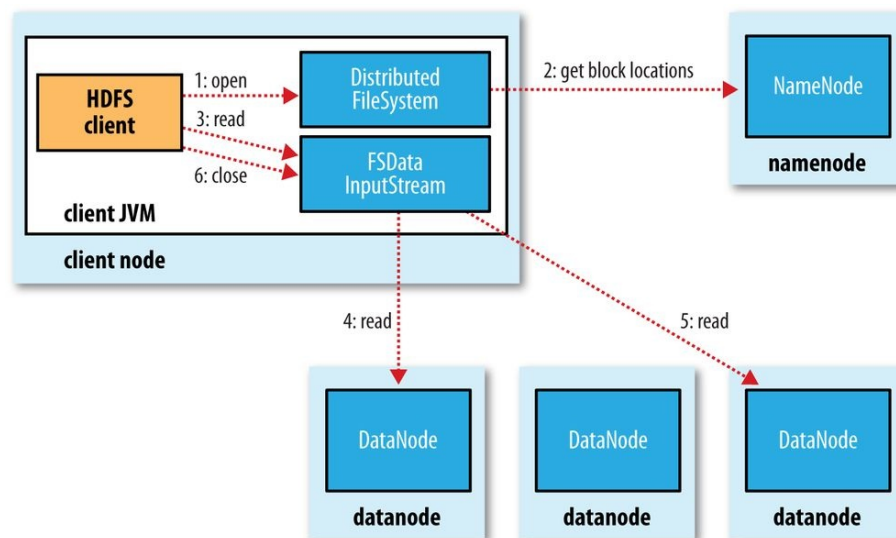


Abbildung 1.1: Architektur des HDFS(15, S.69, Figure 3-2)

großen Datenmengen umzugehen[16]:

Aufteilung: Eingabedaten werden in **InputSplits** geteilt verarbeitet. Dadurch verarbeitet ein einzelner Prozess ein logisch zusammenhängendes Datenpaket.

Parallelisierung: InputSplits werden parallel auf mehreren Nodes bearbeitet und die Ausgaben zusammengeführt. Dadurch werden auch bei großen Datenmengen hohe Datendurchsatzraten erreicht.

Datenlokalität: Der erste Teil der Verarbeitungslogik, die Mapping-Phase, wird möglichst nahe an den Daten durchgeführt; wenn möglich auf den Nodes, auf denen die Daten gespeichert sind. Ansonsten wird versucht, die Verarbeitung wenigstens auf dem gleichen Server Rack durchzuführen, um die Belastung der Netzwerkinfrastruktur so gering wie möglich zu halten.

Ein MapReduce Job besteht aus zwei Phasen: der **Map-Phase** und der **Reduce-Phase**. Logisch kann man dazwischen noch die **Sort- und Shuffle-Phase** unterscheiden (siehe Abb. 1.2).

Wie eingangs erwähnt, wird die Eingabe in InputSplits zerteilt. Diese werden wiederum in einzelne Datensätze, die **Records**, aufgespalten. Wie diese Aufteilung abläuft, wird durch das **InputFormat** bestimmt, welches vom Anwender im Programmcode festgelegt und auf das Format der Eingabedaten abgestimmt werden muss. Dabei stehen zum Beispiel *TextInputFormat* oder *KeyValueTextInputFormat* zur Verfügung. Es ist auch möglich, durch Ableiten der abstrakten Java Klasse *InputFormat* eigene InputFormats zu schreiben.[15]

Für jeden InputSplit wird ein eigener Map-Prozess (**Mapper**) gestartet. Dieser erhält alle Records des InputSplits in Form von **Key-Value-Paaren** als Eingabe. Auf jeden Record wird eine vom Anwender geschriebene Map-Funktion angewendet, die oftmals die Daten filtert und vorbereitet, zum Beispiel durch Parsen von Strings in Integer. Die Daten wer-

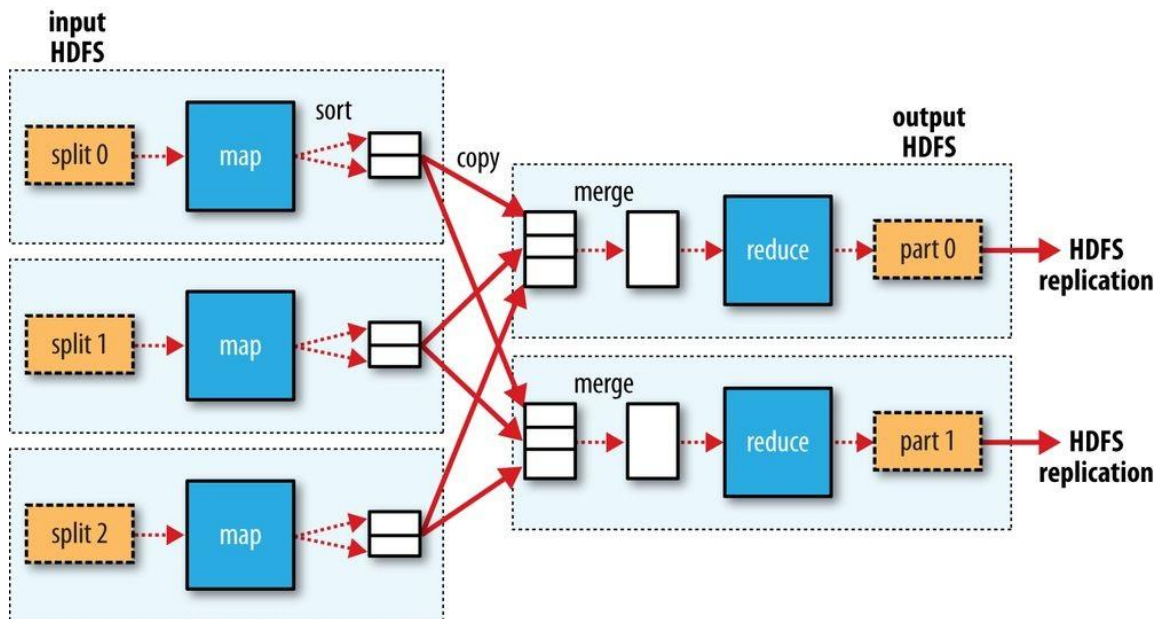


Abbildung 1.2: Die Phasen von MapReduce(15, Seite 34, Figure 2-4)

den vom Eingabeformat in bereinigte Key-Value-Paare **gemappt**. Das Ergebnis wird an Reduce-Prozesse (**Reducer**) weitergegeben. Ein Beispiel dazu wird in Abschnitt 1.2.4 besprochen.

Bevor die Key-Value-Paare an die Reducer gegeben werden, werden sie nach Keys sortiert und gruppiert. Dies geschieht in der Sort- und Shuffle-Phase. Der Input für den Reducer ist dann eine Liste mit Key-Value-Paaren, wobei die Values wiederum Listen mit den Werten sind, die von den Mappern für den jeweiligen Key gefunden wurden (vgl. Abb. 1.3).

Der Reducer wendet eine ebenfalls vom Anwender geschriebene Reduce-Funktion auf die ihm übergebenen Daten an. Für jeden Key wird die Liste aus Values zu einem einzigen Value **reduziert**, zum Beispiel durch Bestimmung des Maximums oder Aufsummierung aller Teilwerte. Die Anzahl der Reduce-Prozesse bestimmt die Anzahl der Ausgabedateien (eine Datei pro Reducer) und *kann* im Programmcode festgelegt werden. Allerdings sollte man gute Gründe haben, um die von Hadoop gewählten Werte zu überschreiben, da dies katastrophale Folgen für die Performance haben kann.[17]

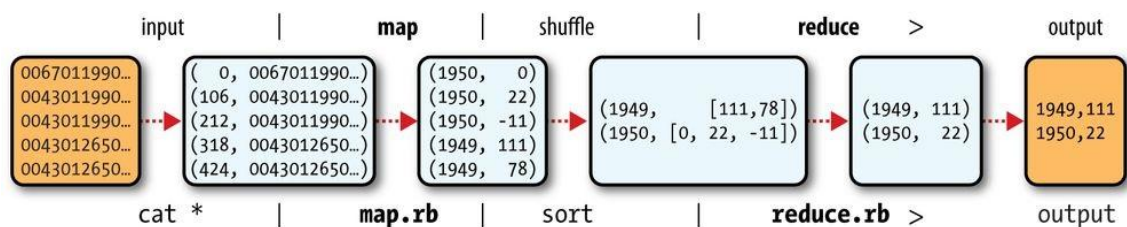


Abbildung 1.3: Der MapReduce Dataflow(15, Seite 24, Figure 2-1)

MapReduce hat immer noch eine Sonderstellung im Hadoop Ecosystem, da es die mitgelieferte Processing Engine ist. Vor Hadoop 2.x war es sogar die einzige Möglichkeit, Daten in einem Hadoop-Cluster zu verarbeiten. Eine MapReduce-Applikation zu entwickeln erfordert allerdings das Schreiben vielen Java-Codes und Problemstellungen müssen in die Phasen von Mapping und Reducing übertragen werden, selbst wenn andere Modellierungen der Problemstellung intuitiver oder leichter zu bearbeiten wären. Außerdem ist MapReduce allein für **Batch Processing** ausgelegt. Das heißt, dass man eine MapReduce-Applikation schreibt, die eine ganz bestimmte Fragestellung zu einem Datensatz beantwortet. Diese wird ausgeführt und erst nachdem alle Daten verarbeitet wurden, sieht man ein Ergebnis. Dies kann viele Minuten, Stunden oder sogar Tage dauern. Will man nun einen Parameter der Fragestellung ändern (zum Beispiel nicht mehr nach Monaten sondern nach Wochen aufgeschlüsselt), muss man die gesamte Verarbeitung des Datensatzes noch einmal durchführen. Dies steht im Konflikt mit der heute üblichen Forderung nach **visueller Datenexploration**[18]. Zuguterletzt ist MapReduce nach heutigen Standards eher langsam. Da es für die Ausführung auf Commodity Hardware entwickelt wurde, schreibt und liest es die Zwischenergebnisse der einzelnen Phasen immer wieder von der Festplatte des DataNodes. Neue Processing Engines (allen voran Apache Spark⁷), setzen viel auf **In-Memory Processing**, halten alle Daten also möglichst während der gesamten Bearbeitungszeit im Arbeitsspeicher. Das ermöglicht bis zu 40 mal schnellere Abfragen bei gleichwertigem Arbeitsaufwand. [vgl. 16, Kap. 3.19]

