Hadoop

HDFS und MapReduce

Seminararbeit im Modul NoSQL-Datenbanken

Bachelorstudiengang Informatik

Universität Leipzig

JOHANNES FREY

UNTER BETREUUNG VON ANIKA GROß

22. Januar 2012

Inhalt

1. Hadoop 2
   1.1 Kurzporträt 2
   1.2 Überblick über die Teilprojekte von Hadoop 2
2. HDFS 3
   2.1 Ziele von HDFS 3
   2.2 Architektur und Design 3
      2.2.1 Dateisystem 4
      2.2.2 Replikationsmechanismus 4
      2.2.3 Metadatenpersistenz 5
      2.2.4 Fehlertoleranz 5
      2.2.5 Dateiübertragung 6
3. Hadoop bei Facebook 8
   3.1 Anforderungen von Facebook 8
   3.2 Probleme mit MySQL 9
   3.3 Begründung für die Wahl von Hadoop mit HBase 9
   3.4 Verbesserungen an HDFS 10
      3.4.1 AvatarNode 10
      3.4.2 Echtzeitkritische Veränderungen 12
4. MapReduce 12
   4.1 MapReduce Framework 12
   4.2 Hadoop MapReduce 13
5. Anwendungen von Hadoop und MapReduce in der Bioinformatik 15
   5.1 Motivation 15
   5.2 Hadoop in der Cloud 16
   5.3 Cloudburst ein Seed-and-Extend Algorithmus für Hadoop 16
   5.4 vereinfachtes MapReduce Beispiel 17
   5.5 Evaluierung von Hadoop und Cloudburst 19
6. Zusammenfassung 19
7. Literatur 19

1. Hadoop

1.1 Kurzporträt

Hadoop [S+10] hat seine Ursprünge bei Yahoo. Yahoo entwickelte eine Open-Source Websuche namens Nutch [K+04]. Allerdings gab es Probleme bei der verteilten Berechnung auf vielen Knoten. Als Google im Jahr 2004 die Konzepte von MapReduce [DG04] und dem Google Filesystem [GGL03] publizierte, übernahm Yahoo diesen vielversprechenden Ansatz zur Lösung der Probleme. Zu Beginn von 2006 wurde Doug Cutting von Yahoo angeheuert, um die MapReduce Implementierung und das verteilte Dateisystem von Nutch als eigenes Projekt auszugliedern. Doug Cutting initiierte dieses Projekt unter dem Namen Hadoop als ein Teilprojekt von Lucene. Bei der Namenswahl ließ er sich von seinem Sohn inspirieren, welcher seinen gelben Stoffelefanten so nannte. Seit 2008 ist Hadoop ein Apache-Top-Level-Projekt und trägt als Logo den gelben Elefanten. Während die Anfänge von Nutch in C++ geschrieben waren, ist Hadoop größtenteils in Java implementiert [W09, S. 9-13]. Es zählt zur Kategorie der verteilten Dateisysteme und ist als Open-Source unter der Apache-Lizenz verfügbar [W11].

1.2 Überblick über die Teilprojekte von Hadoop

Hadoop[1] umfasst zahlreiche Teilprojekte. In der folgenden Auflistung sind die wichtigsten Teilprojekte kurz beschrieben.

Avro[2]	Ein System zur Datenserialisierung und persistenten Datenspeicherung.
Cassandra[3]	Eine skalierbare Multi-Master-Datenbank ohne „Single Point of Failure“.
Core	Eine Sammlung von Komponenten und Schnittstellen für verteiltes Dateisystem und I/O Verarbeitung.
Chukwa[4]	Ermöglicht verteiltes Sammeln und Analysieren von Daten
MapReduce	Implementierung und API von MapReduce
HBase[5]	Verteilte spaltenorientierte Datenbank
HDFS	Verteiltes Dateisystem auf Basis von GFS
Hive[6]	Verteiltes Data-Warehouse mit SQL-Dialekt
Mahout[7]	Bibliothek für Machine-Learning und Data-Mining
Pig[8]	Skriptsprache zur Datenflussbeschreibung für verteiltes Rechnen
ZooKeeper[9]	Koordinierungsdienst für verteiltes Rechnen

[1] http://hadoop.apache.org/
[2] http://avro.apache.org/
[3] http://cassandra.apache.org/
[4] http://incubator.apache.org/chukwa/
[5] http://hbase.apache.org/
[6] http://hive.apache.org/
[7] http://mahout.apache.org/
[8] http://pig.apache.org/
[9] http://zookeeper.apache.org/

2. HDFS

Im folgenden Abschnitt werden zunächst die Ziele vom Hadoop Distributed File System (kurz HDFS) vorgestellt [B11][W09, S. 41-42]. Im Anschluss daran werden die Architektur und das Design von HDFS im Detail präsentiert.

2.1 Ziele von HDFS

- Eine HDFS-Instanz kann aus Hunderten oder Tausenden von Rechnern bestehen. Diese verfügen wiederum über zahlreiche Komponenten, die größtenteils sehr kostengünstige Serverhardware ist. Daher stellen Hardwarefehler eher die Regel als die Ausnahme dar. Ein Kernziel von HDFS ist es, diese Fehler zu erkennen und die damit verbundene Störung schnell und automatisch zu beheben.
- Die Anwendungen, die auf HDFS betrieben werden sollen, sind keine Anwendungen, die auf gewöhnlichen Dateisystemen betrieben werden, sondern zeichnen sich durch sequentiellen Zugriff (Streaming) auf die Daten aus. Daher steht ein großer Datendurchsatz im Gegensatz zu einer geringen Latenzzeit im Vordergrund. HDFS ist somit eher als Stapelverarbeitungssystem als für interaktive Anwendungen gedacht.
- HDFS-Anwendungen sollen auf riesigen Datenmengen arbeiten. Eine Datei kann von einem Gigabyte bis mehrere Terabyte groß sein. HDFS soll Millionen solcher Dateien in einer Instanz unterstützen und über Hunderte von Rechnern in einem Cluster skalieren.
- Die HDFS-Anwendungen zeichnen sich durch ein besonderes I/O-Modell aus. Es werden einmal geschriebene Daten oft gelesen aber nicht mehr verändert. Dies erfordert keine komplexen Kohärenzalgorithmen und erhöht somit den Durchsatz.
- Weiterhin wird in HDFS die Annahme getroffen, dass die Netzwerkbandbreite limitiert ist und somit der Datenverkehr gering gehalten werden muss. Die einzelnen Berechnungen sollen so verteilt werden, dass die Berechnungen möglichst nah an den Daten ausgeführt werden.
- Zuletzt soll HDFS portabel und plattformunabhängig sein, um eine große Vielfalt an Anwendungen zu unterstützen.

2.2 Architektur und Design

HDFS verfügt über eine in Abbildung 1 dargestellte Master/Slave Architektur. Ein Cluster besteht aus einem einzelnen Master, dem NameNode, welcher das Dateisystem und dessen Namensraum verwaltet und den Zugriff von Clients regelt. Er führt Dateisystemoperationen wie Öffnen, Schließen, Umbenennen und Ähnliches aus. Die Slaves bilden die DataNodes. Gewöhnlich operiert ein DataNode auf genau einem Knoten, es können aber auch mehrere auf einem Knoten arbeiten. DataNodes verwalten den am Knoten zur Verfügung gestellten Speicher. Eine Datei wird intern in mehrere Blöcke aufgeteilt und auf verschiedenen DataNodes abgelegt. Der NameNode kümmert sich um die Verteilung der Blöcke auf die DataNodes. Diese hingegen sind für Lese- und Schreibanforderungen der Clients zuständig und führen Befehle des NameNodes zum Beispiel Anlegen, Löschen und Replizieren eines Blocks aus [B11]. Über den NameNode werden demnach nie Nutzdaten sondern nur Metadaten übertragen, was die Architektur deutlich vereinfacht und den NameNode als Flaschenhals entschärft.

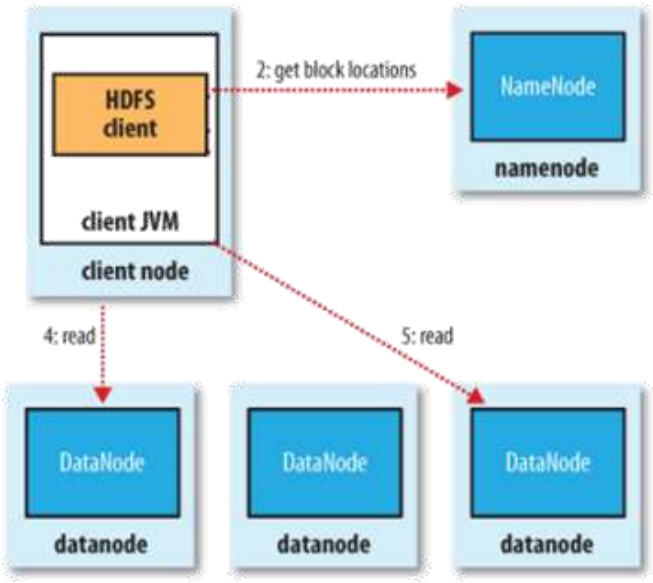

Abbildung 1 – Architekturübersicht Quelle: [W09]

2.2.1 Dateisystem

Der NameNode stellt ein klassisches hierarchisches Dateisystem zur Verfügung. Die gewohnten Operationen wie Verschieben, Löschen, Anlegen und Umbenennen von Dateien und Verzeichnissen können von einer Anwendung oder einem Benutzer ausgeführt werden. Hardlinks, Softlinks und erweiterte Benutzerrechte werden jedoch nicht unterstützt [B11]. Der NameNode protokolliert jede Veränderung am Dateisystem. Weiterhin vermerkt er sich zu jeder Datei den Replikationsfaktor. Dieser ist gewöhnlich 3, kann aber von der Anwendung bei Erstellung der Datei oder auch nachträglich geändert werden. Der Replikationsfaktor gibt an, wie viele Kopien von einer Datei auf unterschiedlichen Knoten vorhanden sein sollen.

