Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Technische Informatik, Zuverlässige Systeme

Leistungsanalyse von XtreemFS als Ersatz für HDFS in Hadoop

Lukas Kairies
Matrikelnummer: 4476422
lkairies@zedat.fu-berlin.de

Betreuer: Dr. Florian Schintke
Eingereicht bei: Prof. Dr. Katinka Wolter

Berlin, 12.September.2013

Lukas Kairies

Zusammenfassung

In dieser Arbeit wird die Leistungsfähigkeit des MapReduce-Framework Hadoop mit XtreemFS als verteiltes und POSIX-konformes Dateisystem bestimmt. Damit wird XtreemFS das für die Nutzung mit Hadoop entwickelte Dateisystem HDFS ersetzen. HDFS und XtreemFS werden verglichen und XtreemFS für Nutzung unter Hadoop konfiguriert. Zudem werden Optimierungen an der Hadoop-Schnittstelle von XtreemFS vorgenommen. Die Leistung von Hadoop mit XtreemFS wird mittels synthetischer Benchmarks und realer Hadoop Anwendungen gemessen und mit der Leistung von Hadoop mit HDFS verglichen.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

12.September.2013

Lukas Kairies

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 2
1.2 Inhalt . . . 3

2 Grundlagen **4**
2.1 Definitionen . . . 4
2.2 MapReduce Programmiermodell . . . 4
2.2.1 Berechnungsmodell . . . 4
2.2.2 Beispielanwendung: WordCount . . . 5
2.3 Apache Hadoop . . . 6
2.3.1 MapReduce-Implementierung . . . 7
2.4 Hadoop Distributed File System (HDFS) . . . 7
2.4.1 Architektur . . . 8
2.4.2 Verteilung . . . 10
2.4.3 Replikation . . . 11
2.5 XtreemFS . . . 12
2.5.1 Architektur . . . 12
2.5.2 Verteilung . . . 13
2.5.3 Replikation . . . 14
2.6 Tabellarischer Vergleich von XtreemFS und HDFS . . . 15

3 Konfiguration und Optimierung von XtreemFS für Hadoop **17**
3.1 Einfluss der Verteilung und Blockgröße auf die Datenverarbeitung . . . 18
3.1.1 Wahl der OSD Selection Policy . . . 18
3.1.2 Einfluss der Blockgröße auf den Durchsatz . . . 20
3.2 Einfluss der Replikation auf die Datenverarbeitung . . . 22
3.2.1 Wahl der Replikationsstrategie . . . 22
3.2.2 Einfluss des Replikationsgrads auf den Durchsatz . . . 23
3.3 Optimierungen der Hadoop-Schnittstelle von XtreemFS . . . 25

4 Test von realen Anwendungen undsynthetischen Benchmarks **28**
4.1 Anwendungen . . . 28
4.2 Testsystem . . . 29
4.3 Durchführung . . . 30
4.3.1 Micro Benchmarks . . . 31
4.3.2 Internetsuchanwendungen . . . 33
4.3.3 Anwendungen zum maschinellen Lernen . . . 36

5 Fazit **39**

1 Einleitung

Das Arbeiten mit großen Mengen von Dateien ist ein aktuelles Thema mit großer Relevanz in der Informatik und für Internet Unternehmen wie Google und Yahoo!.
Eine Möglichkeit große Datenmengen zu verarbeitet ist das Nutzen von Hochleistungsrechnern und entsprechenden parallelen Algorithmen. Als Dateisystem dienen dabei parallele Dateisysteme, die auf parallelen Datenzugriff optimiert sind. Da für Hochleistungsrechner spezielle Hardware eingesetzt wird, sind solche Systeme weniger Fehleranfällig als Systeme die mit herkömmlicher Hardware arbeiten. Allerdings sind die Anschaffungs- und Betriebskosten für solch ein System sehr hoch.
Eine Andere Möglichkeit große Datenmengen zu verarbeiten ist das Nutzen von Systemen, die es ermöglichen Anwendungen verteilt innerhalb eines Clusters auszuführen. Diese Systeme abstrahieren von der Verteilung der Dateien und Anwendung. Für die Verteilung der Dateien werden verteilte Dateisysteme genutzt, die meist speziell für das entsprechendes System entwickelt wurden, um so die Anforderungen des Systems für den Datenzugriff effizient umsetzen zu können.
Ein Modell, das solch ein System beschreibt ist das MapReduce Programmiermodell, welches in einer 2004 veröffentlichen Arbeit von Google[11] beschrieben wurde. Das MapReduce Programmiermodell ist durch die beiden Funktionen Map und Reduce aus funktionalen Programmiersprachen wie Lisp motiviert und soll das parallele Arbeiten auf großen Datenmengen innerhalb eines Clusters ermöglichen. Eine MapReduce Anwendung besteht laut dem Programmiermodell aus einer Map- und einer Reduce-Funktion.
Aufgrund des funktionalen Charakters des MapReduce Programmiermodell bietet dieses eine hohe Parallelität. Ein System, welches das MapReduce Programmiermodell implementiert, kann die hohe Parallelität nutzen um Anwendungen skalierbar auf einer Vielzahl von Knoten auszuführen. Zusätzlich kann solch ein System eine hohe Fehlertoleranz gegenüber Teilausfällen des Clusters bieten, so dass die Knoten eines Cluster aus herkömmlicher Hardware bestehen können. So dass die Anschaffungskosten für ein Cluster gering sind und eine flexible horizontale Skalierung möglich ist.
Eine Umsetzung des MapReduce Programmiermodells ist Teil des Open-Source Projektes Apache Hadoop [12]. Das Projekt umfasst ein Framework, mit welchem es möglich ist MapReduce Anwendungen zu schreiben und auf einem Cluster ausführen zu lassen. Neben der MapReduce Implementierung, beinhaltet Hadoop das verteilte Dateisystem Hadoop Distributed File System (HDFS) [8]. Dieses nutzt Hadoop um die Dateien, auf denen Hadoop-Anwendungen arbeiten, verteilt im Cluster zu speichern. Durch eine Schnittstelle in Hadoop lassen sich aber auch andere Dateisysteme verwenden.

1.1 Motivation

Typischerweise werden Dateien auf denen MapReduce Anwendungen arbeiten einmal geschrieben und mehrfach gelesen (write-once-read-many). Um dieses Zugriffsmuster zu optimieren und den Durchsatz zu erhöhen, verzichtet HDFS auf die Einhaltung der POSIX-Semantik und POSIX-API. So lassen sich Dateien in HDFS nicht von mehreren Klienten gleichzeitig schreiben und nicht beliebig verändern (random-write). Die POSIX-Semantik und POSIX-API werden im POSIX-Standard [6] definiert und beschreiben u.a welche Operationen POSIX-konforme Dateisysteme zur Verfügung stellen und wie diese sich verhalten. Aus der Einhaltung des POSIX-Standards folgt die Kompatibilität mit den meisten Anwendungen. Die fehlende POSIX-Kompatibilität von HDFS führt dazu, dass Anwendungen nicht auf die in HDFS gespeicherten Dateien zugreifen können wenn erstere nicht für den Zugriff auf HDFS entwickelt wurden. Es existieren Anwendungsfälle, in denen dies zu einem Problem führt. So kann es der Fall sein, dass

(1) die Eingabedateien für eine Hadoop Anwendungen von einer nicht mit HDFS kompatiblen Anwendung erzeugt werden oder die Ausgabedateien einer Hadoop Anwendung von weiteren, nicht Hadoop-Anwendung als Eingabe benötigt werden.

(2) nicht ausschließlich Hadoop-Anwendungen auf den Dateien in HDFS arbeiten sollen.

Diese Anwendungsfälle lassen sich unter Verwendung von HDFS nur umsetzen, wenn ein zweites, allgemein nutzbares Dateisystem neben HDFS genutzt wird. Beim Nutzen eines zusätzlichen Dateisystems müssten die Ein- und Ausgabedaten vor und nach der Ausführung der Hadoop Anwendung zwischen den Dateisystemen kopiert werden. Da typischerweise große Datenmengen verarbeitet werden, verzögert dieses Vorgehen die Ausführung der Anwendungen, da die Daten erst kopiert werden müssen, und erhöht den Speicherverbrauch, da die Daten mehrfach abgespeichert werden müssen.
Um dies zu vermeiden, soll in dieser Arbeit untersucht werden, inwiefern sich HDFS als spezialisiertes (purpose-build) Dateisystem gegen ein allgemein nutzbares (general-purpose) Dateisystem mit POSIX-Kompatibilität austauschen lässt und welchen Einfluss dies auf die Leistungsfähigkeit von Hadoop hat. So lassen sich die oben beschrieben Anwendungsfälle umsetzen ohne einen Mehraufwand durch den Im- und Export der Dateien zu erzeugen.
Als Ersatz soll das am Zuse-Institut Berlin (ZIB) entwickelte verteilte Dateisystem XtreemFS dienen [17]. XtreemFS bietet bereits eine Integrationsmöglichkeit in Hadoop an und ist im Gegensatz zu HDFS POSIX-konform. So eignet es sich für den Einsatz in einem Cluster, auf welchem nicht ausschließlich Hadoop Anwendungen arbeiten.

Um ein Einfluss vom XtreemFS auf die Leistungsfähigkeit von Hadoop zu bestimmen, werden die Ausführungszeiten von verschiedenen Hadoop-Anwendungen unter Verwendung von XtreemFS und HDFS gemessen. Die Ausführungszeiten werden anschließend gegenüber gestellt und so untersucht ob XtreemFS unter Hadoop einsetzbar ist ohne größere Leistungseinbußen hinnehmen zu müssen.

1.2 Inhalt

In dieser Arbeit soll HDFS mit dem am Zuse Institut Berlin entwickelten verteilten Dateisystem XtreemFS ausgetauscht werden und die Leistung von Hadoop mit XtreemFS untersucht werden. Dazu werden in Abschnitt 2 die Grundlagen für XtreemFS, HDFS und Hadoop vorgestellt. In Abschnitt 3 wird beschrieben wie XtreemFS für die Nutzung mit Hadoop konfiguriert werden kann und erste Vergleiche mit HDFS durchgeführt. Mit der Konfiguration aus Abschnitt 3 werden in Abschnitt 4 verschiedene Hadoop Anwendungen mit XtreemFS und HDFS als Dateisystem für Hadoop ausgeführt und die Ausführungszeiten verglichen. In Abschnitt 5 werden die Ergebnisse aus den vorangegangen Abschnitten zusammengefasst.

2 Grundlagen

2.1 Definitionen

Ein MapReduce Auftrag in Hadoop soll im weiteren *MR-Job* genannt werden. Ein MR-Job in Hadoop besteht im wesentlichen aus folgenden Teilen:

1. Ein- und Ausgabepfad der Dateien bzw. des Ergebnisses
2. Implementierung der Map- und Reduce-Funktion

Die Eingabedateien für einen MR-Job in Hadoop werden in mehreren Blöcken, auch *Chunks* genannt, gespeichert. Auf die Eingabemenge werden mehrere Ausführungen der Map-Funktionen angewandt, wobei jede Ausführung einen Teil der Eingabedateien (bspw. einen Block) verarbeitet. Eine Ausführung der Map-Funktion wird *Map-Task* genannt. Die Ergebnisse aller Map-Tasks werden von mehreren Ausführungen der Reduce-Funktion weiterverarbeitet. Eine Ausführung der Reduce-Funktion wird analog zur Ausführung der Map-Funktion *Reduce-Task* genannt. Jede Reduce-Task schreibt das Ergebnis ihrer Berechnung in eine eigene Datei im Ausgabepfad des MR-Jobs.

2.2 MapReduce Programmiermodell

MapReduce ist ein Programmiermodell, mit dem es möglich ist große Datenmengen zu verarbeiten. Dabei orientiert es sich an den aus funktionalen Programmiersprachen wie Lisp bekannten Operationen Map und Reduce. Anwendungen die in diesem funktionalen Stil geschrieben sind, lassen sich automatisch parallelisieren [11]. Dies ermöglicht dem Anwender seine Berechnungen verteilt und skalierbar auf mehreren Rechnern auszuführen. Die nötige Verteilung und Koordination der Prozesse und die Fehlerbehandlung bei Ausfällen wird dabei von der MapReduce-Implementierung übernommen, so dass der Nutzer lediglich die Map- und Reduce-Funktion definieren muss. Dabei benötigt er keine Kenntnisse über die Verteilung, was die Entwickelung solcher verteilten Anwendungen vereinfacht [11].