1.2.3 YARN

In Version 1.x von Hadoop war MapReduce sowohl für die Verarbeitung der Daten, als auch für die Ressourcenzuteilung im Cluster zuständig. Das bedeutete, dass man zwingend das MapReduce-Programmiermodell nutzen musste, um die im Hadoop Cluster gespeicherten Daten auszuwerten. Die Ressourcenverwaltung war damit ein mögliches Bottleneck, da sie bei mehreren parallel laufenden Jobs auf einem Node um Rechenzeit mit der Datenverarbeitung konkurrieren musste und neue Jobs gegebenenfalls lange nicht gestartet wurden.[16] Die größte Änderung in Hadoop 2.x war dann die Ausgliederung der Ressourcenverwaltung aus MapReduce und die Einführung einer dedizierten Ressourcenverwaltungsanwendung - **YARN** - *Yet Another Resource Negotiator*. YARN teilt eingehenden Jobs Cluster-Ressourcen zu und startet fehlgeschlagene Jobs gegebenenfalls neu. Ähnlich wie das HDFS bringt YARN eine Reihe von Prozessen mit sich, die Master- und Worker-Rollen einnehmen. Auf dem vom HDFS designierten NameNode läuft der **Resource Manager**. Dieser unterteilt sich wiederum in **Application Manager** und **Scheduler**. Auf allen DataNodes läuft jeweils ein **Node Manager**. [16]

Durch das Zusammenspiel dieser Prozesse bietet sich dem Anwender ein Interface zur verteilten Ausführung von Anwendungslogik, bei dem man sich nicht an das MapReduce-Programmiermodell halten muss. Startet man in Hadoop 2.x eine MapReduce-Applikation, ist diese eigentlich eine YARN-Applikation, bei der einem schon ein Teil des Programmieraufwands abgenommen wurde. Eine eigene YARN-Applikation zu schreiben bedeutet

⁷<https://spark.apache.org/>

hingegen, sich selbst um die logische Aufteilung der Daten zu kümmern, Cluster-Ressourcen wie CPU und RAM in Form sogenannter **Container** von YARN anzufordern und dafür zu sorgen, dass der auszuführende Programmcode für alle DataNodes (am besten gespeichert im HDFS) verfügbar ist. YARN reiht die Ausführung der angeforderten Container auf verschiedenen DataNodes in Warteschlangen ein, kopiert den Anwendungscode aus dem HDFS auf diese Nodes und überwacht die erfolgreiche Ausführung der Anwendung.

1.2.4 Umgang mit HDFS und MapReduce

Zur Installation von Hadoop kann man die offizielle Distribution⁸ benutzen und komplett selbst konfigurieren. Dabei besteht die Möglichkeit, Hadoop in drei verschiedenen Modi zu betreiben: **Single Node**, **Pseudo-distributed** und **Fully-distributed**. Erstere beide sind zum Testen und Entwickeln, letztere für den tatsächlichen Einsatz im Cluster gedacht[vgl. 16, Kap. 3.4]. Weiterhin haben diverse kommerzielle Anbieter wie Cloudera⁹ eigene Hadoop Distributionen entwickelt, die sie in Form von vorkonfigurierten VM- oder Docker-Images teilweise kostenlos zur Verfügung stellen. Cloudera zum Beispiel ergänzt diese Distributionen aber mittlerweile durch Cloudlösungen¹⁰. Cloudanbieter wie Amazon, Google und Microsoft bieten fertig konfigurierte und voll verwaltete Cluster auf ihren jeweiligen Cloudplattformen an (Amazon EMR¹¹, Google Dataproc¹² und Azure HDInsight¹³).

Single Node Setup

Hadoop bietet unzählige Einstellungsmöglichkeiten während der Installation und sie alle zu behandeln würde den Rahmen dieser Arbeit sprengen. Daher wird in diesem Abschnitt ein VirtualBox Image von Cloudera (die Hortonworks Data Platform (HDP) Sandbox)¹⁴ genutzt. Dieses kann auf dem eigenen Rechner oder auf einem Remote-Host gestartet werden¹⁵ und bietet Zugriff auf eine voll konfigurierte Installation von Hadoop im Single Node Modus. Zusätzlich sind noch ergänzende Komponenten aus dem Hadoop Ecosystem installiert, auf die in späteren Kapiteln eingegangen wird.

Hadoop und Ambari in der HDP Sandbox VM

Im Single Node Modus laufen alle Hadoop-Prozesse auf *einem* Host(sprich Rechner). Dieser Modus ist für die Entwicklung und zum Testen von Hadoop gedacht, da es keinen praktischen Nutzen bringt, ein Framework zur verteilten Datenverarbeitung ohne die entsprechende Verteilung über einen Cluster zu betreiben. Auch wenn in dieser VM nur ein Cluster

⁸<https://hadoop.apache.org/releases.html>

⁹<https://de.cloudera.com/>

¹⁰<https://de.cloudera.com/products/cloudera-data-platform.html>

¹¹<https://aws.amazon.com/emr/features/hadoop/>

¹²<https://cloud.google.com/dataproc>

¹³<https://azure.microsoft.com/en-us/products/hdinsight/#overview>

¹⁴Download: <https://www.cloudera.com/downloads/hortonworks-sandbox/hdp.html>

¹⁵Installation: <https://www.cloudera.com/tutorials/sandbox-deployment-and-install-guide.html>

bestehend aus einem Node aufgesetzt wurde, steht das Cluster-Verwaltungs-Tool **Apache Ambari**¹⁶ zur Verfügung. Normalerweise ist das Aufsetzen eines Hadoop Clusters mit dem Bearbeiten vieler XML-Konfigurationsdateien und der Ausführung von Start-Bash-Skripten auf allen Nodes verbunden. Auf diese Weise bestimmt man, welcher Node der NameNode des HDFS werden soll, welche Nodes DataNodes werden, wo der ResourceManager von YARN läuft und so weiter. Ambari bietet einem für all das (und noch viel mehr) eine übersichtliche Weboberfläche. Diese erreicht man nach Starten der HDP Sandbox unter <http://localhost:8080>. Die Rechte und Anwendungsszenarien der verschiedenen Nutzeraccounts, eine Anleitung zum (Zurück)setzen des Admin-Passworts und weitere Schritte nach der Installation findet man auf Clouderas Hilfeseite zur Sandbox¹⁷. Loggt man sich als 'admin' zeitnah nach dem Start der VM in das Ambari Dashboard ein, kann man im Header des Dashboards auf das Zahnrad klicken und den Prozess in Arbeit sehen, der die Dienste aller Hadoop Komponenten startet (siehe Abb. 1.4).

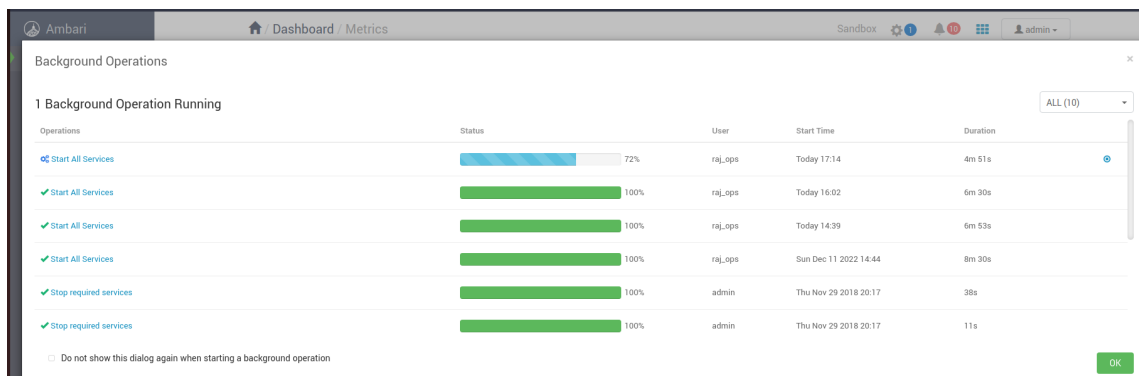


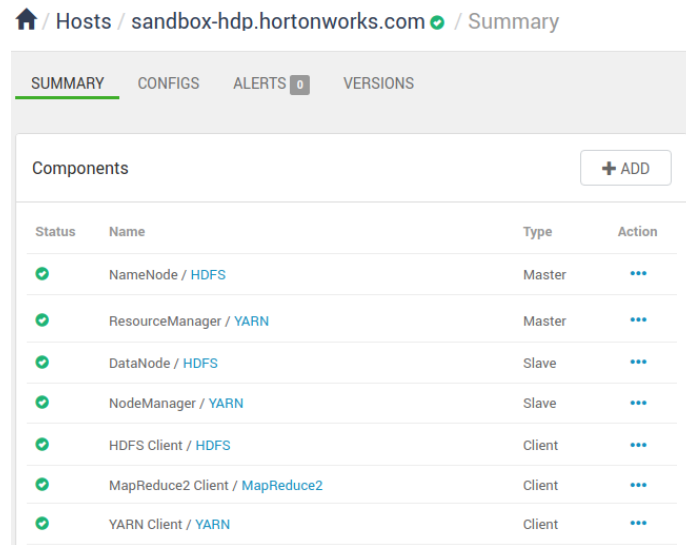
Abbildung 1.4: Start aller Dienste beim Cluster Startup

Ist der Prozess abgeschlossen, kann man im Navigationsmenü auf der linken Seite unter dem Eintrag *Hosts* eine Auflistung aller Nodes im Cluster (hier nur ein einziger) sehen. Durch Anklicken des Hostnamens gelangt man in die Host-Übersicht, wo man unter anderem die Liste der laufenden Komponenten findet. Abbildung 1.5 zeigt ein Bild davon. Der Übersicht halber wurden nur die Hadoop Core Komponenten abgebildet. Wie man sehen kann, laufen auf diesem Host alle Master-, Worker-, und Client-Prozesse gleichzeitig. Man findet die in Abschnitt 1.2.3 angesprochenen YARN-Komponenten ResourceManager und NodeManager, die HDFS-Komponenten DataNode und NameNode aus Abschnitt 1.2.1, sowie die Client-Dienste für YARN, HDFS und MapReduce wieder.

