Hadoop

Hadoop ist ein in Java geschriebenes und quelloffenes Framework für das Verarbeiten und Analysieren großer Datenmengen auf verteilten Systemen der Apache Software Foundation. Ursprünglich wurde es 2005 von Doug Cutting und Mike Cafarella bei Yahoo! entwickelt.

Für einen effizienten Umgang mit enormen Datenmengen werden diese nicht als Ganzes verwertet, sondern in kleinere Stücke zerteilt, parallel verarbeitet und anschließend wieder zusammengeführt. Im Kern besteht Hadoop im Grunde aus zwei Komponenten: Dem „**Hadoop distributed file system**“, ein Dateisystem für verteilte Anwendungen, das seinen Ursprung in Googles „Google File System“ (Ghemawat et al. 2003) hat und „**MapReduce**“, das auf Googles MapReduce-Modell (Dean/Ghemawat 2004) basiert und für die Verteilung der Aufgaben auf die Knoten und das anschließende Zusammenführen zuständig ist. Dazu kommt **Hadoop Common**, das gewisse Grundfunktionalitäten und Schnittstellen zur Verfügung stellt.

In gewisser Weise übernimmt Hadoop für verteilte Systeme die Rolle des Betriebssystems auf Cluster-Ebene (die einzelnen Maschinen haben immer noch ihr eigenes) (Barroso et al. 2013: S. 33). Es fasst alle Rechner zusammen und stellt mit HDFS ein Dateisystem zur Verfügung und verwaltet zudem Ressourcen, teilt sie den Prozessen zu und überwacht diese.

(Grafik-Quelle: GeekFluent 2013)

Hadoop Common

Grundlage eines Hadoop Clusters ist **Hadoop Common**, das eine Sammlung der notwendigen Dienstprogramme und Bibliotheken bereitstellt und als Schnittstelle zu den anderen Bestandteilen dient. Es enthält auch die nötigen JAR-Files, die es braucht, um Hadoop zu starten.

Hadoop Distributed File System

Das **Hadoop Distributed File System** (**HDFS**) ist das Standard-Dateisystem Hadoops zum Speichern großer Datenmengen (bis in den Petabyte-Bereich) auf verteilten Systemen. Es basiert auf dem Google File System und zeichnet sich durch seine ähnlich hohe Skalierbarkeit aus. Damit einher geht das Verlangen nach hoher Fehlertoleranz, da es für den Einsatz auf mehreren Tausend Maschinen ausgelegt ist, was einen Hardwaredefekt oder kompletten Ausfall sehr wahrscheinlich macht, zumal es auch für den Betrieb auf kostengünstiger Hardware gedacht ist. Daher wurde beim Design davon ausgegangen, dass ein Hardwareversagen keine Ausnahme, sondern die Regel ist.

Hadoop wurde für den Einsatz auf kostengünstiger Commodity-Hardware entwickelt und somit kann sowohl auf hochwertigen Servern, als auch auf einfachen Desktoprechnern verwendet werden. Einzig der Masterknoten muss über mehr Leistung und Arbeitsspeicher verfügen, da auf ihm Framework-Komponenten laufen, die ihre Daten im RAM halten (Fischer 2010).

Aufbau

Der Aufbau eines Hadoop-Clusters ist nach dem Master-Slave-Konzept gestaltet und besteht aus einem **NameNode** (Master) und einem Cluster von **DataNodes** (Slave). Der NameNode verwaltet die Namespace-Operationen und Client-Zugriffe und ist für die Organisation der Replikationen und der Datenverteilung auf die DataNodes verantwortlich und bearbeitet eingehende Datenanfragen. Die DataNodes übernehmen die Aufgaben der Verwaltung und Verarbeitung (Lesen, Schreiben) der Daten im Cluster. Sie sind nur für die auf ihnen gespeicherten Daten zuständig. Auf Anweisung des NameNodes können sie auch Blöcke erstellen, löschen und replizieren.