2.2.1 Berechnungsmodell

Eine Berechnung im MapReduce-Programmiermodell nimmt eine Liste von Schlüssel-Wert-Paaren als Eingabe und gibt eine Liste von Schlüssel-Wert-Paaren zurück. Wie bereits erwähnt, definiert der Nutzer für eine Berechnung zwei Funktionen: Map und Reduce. Das Verhalten der Funktionen ist wie folgt beschrieben [11]:

Map $(k1, v1) \rightarrow list(k2, v2)$: Die Map-Funktion nimmt als Eingabe ein Schlüssel-Wert-Paar aus der Eingabemenge und berechnet daraus als

Zwischenergebnis eine Liste von Schlüssel-Wert-Paaren. Dabei müssen die Typen der Paare der Eingabe nicht mit denen des Zwischenergebnisses übereinstimmen. Im Zwischenergebnis kann der selbe Schlüssel mehrfach vorkommen.

Reduce $(k2, list(v2)) \rightarrow list(v2)$: Vor der Ausführung der Reduce-Funktion werden die Werte aller Zwischenergebnisse mit dem selben Schlüssel von der MapReduce Umgebung in einer Liste zusammengefasst. Die neuen Paare des Zwischenergebnisses bestehen aus einem Schlüssel und einer Liste von Werten. Die Reduce-Funktion nimmt ein Paar aus diesem Zwischenergebnis und verarbeitet die Liste der Werte zu einer neuen Liste von Werten. Diese wird als Ergebnis zurückgegeben.

2.2.2 Beispielanwendung: WordCount

Am folgenden Beispiel soll deutlich gemacht werden, wie sich mit Hilfe von MapReduce eine Eingabemenge parallel verarbeiten lässt [11]:
Als Eingabe soll eine Menge von Dokumenten dienen. In dieser sollen die Vorkommen aller Wörter gezählt werden. Der Inhalt aller Dokumente wird in mehrere Teile zerteilt und auf jedem Teil wird die Map-Funktion aufgerufen. Diese nimmt als Eingabe den Namen des Dokumentes als Schlüssel und den Teil des Inhalts als Wert. In der Map-Funktion wird nun für jedes Wort w in der Eingabe ein Paar $(w, 1)$ erzeugt. Die Ausführungen der Map-Funktion lassen sich parallelisieren, da die Berechnung auf jedem Teil unabhängig von den Übrigen ist. Bevor das Zwischenergebnis an die Reduce-Funktion übergeben wird, werden alle Paare mit dem selben Wort als Schlüssel zusammengefasst. Dieser Schritt wird sort genannt. Die neuen Schlüssel-Wert-Paare haben die Form $(w, [1, ..., 1])$. Auf jedes dieser Paare des Zwischenergebnis wird die Reduce-Funktion aufgerufen. Diese addiert die Liste der Einsen auf und gibt das Paar (w, n) zurück, wobei n die Anzahl der Vorkommen vom Wort w in den Dokumenten ist. Die Ausführungen der Reduce-Funktion lassen sich ebenfalls parallelisieren. Das Ergebnis der Berechnung ist eine Auflistung aller Wörter mit der Anzahl ihrer Vorkommen.

Algorithmus 1 WordCount Anwendung in Pseudocode [11]

```
map(String key, String value):
// key: Dokumentenname
// value: Dokumenteninhalt
for each word w in value:
EmitIntermediate(w, "1");

reduce(String key, Iterator values):
// key: ein Word
// values: Eine Liste mit Zählerwerten
int result = 0;
for each v in values:
result += ParseInt(v);
Emit(AsString(result));
```

Algorithmus 1 zeigt eine Implementierung des oben beschriebenen Algorithmus für die WordCount Anwendung in Pseudocode.

2.3 Apache Hadoop

Neben Googles eigener Implementierung des oben beschriebenen Modells, entwickelt die Apache Software Foundation mit dem Apache Hadoop Projekt eine Open-Source Implementierung des MapReduce-Programmiermodells [12]. Hadoop ermöglicht es, die Ausführungen der Map- und Reduce-Funktion als Prozesse verteilt auf verschiedenen Knoten eines Clusters auszuführen. Dabei nutzt Hadoop Informationen über die Speicherorte der Dateien im Cluster, um die Berechnung auf den Dateien auf den Knoten durchzuführen, auf denen auch die Dateien gespeichert sind bzw. die Entfernung zwischen Dateien und den Knoten auf denen die Berechnung durchgeführt wird möglichst gering zu halten. So wird versucht die Netzwerklast und die Wartezeiten, die durch das Kopieren der Daten im Cluster entstehen, zu minimieren. Dazu nutzt Hadoop ein verteiltes Dateisystem, welches die Dateien über die Knoten verteilt und repliziert abspeichert. Neben dem Hadoop Distributed File System (HDFS), welches Teil des Apache Hadoop Projekts ist, kann Hadoop über eine Schnittstelle auch auf anderen Dateisystemen arbeiten. Um mit Hadoop genutzt werden zu können muss ein Dateisystem die in der Schnittstelle beschriebenen Methoden implementieren [1]. Im wesentlichen beinhaltet die Schnittstelle Methoden zum Bereitstellen von Ein- und Ausgabeströmen für den Zugriff auf die Dateien und für das zur Verfügung stellen der Information über den Speicherort der Dateiblöcke für die lokale Platzierung der Map-Taks.

2.3.1 MapReduce-Implementierung

Die MapReduce-Implementierung von Hadoop ist eine Master/Slave-Architektur, in der verschiedene Komponenten auf den Knoten des Clusters laufen. In einem Cluster gibt es einen Master-Knoten und mehrere Slave-Knoten. Auf dem Master-Knoten läuft der JobTracker. Diese Komponente der MapReduce-Implementierung verwaltet und koordiniert die Ausführung der MR-Jobs. Auf jedem Slave-Knoten läuft ein TaskTracker. Dieser verwaltet die Aufgaben, also die Ausführung der Map- und Reduce-Task auf den einzelnen Knoten. Wie in Abschnitt 2.1 beschrieben stellt eine Map- bzw. Reduce-Task die Ausführung der Map- bzw. Reduce-Funktion auf einem Block bzw. Zwischenergebnis dar. Wird ein MR-Job an das System übergeben, verteilt der JobTracker die Map- und Reduce-Tasks an die TaskTracker. Bei der Zuordnung der Tasks berücksichtigt der JobTracker die Speicherorte der Dateiblöcke auf denen gearbeitet werden soll. Die TaskTracker erzeugen Arbeiterprozesse, welche die Map- und Reduce-Tasks ausführen. Ein MR-Job in Hadoop besteht aus folgenden Teilen (siehe auch Abschnitt 1):

1. Spezifikation der Map- und Reduce-Funktion
2. Eingabe- und Ausgabepfad der Dateien.

Zusätzlich zur Spezifikation der Map- und Reduce-Funktion kann eine optionale Combine-Funktion angegeben werden. Die Combine-Funktion kann genutzt, werden um die Ergebnisse der Map-Tasks bereits vor der Weitergabe an die Reduce-Tasks anhand des Schlüssels zusammenzufassen. Dies verringert die Datenmenge, die über das Netz gesendet werden muss.
Jeder Map-Task wird ein Teil der Eingabedaten zur Verarbeitung übergeben. Eine weitere Verfeinerung des Modells ist die Zuordnung der Zwischenergebnisse über eine Partitionierungsfunktion zu bestimmen. Diese verteilt die Ergebnisse anhand ihrer Schlüssel und ist typischerweise eine Hashfunktion (z.B. „$hash(key) \, mod \, R$" mit R gleich der Anzahl der Reduce-Tasks).
Eine Reduce-Task besteht neben der eigentlichen Reduce-Funktion aus zwei weiteren Teilen, Shuffle und Sort. In der Shuffle-Phase werden alle Zwischenergebnisse, die der Reduce-Task zugeordnet werden, abgerufen und anhand ihres Schüssel zusammengefügt, so dass alle Werte mit dem selben Schlüssel in einem neuen Schlüssel-Wert-Paar zusammengefasst sind. In der Sort-Phase werden die so erzeugten Schlüssel-Wert-Paare nach ihrem Schlüssel sortiert und nacheinander in geordneter Reihenfolge der Reduce-Funktion übergeben. Zum Schluss schreibt jede Reduce-Task sein Ergebnis in eine eigene Datei im Ausgabepfad des MR-Jobs.

2.4 Hadoop Distributed File System (HDFS)

Das Hadoop Distributed File System (HDFS) wird im Rahmen des Apache Hadoop Projekts entwickelt. Dateien in HDFS werden in mehrere Blöcken

geteilt und repliziert gespeichert. Dabei werden die Metadaten und der Dateiinhalt separat gespeichert und verwaltet. HDFS wurde, so wie Hadoop selbst, für den Einsatz in Clustern mit Standardhardware entwickelt. Dazu ist HDFS fehlertolerant, so dass die Unzuverlässigkeit der Hardware keinen oder nur geringen Einfluss auf den Betrieb von Hadoop hat. So ist ein problemloser Betrieb auch auf großen Clustern mit einer daraus folgenden hohen Hardwareausfallwahrscheinlichkeit möglich [7]. Da HDFS primär für den Einsatz mit Hadoop entwickelt wird, ist es auf diesen Anwendungsfall hin optimiert. Das heißt vor allem, dass HDFS das typische write-once-read-many Zugriffsmuster von Hadoop-Anwendungen effizient ausführen kann. Das write-once-read-many Zugriffsmuster besagt, dass eine Datei beim Erzeugen geschrieben und nach dem erstmaligen Schließen der Datei nicht mehr verändert wird. Nach dem Schließen wird ausschließlich lesend auf die Datei zugegriffen, so dass die Optimierung der Lesegeschwindigkeit für dieses Zugriffsmuter im Vordergrund steht. Aus dieser Anforderung ergeben sich mehrere Einschränkungen die im Hinblick auf die POSIX-Semantik gemacht werden können, um im Gegenzug die Lesegeschwindigkeit auf die Daten zu erhöhen. Dies ermöglicht eine simple und effiziente Replikation der Dateien, in der, bis auf das einmalige Schreiben der Replikate, keine Koordination zwischen den Replikaten nötig ist und von jedem Replikat gelesen werden kann. Eine Datei kann in HDFS nur von einem Klienten gleichzeitig geschrieben werden, um so auf die komplexe Koordination von parallelen Schreibzugriffen zu verzichten. Diese Maßnahmen führen bei Hadoop-Anwendungen zwar zu optimierten Schreib- und Lesezugriffen, schließen aber aufgrund der fehlenden POSIX-Kompatibilität den Einsatz von anderen Anwendungen aus.

2.4.1 Architektur

HDFS basiert auf einer Master/Slave-Architektur. In einem Cluster gibt es einen sogenannten NameNode als Master und mehrere DataNodes als Slaves. Der NameNode verwaltet die Metadaten und reguliert den Klientenzugriff auf die Dateien. Auf ihm werde alle Metadaten-Operationen wie das Öffnen, Schließen und Umbenennen von Dateien ausgeführt. Der NameNode wählt auch den Speicherort für die Blöcke und die Replikate der Dateien aus.