Im Tab *Configs* kann man die sonst über viele XML-Dateien verstreuten Einstellungen der Komponenten vornehmen. So kann man zum Beispiel unter *HDFS -> Advanced -> General -> Block replication* (siehe Abb. 1.6) den Replication Factor des HDFS verändern. Dieser ist in der Sandbox auf 1 gestellt, da es im Single Node Modus nur einen Node gibt und das Speichern mehrerer Block-Replika auf dem gleichen Node keinen Vorteil bringt.

¹⁶<https://ambari.apache.org/>

¹⁷<https://www.cloudera.com/tutorials/learning-the-ropes-of-the-hdp-sandbox.html>



The screenshot shows the 'Summary' page of the Hadoop Sandbox. At the top, there is a breadcrumb navigation: Home / Hosts / sandbox-hdp.hortonworks.com / Summary. Below this is a tabbed interface with 'SUMMARY' selected, and other tabs for 'CONFIGS', 'ALERTS' (with a count of 0), and 'VERSIONS'. The main content area is titled 'Components' and features a '+ ADD' button. A table lists the components with columns for Status, Name, Type, and Action. All components are shown with a green status icon.

Status	Name	Type	Action
✓	NameNode / HDFS	Master	...
✓	ResourceManager / YARN	Master	...
✓	DataNode / HDFS	Slave	...
✓	NodeManager / YARN	Slave	...
✓	HDFS Client / HDFS	Client	...
✓	MapReduce2 Client / MapReduce2	Client	...
✓	YARN Client / YARN	Client	...

Abbildung 1.5: Hadoop Core Prozesse in der HDP Sandbox

Fallstudie Globales Wetter

Als Fallstudie für die Nutzung von Hadoop werden die täglichen Zusammenfassungen aller Wetterstationen der Welt aus dem Katalog der National Centers for Environmental Information¹⁸ benutzt. Dieses Beispiel wurde größtenteils entnommen aus Hadoop: the definitive guide, (S. 19-30, 693-695). Da durch das lokale Setup die Kapazitäten von Hadoop auf die des Host-Computers beschränkt sind, werden hier nur die Datensätze der Jahre 2016 bis 2022 in komprimierten Archiven heruntergeladen¹⁹. Diese enthalten jeweils die gesammelten Aufzeichnungen eines Jahres in Form von CSV-Dateien mit geringer Dateigröße (siehe Abb. 1.7). Das HDFS zeigt jedoch seine Stärken erst bei Datensätzen im Gigabyte- bis Terabyte-Bereich und kann nicht effizient mit vielen kleinen Dateien umgehen²⁰.

Das HDFS und MapReduce arbeiten außerdem mit wenigen großen Dateien wesentlich besser als mit vielen kleinen Dateien.[15, (S. 19-30, 693-695)] Deshalb wird der erste Schritt sein, die Daten in das HDFS zu laden. Anschließend wird eine MapReduce-Applikation mit der MapReduce Streaming-API geschrieben. Diese wird nur aus einer Map-Phase bestehen und die Dateien säubern und zusammenzuführen. Zuguterletzt wird eine weitere MapReduce-Applikation mit der Java-API geschrieben, die die Daten auswertet. In späteren Abschnitten werden weitere Komponenten des Hadoop Ecosystems vorgestellt, mit denen die Dateien weiter verarbeitet werden.

¹⁸<https://www.ncei.noaa.gov/access/search/data-search/global-summary-of-the-day>

¹⁹Download: <https://www.ncei.noaa.gov/data/global-summary-of-the-day/archive/>

²⁰vgl. 19, Assumptions and Goals -> Large Data Sets.

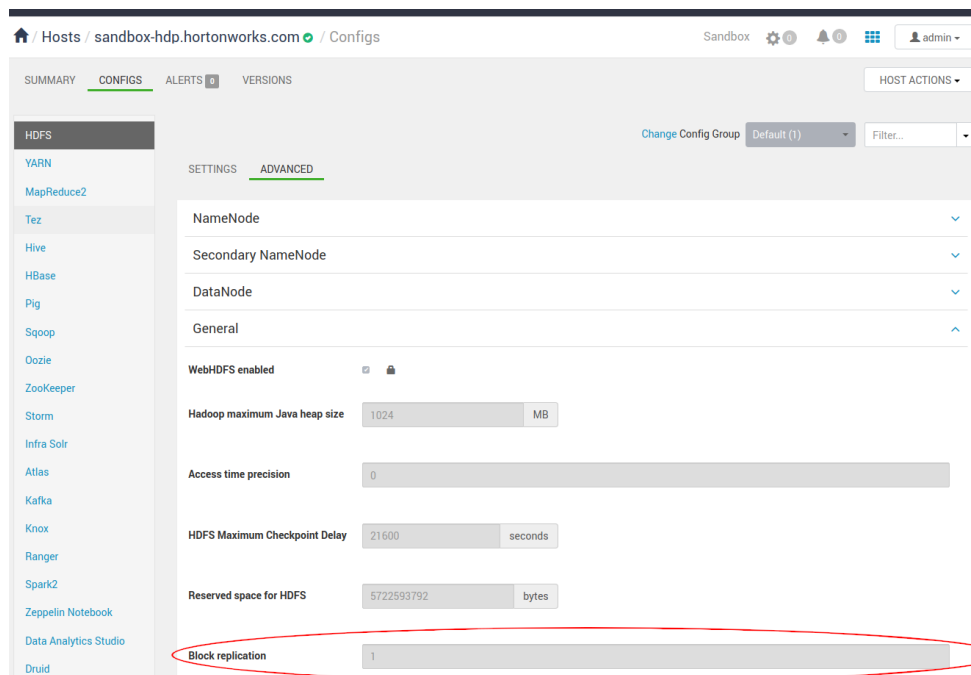


Abbildung 1.6: Replication Factor Einstellung in Ambari

Umgang mit dem HDFS

Es gibt verschiedene Wege, um Dateien ins HDFS zu laden²¹. In der Sandbox wurde ein Ambari View, der **Files View**²², erstellt. Diesen erreicht man über das Kachel-Icon im Header des Dashboards²³. Der Files View bietet einen HDFS Dateibrowser mit Funktionalitäten wie Datei-Upload direkt von der lokalen Maschine, Ordnererstellung und Rechteverwaltung. Nicht alle Nutzer der Sandbox haben die nötigen Rechte zur Ausführung der Befehle. Daher sollte man sich mit dem dafür vorgesehenen Nutzer²⁴ `user: maria_dev` `password: maria_dev` in Ambari einloggen. Man kann über den Files View aber nur eine Datei zur Zeit hochladen. Das Hochladen von Archiven (gestestet mit `.tar.gz` und `.jar`) schließt ebenfalls nie ab. Da für den nachfolgenden Schritt ein JAR aus dem HDFS als Input eingebunden werden soll, wird ein HDFS Client Prozess verwendet, der mit dem Hadoop Cluster verbunden ist. Dafür ist es nötig, Hadoop lokal installiert zu haben, oder man kopiert erst per `scp` die Dateien auf das lokale Dateisystem eines Nodes im Cluster, auf dem ein HDFS Client läuft. Hierfür verwendet man zum Beispiel den NameNode.

Schritt 1 ist das Erstellen des JARs `ncdc.jar` mit den Eingabedateien des NCDC (siehe

²¹ 3 Beispiele: <https://community.cloudera.com/t5/Support-Questions/Import-data-from-remote-server-to-HDFS/m-p/233149/highlight/true#M194979>

²² https://docs.cloudera.com/HDPDocuments/Ambari-2.7.4.0/using-ambari-views/content/amb_using_files_view.html

²³ http://<localhost oder Sandbox Hostname>:8080/#/main/view/FILES/auto_files_instance

²⁴ <https://www.cloudera.com/tutorials/learning-the-ropes-of-the-hdp-sandbox.html#login-credentials>

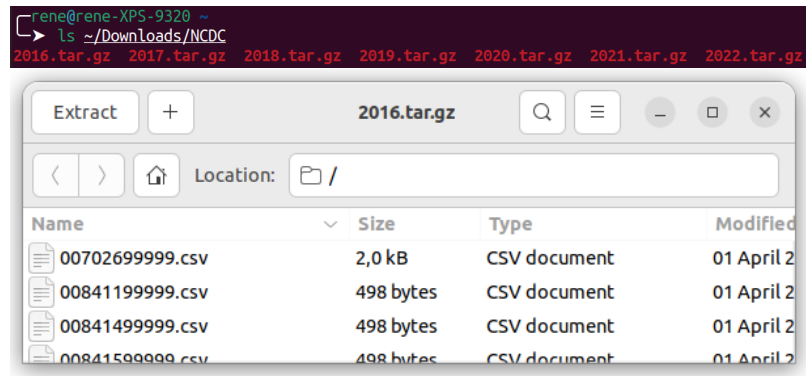


Abbildung 1.7: NCDC Archive für die Fallstudie

Abb. 1.8). Hadoop wird dieses nachher dem MapReduce Job bereitstellen und automatisch entpacken. **Schritt 2** ist das Erstellen der Textdatei `file_names.txt` mit den Dateipfaden der Unterordner des JARs (siehe Abb. 1.9). Diese Datei wird der Input des MapReduce Jobs.