Ähnlich wie ein „normales“ Dateisystem auch, erlaubt HDFS das Anlegen, Löschen, Verschieben oder Umbenennen von Dateien in Ordern und Unterordnern organisiert in hierarchischen Namensräumen. Jedoch ist das HDFS kein vollständiges POSIX Dateisystem. Zugriff auf Dateien beschränkt sich der Zugriff auf das Anfügen von Zeilen an das Dateiende, nicht jedoch in der Mitte. Code, der für POSIX-kompatible Dateisysteme geschrieben wurde, kann daher auch nicht ohne weiteres übernommen werden (Loughran 2013).

Dateien werden dabei zu Blöcken fester Länge mit eigener ID zerteilt und auf die DataNodes im Cluster verteilt. Um Datenverlust zu vermeiden, werden die Blöcke (mit der Standardeinstellung) dreifach auf unterschiedlichen Clusterknoten repliziert. Der blockorientierte Ansatz bringt hier den Vorteil, dass im Falle eines Knotenverlusts nicht die komplette Datei verloren geht, sondern nur ein Teil der Datenblöcke, die sich durch die übrigen Kopien wiederherstellen lassen. (Fischer 2010) Die Informationen über die Verteilung und Organisation der Datenblöcke und ihrer Repliken liegen im NameNode vor.

Will ein Client nun eine Datei lesen, muss er also zunächst eine Verbindung zum NameNode herstellen, der die Knoten ermittelt, auf denen die zugehörigen Blöcke liegen. Um den NameNode nicht zu überlasten, fließen die Daten nicht über ihn. Für den Transfer sind dann der Client und der entsprechende DataNode zuständig. Soll eine neue Datei geschrieben werden, muss der Client die Datei zunächst selbst in Blöcke aufteilen, bevor er den NameNode kontaktiert. Dieser trägt den Dateinamen in das System ein, reserviert einen Datenblock dafür und informiert den Client darüber, welcher Zieldatenblock auf welchem DataNode für ihn zu Verfügung steht, damit er mit der Übertragung beginnen kann. Für die redundante Speicherung kommt das **Replication Pipelining** zum Einsatz. Der Client erhält vom NameNode eine Liste mit DataNodes für das Speichern der Datenblöcke. Dabei sendet er einen Block in kleinen Portionen an den ersten DataNode, der diese nach dem Speichern an einen weiteren DataNode leitet und dieser wiederum an den letzten (abhängig vom eingestellten Replikationsfaktor) Datenknoten. (Vgl. Apache 2015)

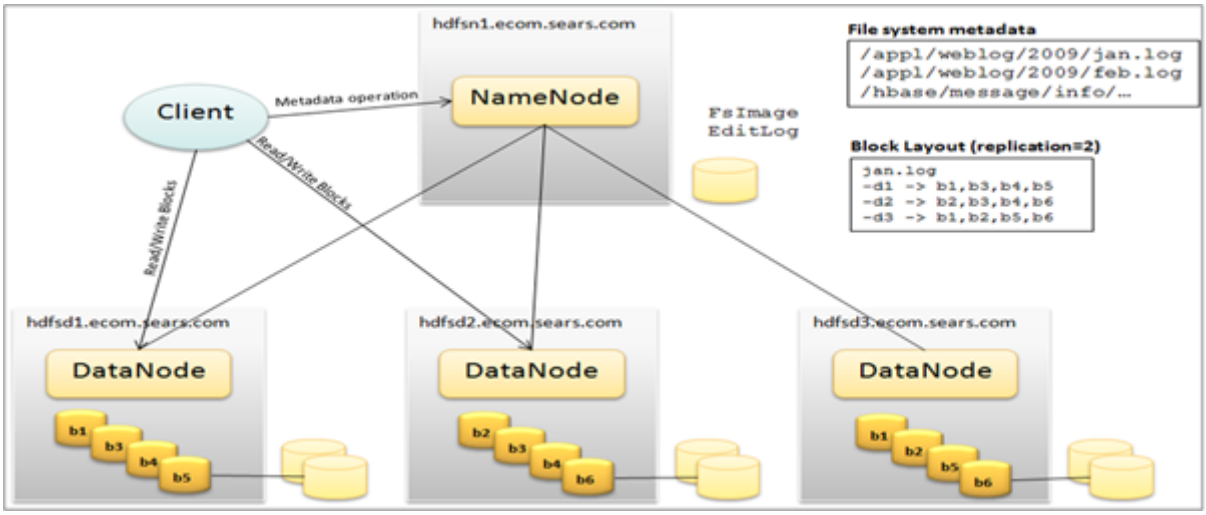
Zur Gewährleistung der Datenkonsistenz und der Verfügbarkeit bei Hardwareausfällen, überprüft der NameNode ständig den Zustand der DataNodes und die Anzahl der Replikationen. Dafür sendet jeder Datenknoten in (konfigurierbar) regelmäßigen Abständen **Blockreports** und **Heartbeats** an den NameNode. Ein Blockreport enthält eine Liste aller Datenblöcke des Knotens und ein Heartbeat signalisiert dem NameNode, dass der DataNode ordnungsgemäß funktioniert. Bleibt ein Heartbeat aus, wird der entsprechende Knoten als funktionsuntüchtig eingestuft und erhält keine Anfragen mehr. Kommt es durch so einen Kontenausfall zu einer Unterschreitung der Mindestzahl von