2.2.2 Replikationsmechanismus

HDFS ist als ein zuverlässiges Dateisystem konzipiert worden. Dieses Konzept wird nicht mit RAID-Systemen erreicht, sondern mit dem Replikationsmechanismus. Dazu wird eine Datei in eine frei definierbare, jedoch feste Blockgröße eingeteilt. Der NameNode verteilt die Blöcke auf die entsprechend dem Replikationsfaktor eingestellte Anzahl verschiedener Knoten. Ein „Heartbeat“-Signal, welches jeder aktive DataNode an den NameNode sendet, gibt dem NameNode Auskunft über die verfügbaren DataNodes. Weiterhin sendet jeder DataNode einen sogenannten Blockreport, eine Liste mit allen Blöcken, die dieser DataNode hält, an den NameNode. Die Verteilung der einzelnen Blöcke auf die Knoten ist entscheidend für Zuverlässigkeit und Geschwindigkeit. Dabei spielt auch die Einteilung der Knoten in Racks eine beachtliche Rolle.

Die Kommunikation von Knoten innerhalb eines Racks ist viel schneller und günstiger als zwischen Knoten verschiedener Racks. Für die Ausfallsicherheit und Verfügbarkeit ist es jedoch wichtig, Daten auf verschiedene Racks zu verteilen. In der Standardkonfiguration ist dieses Rack-spezifische Verhalten jedoch deaktiviert und bedarf einiges an Feintuning durch einen Administrator. Apache bezeichnet die Rack-Strategien als experimentell in Hinsicht auf Effizienz und Verhalten [B11]. Es gibt verschiedene Strategien, wie die Kopien auf die Racks verteilt werden. Die einfachste Form ist das

Kopieren auf nur einen Rack. Dies hat bei einem Ausfall des Racks den Verlust der Daten zur Folge. Der Schreibdurchsatz ist jedoch sehr gut allerdings ist keine gute Balancierung beim Lesen möglich. Andererseits ist es auch möglich, alle Daten auf verschiedene Racks zu verteilen. Dies bietet eine gute Zuverlässigkeit, Verfügbarkeit und Leserate. Allerdings ist der Schreibdurchsatz sehr gering, da die Daten zu verschiedenen Racks Switches passieren müssen. Die Standardkonfiguration von HDFS geht einen Mittelweg. Die erste Kopie wird auf dem Knoten erstellt, auf dem der Client ausgeführt wird (es sei denn dieser ist ausgelastet oder der Client befindet sich außerhalb des Clusters, dann wird zufällig ein Knoten bestimmt). Die zweite Kopie wird auf einem anderen Rack gespeichert. Die dritte Kopie wird auf einem anderen Knoten auf demselben Rack der zweiten Kopie erstellt. Diese Strategie vereint eine Verfügbarkeit und Zuverlässigkeit der Daten mit einem guten (balancierbaren) Lesedurchsatz und einem guten Schreibdurchsatz der Anwendung, da diese nur einmal schreiben muss (im besten Fall sogar lokal). Die Daten müssen nur einmal von einem Rack zu einem anderen übertragen werden [W09, S. 68-69].

2.2.3 Metadatenpersistenz

Der NameNode speichert im sogenannten EditLog, einer Transaktionslogdatei, jegliche Veränderung in den Metadaten des Dateisystems. Das Verändern eines Replikationsfaktors oder Anlegen einer neuen Datei erfordert ergo immer einen neuen Eintrag im EditLog. Der gesamte Namensraum und die Zuordnung der Blöcke werden im Filesystem-Image (FsImage) gespeichert. EditLog und FsImage sind Dateien im Hostbetriebssystem des NameNodes. Der NameNode hält ein Abbild des Namensraums und der Blockzuordnung im Speicher. Dieses Abbild ist komprimiert, sodass 4 GB Hauptspeicher ausreichend für den NameNode sind. Beim Start des NameNodes wird das Abbild aus dem FsImage neu in den Speicher geladen und alle Änderungen aus dem EditLog werden auf das Abbild im Hauptspeicher angewendet. Abschließend wird aus diesem Abbild ein neues FsImage erstellt und das EditLog gelöscht [W09, S. 274-275].

Die DataNodes besitzen kein Wissen über die vorhandenen HDFS-Dateien, sondern speichern jeden Block in einer separaten Datei in ihrem Host-Dateisystem. Weiterhin benutzen sie eine Heuristik um die Anzahl der Dateien pro Verzeichnis im Hostdateisystem möglichst optimal zu gestalten und erstellen automatisch Unterverzeichnisse, um Ineffizienzen des Host-Dateisystems bei zu großen Ordnern zu vermeiden [W09, S. 277-278]. Beim Start eines DataNodes durchsucht dieser sein lokales Dateisystem und stellt eine Liste aller vorhandenen Blöcke zusammen. Diese Liste sendet er in Form des Blockreports an den NameNodes.

2.2.4 Fehlertoleranz

HDFS ist als ein zuverlässiges und verfügbares System konzipiert worden und verfügt somit über Fehlermechanismen für DataNode- und NameNode-Fehler sowie Netzwerkpartitionierung. Im Falle einer Netzwerkpartitionierung, zum Beispiel durch Ausfall eines Switches, können mehrere DataNodes vom NameNode getrennt werden. Dieser erkennt dies durch das fehlende Heartbeat-Signal und kennzeichnet die DataNodes als ungültig, welche nun somit keine IO-Anfragen mehr vom NameNode erhalten. Dies kann zur Folge haben, dass der Replikationsfaktor nicht mehr eingehalten wird. Der NameNode stellt fest, welche Blöcke dies betrifft und veranlasst eine erneute Kopie dieser Blöcke. Wenn ein Block durch fehlerhafte Netzwerkübertragung oder durch einen Festplattenfehler des DataNodes beschädigt wird, so kann eine ganze HDFS-Datei beschädigt werden. Um diesen Fehler zu erkennen, speichert der Client, der eine Datei anlegen möchte, eine versteckte HDFS-

Prüfsummendatei ab. Möchte ein Client eine Datei lesen, vergleicht er diese Prüfsummen mit der gelesenen Datei und kann im Fehlerfall andere Knoten anfragen [W09, S. 75-77].

Der NameNode als Herzstück stellt nach wie vor den „Single Point of Failure“ dar. Fehler im EditLog oder schlimmer im FsImage können fatale Folgen bis zum Totalausfall haben. Daher ist es möglich den NameNode so zu konfigurieren, dass er synchron Veränderung in mehrere Kopien dieser beiden Dateien schreibt. Dies zieht jedoch eine Verschlechterung der Ausführungszeit einer Transaktion nach sich. Weiterhin gab es einen zweiten NameNode, der periodisch das EditLog des primären NameNode auf eine Kopie von dessen FsImage anwendet. Dies hält das EditLog klein und erspart somit beim Ausfall des primären NameNode ein aufwändiges Zusammenführen dieser Dateien und beschleunigt somit den Neustart. Der CheckpointNode übernimmt in der aktuellen Version von HDFS diese Funktion. Hinzugekommen ist noch der BackupNode. Dieser erweitert die Funktion eines CheckpointNodes um ein Abbild des Dateisystems im Hauptspeicher. Dieses synchronisiert er mit dem NameNode und speichert es regelmäßig lokal ab. Im Gegensatz zum CheckpointNode entfällt somit das Übertragen des FsImage und EditLog vom NameNode und das Zurückschreiben des aktualisierten FsImage für jeden Backup-Vorgang [A11].