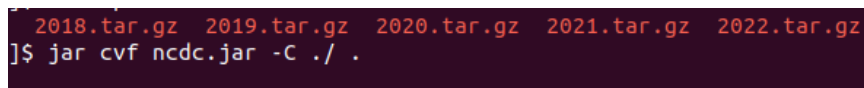


Abbildung 1.8: Erstellung des JARs mit Wetterdaten

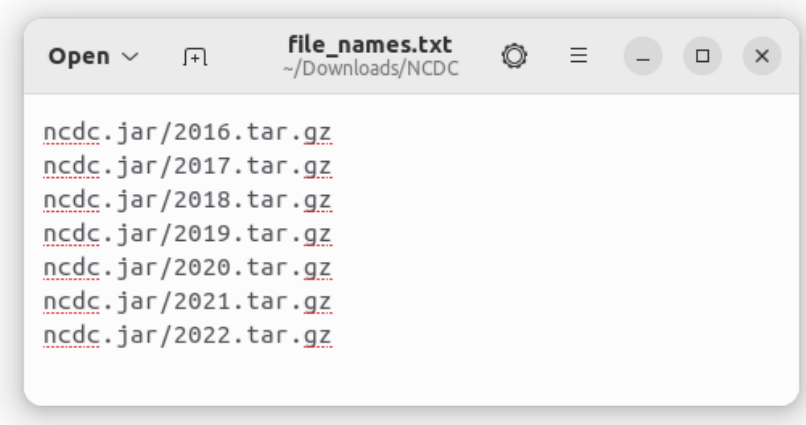


Abbildung 1.9: Inputdatei des MapReduce Jobs

In **Schritt 3** gilt es, die bisher erstellten Dateien ins HDFS zu übertragen. Dazu muss man sich per `ssh` als Benutzer `maria_dev` mit der Sandbox verbinden. Um die Ordnerstruktur im HDFS zu erstellen, nutzt man entweder den Ambari File View, oder den Befehl `hdfs dfs -mkdir -p /user/maria_dev/input`. `hdfs` ist das Programm zur Interaktion mit dem Hadoop Distributed File System. `dfs` startet die **Filesystem Shell**, mit der man eine

Reihe von "shell-artigen" Kommandos²⁵ direkt auf dem HDFS anwenden kann. An dieser Stelle könnte zum Beispiel auch `dfsadmin` stehen, wenn es um die Administration des Dateisystems als solches ginge (einen DataNode abschalten etc.). FS Shell Commands beginnen mit einem Bindestrich, darauf folgt ein Befehl und dahinter kommen die Parameter. `-mkdir -p /user/maria_dev/input` funktioniert genau wie sein Gegenstück aus einer normalen Shell. Der Ordner `input` sollte noch für alle Benutzer als schreibbar markiert werden, da dort hin später die zusammengefassten CSV-Dateien geschrieben werden sollen. Dazu dient der Befehl `hdfs dfs -chmod 777 /user/maria_dev/input`. Ist die Ordnerstruktur erstellt, kann man von seiner lokalen Maschine mit `scp -P <Port> <lokaler Pfad> maria_dev@<Hostname>:<Host Pfad>` die bisher erstellten Dateien (JAR und Textdatei) auf das lokale Dateisystem des NameNodes (der Sandbox) kopieren (Abb. 1.10). Anschließend werden diese per FS Shell mit dem Befehl `-moveFromLocal` in das HDFS geladen, damit die Dateien vom lokalen Dateisystem des NameNodes entfernt werden, nachdem sie im HDFS gespeichert wurden (Abb. 1.11).

```

rene@rene-XPS-9320 ~
➤ scp -P 2222 /home/rene/Downloads/ncdc.jar maria_dev@sandbox-hdp.hortonworks.com:/home/maria_dev/
maria_dev@sandbox-hdp.hortonworks.com's password:
ncdc.jar                                100% 714MB 174.4MB/s   00:04
rene@rene-XPS-9320 ~
➤ scp -P 2222 /home/rene/Downloads/NCDC/file_names.txt maria_dev@sandbox-hdp.hortonworks.com:/home/
maria_dev@sandbox-hdp.hortonworks.com's password:
file_names.txt                          100% 147   155.6KB/s   00:00
rene@rene-XPS-9320 ~
➤

```

Abbildung 1.10: Dateitransfer mit scp

```

rene@rene-XPS-9320 ~
➤ ssh maria_dev@sandbox-hdp.hortonworks.com -p 2222
maria_dev@sandbox-hdp.hortonworks.com's password:
Last login: Wed Dec 14 06:46:18 2022 from 172.18.0.3
[maria_dev@sandbox-hdp ~]$ ll
total 731472
-rw-rw-r-- 1 maria_dev maria_dev    147 Dec 14 06:47 file_names.txt
-rw-rw-r-- 1 maria_dev maria_dev 749018830 Dec 14 06:46 ncdc.jar
[maria_dev@sandbox-hdp ~]$ hdfs dfs -mkdir -p /user/maria_dev/input
[maria_dev@sandbox-hdp ~]$ hdfs dfs -moveFromLocal file_names.txt /user/maria_dev/input/
[maria_dev@sandbox-hdp ~]$ hdfs dfs -moveFromLocal ncdc.jar /user/maria_dev/input/
[maria_dev@sandbox-hdp ~]$ ll
total 0
[maria_dev@sandbox-hdp ~]$ hdfs dfs -ls /user/maria_dev/input/
Found 2 items
-rw-r--r-- 1 maria_dev hdfs      147 2022-12-14 06:51 /user/maria_dev/input/file_names.txt
-rw-r--r-- 1 maria_dev hdfs 749018830 2022-12-14 06:54 /user/maria_dev/input/ncdc.jar
[maria_dev@sandbox-hdp ~]$

```

Abbildung 1.11: Dateiupload in das HDFS

Schritt 4 ist das Schreiben eines Bash Skripts, welches später als Mapper-Klasse fungiert. Zum Zusammenführen der Dateien wird nicht die Java API von MapReduce, sondern die Streaming API von Hadoop²⁶ verwendet. Diese erlaubt es, völlig sprachenunabhängig MapReduce Jobs zu schreiben. Man kann als Mapper und/oder Reducer dabei jeweils eigene Skripte oder Binaries verwenden. Jeder Mapper-Prozess führt eine Instanz des angegebenen Mapper-Skripts aus und füttert ihm, sofern nicht ein anderes InputFormat bestimmt wurde, Zeile für Zeile den Inhalt des ihm zugeteilten InputSplits als Key-Value-Paare auf `stdin`. `stdout` des Skripts wird wiederum vom Mapper zeilenweise als Key-Value-Paar gesammelt

²⁵<https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/FileSystemShell.html>

²⁶<https://hadoop.apache.org/docs/stable/hadoop-streaming/HadoopStreaming.html>

und an die Reducer weitergeleitet, wo der Prozess genauso abläuft. Das Skript ist in Listing 1.1 bis auf einige Kommentare abgebildet. Das gesamte Skript findet sich im Appendix 1.

```
1 read offset inputfile
2 echo "reporter:status:Verarbeite $inputfile" >&2
3
4 # basename gibt den Dateinamen ohne den Rest des Pfades und ohne die Dateiendung zurück
5 target_dir='basename $inputfile .tar.gz'
6
7 # Erstelle für das Jahr ein neues Verzeichnis und entpacke das Archiv dort hin
8 # Mit der Option -C am ENDE wird NACH dem Entpacken in das Verzeichnis gewechselt
9 mkdir -p $target_dir
10 echo "reporter:status:Entpacke $inputfile nach $target_dir" >&2
11 tar xzf $inputfile -C $target_dir
12
13 # Füge alle CSV-Dateien im Ordner in einer Datei mit Endung ".complete" zusammen.
14 echo "report:status:Füge alle Dateien des Jahres $target_dir zusammen" >&2
15 for file in $target_dir/*
16 do
17     cat $file >> $target_dir.complete
18     echo "report:status:Bearbeite $file" >&2
19 done
20
21 # Komprimiere die Datei wieder mit gzip und speichere das Ergebnis im HDFS.
22 # Durch das Argument "-" nach "-put" wird dabei STDIN als Quelle verwendet.
23 echo "report:status:Komprimiere Datei und schreibe ins HDFS" >&2
24 gzip -c $target_dir.complete | /usr/hdp/current/hadoop-hdfs-client/bin/hdfs dfs -put - /user/
    maria_dev/input/processed/$target_dir.gz
```

Listing 1.1: Bash Skript als Mapper-Klasse

Mit `read offset inputfile` liest das Skript das Key-Value-Paar von `stdin` ein. Durch das später bei der Ausführung angegebene InputFormat `NLineInputFormat` ist das erste Wort der Zeile der Offset der Zeile zum Dateianfang. Diese Information ist für diesen Job nicht relevant. Im Rest der Zeile steht der Inhalt der ursprünglichen Zeile, in diesem Fall ein Dateiname wie `ncdc.jar/20xx.tar.gz`. Das Archiv aus dem `ncdc.jar` wird in einen Unterordner in der Laufzeitumgebung des MapReduce Jobs entpackt. Anschließend werden alle darin befindlichen CSV-Dateien zu einer Datei vereint. Diese resultierende Datei wird zum Schluss wieder komprimiert und mit dem Befehl `hdfs dfs -put - <Pfad>` im HDFS abgespeichert.

Schritt 5: Ist das Skript geschrieben, könnte es wieder per `scp` und `hdfs dfs` in das HDFS übertragen werden. An dieser Stelle soll aber der Ambari File View gezeigt werden. Zu diesem navigiert man wie bereits in Abschnitt 1.2.4 beschrieben und wechselt über die grafische Oberfläche in den Zielordner `/user/maria_dev/input`. Über den Upload Button oben rechts kann man die Datei direkt von seinem lokalen Rechner ins HDFS laden. Am Ende sollte der Inhalt des Ordners aussehen wie in Abbildung 1.12. Das Skript muss noch für alle Nutzer als ausführbar markiert werden, damit der MapReduce Job es später benutzen kann. Dafür wählt man die Datei an, klickt oben links auf **Permissions** und setzt sie entsprechend Abbildung 1.13. Man könnte hierfür auch das HDFS CLI mit dem Befehl `hdfs dfs -chmod 755 <Pfad>` verwenden.