Blockreplikationen, werden die übrigen DataNodes mit der Speicherung der verlorengegangenen Datenblöcke beauftragt.

Schwieriger ist es jedoch, wenn der NameNode ausfällt, da er nur einmal vorhanden ist. Er führt ein Transaktionslog „**EditLog**“, in dem er jede Änderung an den Metadaten des Namensraums protokolliert (z.B. Das Anlegen einer neuen Datei). Zusätzlich erstellt er ein Snapshot der Dateisystem-Metadaten, das er ebenfalls im Hostsystem speichern. Dieses Abbild wird FsImage genannt. Beides, Protokoll und Image, sichert er im Dateisystem seines Host-Betriebssystems. Bei Hochfahren, werden Log und Image in den Hauptspeicher geladen und alle Aufzeichnungen aus dem EditLog in das **FsImage** übertragen und das aktualisierte Image wieder gespeichert. Das EditLog kann nun überschrieben werden. Dieser Vorgang wird **Checkpoint** genannt und wird momentan (Version 2.7.1) nur beim Hochfahren des NameNodes ausgeführt. Erst danach kann er wieder Client-Anfragen annehmen. (Vgl. Apache 2015; Fischer 2010)

Um diesen Vorgang zu beschleunigen, gibt es den „**Secondary NameNode**“, der regelmäßig die Änderungen des Logs auf das FsImage anwendet und die aktualisierte Version wieder auf den NameNode lädt. Die Bezeichnung „*Secondary* NameNode“ ist irreführend, da er nicht als Ersatz bei einem Ausfall dienen kann. (Gopalakrishnan 2015)

Mit Hadoop 2 wurde das HDFS dahingehend erweitert, als dass ein Cluster nun von mehreren NameNodes verwaltet werden kann, was bessere Performance durch weiteres horizontales Skalieren oder das Führen mehrerer Namensräume bringen und letztlich auch für mehr Zuverlässigkeit im Betrieb sorgen soll.



(Grafik-Quelle: DigitalMinds o. J.)

MapReduce

Eine der Ideen, die Hadoop zugrunde liegen, ist, dass es effektiver ist, Berechnungen zu den Daten zu bringen, anstatt Die Daten zu verschieben, dass also Applikationen, die mit einem großen Datenvolumen arbeiten auch in der Nähe dieser Daten zur Ausführung gebracht werden, um zeitintensives Verschieben der Daten durch das Netzwerk zu vermeiden. Dies geschieht nach dem Vorbild von Googles MapReduce-Framework, was zwei Phasen der verteilten Verarbeitung beinhaltet: eine **Map**-Phase, in der die Datenknoten ermittelt werden, die die geforderten Daten gespeichert haben und die Arbeitslast auf ebendiese Knoten verteilt wird; und eine **Reduce**-Phase, in der die Zwischenergebnisse zusammengeführt und verarbeitet werden.