2.2.5 Dateiübertragung

Wenn ein Client eine Datei schreiben möchte, so wird diese zunächst in einem lokalen Zwischenspeicher gepuffert. Erst wenn dieser Puffer die eingestellte Blockgröße überschreitet (64 MB in der Standardkonfiguration), kontaktiert der Client den NameNode. Dieser trägt daraufhin den Namen der Datei in sein Verzeichnis ein und weißt den Block einem DataNode zu. Anschließend teilt er dem Client den DataNode und eine ID des Datenblocks mit. Abschließend überträgt der Client direkt die Daten an den DataNode. Abbildung 2 verdeutlicht diesen Ablauf. Der Sendevorgang des Clients erfolgt in eine Pipeline. Der erste DataNode empfängt kleine Teile des Blocks (4KB groß) und speichert diese lokal. Währenddessen überträgt er diese Teile an den zweiten DataNode. Dieser verhält sich ähnlich und sendet die Teile an den dritten DataNode. Die Daten werden an einem DataNode gleichzeitig empfangen und weitergereicht in der Pipeline, solange bis der Replikationsfaktor erreicht ist. Wenn der Client die Datei geschlossen hat, wird der letzte Block übertragen und der NameNode darüber informiert. Erst jetzt hält der NameNode diese Änderung im EditLog fest. Falls der NameNode vorher abstürzt, geht die Datei verloren [W09, S. 66-69][F10].

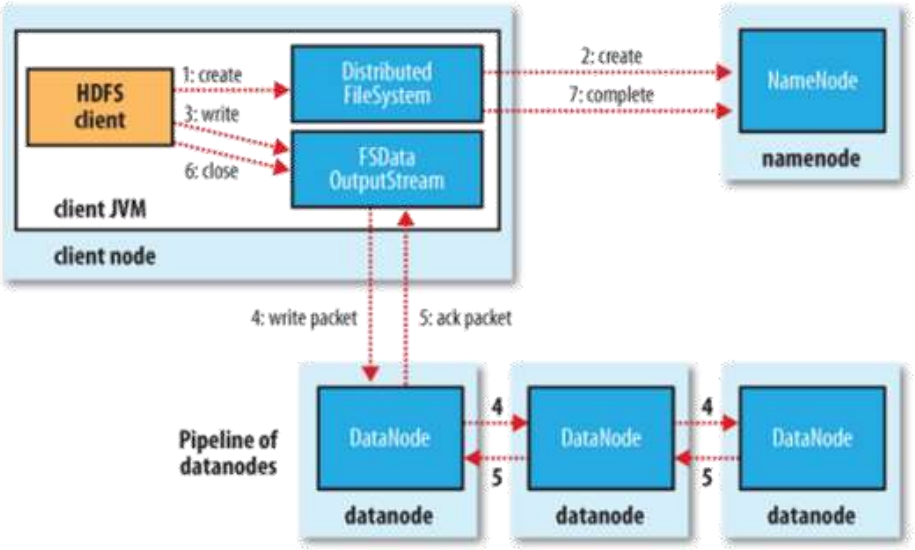

Abbildung 2 – Lesevorgang eines Clients Quelle: [W09]

Wenn der Client eine Datei lesen möchte, so stellt er die Anfrage mit dem entsprechenden HDFS-Dateinamen an den NameNode. Für jeden Block liefert dieser eine Liste mit Adressen der DataNodes zurück, welche diesen Block gespeichert haben. Hierbei sind die Einträge der NameNodes in der Reihenfolge der besten Erreichbarkeit für den Client sortiert. Wenn der Client zum Beispiel auf einem DataNode läuft, der einen Block beherbergt, so steht dieser DataNode für diesen Block ganz oben. Der Client beginnt dann eine Verbindung zu den jeweiligen DataNodes aufzubauen und die Blöcke entsprechend ihrer Reihenfolge von den DataNodes zu lesen. Einen Überblick liefert Abbildung 3. Dieser Prozess geschieht stückweise. Der Client bekommt ergo immer nur Informationen zu ein paar Blöcken und muss, wenn er diese gelesen hat, erneut den NameNode kontaktieren. Wenn die Übertragung eines Blocks fehlschlägt, so fragt der Client den nächsten DataNode für diesen Block in seine Liste an. Falls der Client den Defekt eines Blocks anhand des Abgleichs mit der Prüfsumme feststellt, so informiert er den DataNode darüber [W09, S. 63-65].

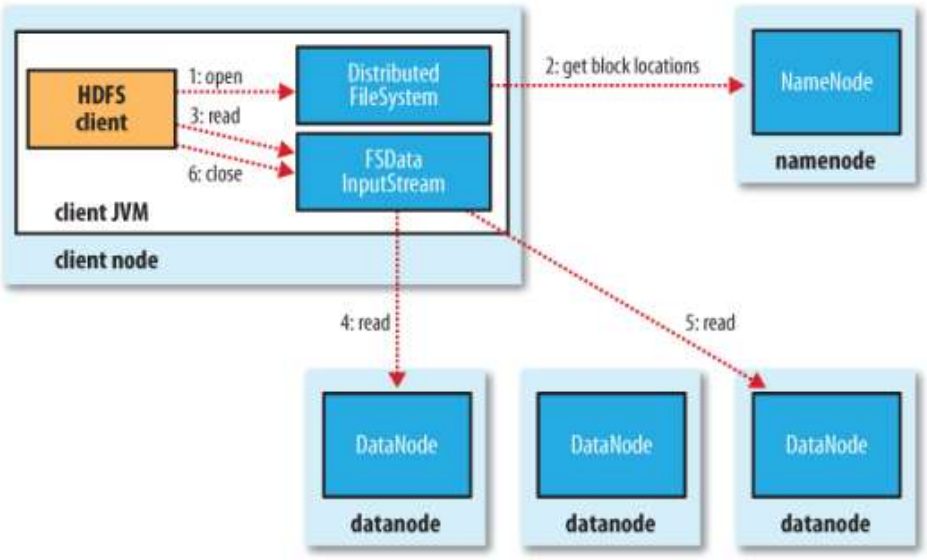

Abbildung 3 – Schreibvorgang eines Clients Quelle: [W09]

Dieses Design der Dateiübertragung ist der Kernpunkt von HDFS. Es erlaubt eine Skalierung über riesige Dateien und viele Clients, die parallel Schreib- und Lesezugriffe ausführen. Die Netzwerklast verteilt sich effizient über das gesamte Netzwerk, da die Daten überall verstreut sind. Der NameNode wird als potenzieller Flaschenhals entschärft, indem er keine Daten versendet, sondern nur der Koordinierung dient. Da er die dazu benötigten Daten im Hauptspeicher hält, geschieht dies sehr schnell und er kann somit viele Clients gleichzeitig verarbeiten.

3. Hadoop bei Facebook

Facebook ist neben Yahoo einer der populärsten Benutzer von Hadoop. In diesem Abschnitt werden zunächst die Charakteristika neuer Facebook-Anwendungen erläutert und die Gründe dafür genannt, dass Facebook Hadoop für die Implementierung dieser Anwendungen gewählt hat. Weiterhin werden einige von Facebook entwickelte Optimierungen und Anpassungen von Hadoop vorgestellt. Grundlage für den kompletten Abschnitt bildet eine Publikation von Facebook [B+11].

3.1 Anforderungen von Facebook

Bei Facebook wurde Hadoop zunächst in Verbindung mit Hive eingesetzt zum Speichern und zur Analyse von Daten. Hive bietet ein Data Warehouse unter Hadoop und ist von Facebook entwickelt worden. Hierbei wurden riesige Datenmengen in Hadoop importiert und dann anschließend offline stapelverarbeitet. Es stehen bei dieser Anwendung der Datendurchsatz und die Effizienz im Vordergrund. Die Ergebnisse wurden anschließend häufig in MySQL Datenbanken mit Memcached-Technologie kopiert, um den Anforderungen einer kurzen Latenzzeit für zufällige Lesezugriffe der Webserver gerecht zu werden.

Facebook hat 2011 einige neue Dienste eingeführt, die ganz andere technische Anforderungen stellen als die bisherigen. Sie benötigen sehr hohe Datendurchsätze bei günstigem und flexibel erweiterbarem Speicher. Weiterhin ist bei ihnen eine kurze Latenzzeit wichtig neben effizientem sequentiellen als auch zufälligem Lesen. Die neuen Anwendungen kann man in zwei Gruppen einteilen. Auf der einen Seite gibt es solche, die einen sequentiellen und nebenläufigen Lesezugriff auf einen sehr großen Echtzeitdatenstrom benötigen. Ein Vertreter davon ist Scribe, ein verteilter Log-Dienst. Auf der anderen Seite gibt es jene Anwendungen, die einen dynamischen Index für eine sehr stark wachsende Datenmenge brauchen, um einen schnellen zufälligen Zugriff auf die Daten zu ermöglich. Hier sei als Beispiel Facebook Messages genannt. Jeder Benutzer von Facebook hat mit dieser Anwendung die Möglichkeit eine Facebook E-Mail-Adresse zu bekommen, an welche sämtliche E-Mails, Chat-Nachrichten sowie SMS, die von einer bestimmten Gruppe von Nutzern geschrieben worden sind, geschickt werden.