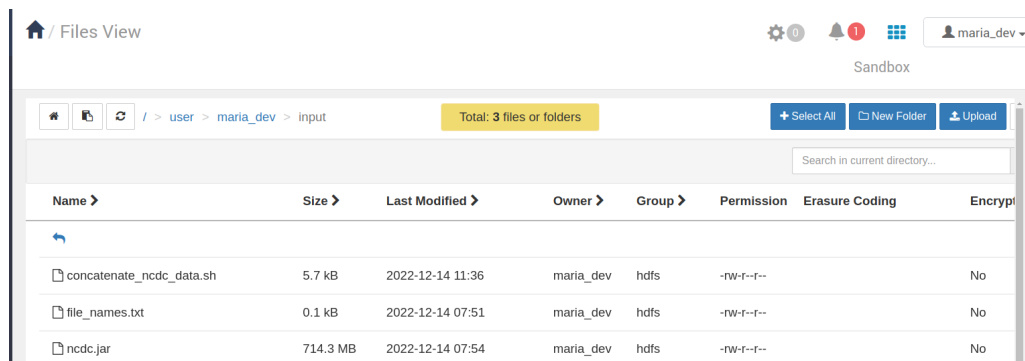


Abbildung 1.12: Upload der NCDC Datensätze

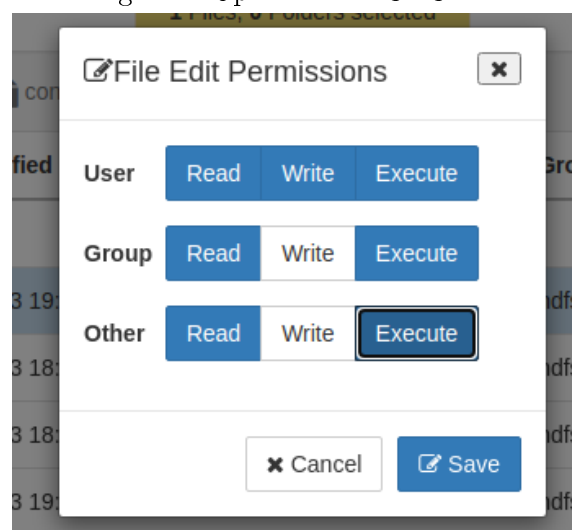


Abbildung 1.13: Setzen von Dateiberechtigungen im Ambari File View

MapReduce mit der Hadoop Streaming API

Um den vorbereiteten MapReduce Job auszuführen, muss der Befehl aus Listing 1.2 abgesetzt werden. Der vollständig dokumentierte Befehl ist in Appendix 2 zu finden.

- Zeile 1 weist Hadoop an, ein JAR auszuführen. Dateipfad des Hadoop Streaming JARs angegeben, welches mit dem Framework mitgeliefert wird.
- Zeile 2 sorgt dafür, dass Hadoop das im HDFS gespeicherte JAR `ncdc.jar` in die Laufzeitumgebung des MapReduce Jobs kopiert und automatisch entpackt. Prozesse können während des Jobs unter dem Pfad `ncdc.jar` auf die darin enthaltenen Dateien (hier die TARs mit den jährlichen Datensätzen) zugreifen.
- Zeile 3 weist Hadoop eigentlich an, die Mapper-Klasse vom lokalen Dateisystem auf alle am Job beteiligten Nodes zu kopieren, damit sie dort lokal zur Verfügung steht. Durch das Präfix `hdfs://host:port/` teilt man Hadoop mit, dass die Datei bereits

im HDFS liegt. Mit `\#concatenate_ncdc_data.sh` am Ende des Pfades gibt man der Datei einen Alias, damit man in der `-mapper` Option nicht wieder den vollen Pfad angeben muss.

- Mit `-D` kriegt ein Parameter Priorität über eine Einstellung, die bereits in Konfigurationsdateien gesetzt sind.
- Zeile 4 macht aus diesem Job einen reinen Map-Job ohne Reduce-Phase, da für die Umwandlung der Dateien keine Reduce-Phase nötig ist.
- Zeile 5 verhindert die sogenannte spekulative Ausführung. Ist diese Option aktiviert, startet Hadoop manchmal mehrere Jobs für einen InputSplit und filtert in der Shuffle-Phase doppelte Ergebnisse. Sind manche Nodes deutlich langsamer als andere, kann das Performancegewinne bringen. In diesem Fall würde das aber dazu führen, dass Dateien doppelt ins HDFS geschrieben würden, da nicht der Output der Mapper verwendet wird, sondern das Skript direkt ins HDFS schreibt.
- Zeile 6 gibt den Pfad zur Datei im HDFS an. Anders als bei der Angabe des mitzuliefernden JARs muss bei den Parametern für Input und Output kein vollständiger HDFS-Pfad angegeben werden. MapReduce erwartet standardmäßig, dass es sich dabei um Verzeichnisse und Dateien im HDFS handelt.
- Zeile 7 gibt die Java Klasse des InputFormats an. Diese gehört zum Hadoop Framework und ist im Java Class Path der Ausführungsumgebung verfügbar. Mit `NLineInputFormat` werden einem Mapper *N* Zeilen aus dem InputSplit als Eingabe gegeben. *N* ist standardmäßig 1.
- Zeile 8 gibt den Pfad an, unter dem die Reducer ihre Ausgabedateien ablegen werden. Es muss sich hierbei um einen Ordner handeln, der noch nicht existiert. Im HDFS können Dateien nicht einfach überschrieben werden. Daher dürfen Ausgabeordner grundsätzlich nicht vorher existieren.
- Zeile 9 gibt die Mapper-Klasse an. Würde die Java API statt der Streaming API verwendet, stünde hier eine Java Klasse. Es wird der Alias aus Zeile 3 benutzt.

```
1 hadoop jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming-*.jar \  
2 -archives hdfs://sandbox-hdp.hortonworks.com:8020/user/aria_dev/input/ncdc.jar \  
3 -files hdfs://sandbox-hdp.hortonworks.com:8020/user/aria_dev/input/concatenate_ncdc_data.sh#  
4   concatenate_ncdc_data.sh \  
5 -D mapred.reduce.tasks=0 \  
6 -D mapred.map.tasks.speculative.execution=false \  
7 -input /user/aria_dev/input/file_names.txt \  
8 -inputformat org.apache.hadoop.mapred.lib.NLineInputFormat \  
9 -mapper concatenate_ncdc_data.sh
```

Listing 1.2: Startskript für den NCDC Concatenation MapReduce Job

Sind alle Dateien an den richtigen Orten platziert, kann man sich erneut als `aria_dev` per `ssh` auf den NameNode verbinden und den Befehl aus Listing 1.2 absetzen. Die Ausführung kann mehrere Minuten dauern. Am Ende sollte die Konsole eine ähnliche Ausgabe wie

in Abbildung 1.14 zeigen. Es wurden der Übersicht halber viele Zeilen in der Abbildung entfernt. Man kann sehen, dass der Job ein Input File (die `file_names.txt`) gelesen und daraus sieben InputSplits erstellt hat, korrespondierend zu den sieben Zeilen (Dateipfaden) in der Datei. Für jeden InputSplit wurde ein Mapper gestartet und alle Mapper sind haben ihre Arbeit erfolgreich beendet. Mit der `applicationId application_xxxxxxxxxx_XXXX` kann man die YARN Logs zum Job Run anzeigen lassen. Darin findet man unter anderem die Statusreports aus dem Skript und eventuelle Fehlermeldungen. Dieser MapReduce Job schließt nämlich auch erfolgreich ab, wenn beim Entpacken oder Zusammenfügen der Archive ein Fehler auftritt, da diese Vorgänge als Unterprozesse des eigentlichen Mappers ohne Fehlerüberprüfung ausgeführt wurden. Sind zum Beispiel die Rechte auf dem Ordner, in den die fertigen Dateien mit `hdfs dfs -put -` geschrieben werden sollen, falsch gesetzt, wird man dies nur anhand der fehlenden Dateien und der YARN Logs bemerken. Existiert jedoch der Ordner bereits, der als Ausgabeverzeichnis des Jobs angegeben wurde, oder kann eine die Eingabedatei nicht gefunden werden, beendet sich der MapReduce Job mit einer entsprechenden Fehlermeldung. Weiter unten in der Konsolenausgabe sieht man, dass die Mapper insgesamt über elf Minuten Laufzeit benötigt haben. Den Zeitstempeln kann man aber entnehmen, dass der Job nur etwas über vier Minuten lief. Daran sieht man den Performancegewinn bei paralleler Verarbeitung der Daten. Im Single Node Setup ist zwar nur ein Node an der Verarbeitung beteiligt, aber Hadoop startet mehrere JVM-Prozesse, welche dann auf mehreren Prozessorkernen gleichzeitig laufen können.

Die Ausgabedateien des Jobs können ebenfalls betrachtet werden. Ein Blick in den File View zeigt, dass Hadoop den Ordner `/user/maria_dev/output`, wie im `-output`-Parameter gefordert, angelegt hat. In diesem Ordner befinden sich sieben Dateien mit den Namen `part-0000x` (siehe Abb. 1.15). Diese Dateien sind die Ausgaben der Mapper. Da der Job keine Reducer verwendet hatte, wurde eine Datei pro Mapper erstellt. Da die Mapper aber keine Ausgaben erzeugt haben, was bei der Streaming API durch Schreiben auf `stdout` passiert wäre, sind diese Dateien vollkommen leer. Eine achte Datei im Ordner gibt mit ihrem Namen den Exit-Status des Jobs an, hier `_SUCCESS`.