Die MapReduce-Engine arbeitet auf dem Dateisystem und besteht aus einem **JobTracker**-Knoten, der die Daten lokalisiert und die Verarbeitungsfunktionen verteilt und koordiniert, und einer Reihe von **TaskTracker**-Knoten, die die map- und reduce-Funktionen ausführen und ihren Status zurück an den JobTracker senden. Es ermöglicht das Bearbeiten sehr großer Datenmengen und das Ausführen komplexer Aufgaben darauf, indem sie auf Unteraufgaben verteilt werden. Im Gegensatz zur klassischen Business Intelligence, werden hier keine ordentlich zusammengestellten Daten benötigt. Es ist egal, ob ihnen ein Schema zugrunde liegt oder nicht oder ob sie strukturiert oder unstrukturiert sind.

MapReduce-Operationen laufen im Wesentlichen in drei Schritten ab:

Map

Zunächst wird ein Inputfile (typischerweise vom HDFS) geladen, in **FileSplits** aufgeteilt und auf unterschiedliche Knoten verteilt. Dies erlaubt eine effiziente Verarbeitung auch sehr großer Inputfiles durch die massiv verteilte, parallele Arbeit der Mapper, die auf je einem solchen Split zum Einsatz kommen. Dabei geht das Splitting nach Bytelänge vor und weiß nichts über die interne Struktur der Dateien (Taggart 2011). Für jeden FileSplit wird dann eine **Map**-Operation gestartet. Über den **RecordReader** liest ein Map-Task dann seinen FileSplit ein und wandelt ihn in Key-Value-Paare um (Yahoo o. J.). Diese Paare werden dann von der benutzerdefinierten Map-Funktion gelesen und ihrer Programmierung entsprechend zu neuen Key-Value-Paaren verarbeitet, die vom **OutputCollector** dann an die Reducer geleitet werden. Die Paare werden nach ihren Schlüsseln in Subsets (oder auch „**Partitionen**“) gruppiert. Jeder Reducer erhält dabei ein eigenes Subset der Schlüsselwerte, damit er nur die Werte einer einzigen Schlüsselgruppe zusammenfassen kann. Dieser Vorgang der Verteilung der Mapper-Outputs an die Reducer wird als „Shuffling“ bezeichnet.

Es stehen einige Inputformate zur Verfügung:

InputFormat	**Beschreibung**	**Key**	**Value**
TextInputFormat	Default; liest Zeilen von Textdateien	Byte-Offset einer Zeile	Zeileninhalt
KeyValueInputFormat	Parst Zeilen in Key-Value-Paare	Alles bis zum ersten Tab-Zeichen	Rest der Zeile
SequenceFileInputFormat	Hadoop-spezifisches Binärformat	Benutzerdefiniert	Benutzerdefiniert

(Nach Yahoo o. J.)

Combine

Die Combine-Phase ist eine optionale Phase, die zu Optimierungszwecken verwendet werden kann. Sie findet nach dem Mapper und vor dem Shuffle statt, also bevor der Output der Map-Phase vom Hauptspeicher auf Disk geschrieben wird. Der **Combiner** wird auch als „Lokaler Reducer“ bezeichnet, da er nur auf den Daten einer Maschine arbeitet. Dabei werden die Key-Value-Paare der Map-Phase nach dem Schlüssel zusammengefasst und die Werte entsprechend zusammengerechnet. Auf diese Weise kann die Datenmenge noch einmal reduziert werden. Der Output des Combiners wird dann als Input an den Reducer übergeben. (Vgl. Dean/Ghemwat 2004: S. 6; Yahoo o. J.)

Reduce

Wenn die Map-Phase abgeschlossen ist, müssen die entstandenen Zwischenergebnisse (Key-Value-Paare), die nun lokal auf ihren Knoten vorliegen, so im Cluster ausgetauscht werden, dass alle Werte mit demselben Schlüssel zu einem Reducer geleitet werden. Dieser Vorgang stellt den einzigen Kommunikationsschritt der Maschinen im MapReduce dar, da ansonsten alle Mapper und Reducer getrennt und unabhängig voneinander parallel auf ihrem eigenen Datenbestand arbeiten (Yahoo o. J.). Nach einem Sortieren der Map-Erzeugnisse, kommt die vom Programmierer definierte Reduce-Funktion darauf zum Einsatz. Wertepaare mit demselben Schlüssel werden dabei aufsummiert. Das Ergebnis ist ein Output-File pro Reduce-Task auf der lokalen Platte oder im HDFS (Yahoo o. J.).