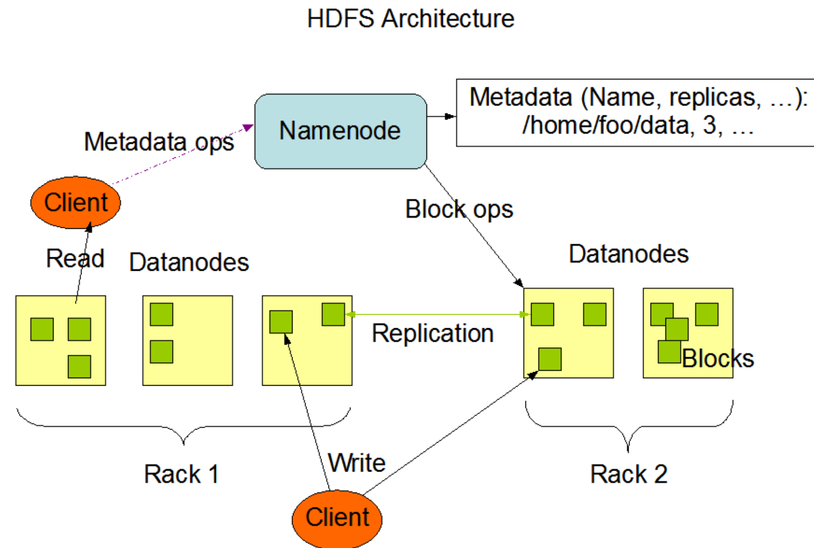

Abbildung 1: HDFS Architektur mit Nutzerzugriff [7]

Abbildung 1 zeigt die Architektur von HDFS mit Nutzerzugriff. Um die Metadaten persistent zu speichern nutzt der NameNode ein Transaktionslog, genannt EditLog, in dem alle Änderungen der Metadaten aufgezeichnet werden. Die gesamte Dateisystemhierarchie, einschließlich der Dateisystemkonfiguration und der Verteilung der Blöcke im Cluster, werden in einer Datei namens FSImage gespeichert. Sowohl das EditLog, also auch das FSImage werden auf dem lokalen Dateisystem des NameNodes gespeichert.
Beim Start lädt der NameNode das FSImage in den Arbeitsspeicher und aktualisiert es mit den Änderungen aus dem Editlog. Alle während des Betriebs gemachten Metadatenänderungen werden sowohl auf dem FSImage im Speicher angewandt, als auch im EditLog aufgezeichnet. So können alle Anfragen vom NameNode schnell verarbeitet werden, da alle nötigen Informationen bereits im Hauptspeicher liegen und die Änderungen gleichzeitig persistent im lokalen Dateisystem gespeichert werden. Im Fall eines Absturz, kann der aktuelle Dateisystemstand aus den lokal gespeicherten FSImage und EditLog wiederhergestellt werden [7].
Nebem dem NameNode gibt es eine weitere optionale Komponente zur Verwaltung der Metdadaten, den Secondary NameNode. Der Secondary NameNode hält ebenfalls eine Kopie des FSImage in seinem lokalen Dateisystem und lädt in Abhängigkeit seiner Konfiguration das aktuelle EditLog vom NameNode. Dies geschieht wenn das EditLog eine festgelegte Größe erreicht, spätestens aber nach Ablauf eines ebenfalls festlegbaren Zeitintervalls. Die Änderungen im EditLog werden auf das FSImage angewandt und dieses zurück an den NameNode geschickt. So müssen bei einem Neustart des NameNode nur wenige Änderungen vom EditLog übernommen werden, was

den Neustart beschleunigt[14].
Die DataNodes verwalten den ihnen auf einem Knoten zugeordneten Speicher und dienen so als Speicherort für die Blöcke der Dateien im HDFS Clusters. Sie führen die Schreib- und Leseoperationen der Klienten aus und sind auch für das Löschen, Erzeugen und Replizieren der Blöcke unter Anweisung des NameNodes zuständig. Die DataNodes selbst halten keine Informationen über die Dateien in HDFS, sie speichern lediglich die Blöcke als Dateien in ihrem lokalen Dateisystem. Beim Start eines DataNodes scannt dieser über das Dateisystem und erzeugt eine Liste aller Blöcke, die er in Form von lokalen Dateien gespeichert hat. Diese Liste, Blockreport genannt, schickt er an den NameNode. Während des Betriebs wird das Scannen des Dateisystems und Erstellen des Blockreports in periodischen Zeitintervallen wiederholt. Neben den Informationen aus den Blockreports, welche der NameNode nutzt, um den Replikationsgrad der Blöcke aufrecht zu erhalten, teilen die DataNodes so auch ihre Verfügbarkeit dem NameNode gegenüber mit. Erhält der NameNode innerhalb des festgelegten Intervalls keinen Blockreport eines DataNodes kann dieser davon ausgehen das der DataNode nicht mehr verfügbar ist, da er bspw. abgestürzt ist und kann die Wiederherstellung der so verlorenen Blöcke auf den übrigen DataNodes anstoßen [7].

2.4.2 Verteilung

Dateien in HDFS werden in Blöcken mit konfigurierbarer Größe über mehrere DataNodes verteilt gespeichert. Die Blockgröße lässt sich für jede Datei einzeln und global für alle Dateien festlegen, wobei die Standardgröße 64 MB beträgt. Wird eine neue Datei geschrieben, wird der Inhalt zunächst klientenseitig gepuffert. Das Puffern der geschriebenen Bytes beim Klienten soll den Einfluss der Netzwerkgeschwindigkeit und einer möglichen Netzwerküberlastung auf die Schreibgeschwindigkeit minimieren. Für Anwendungen, die eine Datei über den Klienten schreiben, geschieht dies transparent. Das heißt, dass alle Schreiboperationen auf den lokalen Puffer umgeleitet werden. Damit ist der Schreibvorgang für eine Anwendung mit dem Schreiben der Dateien in der Puffer abgeschlossen.
Hält der Puffer genug Bytes für einen Block wird der Puffer geleert und der NameNode wird informiert. Dieser wählt einen DataNode aus, auf dem der Block gespeichert werden soll. Sollte sich der Klient auf einem DataNode befinden, wird dieser gewählt, sonst bestimmt der NameNode einen zufälligen DataNode. Der NameNode teilt dem Klienten den DataNode mit und ersterer schickt den Block an letzteren, wo der Block gespeichert wird. Wird die Datei geschlossen, wird der restliche Puffer geleert und der letzte Block gespeichert. Anschließend teilt der Klient dem NameNode mit, dass die Datei geschlossen wurde. Ab diesem Punkt sind die Metadaten persistent im NameNode gespeichert. Daraus folgt, dass ein Abstürzen des NameNodes während des Schreibprozesses zum Verlust der Datei führt.

2.4.3 Replikation

HDFS nutzt Replikation, um unter anderem die Verfügbarkeit der Dateien bei Teilausfällen des Clusters zu gewährleisten. Zudem ermöglicht die Replikation der Blöcke eine flexiblere Verteilung der Map-Tasks durch den JobTracker. Wie auch die Blockgröße kann der Replikationsgrad für jede Datei einzeln oder global für alle Dateien festgelegt werden. Der Replikationsgrad kann nachträglich geändert werden. Die Verwaltung der Replikation ist Aufgabe des NamenNodes, während die Durchführung durch die DataNodes geschieht.

HDFS-Instanzen laufen meist auf Clustern mit mehreren Racks, also Verbunden von Knoten innerhalb eines Netzwerkschrankes. Typsicherwiese ist die Netzwerkbandbreite innerhalb eines Racks größer als die zwischen verschiedenen Racks. So kann die Replikation innerhalb eines Racks schneller durchgeführt werden kann. Fällt ein Rack aus sind allerdings alle Replikate innerhalb dieses Racks nicht mehr verfügbar, so dass die Replikation über mehrere Racks durchgeführt werden sollte. Aus diesem Umstand heraus verfolgt HDFS für die Replikation die sogenannte rack-aware Strategie. Angenommen der Replikationsgrad liegt bei drei (Standardwert in HDFS), so wird das erste Replikat wie oben beschrieben auf dem lokalen DataNode des schreibenden Klienten oder auf einem zufällig ausgewählten DataNode gespeichert. Das zweite Replikat und dritte Replikat werden auf zwei zufälligen Knoten innerhalb eines anderen Racks gespeichert. So ist ist die Verfügbarkeit der Datei auch beim Ausfall eines der beiden Racks gesichert und die Durchführung der Replikation durch die Platzierung der beiden letzten Replikate innerhalb eines Racks beschleunigt. Werden mehr als drei Replikate erzeugt, werden die restlichen Replikate gleichmäßig über alle Racks des Clusters verteilt.

Ist ein Block voll beschrieben und sind die DataNodes für die Speicherung des Blocks ausgewählt, wird die Replikation mittels *Replication Pipelining* durchgeführt. Dabei erhält der Klient wie bereits in Abschnitt 2.4.2 beschrieben für jeden Block eine Liste von n DataNodes vom NameNode, wobei n dem Replikationsgrad der Datei entspricht. Angenommen der Replikationsgrad sei wieder drei, dann schickt der Klient den Blockinhalt in mehreren 4 kB großen Teilen an den ersten Knoten. Zusätzlich schickt er eine Liste bestehend aus den übrigen 2 Knoten mit. Sobald der erste Knoten den ersten Teil des Blockinhalts erhält, speichert er diesen und alle folgenden Teile und schickt diese gleichzeitig an den zweiten Knoten der Liste mit der Information darüber welcher Knoten der dritte und somit letzte Knoten ist. Erhält der zweite Knoten die Teile speichert er diese ebenfalls und schickt sie an den letzten Knoten der Liste, wo sie auch gespeichert werden. Somit wird jeder Block auf drei Knoten gespeichert.

2.5 XtreemFS

XtreemFS ist ein am Zuse Institut Berlin (ZIB) entwickeltes verteiltes Dateisystem [17]. Der Inhalt und die Metadaten von Dateien werden in XtreemFS separat gespeichert, wobei die Dateiinhalte über mehrere Knoten verteilt und repliziert werden. Dateien werden in Volumes verwaltet, welche sich über den XtreemFS-Client in das lokale System einbinden lassen. Der Zugriff auf die Dateien in den eingebunden Volumes erfolgt entsprechend der POSIX-Semantik. So unterscheidet sich der Umgang mit XtreemFS Volumes nicht von dem anderer POSIX-konformer Dateisysteme [20].

2.5.1 Architektur

Dateiinhalte und Metadaten werden in XtreemFS separat gespeichert und verwaltet. Die Metadaten werden auf dem Metadata and Replica Catalog (MRC) und die Dateiinhalte auf dem Object Storage Device (OSD) gespeichert. Zusätzlich gibt es den Directory Service (DIR) als zentrale Stelle an der alle anderen Knoten registriert sind. Die drei Knoten und deren Aufgaben sind im Folgenden beschrieben [20]:

Metadata and Replica Catalog (MRC): Der MRC speichert und verwaltet die Metadaten der Dateien. Darüber hinaus überprüft er die Berechtigungen für den Dateizugriff. Die Metadaten werden in der mit XtreemFS entwickelten Datenbank BabuDB in Form von Schlüssel-Wert-Paaren gespeichert. Da der MRC die Metadaten verwaltet, kann er zusammen mit dem DIR als Analogie zum NameNode in HDFS betrachtet werden.

Object Storage Device (OSD): Ein OSD speichert die Dateiinhalte. Alle Lese- und Schreiboperationen der Klienten werden auf den OSDs ausgeführt. Ein OSD in XtreemFS ist damit analog zu einem NameNode in HDFS.

Directory Service (DIR): Der Directory Service ist die zentrale Stelle an der alle Knoten des Clusters registriert sind. Er wird unter anderem vom MRC genutzt um die OSDs im Cluster zu finden.

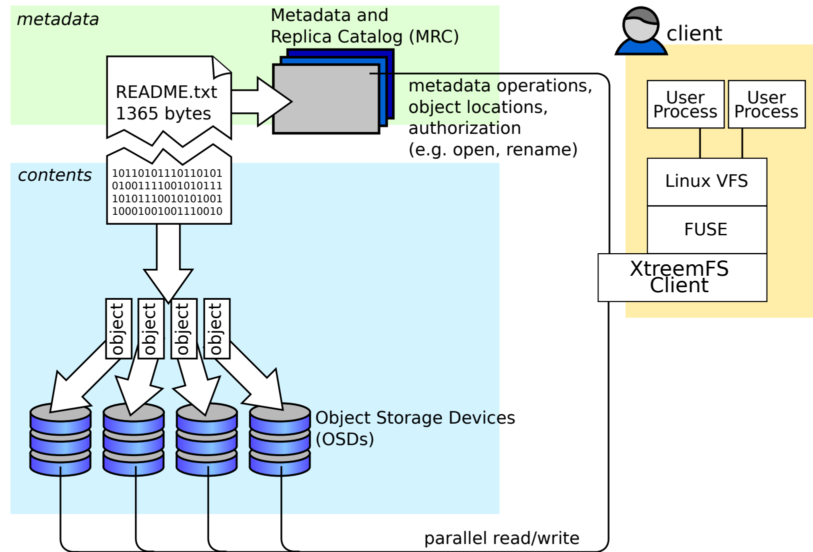

Abbildung 2: XtreemFS Architektur mit Nutzerzugriff auf eine Datei[20]

In Abbildung 2 ist die Architektur von XtreemFS mit Nutzerzugriff auf eine Datei dargestellt. Alle Knoten besitzen einen eindeutigen Identifikator (UUID) über welchen sie identifizierbar sind. Eine minimale Konfiguration für ein XtreemFS Cluster besteht aus einem DIR, einem MRC und mindestens einem OSD. Im Produktiveinsatz werden typischerweise deutlich mehr OSDs, als MRCs und DIRs genutzt.