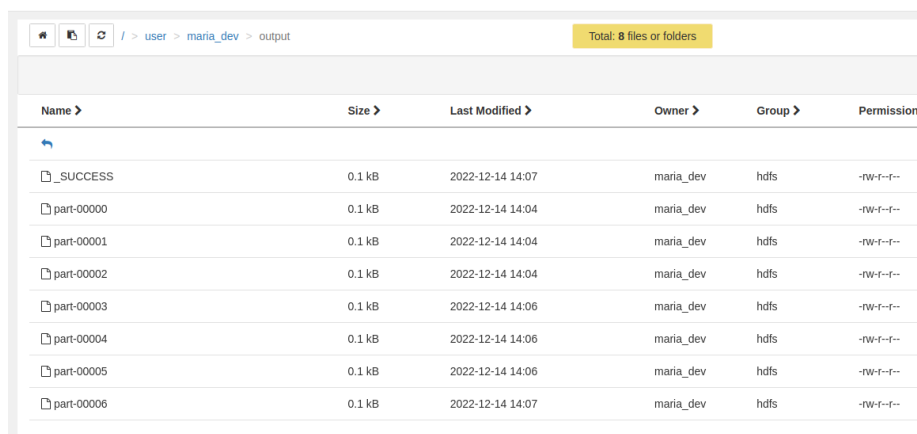
Im Ordner `/user/maria_dev/input/processed` liegen die zusammengefassten und erneut komprimierten Dateien ab, die während der Map-Phase des Jobs erstellt wurden (siehe Abb. 1.16). Diese kann man im Webbrowser über den *Open* Button direkt betrachten. Die Anzeige funktioniert nicht fehlerfrei; es sind abgehackte und duplizierte Zeilen zu sehen (siehe Abb. 1.17). Dies sind allerdings nur Fehler des File Views. Lädt man die Dateien wieder herunter und betrachtet sie lokal, wird man feststellen, dass das Zusammenfügen funktioniert hat. Lediglich die Headerzeile wiederholt sich mehrmals in jeder Datei, was im weiteren Verlauf der Bearbeitung beachtet werden muss.

Die MapReduce Java API

1 Hadoop Grundlagen

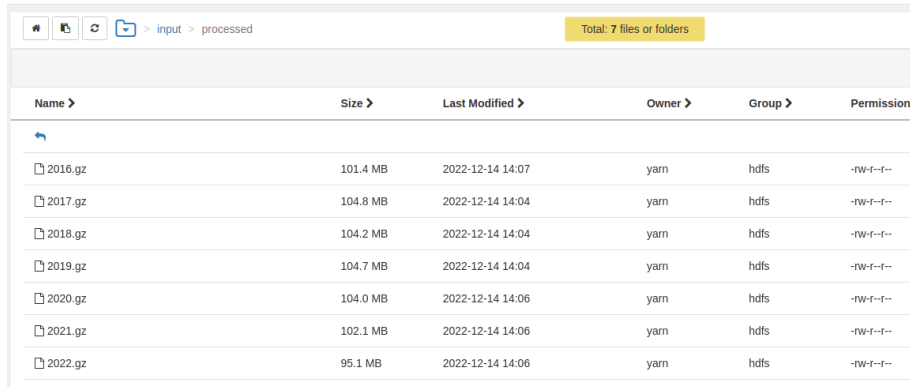
```
rene@rene-XPS-9320 ~
└─ ssh maria_dev@sandbox-hdp.hortonworks.com -p 2222
maria_dev@sandbox-hdp.hortonworks.com's password:
Last login: Wed Dec 14 10:54:00 2022 from 172.18.0.3
[maria_dev@sandbox-hdp ~]$ hadoop jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming-*.jar \
> -archives hdfs://sandbox-hdp.hortonworks.com:8020/user/maria_dev/input/ncdc.jar \
> -files hdfs://sandbox-hdp.hortonworks.com:8020/user/maria_dev/input/concatenate_ncdc_data.sh#concatenate_ncdc_data.sh \
> -D mapred.reduce.tasks=0 \
> -D mapred.map.tasks.speculative.execution=false \
> -input /user/maria_dev/input/file_names.txt \
> -inputformat org.apache.hadoop.mapred.lib.NLLineInputFormat \
> -output /user/maria_dev/output \
> -mapper concatenate_ncdc_data.sh
22/12/14 11:01:04 INFO mapred.FileInputFormat: Total input files to process : 1
22/12/14 11:01:05 INFO mapreduce.JobSubmitter: number of splits:7
22/12/14 11:01:06 INFO impl.YarnClientImpl: Submitted application application_1670997201911_0003
22/12/14 11:01:06 INFO mapreduce.Job: The url to track the job: http://sandbox-hdp.hortonworks.com:8088/proxy/application_1670997201911_0003/
22/12/14 11:01:06 INFO mapreduce.Job: Running job: job_1670997201911_0003
22/12/14 11:01:10 INFO mapreduce.Job: Job job_1670997201911_0003 running in uber mode : false
22/12/14 11:01:10 INFO mapreduce.Job: map 0% reduce 0%
22/12/14 11:01:20 INFO mapreduce.Job: map 43% reduce 0%
22/12/14 11:03:21 INFO mapreduce.Job: map 71% reduce 0%
22/12/14 11:03:36 INFO mapreduce.Job: map 86% reduce 0%
22/12/14 11:04:44 INFO mapreduce.Job: map 100% reduce 0%
22/12/14 11:05:28 INFO mapreduce.Job: Job job_1670997201911_0003 completed successfully
22/12/14 11:05:29 INFO mapreduce.Job: Counters: 32
  File System Counters
    FILE: Number of bytes read=0
    FILE: Number of bytes written=1667113
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
    HDFS: Number of bytes read=1483
    HDFS: Number of bytes written=0
    HDFS: Number of read operations=49
    HDFS: Number of large read operations=0
    HDFS: Number of write operations=14
  Job Counters
    Launched map tasks=7
    Total time spent by all reduces in occupied slots (ms)=0
    Total time spent by all map tasks (ms)=662095
  Map-Reduce Framework
    Map input records=7
    Map output records=0
    Input split bytes=889
    Spilled Records=0
    Failed Shuffles=0
    Merged Map outputs=0
  File Input Format Counters
    Bytes Read=594
  File Output Format Counters
    Bytes Written=0
22/12/14 11:05:29 INFO Streaming.StreamJob: Output directory: /user/maria_dev/output
[maria_dev@sandbox-hdp ~]$
```

Abbildung 1.14: MapReduce Konsolenausgabe



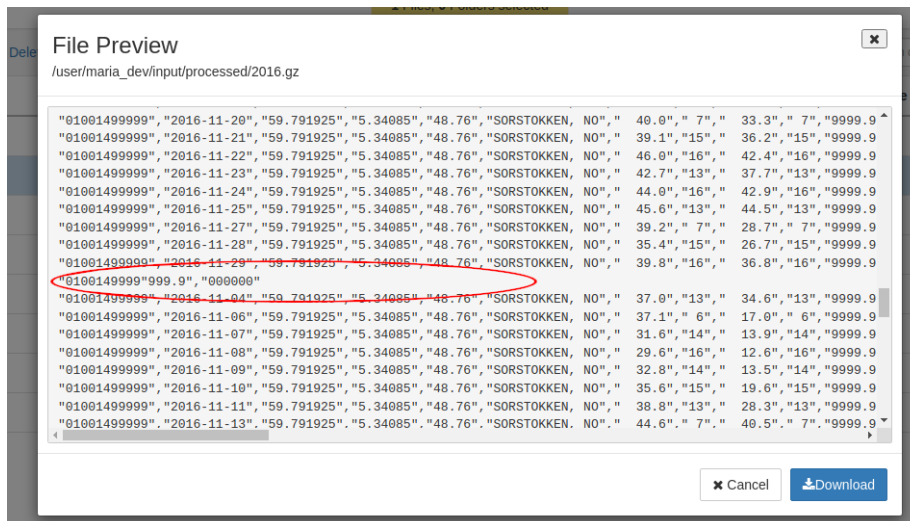
Name	Size	Last Modified	Owner	Group	Permission
._SUCCESS	0.1 kB	2022-12-14 14:07	maria_dev	hdfs	-rw-r--r--
part-00000	0.1 kB	2022-12-14 14:04	maria_dev	hdfs	-rw-r--r--
part-00001	0.1 kB	2022-12-14 14:04	maria_dev	hdfs	-rw-r--r--
part-00002	0.1 kB	2022-12-14 14:04	maria_dev	hdfs	-rw-r--r--
part-00003	0.1 kB	2022-12-14 14:06	maria_dev	hdfs	-rw-r--r--
part-00004	0.1 kB	2022-12-14 14:06	maria_dev	hdfs	-rw-r--r--
part-00005	0.1 kB	2022-12-14 14:06	maria_dev	hdfs	-rw-r--r--
part-00006	0.1 kB	2022-12-14 14:07	maria_dev	hdfs	-rw-r--r--

Abbildung 1.15: MapReduce Ausgabedateien



Name >	Size >	Last Modified >	Owner >	Group >	Permission
2016.gz	101.4 MB	2022-12-14 14:07	yarn	hdfs	-rw-r--r--
2017.gz	104.8 MB	2022-12-14 14:04	yarn	hdfs	-rw-r--r--
2018.gz	104.2 MB	2022-12-14 14:04	yarn	hdfs	-rw-r--r--
2019.gz	104.7 MB	2022-12-14 14:04	yarn	hdfs	-rw-r--r--
2020.gz	104.0 MB	2022-12-14 14:06	yarn	hdfs	-rw-r--r--
2021.gz	102.1 MB	2022-12-14 14:06	yarn	hdfs	-rw-r--r--
2022.gz	95.1 MB	2022-12-14 14:06	yarn	hdfs	-rw-r--r--