Für Facebook Messages sind folgende Belastungsszenarien hervorzuheben. Ein sehr hoher Schreibdurchsatz entsteht durch die Tatsache, dass die Nutzer am Tag mehrere Milliarden von Instant-Messages versenden. Diese werden aufgrund der denormalisierten Struktur auch noch mehrfach geschrieben. Weiterhin entstehen bei der Speicherung der Nachrichten riesige Tabellen. Die Nachrichten werden in der Regel nicht gelöscht und wachsen somit stetig an. Außerdem können sie nicht archiviert werden, weil ein dauerhafter Zugriff mit kurzer Latenzzeit auf sie möglich sein muss. Der Import der Daten stellt ebenfalls eine große Herausforderung dar, denn alle bereits existierenden Nachrichten eines Benutzers müssen in das neue System eingespeist werden. Dazu

benötigt man die Möglichkeit eines Bulk-Imports sowie Suchanfragen über riesige Bereiche (large scan queries) zu stellen.

3.2 Probleme mit MySQL

MySQL-Datenbanken bieten zwar ein gutes Leseverhalten für zufällige Zugriffe, allerdings erzielen sie beim zufälligen Schreiben keine gute Performanz. Weiterhin skalieren sie relativ schlecht. Eine effiziente Lastverteilung bei voller Verfügbarkeit erfordert einen enormen Verwaltungsaufwand und in der Regel teure Hardware. Die Performanz von MySQL wird immer schlechter mit zunehmender Anzahl an Spalten in einer Tabelle und ist somit bei solch riesigen Tabellen relativ schlecht. Ein weiteres Problem stellt das automatische Fragmentieren und Verteilen (Sharding) von Tabellen auf unterschiedliche Server da. Da dem Datenzuwachs kein regelmäßiges Muster zugrunde liegt, kommt es häufig vor, dass ein MySQL-Server schlagartig mehr Daten verarbeiten muss als er imstande ist und somit ein zusätzliches manuelles Sharding durchgeführt werden muss. Die Lastverteilung ist somit nicht immer ideal gewährleistet und bedarf ständiger Kontrolle und Anpassung.

3.3 Begründung für die Wahl von Hadoop mit HBase

Facebook hat aufgrund der neuen Anwendungen folgende funktionale Anforderungen an ein Datenbanksystem. Hadoop in Kombination mit HBase erfüllt nahezu alle Anforderungen vollständig.

Elastizität – Es muss möglich sein schnell und ohne großen Aufwand die Speicherkapazität zu erhöhen. Dies soll ohne Unterbrechung des Produktionsbetriebs vonstattengehen und anschließend soll die Last automatisch auf den neuen freien Speicher verteilt werden. HDFS unterstützt alle diese Aspekte, wie bereits im Absatz zur Architektur beschrieben.

Hoher Schreibdurchsatz – Einige Anwendungen erfordern einen sehr hohen Schreibdurchsatz. Hoher Datendurchsatz ist eine Kernanforderung von HDFS.

Konsistenz innerhalb eines Data-Centers – Die Anzeige der ungelesenen Nachrichten bei Facebook-Messages soll der tatsächlichen Anzahl auch entsprechen. Für das Log-System vereinfachen Konsistenzeigenschaften die Programmierung. HDFS garantiert einfache Konsistenz der Dateien von HBase durch die Prüfsumme. HBase sichert die Konsistenz der Einträge.

Effiziente zufällige Lesevorgänge – Trotz ausgeklügelter Cache-Strategien, treffen viele Anfragen nicht den Cache, sondern das Datenbanksystem direkt. Daher sind kurze Lesevorgänge für ein flüssiges Benutzerverhalten wichtig. HBase bietet eine gute Leserate für zufällige Zugriffe.

Hohe Verfügbarkeit und Fehlertoleranz – Die Dienste bei Facebook sollen permanent verfügbar sein. Im Fehlerfall darf nicht das gesamte System ausfallen. Bei Hadoop sind beim Ausfall einiger DataNodes in der Regel nur wenige Daten nicht verfügbar. Außerdem hat Facebook schon positive Erfahrung mit Hadoop als Data-Warehouse gesammelt. Eine Schwachstelle ist jedoch der NameNode.

Atomare Lese-und Schreibmethoden – Für Facebook ist Atomarität für die Entwicklung von Anwendungen, die parallel Daten ohne gegenseitige Sperrung verändern, unverzichtbar. HBase garantiert Atomarität auf Zeilenebene.

Effiziente Bereichsanfragen – Einige Anwendungen benötigen einen effizienten Zugriff auf einen bestimmten Bereich der Daten, zum Beispiel auf die letzten hundert Nachrichten eines Nutzers. HDFS bietet ein gutes Leseverhalten bei solchen sequentiellen Daten.

Weiterhin sind noch folgende nichtfunktionale Anforderungen von Bedeutung. Es sollte ein System verwendet werden, mit dem Facebook möglichst schon Erfahrung gesammelt hat. Hadoop hatte Facebook zu diesem Zeitpunkt bereits im Produktiveinsatz und verfügte somit über Experten, die damit vertraut waren. Weiterhin spielte die Größe der Community eine Rolle. Als Apache Top-Level-Projekt schien Hadoop als besonders geeignet. Außerdem sollte mit dem Einsatz des Systems ein breiter Einsatzbereich erschlossen werden, und das System nicht nur für die aktuellen Anwendungen optimiert werden.

3.4 Verbesserungen an HDFS

HDFS ist ursprünglich für riesige Stapelverarbeitungsprozesse entwickelt worden. Um den Echtzeitanforderungen der Anwendungen gerecht zu werden und eine sehr hohe Verfügbarkeit zu gewährleisten, hat Facebook einige Details in der Architektur und der Implementierung von HDFS und Hadoop geändert.

3.4.1 AvatarNode

Der Schwachpunkt von HDFS ist der NameNode. Beim Ausfall des NameNodes ist der Betrieb von Hadoop nicht mehr möglich. Facebook benennt als größte Ausfallzeit nicht den Fehlerfall, sondern geplante Softwareupdates von HDFS, welche einen Neustart des NameNodes zur Folge haben. Ein Störfaktor für ein hochverfügbares Echtzeitsystem ist die lange Startzeit des NameNodes. Durch das Einlesen des FsImage und EditLog in den NameNode benötigt HDFS zu viel Zeit, um den Betrieb wieder aufzunehmen. Selbst eine Verwendung eines BackupNodes birgt immer noch das Problem der Blockreports, die jeder DataNode an den NameNode schickt beim Start des NameNodes. Das Facebook-Cluster verfügt über 150 Millionen Dateien und somit benötigt die Startphase des NameNodes mit BackupNode etwa 10 Minuten (im Gegensatz zu 45 Minuten ohne BackupNode). Außerdem bemängeln Facebooks Ingenieure die Zuverlässigkeit des Systems, da der BackupNode über jede Transaktion informiert werden muss, und sich in dieser Prozedur Fehler einschleichen können. Aus diesen Gründen ist der AvatarNode von Facebook entwickelt worden.

Ein Cluster verfügt jeweils über zwei AvatarNodes einem aktiven und einem passiven AvatarNode. Ein AvatarNode ist ein Wrapper eines NameNodes. Bei Facebook werden eine Kopie vom FsImage und EditLog über NFS angelegt. Der aktive AvatarNode schreibt seine Änderungen via NFS in die Kopie des EditLogs. Währenddessen liest der passive AvatarNode die Änderungen aus diesem EditLog ebenfalls via NFS und wendet diese auf sein lokales FsImage an. Bis auf eine geringe Verzögerung durch die NFS-Übertragung besitzt der passive NameNode dieselben Metadaten wie der aktive. Außerdem übernimmt der passive AvatarNode die Funktion des SecondaryNameNodes und erstellt in regelmäßigen Abständen ein aktuelles FsImage. Abbildung 4 zeigt das Zusammenwirken der beiden AvatarNodes.

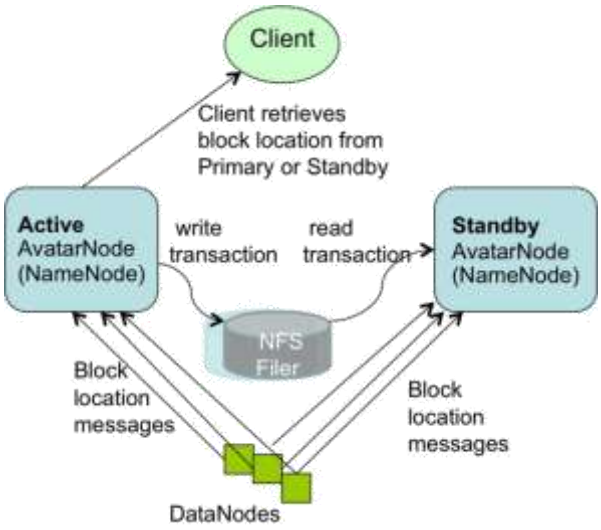

Abbildung 4 – Facebook AvatarNode Architektur Quelle: [B+11]

Die DataNodes wurden ebenfalls verändert. Sie kommunizieren mit beiden AvatarNodes anstatt nur mit einem und werden deswegen AvatarDataNodes genannt. Die AvatarDataNodes senden BlockReports und Heartbeats an beide AvatarNodes. Der passive AvatarNode verfügt somit über das aktuelle Wissen, welche Blöcke sich wo befinden. Beim Ausfall des aktiven AvatarNodes kann er dessen Funktion in weniger als einer Minute übernehmen. Mittels ZooKeeper wissen die AvatarDataNodes welcher der aktive AvatarNode ist und nehmen nur von diesem Dateianfragen entgegen.