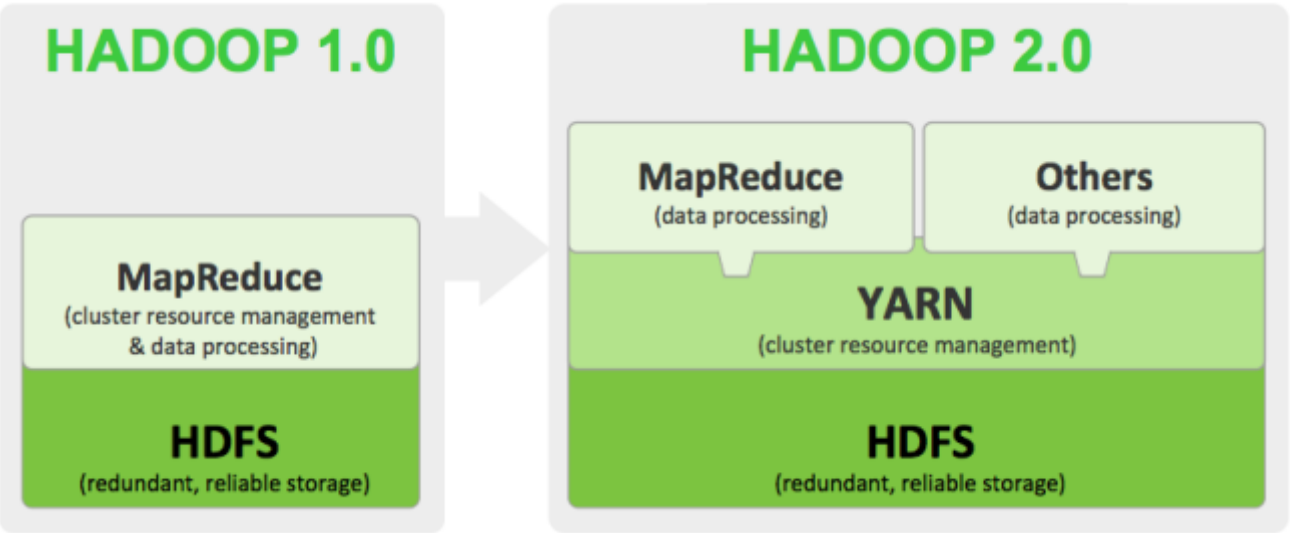
Das OutputFormat kann dabei bestimmt werden, ähnlich wie das InputFormat:

InputFormat	**Beschreibung**
TextOnputFormat	Default; schreibt Zeilen in der Form „key \t value“
SequenceFileOutputFormat	Schreibt binäre Files für das Lesen in darauffolgenden MapReduce-Jobs
NullOputFormat	Lässt seine Inputs außer Acht (erzeugt keinen MapReduce-Output)

(Nach Yahoo o. J.)

YARN

YARN steht für „**Y**et **A**nother **R**esource **N**egotiator“ („noch ein Ressourcen-Vermittler“) oder auch **MapReduce 2.0** (**MRv2**) und kam als wichtigste neue Komponente des Hadoop 2 Upgrades und übernimmt den Part des Ressourcen-Managements und Job-Schedulings und kommt so als Nachfolger des MapReduce-Frameworks daher. Es bildet eine neue Abstraktionsschicht, die das Cluster-Ressourcen-Management von der Datenverarbeitung durch MapReduce trennt, sodass MapReduce zwar weiterhinals Verarbeitungsmodell verwendet werden kann, aber daneben nun auch andere Alternativen verfügbar werden.