2.5.2 Verteilung

Der Inhalt einer Datei wird in mehreren gleich großen Blöcken, auch Chunks genannt, auf den OSDs gespeichert. Es besteht die Möglichkeit, die Blöcke einer Datei über mehrere OSDs verteilt zu speichern. Diese Art der Verteilung der Blöcke wird in XtreemFS mit Striping bezeichnet und soll den Lese- und Schreibdurchsatz durch den parallelen Zugriff auf mehrere OSDs erhöhen. Wie die Blöcke im Cluster verteilt werden sollen wird in XtreemFS über die *Default Striping Policy* (DSP) bestimmt. Diese lässt sich für jede Datei einzeln oder für ein gesamtes Volume festlegen. In der aktuellen Version 1.4 von XtreemFS wird nur die RAID0 Strategie umgesetzt. Das heißt, die Objekte einer Datei werden zyklisch über die ausgewählten OSDs verteilt. Dabei lassen sich die Größe der Blöcke (*stripe-size*) und die Anzahl der OSDs über die verteilt werden soll (*stripe-width*) festlegen.
Für jedes Volume wird eine sogenannte *OSD Selection Policy* (OSP) festgelegt, anhand derer der MRC die OSDs und deren Reihenfolge für neue Dateien wählt. Eine OSD Selection Policy besteht aus einer oder mehreren vordefinierter Basisstrategien. Jede Basisstrategie erwartet als Eingabe eine Liste von OSDs und gibt eine Liste von OSDs zurück. So lassen sich mehrere

Basisstrategien hintereinander ausführen um aus der Liste aller registrierten OSDs eine geeignete Teilmenge auszuwählen. Zusätzlich lassen sich für mehrere Basisstrategien Attribute festlegen, mit denen sich die Auswahl weiter verfeinern lässt. Die Basisstrategien sind in drei Kategorien geordnet, welche im folgenden erläutert werden:

Filtering Policies: Filtering Policies erzeugen aus einer Eingabeliste von OSDs eine Teilliste. Alle OSDs der Teilliste erfüllen eine vordefinierte Bedingung. Die Liste der OSDs lässt sich nach bestimmten UUIDs oder Domains filtern, aber auch nach Eigenschaften wie der Mindestgröße des freien Speichers und der maximalen Antwortzeit.

Grouping Policies: Analog zu Filtering Policies filtern Grouping Policies die Eingabeliste nach bestimmten Kriterien. Dabei wird immer eine feste Anzahl von OSDs zurück gegeben. So kann die Gruppierung beispielsweise über eine sogenannte Datacenter Map erfolgen, in der die OSDs anhand ihrer IP in Datacenter gruppiert werden und die Entfernungen zwischen den Datacentern festgelegt wird. Dabei wird die Gruppe zurück gegeben, die am nächsten zum Klienten befindet.

Sorting Policies: Sorting Policies sortieren die Eingabeliste ohne OSDs zu entfernen. Das Sortieren kann beispielsweise zufällig, nach den Entfernungsangaben aus der Datacenter Map oder der Entfernung zwischen den Vivaldi Koordinaten der OSDs und des Klienten erfolgen. Die Vavaldi Koordinaten werden über die Antwortzeiten zwischen allen OSDs untereinander und dem Klienten bestimmt [10].

2.5.3 Replikation

Um die Verfügbarkeit und Zuverlässigkeit der Dateien zu gewährleisten, werden Dateien repliziert gespeichert. Die OSDs für Replikation werden durch die OSD Selection Policy bestimmt. XtreemFS unterstützt zwei Arten der Replikation:

read-only: Erzeugt ein Klient eine Datei, kann er diese bis zum erstmaligen Schließen beliebig oft schreiben. Nach dem Schließen wird die Datei als schreibgeschützt markiert und die Replikation gestartet. Analog zur Replikation in HDFS ist auch hier beim Zugriff auf ein Replikat keine weitere Kommunikation zwischen den Replikaten nötig und es kann von jedem Replikat gelesen werden. Bei der read-only Replikation lässt sich ein Replikat als volles oder partielles Replikat markieren.
Volle Replikate werden zeitnah nach dem Schließen der Datei erzeugt, wobei alle Blöcke der Datei repliziert werden. Durch die Nutzung von vollen Replikaten ergibt sich eine erhöhte Netzwerklast während der Replikation und der Zeitaufwand für die Replikation nimmt zu.
Partielle Replikate werden beim ersten Zugriff erzeugt und rufen nur

die Blöcke ab, die für die entsprechenden Operationen auf dem Replikat benötigt werden.

read-write: Die read-write Replikation erlaubt das Verändern der Dateien auch nach ihrer Erzeugung. Die read-write Replikation nutzt ein Quorum-Protokoll, das heißt, eine Anfrage wird nur dann ausgeführt, wenn die Mehrheit der Replikate darauf antwortet. Bei der read-write Replikation in XtreemFS wird zwischen dem primären Replikat und den Sicherungsreplikaten unterschieden. Es gibt immer nur ein primäres Replikat, wobei dieses Replikat den aktuellen Stand der Datei hat bzw. ihn erhält sobald es für eine festgelegte Zeit zum primären Replikat wird. Sollte es kein primäres Replikat geben, wird das zuerst angefragte Replikat zum primären Replikat [19].
Eine Anfrage von einem Klienten an eine Datei, repräsentiert durch ihr primäres Replikat, wird über dieses an alle Sicherungsreplikate weitergeleitet. Antwortet die Mehrheit der Replikate auf diese Anfrage wird die Operation ausgeführt. Sollte dies nicht der Fall sein, wird die Operation abgelehnt und ein Fehler an den Klienten geschickt.
In der aktuellen Version 1.4 von XtreemFS können die Blöcke bei der read-write Replikation nicht über mehrere OSD verteilt werden. Die Verteilung der Dateien erfolgt in diesem Fall ausschließlich über die Replikation.

Öffnet ein Klient eine Datei erhält er, vorausgesetzt er besitzt die entsprechenden Rechte, die Metadaten, die sortierte Liste aller Replikate und die sogenannte Capability vom MRC. Von da an werden alle weiteren Operationen vom Klienten auf der Datei auf den entsprechenden OSDs ausgeführt. Mit der Capability authentifiziert der Klient den Dateizugriff auf den OSDs. Die Sortierung der Liste wird für jedes Volume über die sogenannte *Replica Selection Policy* (RSP) bestimmt. Der Klient führt die Operationen auf dem ersten Replikat der Liste aus, sollte dieses nicht verfügbar sein, iteriert er über die Liste bis zum ersten verfügbaren Replikat und führt auf diesem die Operationen aus. Analog zur OSD Selection Policy wird die Replica Selection Policy aus einer oder mehreren Basisstrategien aus den bereits vorgestellten Kategorien bestimmt.

2.6 Tabellarischer Vergleich von XtreemFS und HDFS

Abschließend sollen in Tabelle 1 die Eigenschaften von XtreemFS und HDFS tabellarisch gegenübergestellt werden.

	XtreemFS	HDFS
Speicherort: Metadaten	MRC (+DIR)	NameNode
Speicherort: Dateiinhalt	OSD	DataNode
Verteilung	OSP und DSP	zufällig (lokal wenn möglich)
Replikation	OSP und RSP	rack-aware Strategie
POSIX-Kompatibilität	Ja	Nein

Tabelle 1: Tabellarischer Vergleich von XtreemFS und HDFS

3 Konfiguration und Optimierung von XtreemFS für Hadoop

In diesem Abschnitt soll die Konfiguration von XtreemFS für die Nutzung unter Hadoop festgelegt werden und Optimierungen an der Hadoop-Schnittstelle von XtreemFS vorgenommen werden. Dazu wird der Einfluss der Verteilung und der Replikation auf den Schreib- und Lesedurchsatz untersucht und anhand der Ergebnisse die Konfiguration vorgenommen.
Für die Verteilung wird eine OSD Selection Policy (siehe Abschnitt 2.5.2) bestimmt und die stripe-width und Blockgröße festgelegt. Der Einfluss der Blockgröße wird experimentell bestimmt.
Für die Replikation wird eine Replica Selection Policy bestimmt und der Replikationsgrad festgelegt. Wie bei der Blöckgröße soll auch der Einfluss des Replikationsgrads experimentell bestimmt werden.
Die Experimente werden mittels des DFSIO Benchmark durchgeführt. Der DFSIO Benchmark ist Teil des Apache Hadoop Projekts und ist zur Messung der Dateisystemleistung unter Hadoop entwickelt worden. Ermittelt wird die durchschnittliche IO-Rate pro Map-Task und der durchschnittliche Durchsatz pro Map-Task. Insgesamt werden N Map-Tasks ausgeführt und jede Map-Task i liest bzw. schreibt dabei eine einzelne Datei mit vordefinierter Größe in das Dateisystem und misst dabei die Dateigröße (S_i) und Ausführungsdauer (T_i). Anschließend wird eine Reduce-Task ausgeführt, welcher die Messergebnisse sammelt und das Ergebnis berechnet. Berechnet werden die durchschnittliche IO-Rate und der durchschnittlicher Durchsatz wie folgt [16]:

- durchschnittliche IO-Rate pro Map-Task $= \sum_{i=1}^{N}(S_i/T_i)/N$
- durchschnittlicher Durchsatz pro Map-Task $= \sum_{i=1}^{N} S_i / \sum_{i=1}^{N} T_i$

Der DFSIO Benchmark wird für alle Experimente auf 17 Knoten innerhalb eines Clusters ausgeführt, wobei 16 Knoten als Slave-Knoten genutzt werden und ein Knoten als Master-Knoten. Alle Knoten haben die gleiche Hardwarespezifikation und nutzen das gleiche Betriebssystem. Die Knoten sind mit einem 1 GBit Switch miteinander verbunden. In Tabelle 2 ist die Spezifikation der Knoten angegeben.

OS	64-Bit Linux, Fedora 12
CPU	64-Bit, 8-Core CPU mit je 2.3 GHz
HDD	1 Terabyte
RAM	8 Gigabyte
Netzwerk	1 GBit/s LAN

Tabelle 2: Spezifikation der Knoten

Auf jedem Slave-Knoten läuft ein TaskTracker und OSD bzw. DataNode und auf dem Master Knoten läuft ein MRC und DIR bzw. NameNode und Secondary NameNode. Je Slave-Knoten werden acht 2 GB große Dateien gelesen bzw. geschrieben, so dass insgesamt 256 GB Daten gelesen bzw. geschrieben werden.

3.1 Einfluss der Verteilung und Blockgröße auf die Datenverarbeitung

Um die Auswirkungen der Konfiguration der Verteilung auf den Durchsatz bestimmen zu können, soll zunächst nur die Verteilung betrachtet werden und keine Replikation genutzt werden. Dazu wird die Verteilung in HDFS analysiert und aus den gewonnenen Erkenntnissen die geeigneten OSD Selection Policys in XtreemFS betrachtet und eine dieser für den weiteren Vergleich festgelegt. Anschließend soll der Schreib- und Lesedurchsatz bei unterschiedlichen Blockgröße experimentell bestimmt und verglichen werden. Abschließend wird eine geeignete Blockgröße für den weiteren Vergleich festgelegt.