Weiterhin wurde noch eine Änderung am EditLog vorgenommen. Wenn in eine Datei geschrieben wird, so werden die neu angelegten Blöcke erst im EditLog vermerkt, wenn die Datei geschlossen worden ist. Deswegen ist eine neue Transaktion eingeführt worden, damit der DataNode über den Verlauf einer Dateiübertragung in Kenntnis gesetzt wird. Somit kann dieser dann die Dateiübertragung eines Clients fortsetzen. Um beim Lesen des EditLogs via NFS zu verhindern, dass unvollständige Transaktionen gelesen werden, ist das Format um eine Transaktionslänge und -ID sowie eine Prüfsumme ergänzt worden.

Außerdem haben sie das DistributedFileSystem, ein Teil der Java-API, welcher der Client benutzt um Dateioperation auf HDFS auszuführen, durch ein eigenes AvatarDistributedFileSystem (ADFS) ersetzt. In ADFS ist ZooKeeper integriert worden. Für ein Cluster besitzt ZooKeeper einen zNode, welcher die physikalische Adresse des aktiven AvatarNodes repräsentiert. Wenn ein Client sich mit einem Cluster verbindet, ermittelt ADFS die Adresse des entsprechenden zNodes und leitet alle Aufrufe an diese weiter. Schlägt ein Aufruf fehl, so überprüft DAFS, ob sich die Adresse des zNodes geändert hat. Wenn der aktive AvatarNode ausgefallen ist, blockiert DAFS zunächst und nachdem der passive AvatarNode zum aktiven geworden ist, hält der zNode dessen Adresse. Der Client stellt nun die Anfrage gegebenenfalls an die neue Adresse.

Laut Facebook Ingenieuren passt die Standard-Rack-Strategie sich nicht gut der Topologie des Cluster an, da Knoten teilweise zufällig ausgewählt werden. Ihre Strategie ist, die Blöcke auf einen logischen Ring von Racks, wobei jeder Rack wieder einen logischen Ring von Knoten besitzt, zu verteilen. Je nach Definition der Größe der Ringe können sie damit eine deutlich geringere Ausfallwahrscheinlichkeit von Daten erzielen.

3.4.2 Echtzeitkritische Veränderungen

Diese Änderungen dienen alle der Verbesserung der Verfügbarkeit von Hadoop. Die folgenden Anpassungen haben ein verbessertes Echtzeitverhalten zum Ziel.

Die Knoten nutzen Remote Procedure Calls (RPC) über TCP-Verbindungen, um miteinander zu kommunizieren und interagieren. Wenn ein Client eine Datei mittels RPC anfordert von einem DataNode und dieser nicht antwortet, so überprüft der Client zunächst, ob er noch verfügbar ist und falls dem so ist, wartet er auf eine Antwort. Dieses Vorgehen soll die Netzwerklast reduzieren und verhindern, dass der Client seine Anfrage unnötig verwirft. Eine solche Wartezeit ist für Echtzeitanwendungen fatal. Tatsächlich könnte der Client einen anderen DataNode anfragen. Aus diesem Grund ist ein RPC-Timeout eingeführt worden, der dem Client erlaubt dies zu tun und damit die Latenzzeit im Fehlerfall enorm verkürzt.

HDFS unterstützt nur einen Schreibvorgang pro Datei. Der NameNode vergibt ein Schreibrecht an genau einen Client. Der Zugriff auf diese Datei, sowohl schreibend als auch lesend, von anderen Clients ist somit nicht mehr möglich, bis der schreibende Client die Datei ordnungsgemäß geschlossen hat. HDFS bietet eine indirekte Funktion, die einem Client das Schreibrecht entzieht, falls dieser z. B. abgestürzt ist. Diese Funktion benötigt jedoch viel Zeit und erfordert einen großen Mehraufwand. Aus diesem Grund ist die API um eine Funktion ergänzt worden, die dies explizit erledigt. Zusätzlich ist HDFS um die Möglichkeit hinzugekommen, auf eine Datei lesend zuzugreifen, während sie noch geschrieben wird. Der Client fragt dabei zunächst die Metadaten zu dieser Datei vom NameNode ab und bezieht dann die fehlenden Informationen direkt von einem der DataNodes aus der Pipeline. Von diesem DataNode kann er dann die Datei lesen. Weiterhin erkennen die Clients das Vorkommen einer Datei auf dem lokalen Hostdateisystem des DataNodes auf dem sie laufen. Sie lesen diese Datei dann direkt, anstatt sie über den DataNode zu beziehen. Dies ist um einiges effizienter und steigert den Durchsatz und verkürzt die Latenz bei einigen Anwendungen. Zuletzt ist noch die Funktion Hflush/sync, welche das Übertragen des lokalen Schreibpuffers eines Clients in HDFS veranlasst zu einer nicht blockierenden Funktion umgeschrieben. Da diese Funktion sehr oft unter HBase genutzt wird, konnte somit eine Schreibdurchsatzsteigerung erzielt werden.

4. MapReduce

Während das von GFS inspirierte HDFS, dem verteilten Speichern von riesigen Datenmengen dient, koordiniert das MapReduce Teilprojekt von Hadoop nebenläufige und verteilte Berechnung auf den Datenmengen. Dieser Abschnitt thematisiert das MapReduce Framework von Google und die Realisierung dessen in Hadoop.

4.1 MapReduce Framework

MapReduce [DG04] ist ein von Google im Jahr 2004 vorgestelltes Framework. Es ist mit dem Ziel entwickelt worden, nebenläufige und verteilte Berechnungen auf riesigen Datenmengen (im Petabyte-Bereich) auszuführen. Die zugrundeliegende MapReduce-Funktion berechnet aus einer Liste von Paaren aus Schlüsseln und Werten eine neue Liste aus Schlüssel-Werte-Paaren. Es wird in 3 Phasen unterschieden: Map-Phase, Reduce-Phase und Combine-Phase. In der Map-Phase wird eine vom Nutzer definierte und von der Anwendung abhängige Map-Funktion aufgerufen. Die Map-Funktion bildet ein Schlüssel-Wert-Paar auf eine Zwischenergebnis-Liste von neuen Schlüssel-Werte-Paaren ab. Sie wird auf alle Eingabepaare angewendet. Die Berechnung der Map-Phase kann parallel

ablaufen, da die einzelnen Aufrufe der Map-Funktion unabhängig voneinander sind. Abschließend werden die Werte, die alle den gleichen Schlüssel besitzen, in einer Liste zusammengefasst. Diese Listen bilden jeweils zusammen mit den gemeinsamen Schlüsseln neue Paare. Die Reduce-Phase folgt nach Abschluss der Map-Phase. In ihr wird die ebenfalls zu definierende Reduce-Funktion auf die neu gruppierten Paare angewendet. Die Reduce-Funktion bildet ein solches Paar auf eine Liste von Ergebniswerten ab. Die Aufrufe können auch wieder parallelisiert werden. Die Berechnung ist abgeschlossen, wenn alle Ergebnislisten vorliegen. Die Combine-Phase ist optional und kann nach der Map-Phase eingeschoben werden. Es ist eine Art vorgestellte einfachere Reduce-Funktion, die auf dem gleichen Knoten der Map-Funktion stattfindet und deren Ergebnis schon vorverarbeitet oder zusammenfasst, um die Netzwerklast zu verkleinern [WM11].

Ein Beispiel, welches den Ablauf der Map- und Reduce-Phase zeigt, ist in Abbildung 5 zu sehen. Als Eingabemenge dient eine Liste von Wetterdaten aus zahlreichen Messstationen, aus der in der Map-Phase das Jahr und der dazugehörige Temperaturwert extrahiert werden. In der Reduce-Phase wird anschließend pro Jahr der höchste Temperaturwert ermittelt.