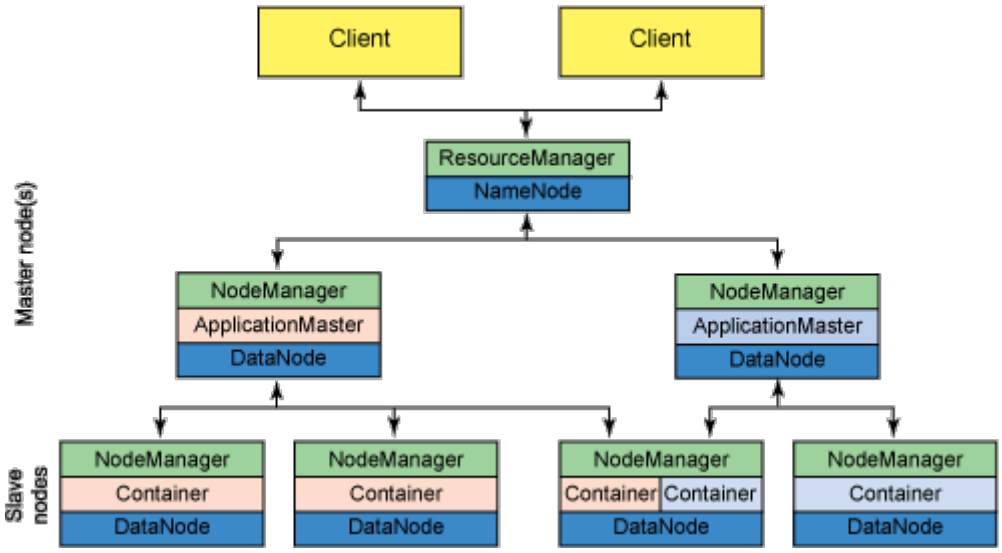
(Bild-Quelle: Sullivan 2014)

Architektur

Ein YARN-Cluster besteht aus folgenden Komponenten:

- **ResourceManager** (global): Er kontrolliert alle Programme/Jobs in einem Cluster vom Start bis zu ihrem Ende und vermittelt ihnen die nötigen Ressourcen (RAM, CPU, etc.). Er besteht aus einem *Scheduler* und einem *ApplicationManager*. Der Scheduler ist ein reiner Scheduler, führt also auch keinerlei Überwachung durch. Er richtet sich nach dem Ressourcenbedarf. Der ApplicationManager nimmt Job-Einreichungen entgegen und ist für die Vergabe der Container an den ApplicationMaster zu Programmbeginn und seinen Neustart bei einem Fehler verantwortlich.
- **ApplicationMaster** (pro Applikation): Fordert beim ResourceManager benötigte Ressourcen an und arbeitet mit dem NodeManager zusammen, um Aufgaben auszuführen und zu überwachen. Er kontrolliert dabei auch die Container und meldet sie beim ResourceManager, wenn der Job erledigt ist. Er ist nur für seine Applikation zuständig und arbeitet unabhängig von anderen Jobs.
- **NodeManager** (pro Knoten): Kontrolliert die Container und den Ressourcenverbrauch der Anwendungen und sendet Status-Reports darüber und über den Zustand des Knotens, auf dem er arbeitet, an den ResourceManager/Scheduler. Fällt ein Knoten aus oder wird unzuverlässig, so wird dies vermerkt und er bekommt auch keine Aufträge und Ressourcen mehr, ohne ein Zutun des Anwenders.
- **Container** (pro Applikation): Meint die Ressourcen, die einer Applikation pro Knoten zur Verfügung stehen.

(Vgl. Apache 2014; Jones/Nelson 2013)



(Bild-Quelle: Jones/Nelson 2013)

Das Trennen des Ressourcenmanagements von MapReduce durch die YARN-Architektur, die nun einen ResoucreManager mit der Aufgabe betraut, können über die ApplicationMaster, die für die Ausführung eines Jobs verantwortlich sind, nun auch mehrere verschiedene Anwendungen gleichzeitig in der Hadoop-Umgebung laufen (seien sie nun MapReduce-Jobs, graphbasierte Verarbeitung, effizientere Echtzeit-Verarbeitung oder Machine Learning). YARN bleibt dabei kompatibel zu MapReduce-Anwndungen, die unter Hadoop 1 geschrieben wurden.