3.1.1 Wahl der OSD Selection Policy

Wie in Abschnitt 2.4.2 beschrieben, werden die Blöcke in HDFS zufällig verteilt, außer der schreibende Klient befindet sich auf einem DataNode des Clusters. In diesem Fall wird der Block auf dem lokalen DataNode gespeichert. Für die Nutzung von HDFS unter Hadoop ist das zufällige Verteilen und das lokale Schreiben voraussetzend für die effiziente Ausführung der MR-Jobs. Daher sollen beide Schreibvorgänge näher betrachtet werden.
Zunächst betrachten wir das Kopieren der Eingabedaten eines MR-Jobs nach HDFS. Angenommen der schreibende Klient befindet sich nicht auf einem DataNode folgt, wie beschrieben, dass die Blöcke zufällig im Cluster verteilt werden. Unter der Annahme, dass um ein vielfaches mehr Blöcke geschrieben werden als Knoten vorhanden sind, ist die Verteilung gleichmäßig. Diese gleichmäßige Verteilung der Blöcke begünstigt die Task-Plazierung beim Ausführen eines MR-Job, da jeder Knoten ähnliche viele Blöcke hält und so die Map-Tasks gleichmäßig verteilt werden können.
Im Folgenden soll das Schreiben auf HDFS innerhalb eines MR-Jobs betrachtet werden. Dabei wird angenommen, dass auf jedem Knoten ein Dat-

aNode und TaskTracker gemeinsam laufen. Wird eine Datei innerhalb eines MR-Jobs geschrieben, geschieht dies durch die Map- und Reduce-Tasks auf den entsprechenden Knoten. Da sich der schreibende Klient, also die Map- bzw. Reduce-Task, auf einem DataNode befindet wird die Datei lokal auf den DataNode geschrieben. Das lokale Schreiben führt zu einem hohen Datendurchsatz, da die Blöcke nicht über das Netzwerk geschrieben werden, sondern direkt in das lokale Dateisystem. Dies beschleunigt die Ausführung der Map- und Reduce-Tasks und somit auch des gesamten MR-Jobs. Eine Verteilung der Blöcke ist in den meisten Fällen nicht nötig, da vor allem die Ergebnisse der Reduce-Task geschrieben werden und dabei keine gleichmäßige Verteilung nötig ist. Sollten die Ergebnisse aber bspw. für einen weiteren MR-Job genutzt werden, kann die Verteilung durch das Nutzen mehrerer Reduce-Tasks umgesetzt werden. So schreibt jede Reduce-Task einen Teil des Ergebnisses auf den lokalen Knoten. Werden mindestens so viele Reduce-Tasks ausgeführt wie Knoten existieren, speichert jeder Knoten am Ende einen Teil des Ergebnis.
Da die in HDFS vorgenommene Verteilung einen guten Ansatz für den Einsatz von HDFS mit Hadoop darstellt, soll die Verteilung in XtreemFS der in HDFS angepasst werden. Dazu müssen die beiden oben beschriebenen Mechanismen umgesetzt werden, also

(a) das zufällige Verteilen der Blöcke, wenn der schreibende Klient sich nicht auf einem OSD befindet und

(b) das lokale Schreiben der Blöcke, wenn der schreibende Klient sich auf einem OSD befindet.

Das zufällige Verteilen lässt sich in XtreemFS mit der Shuffle Policy umsetzten. Diese bringt die Liste der verfügbaren OSDs in eine zufällige Reihenfolge. Die stripe-width der Default Striping Policy soll dabei gleich der Anzahl von OSDs sein, damit entsprechend große Dateien über alle Knoten im Cluster verteilt werden können. Für genügend große Dateien (Dateigröße $\geq$ stripe-size $\cdot$ stripe-width) ist die Verteilung in jedem Fall gleichmäßig, da die Datei zyklisch über alle OSDs verteilt wird.
Das lokale Schreiben kann mit der FQDN Filtering Policy umgesetzt werden. Diese Strategie sortiert die Liste der verfügbaren OSDs nach der Übereinstimmung der Domain des OSDs mit der Domain des Klienten. Wenn der Klient sich auf einem OSD befindet, steht dieser an erster Stelle der Liste. Damit die Blöcke ausschließlich auf dem lokalen OSD gespeichert werden, muss das Striping deaktiviert werden.
Für Anwendungsfälle in denen nur einer der beiden Schreibvorgänge eintritt, genügt es die jeweilige OSD Selection Policy zu wählen. Sollen jedoch beide Mechanismen unterstützt werden, muss für XtreemFS eine neue OSD Selection Policy implementiert werden, die ein ähnliches Verhalten in der Verteilung aufweist wie HDFS. Da für das weitere Vorgehen eine solche

OSD Selection Policy benötigt wird, soll diese hier beschrieben werden.

Algorithmus 2 SortHDFSPolicy als Pseudocode

```
getOSD(allOSDs, clientIP){
    allOSDs.shuffle()
(1)
    for (osd : allOSDs) {
        if (osd.getIP() == clientIP) {
            allOSDs.swapOSDs(0,allOSDs.getIndex(osd))
(2)
        }
    }
    return allOSDs
}
```

Algorithmus 2 zeigt, wie die neue OSD Selection Policy, genannt SortHDFS-Policy, die Liste aller OSDs sortiert. Der Algorithmus geht wie folgt vor:

(1) Die Liste aller OSDs wird zunächst zufällig sortiert. Dies gewährleistet die zufällige Verteilung der Blöcke.

(2) Die IP-Adresse aller OSDs wird mit der des Klienten verglichen. Sollte eine der IP-Adressen mit der des Klienten übereinstimmen, wird der erste Eintrag der Liste und der Eintrag des lokalen OSD getauscht. Wenn keine IP-Adresse mit der des Klienten übereinstimmt, bleibt die Liste zufällig sortiert.

Das Striping für die SortHDFSPolicy wird deaktiviert. Damit werden wie in HDFS die Dateien von lokalen Klienten vollständig lokal geschrieben. Allerdings führt das Deaktivieren des Stripings auch dazu, dass eine Datei bei der zufälligen Wahl des OSDs nur auf diesem gespeichert wird. Ähnlich wie bei HDFS reicht es aber aus um ein vielfaches mehr Dateien als Knoten in das Dateisystem zu schreiben. So hält jeder Knoten ähnlich viele Dateien und eine gleichmäßige Verteilung kann angenommen werden.
Der Javacode für die Implementierung des oben beschrieben Pseudocode ist auf der beigelegten CD zu finden.

3.1.2 Einfluss der Blockgröße auf den Durchsatz

Nachdem die OSD Selection Policy festgelegt und das Striping für diese deaktiviert wurde wurde, soll nun der Einfluss der Blockgröße auf den Schreib- und Lesedurchsatz untersucht werden. Dies erfolgt experimentell anhand des DFSIO Benchmarks. Der DFSIO Benchmark wird nacheinander mit

fünf verschiedenen Blockgrößen ausgeführt und die so entstanden Messwerte miteinander verglichen. Die Blockgröße mit den geeignetsten Messwerten wird für den weitere Vergleich genutzt. Entsprechend dem write-once-read-many Zugriffsmusters wird hier vor allem der Lesedurchsatz betrachtet. Als Blockgrößen werden 4MB, 8MB, 16MB, 32MB und 64MB getestet.

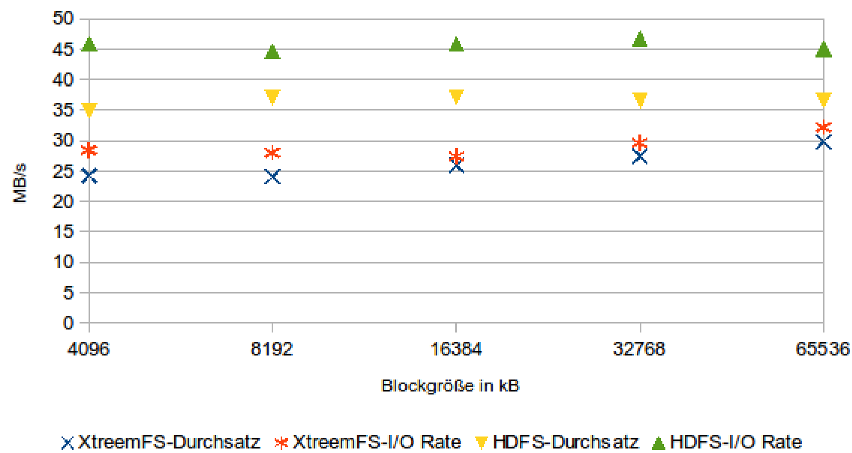

Abbildung 3: DFSIO Benchmark - Lesedurchsatz und IO-Rate für unterschiedliche Blockgrößen

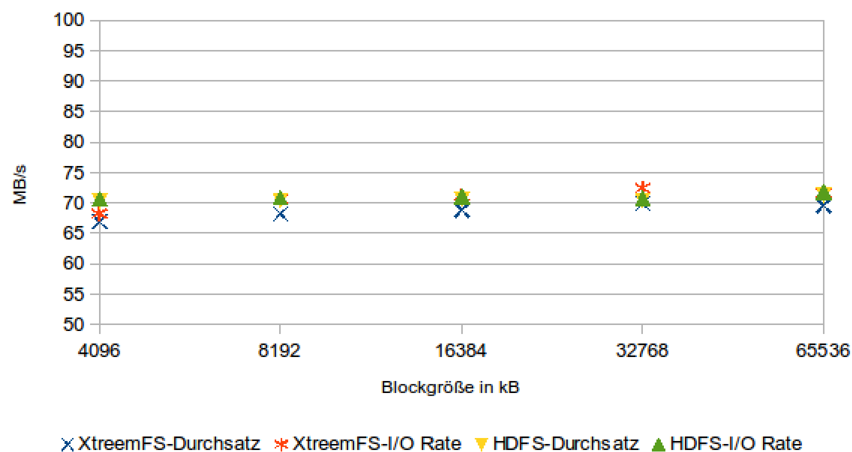

Abbildung 4: DFSIO Benchmark - Schreibdurchsatz und IO-Rate für unterschiedliche Blockgrößen

In Abbildung 3 wird der Durchsatz und die IO-Rate beim Lesen durch den

DFSIO Benchmark und in Abbildung 4 der Schreibdurchsatz und die IO-Rate beim Schreiben durch den DFSIO Benchmark für XtreemFS und HDFS dargestellt. Da pro Knoten für diesen Test eine Map-Task gleichzeitig ausgeführt führt, entsprechen die gegeben Messwerten dem Durchsatz und der I/O-Rate pro Knoten.
Aus den Messwerten geht hervor, dass XtreemFS und HDFS einen ähnlichen Schreibdurchsatz für alle Blockgrößen haben und HDFS einen besseren Lesedurchsatz als XtreemFS für alle Blockgrößen hat. Daraus lässt sich folgern, leseintensive Anwendungen mit HDFS eine bessere Ausführungszeit haben könnten. Ob dies tatsächlich eintritt wird im weiteren untersucht.
Aus beiden Abbildungen lässt sich schließen, dass für diesen Benchmark die Wahl der Blockgröße in HDFS keinen und in XtreemFS einen geringen Einfluss auf den Lese- und Schreibdurchsatz hat. Da der DFSIO Benchmark keine Berechnungen auf den Daten durchführt und je eine Task eine Datei einliest ist das Ergebniss nicht repräsentativ für alle Arten von Hadoop Anwendungen, so dass es bei anderen Anwendungen zu anderen Unterschieden beim Lese- und Schreibdurchsatz kommen kann. Daher wird für die Blockgröße für XtreemFS und HDFS der HDFS Standardwert 64 MB gewählt. Damit ist im Allgemeinen von einem gutem Durchsatz für die meisten Arten von Hadoop Anwendungen auszugehen.

3.2 Einfluss der Replikation auf die Datenverarbeitung

In diesem Abschnitt soll der Einfluss der Replikation auf den Schreib- und Lesedurchsatz von XtreemFS unter Hadoop untersucht werden. Dazu werden die beiden Replikationsarten read-only und read-write miteinander verglichen und eine der beiden für den weiteren Vergleich festgelegt. Zudem wird eine Replica Selection Policy gewählt und der Schreib- und Lesedurchsatz mit unterschiedlichen Replikationsgraden miteinander Verglichen. Die Durchsätze werden anhand des DFSIO Benchmarks ermittelt. Abschließend wird ein Replikationsgrad für den weiteren Vergleich festgelegt.