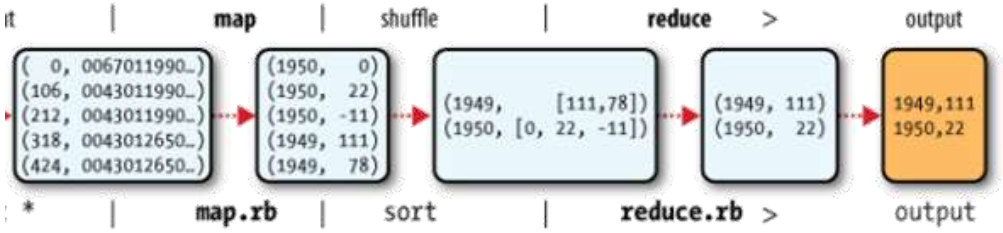

Abbildung 5 – Jahreshöchsttemperaturberechnung mit MapReduce Quelle: [W09]

4.2 Hadoop MapReduce

Hadoop bietet eine Implementierung des MapReduce Frameworks. Es ist in Java geschrieben, aber es ist auch möglich MapReduce-Anwendungen, die in einer anderen Programmiersprache als Java geschrieben sind auszuführen. In Hadoop können Clients Berechnungen mittels Jobs durchführen. Ein Job besteht aus einer Eingabedatenmenge, einem MapReduce-Programm und einigen Konfigurationsdaten. Hadoop unterteilt den Job in Tasks. Es wird in zwei verschiedene Typen von Tasks unterschieden: Map-Tasks und Reduce-Tasks. Für die Ausführung eines Jobs sind zwei verschiedene Typen von Knoten verantwortlich, welche in Abbildung 6 dargestellt werden: der Jobtracker und der Tasktracker. Der Jobtracker koordiniert alle Jobs, die auf einem System laufen, und verteilt sie auf diverse Tasktracker. Die Tasktracker sind für die Ausführung der Tasks verantwortlich und senden Statusberichte über den Fortschritt der Bearbeitung an den Jobtracker. Der Jobtracker besitzt ein Verzeichnis über alle laufenden Tasks und vergibt beim Abbruch eines Tasks durch irgendeinen Fehler den Task erneut. Hadoop unterteilt die Eingabedaten in Stücke einer festen Größe – sogenannte Splits. Es wird für jeden dieser Splits ein eigener Map-Task erstellt, welcher die entsprechende Map-Funktion auf jeden Eintrag in dem Split anwendet. Auch hier stellt die Größe eines Splits wieder eine entscheidende Rolle für die Performance, in diesem Fall bei der Ausführung eines Jobs, dar. Es ist von Vorteil die Splits ziemlich klein zu gestalten, da so die Last besser im Cluster verteilt werden kann. Andererseits, wenn die Splits zu klein sind, steigt der Verwaltungsaufwand für die Erstellung neuer Tasks und steht in keinem guten Verhältnis mehr zum Rechenaufwand pro Split. In der Regel ist für die meisten Jobs die Standardblockgröße von 64 MB eine gute Wahl. Entscheidend für die Performance ist aber außerdem auch, dass die Daten der Splits

dem Task lokal zur Verfügung stehen. Da die Blockgröße 64 MB groß ist, kann somit garantiert werden, dass alle Daten eines Splits auf einem Knoten lokal zur Verfügung stehen. Unter HDFS verfügt der Jobtracker über das Wissen, welche Daten sich auf den Knoten befinden. Er verteilt die Tasks so, dass die Daten lokal vorhanden sind. Ist das nicht möglich, so bevorzugt er Knoten, die sich im selben Rack befinden [W09,S. 153-166][U11].

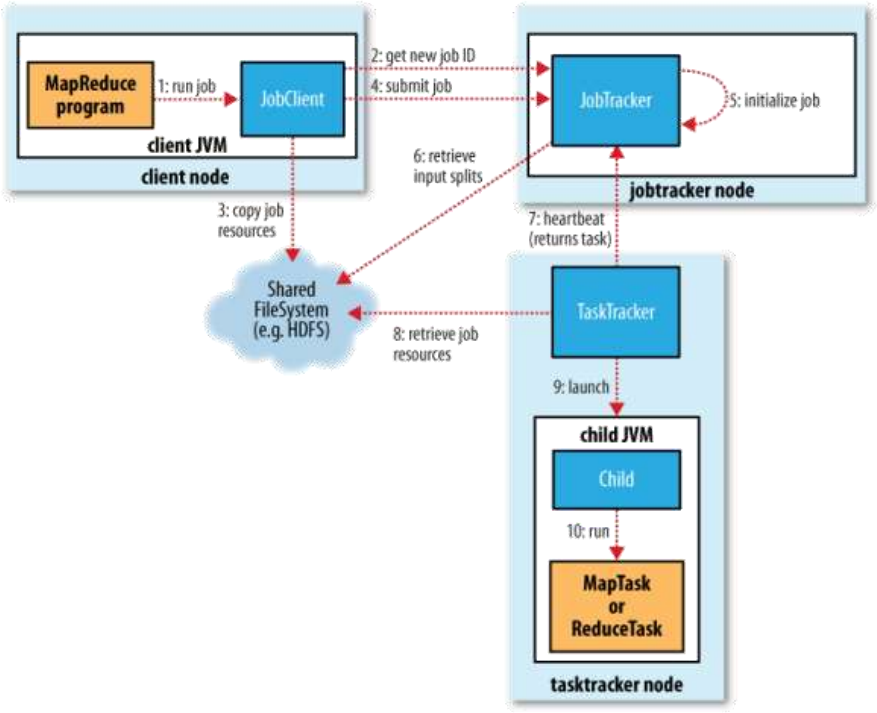

Abbildung 6 – Jobkoordinierung mittels Jobtracker und Tasktracker Quelle: [W09]

Die Ausgabedaten der Map-Funktionen werden nicht im HDFS-Dateisystem gespeichert, sondern im lokalen Hostsystem, da es sich um Zwischenergebnisse handelt, die nicht persistent sein müssen. Nach Abschluss der Berechnung der Map-Funktion werden diese Daten noch lokal nach Schlüsseln sortiert. Bei den Reduce-Tasks ist es nicht möglich die Daten lokal zur Verfügung zu haben, da die Ergebnisse der Map-Tasks auf verschiedenen Knoten verteilt liegen. Dementsprechend müssen die Daten zu den Knoten gesendet werden, auf dem der Reduce-Task läuft. Um die Netzwerkauslastung gering zu halten, ist es daher möglich eine Combine-Funktion zu definieren. Es wird allerdings von Hadoop nicht garantiert, ob und wie oft diese Combine-Funktion angewendet wird. Daher muss bei der Definition der Combine-Funktion darauf geachtet werden, dass die Reduce-Funktion dasselbe Ergebnis zurückgibt, unabhängig von der Anwendung der Combine-Funktion. Es ist nicht möglich, für jedes Problem eine solche Funktion zu finden. Weiterhin können die Daten auch vor dem Versenden komprimiert werden [W09, S. 29-30].

Die Anzahl der Reduce-Tasks kann vom Benutzer eingestellt werden. Um die Ausgaben der Map-Tasks auf mehrere Reduce-Tasks aufzuteilen, gibt es Partitioner. Für jeden Map-Task wird ein Partitioner gestartet. Ein Partitioner unterteilt die Daten eines Map-Tasks jeweils in so viele Partitionen, wie es Reduce-Tasks gibt. Anhand ihres Schlüssels, standardmäßig über eine Hashfunktion, werden die Einträge jeweils einer Partition zugewiesen. Anschließend werden die Partitionen, die jeweils zu einem Reduce-Task gehören, zu diesem kopiert. Die Reduce-Tasks fügen jeweils lokal die einzelnen Partitionen zusammen und sortieren sie dabei erneut nach dem Schlüssel.

Dies geschieht bereits während der Übertragung. Danach wird die Reduce-Funktion auf alle Einträge angewendet. Abschließend werden die Ergebnisse der einzelnen Tasks im HDFS gespeichert. Hierbei entsteht jeweils eine HDFS-Datei pro Reduce-Task. Es ist auch möglich keinerlei Reduce-Tasks zu starten, dann werden die Ergebnisse der Map-Tasks direkt im HDFS gespeichert [W09, S. 27-29]. Eine vereinfachte Darstellung der Partitionierung und der vorher beschriebenen Datenverteilung liefert Abbildung 7.

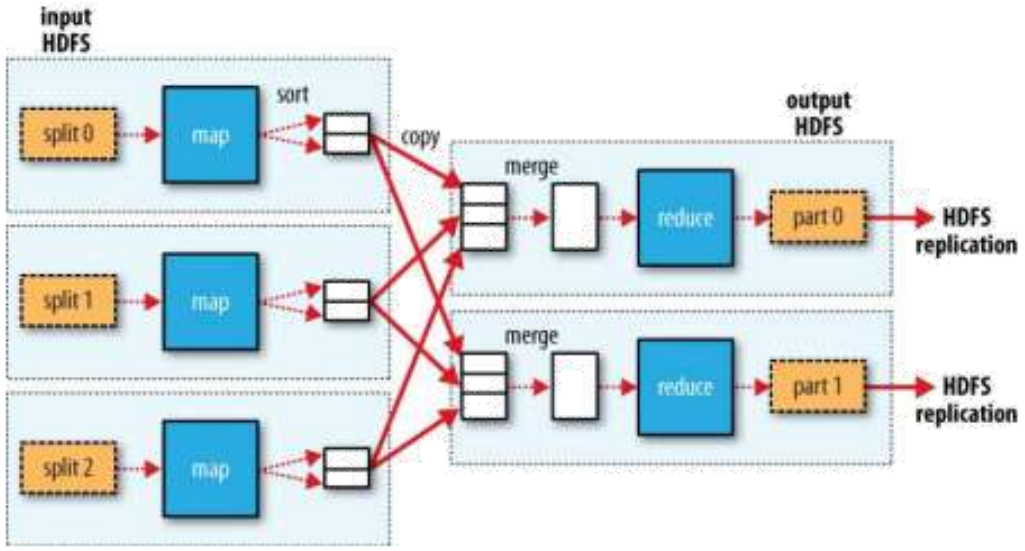

Abbildung 7 – Verteilung der Daten eines Jobs auf Map- und Reduce-Tasks Quelle: [W09]

5. Anwendungen von Hadoop und MapReduce in der Bioinformatik

Neben zahlreichen E-Commerce- und Web-Anwendungen kommen Hadoop und MapReduce auch in der Bioinformatik zum Einsatz. Der nachfolgende Abschnitt erläutert zunächst ausgewählte Probleme in der Bioinformatik, die mittels Cloud-Computing und Hadoop gelöst werden können. Zum besseren Verständnis wird exemplarisch ein MapReduce-Algorithmus vorgestellt. Ein vereinfachtes Beispiel zeigt den grundlegenden Ablauf des Algorithmus.