Abbildung 1.16: Zusammengeführte, komprimierte Wetterdaten im HDFS



File Preview
/user/maria_dev/input/processed/2016.gz

"01001499999", "2016-11-20", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "40.6", "7", "33.3", "7", "9999.9"
"01001499999", "2016-11-21", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "39.1", "15", "36.2", "15", "9999.9"
"01001499999", "2016-11-22", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "46.0", "16", "42.4", "16", "9999.9"
"01001499999", "2016-11-23", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "42.7", "13", "37.7", "13", "9999.9"
"01001499999", "2016-11-24", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "44.0", "16", "42.9", "16", "9999.9"
"01001499999", "2016-11-25", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "45.6", "13", "44.5", "13", "9999.9"
"01001499999", "2016-11-27", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "39.2", "7", "28.7", "7", "9999.9"
"01001499999", "2016-11-28", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "35.4", "15", "26.7", "15", "9999.9"
"01001499999", "2016-11-29", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "39.8", "16", "36.8", "16", "9999.9"
"01001499999", "2016-11-29", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "39.8", "16", "36.8", "16", "9999.9"
"01001499999", "2016-11-04", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "37.0", "13", "34.6", "13", "9999.9"
"01001499999", "2016-11-06", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "37.1", "6", "17.0", "6", "9999.9"
"01001499999", "2016-11-07", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "31.6", "14", "13.9", "14", "9999.9"
"01001499999", "2016-11-08", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "29.6", "16", "12.6", "16", "9999.9"
"01001499999", "2016-11-09", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "32.8", "14", "13.5", "14", "9999.9"
"01001499999", "2016-11-10", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "35.6", "15", "19.6", "15", "9999.9"
"01001499999", "2016-11-11", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "38.8", "13", "28.3", "13", "9999.9"
"01001499999", "2016-11-13", "59.791925", "5.34085", "48.76", "SORSTOKKEN, NO", "44.6", "7", "40.5", "7", "9999.9"

Abbildung 1.17: Fehlerhafte Anzeige der Daten im File View

2 ETL mit Pig

Auch wenn es vermehrt von Spark verdrängt wird

2.1 Anwendungsfälle

Welche neuen Dinge ermöglicht dieses Tool Eine Zeile Pig Latin entspricht vielen Zeilen MapReduce

2.2 Architektur

2.3 Pig Latin

2.4 Praxis

2.4.1 Hinzufügen zum Cluster

2.4.2 Anwendung auf dem Cluster

3 Data Ingestion

3.1 Sqoop

3.2 Flume

4 Datawarehousing mit Hive

4.1 Anwendungsfälle

Welche neuen Dinge ermöglicht dieses Tool

4.1.1 Unterschiede zu Pig

4.2 Architektur

4.3 Interaktion

4.3.1 HiveQL

4.3.2 CLI

4.3.3 Java API

4.4 Praxis

4.4.1 Hinzufügen zum Cluster

4.4.2 Einrichtung einer Datenbank

4.4.3 Einlesen von Daten im CLI

4.4.4 Einlesen von Daten mit Sqoop

4.4.5 Absetzen einer Query

5 NoSQL mit HBase

5.1 Anwendungsfälle

Welche neuen Dinge ermöglicht dieses Tool

5.1.1 CAP-Theorem

5.1.2 ACID und BASE

5.2 Architektur

5.3 Interaktion

5.3.1 HBase Shell

5.3.2 Java API

5.4 Praxis

5.4.1 Hinzufügen zum Cluster

5.4.2 Einrichtung einer Datenbank

5.4.3 Einlesen von Daten

5.4.4 Datenmigration aus einem RDBMS

5.4.5 Absetzen einer Query

6 Streaming mit Kafka

6.1 Anwendungsfälle

Welche neuen Dinge ermöglicht dieses Tool Ersetzt durch Spark Streaming

6.2 Architektur

6.3 Interaktion

6.3.1 Who knows

6.4 Praxis

6.4.1 Hinzufügen zum Cluster

6.4.2 Maybe, vielleicht kann man ja was zeigen

7 Hadoop heute

7.1 Aktuelle Anwendungsbeispiele zu Hadoop

7.1.1 AirBnB

7.2 Apache Spark als Gold Standard

7.2.1 Kann eh alles besser

Appendix

1 Mapper-Skript für Hadoop Streaming

```
1 #!/usr/bin/env bash
2
3 # Dies ist ein Helper-Skript zum Zusammenfügen aller CSV-Dateien in den
4 # komprimierten Archiven (.tar.gz) der einzelnen Jahre im NCDC Wetter-Datensatz.
5 # Es basiert auf der Anleitung zum Präparieren des NCDC Datensatzes
6 # aus White, T. E. (2015). Hadoop: The Definitive Guide (4th edition). O'Reilly Media.
7 # Siehe auch: https://github.com/tomwhite/hadoop-book/tree/master/appc/src/main/sh
8 # bzw. https://github.com/Resteklicken/Hauptseminar-Hadoop für Code und Write-Up
9
10 # Als Eingabedatei wird eine Textdatei mit den Dateinamen der Archive erwartet:
11
12 # cat file_names.txt
13 # ncdc.jar/2016.tar.gz
14 # ncdc.jar/2017.tar.gz
15 # ...
16 # ncdc.jar/2022.tar.gz
17
18 # Für jede Zeile in der Eingabedatei wird von Hadoop ein Map-Prozess gestartet.
19 # ncdc.jar ist ein JAR, welches im HDFS abliegt und die Ordner der einzelnen Jahre bündelt.
20 # Dies ist nötig, da man Hadoop nur eine Liste mit Dateien oder JARs auf einen Job Run
    mitgeben kann.
21 # Das Skript sollte wenigstens etwas flexibel bleiben, daher wurde davon abgesehen, alle
    Dateinamen
22 # als Liste bei der Ausführung des Befehls zu übergeben.
23 # NLineInputFormat gibt jedem Mapper eine einzige Zeile aus der Eingabedatei als Key-Value-
    Paar als Input.
24 # Der Key ist der Offset der Zeile zum Dateianfang, an dieser Stelle nicht weiter interessant
    .
25 # Der Value ist der Inhalt der Zeile, der hier in die Variable inputfile gelesen wird.
26 # Nachrichten auf STDERR mit dem Präfix "reporter:status:" werden von Hadoop als MapReduce
    Statusupdates interpretiert.
27 # Dadurch denkt Hadoop nicht, dass der Job sich aufgehängt hätte.
28 read offset inputfile
29 echo "reporter:status:Verarbeite $inputfile" >&2
30
31 # basename gibt den Dateinamen ohne den Rest des Pfades und ohne die Dateiendung zurück
32 target_dir='basename $inputfile .tar.gz'
33
34 # Erstelle für das Jahr ein neues Verzeichnis und entpacke das Archiv dort hin
35 # Mit der Option -C am ENDE wird NACH dem Entpacken in das Verzeichnis gewechselt
36 mkdir -p $target_dir
37 echo "reporter:status:Entpacke $inputfile nach $target_dir" >&2
```

```
38 tar xzf $inputfile -C $target_dir
39
40 # Füge alle CSV-Dateien im Ordner in einer Datei mit Endung ".complete" zusammen.
41 echo "report:status:Füge alle Dateien des Jahres $target_dir zusammen" >&2
42 for file in $target_dir/*
43 do
44     cat $file >> $target_dir.complete
45     echo "report:status:Bearbeite $file" >&2
46 done
47
48 # Komprimiere die Datei wieder mit gzip und speichere das Ergebnis im HDFS.
49 # Durch das Argument "-" nach "-put" wird dabei STDIN als Quelle verwendet.
50 echo "report:status:Komprimiere Datei und schreibe ins HDFS" >&2
51 gzip -c $target_dir.complete | /usr/hdp/current/hadoop-hdfs-client/bin/hdfs dfs -put - /user/
    maria_dev/input/processed/$target_dir.gz
52 echo "report:status:Fertig" >&2
53
54 # Die Ausführung des Skripts erfolgt mittels des folgenden Befehls, abgesetzt zum Beispiel
55 # nach Verbinden auf die HDP Sandbox per ssh als maria_dev:
56
57 # hadoop jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming-*.jar \
58 # -archives hdfs://sandbox-hdp.hortonworks.com:8020/user/maria_dev/input/ncdc.jar \
59 # -files hdfs://sandbox-hdp.hortonworks.com:8020/user/maria_dev/input/concatenate_ncdc_data.
    sh#concatenate_ncdc_data.sh \
60 # -D mapred.reduce.tasks=0 \
61 # -D mapred.map.tasks.speculative.execution=false \
62 # -input /user/maria_dev/input/file_names.txt \
63 # -inputformat org.apache.hadoop.mapred.lib.NLineInputFormat \
64 # -output /user/maria_dev/output \
65 # -mapper concatenate_ncdc_data.sh
66
67 # hadoop jar weist Hadoop an, ein JAR auszuführen. Dafür wird das Hadoop Streaming Jar ausgew
    ählt ,
68 # welches mit dem Framework mitgeliefert wird und Eingaben von STDIN entgegen nimmt.
69 # -archives hdfs://sandbox-hdp.hortonworks.com:8020/user/maria_dev/input/ncdc.jar sorgt dafür
    , dass Hadoop das im HDFS
70 # gespeicherte JAR ncdc.jar in die Laufzeitumgebung des MapReduce Jobs kopiert und
    automatisch entpackt. Prozesse können
71 # während des Jobs unter ncdc.jar auf die darin enthaltenen Dateien (hier die TARs)
    zugreifen.
72 # -files hdfs://sandbox-hdp.hortonworks.com:8020/user/maria_dev/input/concatenate_ncdc_data.
    sh#concatenate_ncdc_data.sh
73 # weist Hadoop eigentlich an, die Mapper-Klasse auf alle am Job beteiligten Nodes zu
    kopieren, damit sie dort lokal
74 # zur Verfügung steht. Durch das Präfix hdfs://host:port/ teilt man Hadoop mit, dass die
    Datei bereits im HDFS liegt.
75 # Mit #concatenate_ncdc_data.sh am Ende des Pfades gibt man der Datei einen Alias, damit
    man in der -mapper Option
76 # nicht wieder den vollen Pfad angeben muss.
77 # -D überschreibt priorisiert Werte, die bereits in Konfigurationsdateien gesetzt sind.
78 # mapred.reduce.tasks=0 macht aus diesem Job einen reinen Map-Job ohne Reduce-Phase,
79 # da für die Umwandlung der Dateien keine Reduce-Phase nötig ist.
80 # mapred.map.tasks.speculative.execution=false verhindert die sogenannte spekulative Ausfü
    hrung.
```



```

81 # Ist diese Option aktiviert, startet Hadoop manchmal mehrere Jobs für einen InputSplit und
    # filtert
82 # in der Shuffle-Phase doppelte Ergebnisse. Sind manche Nodes deutlich langsamer als andere
    # , kann das
83 # normalerweise Performancegewinne bringen. In diesem Fall würde das aber dazu führen, dass
    # Dateien doppelt
84 # ins HDFS geschrieben würden.
85 # -input /user/aria_dev/input/file_names.txt gibt den Pfad zur Datei im HDFS an
86 # -inputformat org.apache.hadoop.mapred.lib.NLineInputFormat gibt die Java Klasse des
    # InputFormats an.
87 # -output /user/aria_dev/output gibt den Pfad für die Ausgabe an. Es muss sich hierbei um
    # einen Ordner
88 # handeln. HDFS arbeitet nach dem "write once, read many times" Prinzip, daher dürfen
    # Ausgabeordner
89 # grundsätzlich nicht vorher existieren.
90 # -mapper /user/aria_dev/input/concatenate_ncdc_data.sh gibt die Mapper-Klasse an. Würde die
    # Java API statt der
91 # Streaming API verwendet, stünde hier eine Java Klasse