3.2.1 Wahl der Replikationsstrategie

In Abschnitt 2.5.3 wurden bereits die unterschiedlichen Replikationsarten read-only und read-write in XtreemFS beschrieben. Da für die Nutzung von XtreemFS mit Hadoop grundsätzlich beide Replikationsarten in Frage kommen und die Wahl der Replikationsart letztendlich von dem konkreten Anwendungsfall abhängt werden zunächst beide Arten der Replikation untersucht und deren Anwendungsfälle betrachtet. Anschließend wird eine der beiden Replikationsarten für den weiteren Vergleich festgelegt.
Das read-only replizierte Dateien nicht veränderbar sind führt für die Nutzung von XtreemFS unter Hadoop zu keiner Einschränkung, da in Hadoop das nachträgliche Ändern von Dateien nach dem write-once-read-many Zu-

griffsmuster nicht vorgesehen ist. Allerdings ergibt sich daraus eine Einschränkung für alle sonstigen Anwendungen, die auf den Dateien arbeiten sollen, oder Anwendungen, die für die Vor- und Nachverarbeitung der Dateien genutzt werden sollen. Der Einsatz von Anwendungen die das Verändern von Dateien fordern ist hier ausgeschlossen. Wie stark sich diese Einschränkung in der Praxis auswirkt hängt von dem konkreten Anwendungsfall ab und lässt sich nicht allgemein feststellen.

Wie in Abschnitt 2.5.3 beschrieben gibt es bei read-only Replikation volle und partielle Replikate. Unter Hadoop ist die ausschließliche Nutzung von vollen Replikaten zu bevorzugen. Das Platzieren einer Map-Task auf einem Knoten mit einem partiellen Replikat würde keinen Vorteil gegenüber der Platzierung auf einem Knoten ohne Replikat haben, da in beiden Fällen die Blöcke erst von einem anderen Knoten gelesen werden müssen bevor die Berechnung durchgeführt werden kann. Da es durch die ausschließliche Nutzung von vollen Replikaten zu einer erhöhten Netzwerklast und Replikationsdauer kommt, sollte vor dem Ausführen eines MR-Jobs auf den Dateien sichergestellt werden, dass die Replikation abgeschlossen ist.

Bei read-write replizierten Dateien führt jede Operation zu einem Datenaustausch zwischen den Replikaten. Dieser Mehraufwand hat einen negativen Einfluss auf die Ausführungszeit der Operationen und damit auch auf den Schreib- und Lesedurchsatz. Das read-write Dateien beliebig veränderbar sind, bringt bei Nutzung von XtreemFS unter Hadoop keinen Vorteil, da Hadoop in jedem Fall von unveränderbaren Dateien ausgeht.

Aus dem Umstand, dass read-write replizierte Dateien einen schlechteren Schreib- und Lesedurchsatz haben und dass für die Hadoop Anwendungen das nachträgliche Verändern von Dateien keine Relevanz hat, lässt sich für das Nutzen von XtreemFS unter Hadoop folgern, dass die read-only Replikation zu Bevorzugen ist. Da dies aber das Verändern von Dateien ausschließt und Anwendungsfälle denkbar sind in denen das Ändern von Dateien voraussetzend ist, kann auch der Einsatz der read-write Replikation nötig sein. So hängt die Wahl der Replikationsart wie eingangs erwähnt vom konkreten Anwendungsfall ab. Da im Weiteren nur die Leistung von XtreemFS unter Hadoop untersucht werden soll und somit nur Hadoop Anwendungen ausgeführt werden, wird die read-only Replikation für alle weiteren Vergleiche eingesetzt.

Für die Replica Selection Policy kann im Allgemeinen die FQDN Sort Strategie genutzt werden. Sollte ein lokales Replikat vorhanden sein, garantiert die FQDN Sort Strategie den Zugriff auf dieses. Durch den lokalen Zugriff auf die Datei wird ein optimaler Durchsatz gewährleistet.

3.2.2 Einfluss des Replikationsgrads auf den Durchsatz

In diesem Abschnitt soll für die read-only Replikation in XtreemFS der Einfluss des Replikationsgrads auf den Schreib- und Lesedurchsatz untersucht

werden. Dies geschieht über den DFSIO Benchmark. Der Replikationsgrad wird schrittweise erhöht und die Leistung bei jeder Erhöhung der Schreib- und Lesedurchsatz gemessen. Der maximale Replikationsgrad ist mit acht festgelegt, dies entspricht der Hälfte der Anzahl an Knoten. Für die Verteilung der Blöcke wird die SortHDFSPolicy genutzt und die Blockgröße mit 64 MB festgelegt.

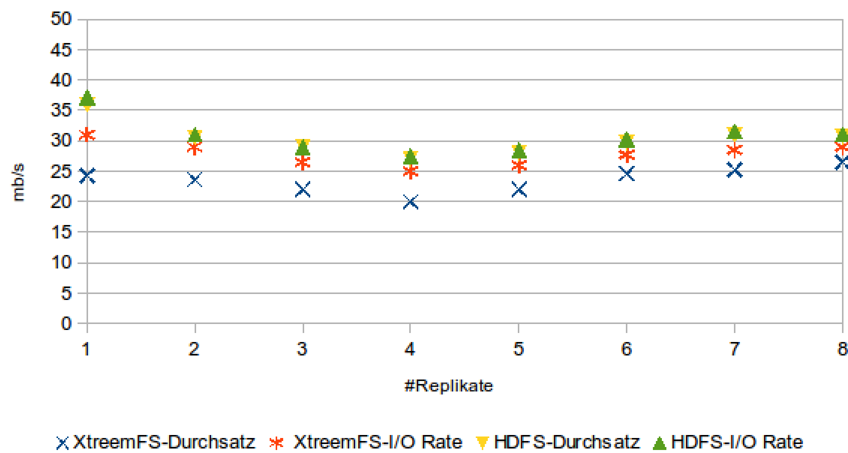

Abbildung 5: DFSIO Benchmark - Lesedurchsatz und IO-Rate für unterschiedliche Replikationsgrade

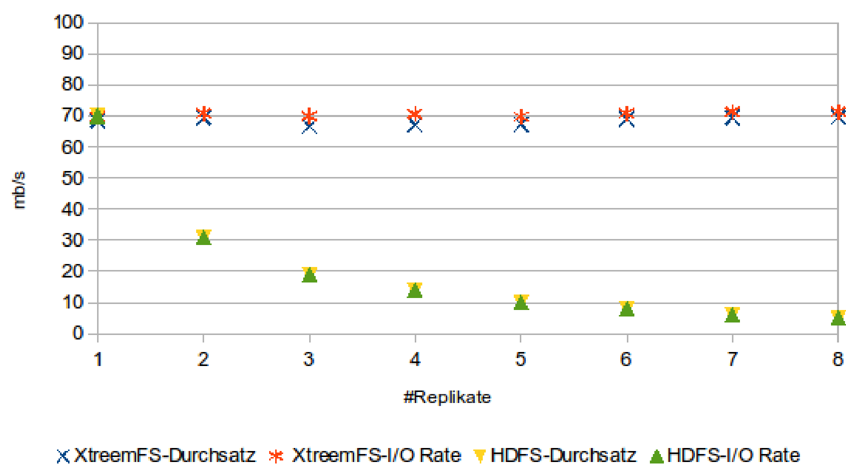

Abbildung 6: DFSIO Benchmark - Schreibdurchsatz und IO-Rate für unterschiedliche Replikationsgrade

In Abbildung 5 ist der Lesedurchsatz und die I/O-Rate beim Lesen von Dateien in XtreemFS und HDFS mit unterschiedlichem Replikationsgrad angegeben. Zu beobachten ist, dass bis zu einem Replikationsgrad von vier der Lesedurchsatz und die IO-Rate beider Dateisysteme um ca. 10 MB/s bei HDFS und und ca. 5 MB/s bei XtreemFS abnimmt. Für höhere Replikationsgrade steigt der Lesedurchsatz und die IO-Rate wieder an.
In Abbildung 6 ist der Schreibdurchsatz und die I/O-Rate beim Schreiben von Dateien in XtreemFS und HDFS mit unterschiedlichem Replikationsgrad angegeben. Dabei ist zu beobachten, dass der Replikationsgrad in HDFS im Gegensatz zu XtreemFS einen deutlichen Einfluss auf den Schreibdurchsatz und die IO-Rate hat. So fällt der Schreibdurchsatz und die IO-Rate in HDFS deutlich mit steigendem Replikationsgrad, während die Messwerte bei XtreemFS linear verlaufen.
Dies lässt sich damit begründen, dass die Dateien in HDFS bereits während des Schreibvorgangs repliziert werden. Dies wird wie in Abschnitt 2.4.3 beschrieben durch Replication Pipelining durchgeführt. Der Schreibvorgang ist erst abgeschlossen, wenn der letzte Knoten in der Pipeline den zu schreibenden Block gespeichert hat. So benötigt der Schreibvorgang mit steigendem Replikationsgrad mehr Zeit und der Durchsatz verringert sich. Das synchrone Durchführen der Replikation führt dazu, dass die Dateien bereits nach dem Schreiben vollständig repliziert und und so unmittelbar nach der Erzeugung auf den Dateien gearbeitet werden kann.
In XtreemFS beginnt die Replikation der Dateien erst nach dem Schließen der Dateien. Daher benötigt die Replikation abhängig vom Replikationsgrad und der Größe der Dateien einige Zeit bis die Dateien vollständig repliziert sind. Abhängig vom Anwendungsfall kann dies einen Einfluss auf die Ausführung von MR-Jobs haben. Sollten die Dateien beispielsweise während oder unmittelbar vor einem MR-Job erzeugt werden, kann dies einen negativen Einfluss auf die Ausführung des MR-Job haben, da Map-Tasks möglicherweise auf Knoten platziert werden, die noch keine vollständigen Replikate der entsprechenden Eingabedaten gespeichert haben und somit die Daten über einen anderen Knoten bezogen werden müssen. Liegen die Dateien bereits vor der Ausführung eines MR-Jobs vollständig repliziert im Dateisystem ergibt sich daraus kein weiterer Nachteil.
Als Folgerung aus den unterschiedlichen Replikationsansätzen wird im weiteren Vergleich auf die Replikation verzichtet. So wird sicher gestellt, dass die Messwerte vergleichbar bleiben. Für HDFS entsteht so kein Nachteil aus der synchronen Replikation und für XtreemFS muss die Replikationszeit nicht berücksichtigt werden.

3.3 Optimierungen der Hadoop-Schnittstelle von XtreemFS

Um auch kleine Schreib- und Leseanfrage effizient auszuführen verfügt HDFS über einen Lese- und Schreibpuffer. Wie in Abschnitt 2.4.2 beschrieben wer-

den Schreibanfragen klientenseitig gepuffert und gesammelt an HDFS geschickt. Beginnt ein Klient eine Datei zu lesen wird die Datei über den angefragten Teil hinaus gelesen und der gelesene Inhalt klientenseitig gepuffert. Alle weiteren Anfragen werden aus dem Puffer heraus beantwortet, bis dieser leer ist und wieder neu befüllt wird.
So wird in HDFS sichergestellt, dass auch kleine Lese- und Schreibanfragen schnell beantwortet werden können. Um dies auch für XtreemFS umzusetzen sollen der Schreib- und Lesepuffer auch in der Hadoop-Schnittstelle von XtreemFS umgesetzt werden.
Während der Schreibpuffer in HDFS die Daten für einen gesamten Block speichert, bevor diese in 4 kB großen Teilen an den entsprechenden DataNode geschickt werden, soll der Puffer in XtreemFS eine vordefinierte Anzahl an Bytes speichern. Ist der Puffer voll, wird er geleert und der Inhalt des Puffers an den entsprechenden OSD geschickt. Die Größe des Puffers für den weiteren Vergleich wird mit 1024 kB festgelegt.
Die Funktionsweise des Lesepuffer entspricht der von HDFS. So wird beim Lesezugriff auf eine Datei unabhängig von der Anzahl der angefragten Bytes eine vordefinierte Größe an Bytes von der Datei gelesen und in den Puffer geschrieben. Die aktuelle und folgende Anfragen werden aus dem Puffer heraus beantwortet, bis dieser leer ist und neu befüllt wird. Für den weiteren Vergleich wird die Größe des Lesepuffers auf 64 kB festgelegt.