5.1 Motivation

In der DNA-Sequenzierung ist es zu einem Umbruch gekommen. DNA-Sequenzierungsgeräte der nächsten Generation zeichnen sich durch einen weit höheren Durchsatz im Leseverhalten von Sequenzinformationen aus. Diese lesen in wenigen Tagen mehr Informationen als früher mehrere Geräte der ersten Generation in einem Jahr und dies bei deutlicher Kostenersparnis. Somit ist es möglich, das menschliche Genom in 2 Monaten komplett zu lesen, wofür sonst mehrere Jahre benötigt worden sind. Ein Knick in den Preiskurven von DNA-Sequenzierung und Festplattenspeicher hat sich eingestellt. Im Gegensatz zu älteren Geräten ist die Sequenzierung deutlich günstiger geworden. Dies hat zur Folge, dass die Kosten pro gelesenem Basenpaar sich schneller halbieren als die Kosten pro Byte einer Festplatte. Die reine Sequenzierung von DNA bringt allerdings wenig wissenschaftlichen Nutzen. Die zum Teil relativ kurzen gelesenen DNA-Sequenzen müssen zu einer kompletten DNA zusammengesetzt werden, oder sollen um Lesefehler bereinigt werden oder mit Referenzgenomen verglichen werden. Die bioinformatische Umsetzung dieser Problemstellungen erfordert enormen Speicher und Rechenleistung. Einige früher häufig genutzte Programme sind

schlicht mit der gewachsenen Menge an Informationen überfordert, da sie z. B. die gesamten Informationen eines Genoms in den Hauptspeicher laden wollen, ohne dass die zugrundeliegende Rechnerarchitektur dies unterstützt. Ein weiteres Problem liegt in der Natur der Arbeitsweise solcher Prozesse wie DNA-Vergleich. Wenn Daten in Zusammenhang mit anderen Daten ausgewertet und in Bezug gesetzt werden, so entstehen wieder enorme Mengen an neuen Daten [B10].

5.2 Hadoop in der Cloud

Eine mögliche und vielversprechende Antwort auf die Probleme der Molekularbiologen lautet Cloud-Computing. Es bietet die Möglichkeit, sowohl Speicherplatz als auch Rechenleistung zu mieten und dynamisch an die aktuellen Daten und Probleme anzupassen. Einer der ersten Cloud-Computing Dienste stammt von Amazon – die „Elastic Compute Cloud“[10] (EC2). Dieser Dienst bietet die Möglichkeit, mit stündlicher Abrechnung virtuelle Rechner zu mieten. Auf den virtuellen Rechnern kann Hadoop installiert werden, zum Speichern von Dateien und zur Durchführung von verteilten Berechnungen. EC2 verfügt mittlerweile über einige Dienstleistungen speziell für die Anwendung der Genforschung. So ist zum Beispiel eine Kopie der Daten des „1000-Genome-Projekts“[11] lokal an den Cloud-Dienst angebunden, was vermeidet, dass diese Daten erst heruntergeladen werden müssen. Außerdem liegen bereits Konfigurationsdaten für verschieden Hadoop Versionen vor, sodass eine schnelle und somit kostengünstige Initialisierung des Clusters möglich ist [B10]. Dies und das Erscheinen einiger Algorithmen für Hadoop, so gibt es zum Beispiel eine Biodoop [LSZ09] genannte Sammlung von Anwendungen und Algorithmen direkt für den Einsatz auf Hadoop, haben zu einer guten Akzeptanz von Hadoop und einem breiten Einsatzspektrum in der Genforschung geführt. Typische Anwendungsszenarien sind DNA-Sequenzierung[12], Alignments[13], Genexpressions-Analyse[14] sowie topologische Untersuchung biologischer Netzwerke [T10].

5.3 Cloudburst ein Seed-and-Extend Algorithmus für Hadoop

Viele herkömmliche Algorithmen, die mit der wachsenden Menge an Daten nicht mehr umgehen können, sind nicht für eine parallelisierte Ausführung im großen Stil konzipiert. Um eine schnelle Berechnung in der Cloud auf einem Hadoop Cluster durchführen zu können, bedarf es spezieller Algorithmen und Verfahren. So muss es möglich sein, dass eine Berechnung in viele parallel und unabhängig voneinander ablaufende Teilaufgaben zerlegt werden kann. Erst dann ist die Implementierung mit Hadoop MapReduce möglich. Ein Verfahren, welches allerdings schon vor Erscheinen der neuen Sequenzierungsgeräte eingesetzt worden ist, sich jedoch hervorragend parallelisieren lässt, bezeichnet man als Seed-and-Extend.

Mittels Seed-and-Extend kann das Vorkommen vieler kurzer Reads in einem relativ langen Referenzgenom effizient untersucht werden – man spricht vom sogenannten Read-Mapping. Cloudburst [S09] ist ein für Hadoop MapReduce entwickelter Algorithmus auf Basis des Seed-and-Extend-Verfahrens. Beim Seed-and-Extend-Verfahren in Cloudburst werden die Reads entsprechend einer einstellbaren Fehleranzahl k in k+1 gleich große Teile mit der Länge l zerlegt. Diese Teile schließen exakt aneinander an und dürfen sich nicht überlappen. Fehler sind hierbei Abweichungen einzelner Basenpaare der Reads vom Referenzgenom durch z. B. Austausch oder Verlust oder

[10] http://aws.amazon.com/de/ec2/
[11] http://www.1000genomes.org/
[12] http://de.wikipedia.org/wiki/DNA-Sequenzierung
[13] http://de.wikipedia.org/wiki/Sequenzalignment
[14] http://de.wikipedia.org/wiki/Genexpressionsanalyse

Einfügung eines Basenpaars. Anschließend wird das Genom in l-Gramme, Substrings der Länge l, zerlegt. Diese Teile überlappen jedoch. Alle nun entstandenen Strings der Länge l werden als Seeds bezeichnet. Danach werden die Seeds des Referenzgenoms mit den Seeds der Reads verglichen und es wird nach exakten Vorkommen, also äquivalenten Seeds, gesucht. Auf jedes Paar an übereinstimmenden Seeds wird der Extend-Teil angewendet. Hierbei werden durch Verlängerung der Seeds entsprechend ihrer ursprünglichen Nachbarbasenpaare Alignments gesucht, die maximal k Fehler besitzen.

5.4 vereinfachtes MapReduce Beispiel

Da der Cloudburst-Algorithmus relativ komplex ist, soll ein vereinfachtes Beispiel die Parallelisierung veranschaulichen.

Man hat zunächst folgende Eingabedaten zur Verfügung: ein Referenzgenom im Beispiel AATTCGAG und einige Reads TTC, CGA, TTT. Ein Buchstabe steht jeweils für eine Nukleotidbase. Hierbei sei angemerkt, dass Referenzgenome erheblich länger sind und die Anzahl der Reads um ein Vielfaches größer ist. Zum Vergleich: Das menschliche Genom besitzt nahezu eine 3 Milliarden Basenpaare lange Sequenz. Reads hingegen sind in der Regel 25-250 Basenpaare lang. Die Fragestellung im Beispiel lautet, wie oft und an welcher Stelle die Reads im Referenzgenom vorkommen. Dies ist ein Spezialfall im Seed-and-Extend-Verfahren bei dem k=0 ist, also keinerlei Fehler auftreten dürfen. In der Praxis wird jedoch selten nach exakten Vorkommen gefragt. Die vorliegenden Daten werden in Form von Schlüssel-Werte-Paaren dargestellt. Im Beispiel ist eine Notation gewählt worden, wo der Schlüssel eine ID des Genoms bzw. der Reads darstellt. Die eigentliche Basen-Sequenz wird im Wert gespeichert.