```

Listing 1: Bash Skript als Mapper-Klasse

2 Startskript für den NCDC Concatenation Job

```

1 #!/usr/bin/env bash
2
3 # Hiermit wird der MapReduce-Job zum Zusammenführen des NCDC Datensatzes ausgeführt.
4 # Das Skript soll nach Verbinden auf die HDP Sandbox per ssh als aria_dev benutzt werden.
5 # Siehe https://github.com/Resteklicken/Hauptseminar-Hadoop für mehr Informationen.
6
7 hadoop jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming-*.jar \
8 -archives hdfs://sandbox-hdp.hortonworks.com:8020/user/aria_dev/input/ncdc.jar \
9 -files hdfs://sandbox-hdp.hortonworks.com:8020/user/aria_dev/input/concatenate_ncdc_data.sh#
    concatenate_ncdc_data.sh \
10 -D mapred.reduce.tasks=0 \
11 -D mapred.map.tasks.speculative.execution=false \
12 -input /user/aria_dev/input/file_names.txt \
13 -inputformat org.apache.hadoop.mapred.lib.NLineInputFormat \
14 -output /user/aria_dev/output \
15 -mapper concatenate_ncdc_data.sh
16
17 # hadoop jar weist Hadoop an, ein JAR auszuführen. Dafür wird das Hadoop Streaming Jar ausgew
    # ählt, welches mit dem Framework mitgeliefert wird und Eingaben von STDIN entgegen nimmt.
18 # -archives hdfs://sandbox-hdp.hortonworks.com:8020/user/aria_dev/input/ncdc.jar sorgt dafür
    # , dass Hadoop das im HDFS gespeicherte JAR ncdc.jar in die Laufzeitumgebung des
    # MapReduce Jobs kopiert und automatisch entpackt. Prozesse können während des Jobs unter
    # ncdc.jar auf die darin enthaltenen Dateien (hier die TARs) zugreifen.
19 # -files hdfs://sandbox-hdp.hortonworks.com:8020/user/aria_dev/input/concatenate_ncdc_data.
    # sh#concatenate_ncdc_data.sh weist Hadoop eigentlich an, die Mapper-Klasse auf alle am
    # Job beteiligten Nodes zu kopieren, damit sie dort lokal zur Verfügung steht. Durch das
    # Präfix hdfs://host:port/ teilt man Hadoop mit, dass die Datei bereits im HDFS liegt. Mit
    # #concatenate_ncdc_data.sh am Ende des Pfades gibt man der Datei einen Alias, damit man
    # in der -mapper Option nicht wieder den vollen Pfad angeben muss.

```

```
20 # -D überschreibt priorisiert Werte, die bereits in Konfigurationsdateien gesetzt sind.
21 # mapred.reduce.tasks=0 macht aus diesem Job einen reinen Map-Job ohne Reduce-Phase, da für
    die Umwandlung der Dateien keine Reduce-Phase nötig ist.
22 # mapred.map.tasks.speculative.execution=false verhindert die sogenannte spekulative Ausfü-
    hrung. Ist diese Option aktiviert, startet Hadoop manchmal mehrere Jobs für einen
    InputSplit und filtert in der Shuffle-Phase doppelte Ergebnisse. Sind manche Nodes
    deutlich langsamer als andere, kann das normalerweise Performancegewinne bringen. In
    diesem Fall würde das aber dazu führen, dass Dateien doppelt ins HDFS geschrieben würden
    .
23 # -input /user/maria_dev/input/file_names.txt gibt den Pfad zur Datei im HDFS an
24 # -inputformat org.apache.hadoop.mapred.lib.NLineInputFormat gibt die Java Klasse des
    InputFormats an.
25 # -output /user/maria_dev/output gibt den Pfad für die Ausgabe an. Es muss sich hierbei um
    einen Ordner handeln. HDFS arbeitet nach dem "write once, read many times" Prinzip,
    daher dürfen Ausgabeordner grundsätzlich nicht vorher existieren.
26 # -mapper /user/maria_dev/input/concatenate_ncdc_data.sh gibt die Mapper-Klasse an. Würde die
    Java API statt der Streaming API verwendet, stünde hier eine Java Klasse
```

Listing 2: Startskript für den NCDC Concatenation MapReduce Job

Literatur

1. *Apache Hadoop* [Apache Hadoop Main Page] [online]. [besucht am 2022-12-07]. Abger. unter: <https://hadoop.apache.org/>.
2. *Big Data: was es ist und was man darüber wissen sollte* [online]. [besucht am 2022-12-05]. Abger. unter: https://www.sas.com/de_de/insights/big-data/what-is-big-data.html.
3. GRIFFITHS, Richard T. *Search Engines* [History of the Internet] [online]. 2007-06-21. [besucht am 2022-12-05]. Abger. unter: <https://web.archive.org/web/20070621143859/http://www.internethistory.leidenuniv.nl/index.php3?m=6&c=7#how>.
4. ZAKON, Robert H'obbes'. *Hobbes' Internet Timeline - the definitive ARPAnet & Internet history* [Hobbes' Internet Timeline] [online]. 2018-01-01. [besucht am 2022-12-05]. Abger. unter: <https://www.zakon.org/robert/internet/timeline/#Growth>.
5. BEAUMONT, David. *How to explain vertical and horizontal scaling in the cloud* [Cloud computing news] [online]. 2014-04-09. [besucht am 2022-12-07]. Abger. unter: <https://www.ibm.com/blogs/cloud-computing/2014/04/09/explain-vertical-horizontal-scaling-cloud/>.
6. GUSTAFSON, John L. Amdahl's Law. In: PADUA, David (Hrsg.). *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011, S. 53–60. ISBN 978-0-387-09766-4. Abger. unter DOI: [10.1007/978-0-387-09766-4_77](https://doi.org/10.1007/978-0-387-09766-4_77).
7. *Horizontal Vs. Vertical Scaling Comparison Guide* [MongoDB] [online]. [besucht am 2022-12-07]. Abger. unter: <https://www.mongodb.com/basics/horizontal-vs-vertical-scaling>.
8. *Pricing - Linux Virtual Machines Scale Sets / Microsoft Azure* [online]. [besucht am 2022-12-07]. Abger. unter: <https://azure.microsoft.com/en-us/pricing/details/virtual-machine-scale-sets/linux/>.
9. ATHOW, Desire. *At 100TB, the world's biggest SSD gets an (eye-watering) price tag* [TechRadar] [online]. 2020-07-07. [besucht am 2022-12-07]. Abger. unter: <https://www.techradar.com/news/at-100tb-the-worlds-biggest-ssd-gets-an-eye-watering-price-tag>.
10. *who was who in SSD? - StorageTek* [online]. [besucht am 2022-12-07]. Abger. unter: <http://www.storagesearch.com/storagetek.html>.
11. *What is a Computer Cluster? / Answer from SUSE Defines* [SUSE Defines] [online]. [besucht am 2022-12-07]. Abger. unter: <https://www.suse.com/suse-defines/definition/computer-cluster/>.

12. GHEMAWAT, Sanjay; GOBIOFF, Howard; LEUNG, Shun-Tak. The Google File System. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Bolton Landing, NY, 2003, S. 20–43.
13. DEAN, Jeffrey; GHEMAWAT, Sanjay. MapReduce: Simplified Data Processing on Large Clusters. In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, S. 137–150.
14. CUTTING, Doug; CAFARELLA, Mike; LORICA, Ben. *The next 10 years of Apache Hadoop* [online]. 2016. [besucht am 2022-12-08]. Abger. unter: <https://www.oreilly.com/content/the-next-10-years-of-apache-hadoop/>.
15. WHITE, Tom. *Hadoop: the definitive guide*. Fourth edition. Beijing: O'Reilly, 2015. ISBN 978-1-4919-0163-2. OCLC: ocn904818464.
16. FREIKNECHT, Jonas; PAPP, Stefan. *Big Data in der Praxis: Lösungen mit Hadoop, Spark, HBase und Hive ; Daten speichern, aufbereiten, visualisieren*. 2., erweiterte Auflage. München: Hanser, 2018. ISBN 978-3-446-45396-8.
17. INFRABOT, ASF. *HowManyMapsAndReduces* [HADOOP2 - Apache Software Foundation] [online]. 2019-07-09. [besucht am 2022-12-10]. Abger. unter: <https://cwiki.apache.org/confluence/display/HADOOP2/HowManyMapsAndReduces>.
18. KEIM, Daniel A. Datenvisualisierung und Data Mining. *Datenbank Spektrum* [online]. [o. D.], Jg. 1, Nr. 2, S. 22 [besucht am 2022-12-11]. Abger. unter: <https://fusion.cs.uni-magdeburg.de/pubs/spektrum.pdf>.
19. *Apache Hadoop 3.3.4 – HDFS Architecture* [online]. [besucht am 2022-12-09]. Abger. unter: <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html>.