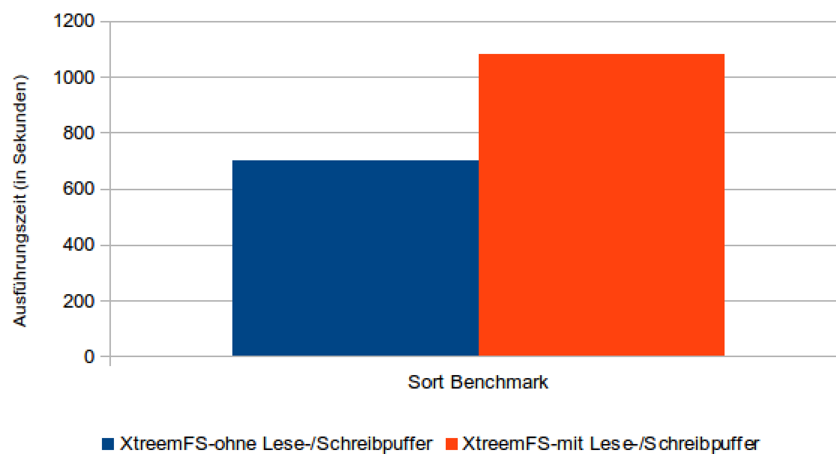

Abbildung 7: Ausführungszeiten des Sort Benchmarks mit und ohne Lese- und Schreibpuffer in XtreemFS

In Abbildung 7 sind die Ausführungszeiten des Sort Benchmarks aus Abschnitt 4 mit und ohne Lese- und Schreibpuffer in XtreemFS dargestellt. Der Sort Benchmark gehört zu den Anwendungen die viele kleine Schreib- und Leseanfragen an das Dateisystem unter Hadoop stellen und soll so als

Beispielanwendung für dieses Zugriffsmuster dienen.
Die Ausführungszeit des Sort Benchmarks unter XtreemFS mit Schreib- und Lesepuffer ist deutlich geringer, da die meisten Anfragen direkt aus dem Lese- bzw. Schreibpuffer beantwortet werden können und so nicht über das Netzwerk geschickt werden müssen.
Im weiteren soll XtreemFS nur noch mit Lese- und Schreibpuffer genutzt werden.
Die Java Implementierung der oben beschriebenen Schreib- und Lesepuffer in die Hadoopschnittstelle von XtreemFS ist auf der beigefügten CD zu finden.

4 Test von realen Anwendungen undsynthetischen Benchmarks

4.1 Anwendungen

Der Vergleich von XtreemFS und HDFS wurde mittels HiBench durchgeführt [16]. HiBench ist eine von Intel zusammengestellte Benchmark Suite für Hadoop und besteht aus verschiedenen Arten von realen Anwendungen und synthetischen Benchmarks, welche einen großen Teil der Anwendungsfälle von Hadoop abdecken sollen. Insgesamt beinhaltet HiBench acht Anwendungen, die in vier Kategorien zusammengefasst sind. Im weiteren sollen die Kategorien und deren Anwendungen vorgestellt werden [16].

Micro Benchmarks: Diese Kategorie beinhaltet die Sort, WordCount und TeraSort Anwendungen, welche auch Teil der Hadoop Distribution sind. Sort und WordCount repräsentieren eine große Teilmenge der realen MapReduce Anwendungen. Diese Anwendungen lassen sich in zwei Klassen aufteilen. Die eine Klasse der Anwendungen wandelt die Daten von einer Repräsentation in eine andere um (Sort) und die andere Klasse erzeugt aus einer großen Datenmenge, eine kleine Menge von relevanten Informationen (WordCount) [11]. Die Sort Anwendung nutzt die Sortiertung der Schlüssel-Wert-Paare des Hadoop Frameworks, so dass die Map- und Reduce-Funktionen Identitätsfunktionen sind und keine Berechnung durchführen. Die TeraSort Anwendung sortiert 1 Terabyte Daten und ermittelt die benötigte Zeit. Da sich Sort und TeraSort Anwendung nur wenig voneinander unterscheiden, soll im weiteren nur der Sort Benchmark betrachtet werden.

Internetsuchanwendungen: In dieser Kategorie befinden sich zwei typische Anwendungen aus dem Bereich der Internetsuche, die Nutch Index Anwendung und eine PageRank Implementierung. Damit repräsentiert diese Kategorie den Anwendungsfall der Verarbeitung und Indizieren von Webseiten zur optimierten Suche.
Das Nutch Indexing ist ein Indizierungssystem von Nutch [18], einer Open Source Internetsuchengine von Apache. Als Eingabe dient hier eine Menge von 2,4 Millionen Webseiten. Aus den Links innerhalb der Webseiten wird eine invertierte Indexdatei erzeugt.
Die PageRank Implementierung ist Teil eines Testfalls für das SmartFrog Framework [4] und implementiert den PageRank Algorithmus [9]. SmartFrog ist ein Open Source Framework zum Verwalten von verteilten Systemen. Der PageRank Algorithmus berechnet in einer Menge von Webseiten den Rang jeder Webseite anhand der Anzahl von eingehenden Verlinkungen auf diese. Die PageRank Implementierung iteriert mehrere Hadoop Jobs so lange bis ein gewünschter Deckungs-

grad über alle Webseiten erreicht ist. Als Eingabe dient Wikipedias page-to-page Link Datenbank [15].

Maschinelles Lernen: Diese Kategorie beinhaltet die Bayesian Classification und k-Means Clustering Algorithmen, die beide in Mahout implementiert sind. Mahout [13] ist ein Open Source Bibliothek für das maschinelle Lernen in Hadoop Anwendungen. Beide Algorithmen repräsentieren einen weiteren wichtigen Anwendungsfall von MapReduce, den des maschinellen Lernens.
Die Bayesian Classification Implementierung beinhaltet den Trainingsteil des Naiven Bayesian Algorithmus [3], einem Klassifikationsalgorithmus zur Wissensentdeckung und Data Mining. Die Implementierung besteht aus vier MR-Jobs. Als Eingabe wird eine Teilmenge aller Wikipediaeinträge [5] genutzt.
Die k-Means Clustering Anwendung implementiert k-Means [2], einen Algorithmus zu Clusteranalyse für Wissensentdeckung und Data-Mining. Als Eingabe werden zufällig generierte Beispiele genutzt, mit welchen im ersten Schritt der Schwerpunkt aller Cluster anhand eines iterativen MR-Jobs berechnet werden und im zweiten Schritt jedes Beispiel einem Cluster zugeordnet wird.

HDFS Benchmark: Diese Kategorie soll die reine Leistung von HDFS bzw. des unter Hadoop eingesetzten Dateisystems ermitteln. Dabei wird eine Erweiterung des im Abschnitt 3 genutzten DFSIO Benchmarks angewendet, welche zusätzlich zum durchschnittlichen Durchsatz und der I/O Rate den aggregierten Durchsatz ermittelt [16]. Da bereits in Abschnitt 3 auf die reine Dateisystemleistung unter Hadoop eingegangen wurde, soll diese Kategorie im weiteren nicht betrachtet werden.

4.2 Testsystem

Der Benchmark wird auf einem Testsystem mit 17 Knoten durchgeführt, wobei 16 Knoten als Slaves Knoten genutzt werden und 1 Knoten der Master ist. Auf jedem Slave läuft ein TaskTracker und OSD bzw. DataNode und auf dem Master läuft ein MRC und DIR bzw. NameNode und Secondary NameNode (siehe auch Abschnitt 3). Alle Knoten verfügen über die gleiche Spezifikation, welche in Tabelle 2 im Abschnitt 3 dargestellt wurde. Alle Knoten befinden sich in einem Rack und sind mit einem 1 GBit/s Switch miteinander verbunden. Abbildung 4 zeigt die Master/Slave-Architektur von Hadoop mit XtreemFS und HDFS Dateisysteme.

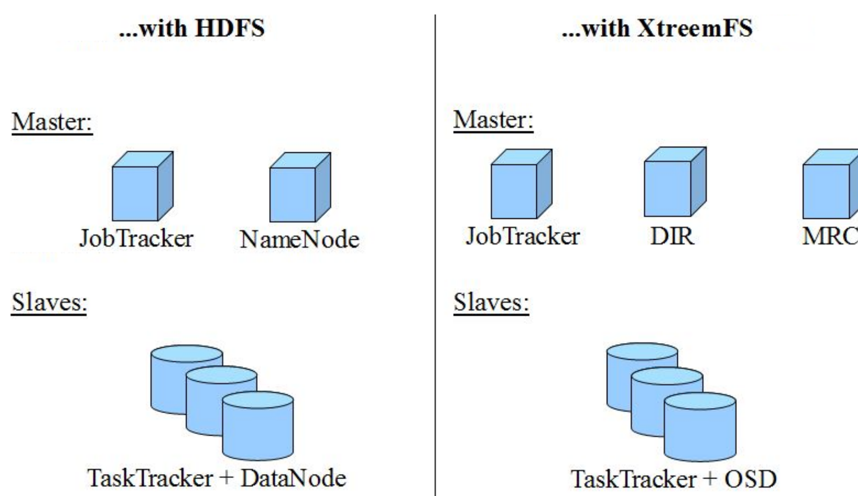

Abbildung 8: Hadoop Master/Slave-Architektur mit XtreemFS und HDFS als Dateisystem [20]

4.3 Durchführung

Im Folgenden werden die Anwendungen aus den in Abschnitt 4.1 beschriebenen Kategorien mit HDFS und XtreemFS als Dateisystem für Hadoop durchgeführt und die Ausführungszeiten miteinander verglichen. Um die Dateizugriffsmuster jeder Anwendung zu charakterisieren wird die Größe der Eingabe- und Ausgabedateien, die Anzahl der Map- und Reduce-Tasks und die durchschnittliche Laufzeit der Map- und Reduce-Tasks angeben. So lässt sich ermitteln, ob es sich um rechenintensive oder lese- bzw. schreibintensive Anwendungen handelt. Zusätzlich werden die Messergebnisse der CPU- und Festplattenauslastung während der Ausführung der einzelnen Anwendungen aus der HiBench Veröffentlichung [16] genutzt, um die Anwendungen zuzuordnen. Die Ausführungszeiten der Anwendungen werden in die Map- und Reduce-Phase unterteilt. Die Zeitmessung der Map-Phase beginnt mit dem Start des MR-Jobs und endet mit der Fertigstellung aller Map-Task. Dies schließt auch die Ausführung der Combine-Funktion ein.
Für die Reduce-Phase beginnt die Zeitmessung mit der Beendigung der Map-Phase und endet mit der Fertigstellung des MR-Jobs. Dabei ist zu beachten, dass innerhalb der Reduce-Phase vor der Ausführung der Reduce-Funktion die Daten kopiert und sortiert werden, so dass die Ausführungszeit der Reduce-Phase nicht ausschließlich von der Ausführungszeit der Reduce-Funktion abhängt. Eine schnellere Ausführung der Reduce-Tasks überträgt sich so nicht im selben Verhältnis auf die Ausführungszeit der Reduce-Phase. Die Gesamtausführungszeit errechnet sich aus der Summe den Ausführungszeiten von Map- und Reduce-Phase.
Von Rechenintensiven Anwendungen ist zu erwarten, dass die Leistung des

Dateisystems einen geringen Einfluss auf die Ausführungszeit hat, da die meiste Zeit für das Rechnen auf den Daten benötigt wird und nicht für das Schreiben und Lesen vom Dateisystem. Entsprechend ist von schreib- und leseintensiven Anwendungen zu erwarten, dass die Ausführungszeit maßgeblich von der Dateisystemleistung abhängt. Aus den Ergebnissen in Abschnitt 3 lässt sich ableiten, dass leseintensive Anwendungen in HDFS und schreibintensive Anwendungen in XtreemFS schneller ausgeführt werden. Bei Anwendungen die schreib- und leseintensiv sind sollte es zu der gleichen oder nur gering abweichenden Ausführungszeiten kommen.