Nachdem die Schlüssel-Werte-Paare in HDFS Dateien abgespeichert worden sind, kann die Berechnung begonnen werden. Abbildung 8 zeigt den Verlauf der Map-Phase. In ihr wird zunächst jeweils eine Instanz der Map-Funktion auf die Reads angewendet. Die Map-Funktion vertauscht bei den Reads lediglich die Werte mit den Schlüsseln. Dieses Vorgehen ist völlig unabhängig von Read zu Read und kann damit parallel auf verschiedenen Knoten und Map-Tasks ablaufen. Wird die Map-Funktion allerdings auf das Referenzgenom angewendet, so wird ein Fenster der Länge 3 schrittweise über das Referenzgenom geschoben und an jeder Position das Trigramm des Fensterinhalts als Schlüssel und ein Offset zur Positionsbestimmung als Wert ausgegeben. Dieser Prozess kann jedoch nur von einer Map-Funktion ausgeführt werden. In der Praxis sind die Referenzgenome allerdings in viele Teilsequenzen untergliedert, wobei eine Teilsequenz durch ein Schlüssel-Wert-Paar dargestellt wird. Der Schlüssel dient als Positionsangabe der Teilsequenz innerhalb des Genoms und der Wert ist die eigentliche Teilsequenz. Somit könnte dann eine Map-Funktion pro Teilsequenz parallel ausgeführt werden. Eine Überlappung der Sequenzen und zusätzliche Informationen ermöglichen die Berechnung von Sequenz übergreifenden Trigrammen.

Abbildung 8 – Map-Phase

Nachdem die Map-Phase abgeschlossen ist, findet die Sortierungs- und Gruppierungsphase statt. Abbildung 9 verdeutlicht dies. Als Ergebnis der Gruppierung befinden sich nun Reads, die im Genom vorkommen, zusammen mit den korrespondierenden Genom-Trigrammen in einem Schlüssel-Wert-Paar.

Abbildung 9 – Sortierungs- und Gruppierungsphase

Abschließend findet die Reduce-Phase statt, die in Abbildung 10 dargestellt wird. In ihr wird die Reduce-Funktion wieder parallel auf jedes der Zwischenergebnispaare angewendet. Die Reduce-Funktion geht dabei die Werteliste durch und zählt für jedes Vorkommen eines Genom-Offsets eine Variable hoch. Abschließend schreibt sie jeden Read als Schlüssel und als Wert einen Tupel, der den Wert der Variable, welche der Anzahl der Vorkommen entspricht, und die jeweiligen Positionen im Genom enthält.

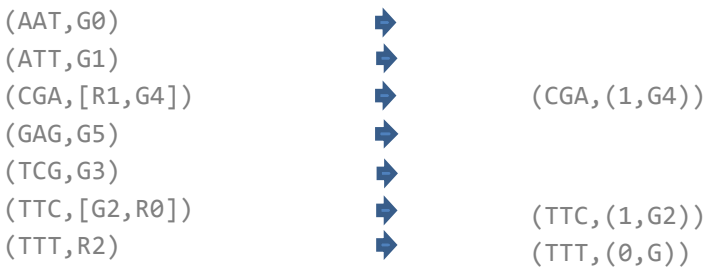

Abbildung 10 – Reduce-Phase

5.5 Evaluierung von Hadoop und Cloudburst

Abbildung 11 zeigt die Ausführungszeit des Cloudburst-Algorithmus in Abhängigkeit der Prozessorkerne. Es wurde jeweils die Durchschnittszeit über 3 Läufe auf Amazon EC2 ermittelt. Die Leistung eines Kerns ist vergleichbar mit einem 1.0–1.2 GHz Intel Xeon von 2007. Ein Knoten verfügt über je 2 Kerne. Es wurden 7 Millionen Reads, die jeweils je 36 Basenpaare lang sind, mit einer maximalen Anzahl von vier Fehlern auf das 22. Humanchromosom (ca. 49.69 Millionen Basenpaare) gemappt. Der Replikationsfaktor der Seeds entsprach jeweils der Hälfte der Kerne [S09].

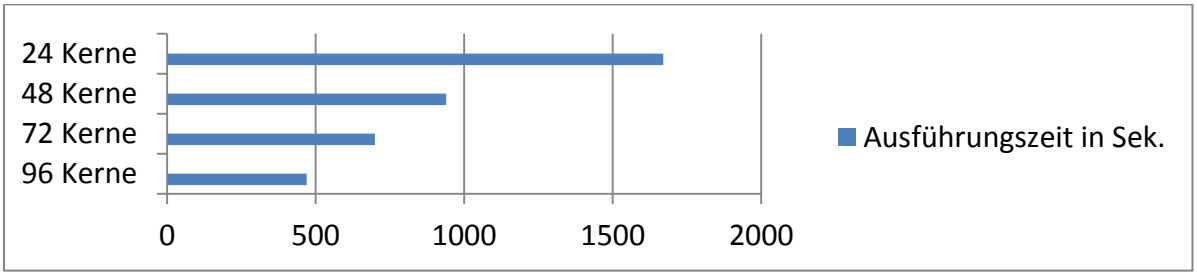

Abbildung 11 – Ausführungszeit von Cloudburst in Abhängigkeit von der Prozessoranzahl

Man kann erkennen, dass bei Vervierfachung der Prozessoranzahl sich die Ausführungszeit um den Faktor 3.5 verkürzt. Dies zeigt das Hadoop und der Cloudburst-Algorithmus nahezu linear skalieren und somit effiziente Berechnungen im Petabyte-Bereich möglich sind.

6. Zusammenfassung

Hadoop ist zuverlässig, verfügbar und verteilt. Es verfügt mit den Rack-Strategien über ein Feature, welches richtig eingesetzt eine gute Performanz gewährleistet. Durch den Wegfall von Lizenzkosten und bedeutend geringere Investitionskosten für Hardware, stellt es eine günstige Alternative zu namhaften Großrechnerinstallationen mit relationalen Datenbanken dar. Die virtuellen EC2-Hadoop-Cluster bieten eine schnelle und relativ einfache Konfiguration von Hadoop. Für den Betrieb eines eigenen Clusters ist jedoch eine Menge an Fachwissen und Feintuning sowie Erfahrung notwendig, um einen reibungslosen und optimalen Betrieb von Hadoop zu gewährleisten. Durch das MapReduce-Paradigma ist es möglich, leistungsfähige und implizit parallele Berechnungen auf riesigen Datenbeständen auszuführen. Allerdings bedarf es grundlegender Überarbeitung bereits bestehender Algorithmen, in vielen Fällen jedoch einer kompletten Neuentwicklung. Hadoop verfügt über eine aktive Entwicklergemeinde und der Einsatz von Hadoop bei Facebook und last.fm zeigen, dass es über großes Potenzial verfügt. Jedoch hat Hadoop sein volles Potenzial noch nicht ausgereizt, denn einige Funktionen sind experimentell oder noch gar nicht implementiert, sodass man gespannt sein darf auf künftige Versionen von Hadoop.

7. Literatur

[S+10] Konstantin Shvachko et al., *The Hadoop Distributed File System*, Mass Storage Systems and Technologies (MSST): 2010 IEEE 26th Symposium, 2010

[K+04] Rohit Khare et al., *Nutch: A Flexible and Scalable Open-Source Web Search Engine*, CommerceNet Labs Technical Report 04-04, 2004

[DG04] Jeffrey Dean, Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04: Sixth Symposium on Operating System Design and Implementation, 2004

[GGL03] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung , *The Google File System*, 19th ACM Symposium on Operating Systems Principles, 2003

[W09] Tom White, *Hadoop – The Definitive Guide*, O´REILLY, 2009

[W11] unbekannte Autoren, *Apache Hadoop*, 2011, http://de.wikipedia.org/wiki/Hadoop, aufgerufen am 8.1.2012

[B11] Dhruba Borthakur, *HDFS Architecture Guide*, 2011, http://hadoop.apache.org/common/docs/stable/hdfs_design.html, zuletzt aufgerufen am 27.12.2011

[A11] unbekannter Autor, *HDFS Users Guide*, 2011, http://hadoop.apache.org/common/docs/stable/hdfs_user_guide.html, zuletzt aufgerufen am 27.12.2011

[F10] Oliver Fischer, *Verarbeiten großer verteilter Datenmengen mit Hadoop,* 2010, http://www.heise.de/developer/artikel/Verarbeiten-grosser-verteilter-Datenmengen-mit-Hadoop-964755.html, zuletzt aufgerufen am 27.12.2011

[B+11] Dhruba Borthakur et al., *Apache Hadoop Goes Realtime at Facebook,* 2011

[U11] unbekannter Autor, *MapReduce Tutorial*, 2011, http://hadoop.apache.org/common/docs/stable/mapred_tutorial.html, zuletzt aufgerufen am 29.12.2011

[WM11] unbekannte Autoren, *MapReduce,* 2011, http://de.wikipedia.org/wiki/MapReduce, zuletzt aufgerufen am 3.1.2012

[B10] Monya Baker, *Next-generation sequencing: adjusting to data overload,* nature methods VOL.7 NO.7, 2010

[LSZ09] Simone Leo, Federico Santoni, Gianluigi Zanetti. *Biodoop: Bioinformatics on Hadoop*, 38th International Conference On Parallel Processing Workshops (ICPPW 2009), 2009

[T10] Ronald C. Taylor, *An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics*, BMCBioinformatics 2010 11(Suppl 12):S1, 2010

[S09] Michael C. Schatz, *CloudBurst: highly sensitive read mapping with MapReduce*, BIOINFORMATICS ORIGINAL PAPER Vol. 25 no.´11 2009, 2009