4.3.1 Micro Benchmarks

Anwendung	Dateisystem	Daten in GB		ØTask-Laufzeit in Sekunden		Anzahl Tasks	
		Eingabe	Ausgabe	Map	Reduce	Map	Reduce
Sort	XtreemFS	394	394	14	772	5888	96
	HDFS			14	831		
WordCount	XtreemFS	60	1,6	27	6	7890	48
	HDFS			23	8		

Tabelle 3: Größe der Ein- und Ausgabedateien, Anzahl und durchschnittliche Ausführungszeiten der Map- und Reduce-Tasks für die Ausführung der Micro Benchmarks

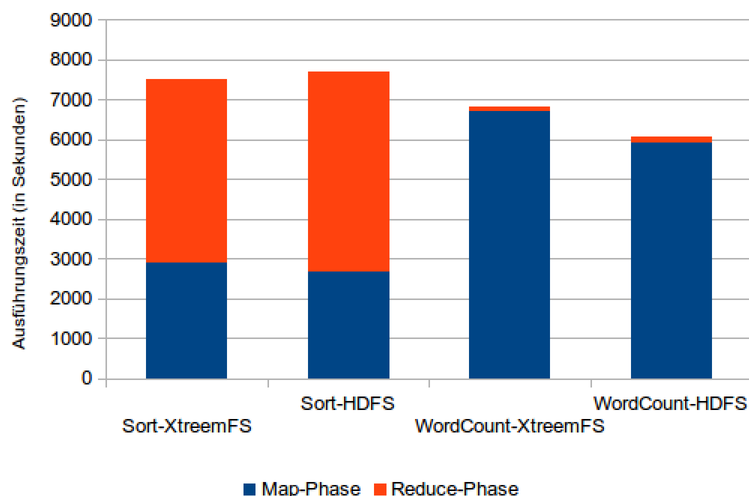

Abbildung 9: Ausführungszeiten der Micro Benchmarks

In Tabelle 3 sind die Größen der Ein- und Ausgabedateien, die Anzahl der Map- und Reduce-Tasks und die durchschnittlichen Laufzeiten der Map- und Reduce-Tasks für die Micro Benchmarks angegeben. In Abbildung 9 sind die Ausführungszeiten der Map- und Reduce-Phase der Micro Benchmarks unter XtreemFS und Hadoop dargestellt.
Sort ist ein schreib- und leseintensive Benchmark, da die gleiche Menge an Daten gelesen und ausgeben wird und die Berechnung auf den Dateien, also das Sortieren, nur einen geringen Anteil an der Ausführungsdauer hat [16]. So unterscheiden sich die Ausführungszeiten von XtreemFS und HDFS nur wenig voneinander. Trotz des größeren Lesedurchsatzes in HDFS unterscheiden sich die durchschnittlichen Ausführungszeiten der Map-Tasks in XtreemFS und HDFS nicht. So wird die Map-Phase unter beiden Dateisystemen ähnlich schnell abgeschlossen. Trotz des ähnlichen Schreibdurchsatz der beiden Dateisystem, ist die Laufzeit der Reduce-Tasks in XtreemFS geringer, was sich auch in der schnellere Ausführung der Reduce-Phase in XtreemFS wiederfindet.
Da der WordCount Benchmark aus einer großen Datenmenge relevante Informationen berechnet und so die CPU beansprucht wird, ist es ein rechenintensiver Benchmark [16].
Entgegen der Annahme wird der WordCount Benchmark als rechenintensive Anwendung nicht gleich schnell mit beiden Systemen ausgeführt. Die Map-Phase in HDFS wird schneller ausgeführt und die Reduce-Phase wird aufgrund des ähnlichen Schreibdurchsatz unter beiden Dateisystemen ähnlich schnell ausgeführt. Um die schnellere Ausführungszeit der Map-Phase in HDFS zu begründen muss die benötigte CPU-Zeit während der Map-Phase

betrachtet werden.

	benötigte CPU-Zeit in der Map-Phase in Sekunden
XtreemFS	6270
HDFS	5904

Tabelle 4: Benötigte CPU-Zeit während der Map-Phase des WordCount Benchmarks

Aus den Werten in Tabelle 4 wird deutlich, das der WordCount Benchmark unter XtreemFS mehr Zeit für die Berechnung auf den Daten benötigt. Zusammen mit dem geringen Lesedurchsatz in XtreemFS lässt sich so die längere Ausführungszeit der Map-Phase begründen.

4.3.2 Internetsuchanwendungen

Anwendung	Dateisystem	Daten in GB		ØTask-Laufzeit in Sekunden		Anzahl Tasks	
		Eingabe	Ausgabe	Map	Reduce	Map	Reduce
Nutch Indexing	XtreemFS	6	3	6	3039	480	1
	HDFS			5	3209		
PageRank-Stage1	XtreemFS	52	60	19	262	779	48
	HDFS			14	259		
PageRank-Stage2	XtreemFS	60	1,5	22	125	912	48
	HDFS			16	113		

Tabelle 5: Größe der Ein- und Ausgabedateien, Anzahl und durchschnittliche Ausführungszeiten der Map- und Reduce-Tasks für die Ausführung der Internetsuchanwendungen

In Tabelle 5 sind die Größen der Ein- und Ausgabedateien, die Anzahl der Map- und Reduce-Tasks und die durchschnittlichen Laufzeiten der Map- und Reduce-Tasks für die Internetsuchanwendungen angegeben.
In Abbildung 10 sind die Ausführungszeiten der Map- und Reduce-Phase der Nutch Indexing Anwendung unter XtreemFS und Hadoop dargestellt.
Die Map-Tasks der Nutch Indexing Anwendung lesen lediglich die Eingabedaten ein, so dass die Ausführung der Map-Tasks leseintensiv ist. Die Reduce-Phase besteht aus einer Reduce-Task, welche aus den Eingabedateien die invertierte Indexdatei erzeugt. Beim Erzeugen der invertierten Indexdatei kommt es zu einer mittleren Auslastung der CPU und durch das Schreiben in das Dateisystem zu einem hohen Schreibaufwand [16].

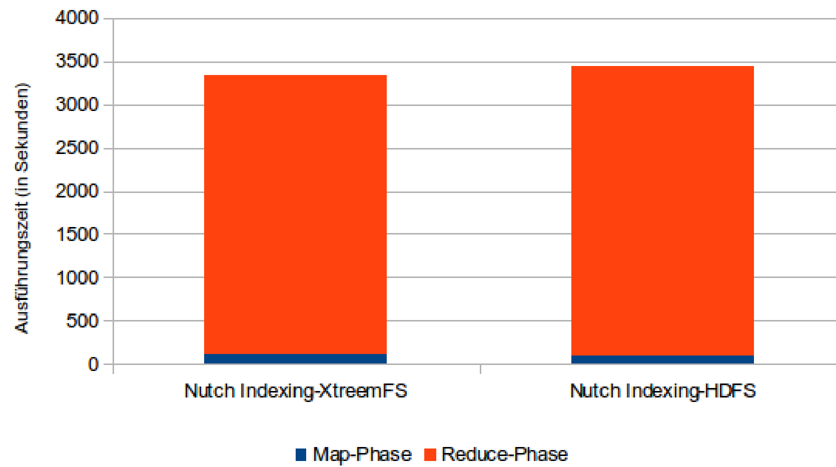

Abbildung 10: Ausführungszeiten der Nutch Indexing Anwendung

Die Ausführungszeiten der Nutch Indexing Anwendung unter XtreemFS und HDFS unterscheiden sich nur wenig voneinander. Dabei wird die Map-Phase aufgrund des höheren Lesedurchsatzes in HDFS geringfügig schneller ausgeführt. Die Reduce-Phase wird unter XtreemFS schneller abgeschlossen, da HDFS während der Reduce-Phase mehr CPU-Zeit benötigt. In Tabelle 6 sind die CPU-Zeiten während der Reduce-Phase angegeben.

	benötigte CPU-Zeit in der Reduce-Phase in Sekunden
XtreemFS	3466
HDFS	3788

Tabelle 6: Benötigte CPU-Zeit während der Reduce-Phase der Nutch Indexing Anwendung

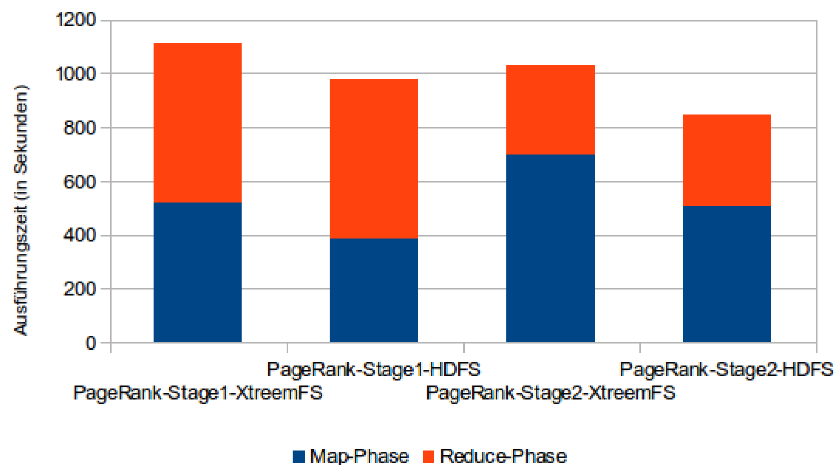

Abbildung 11: Ausführungszeiten der Internetsuchanwendungen

In Abbildung 11 sind die Ausführungszeiten der Map- und Reduce-Phase der PageRank Anwendung unter XtreemFS und Hadoop dargestellt.
Die PageRank Anwendung errechnet ähnlich wie der Wordcount Benchmark aus einer Eingabemenge relevante Informationen, so dass auch hier vor allem die CPU beansprucht wird. Dabei wird der gleiche MR-Job mehrfach hintereinander ausgeführt. In diesem Test wurde der MR-Job zweimal ausgeführt. Wie bei dem WordCount Benchmark sind entgegen der Erwartung die Ausführungszeiten unter XtreemFS und HDFS nicht gleich. Durch den ähnlichen Schreibdurchsatz beider Dateisysteme werden die Reduce-Phasen beider Iteration auf beiden Dateisystem gleich schnell ausgeführt. Die Map-Phase wird bei beiden Iterationen unter HDFS schneller ausgeführt. Dies lässt sich wie bei dem WordCount Benchmark mit dem schnelleren Lesedurchsatz in HDFS und der in XtreemFS erhöhten CPU-Zeit begründen. Die benötigte CPU-Zeit der PageRank Iterationen ist in Tabelle 7 angegeben.

	benötigte CPU-Zeit in der Map-Phase in Sekunden	
	Stage1	Stage2
XtreemFS	13974	13919
HDFS	13028	12495

Tabelle 7: Benötigte CPU-Zeit während der Map-Phase der PageRank

4.3.3 Anwendungen zum maschinellen Lernen

Anwendung	Dateisystem	Daten in GB		ØTask-Laufzeit in Sekunden		Anzahl Tasks	
		Eingabe	Ausgabe	Map	Reduce	Map	Reduce
Naive Bayesian-Job1	XtreemFS	2,1	0,9	17	252	392	5
	HDFS			16	262		
k-Means-Schwerpunkt	XtreemFS	25	5	31	5	360	1
	HDFS			27	7		
k-Means-Clustering	XtreemFS	24	24	32	0	360	0
	HDFS			25	0		

Tabelle 8: Größe der Ein- und Ausgabedateien, Anzahl und durchschnittliche Ausführungszeiten der Map- und Reduce-Tasks für die Ausführung der Anwendungen zum maschinellen Lernen

In Tabelle 8 sind die Größen der Ein- und Ausgabedateien, die Anzahl der Map- und Reduce-Tasks und die durchschnittlichen Laufzeiten der Map- und Reduce-Tasks für die Anwendungen zum maschinellen Lernen angegeben.

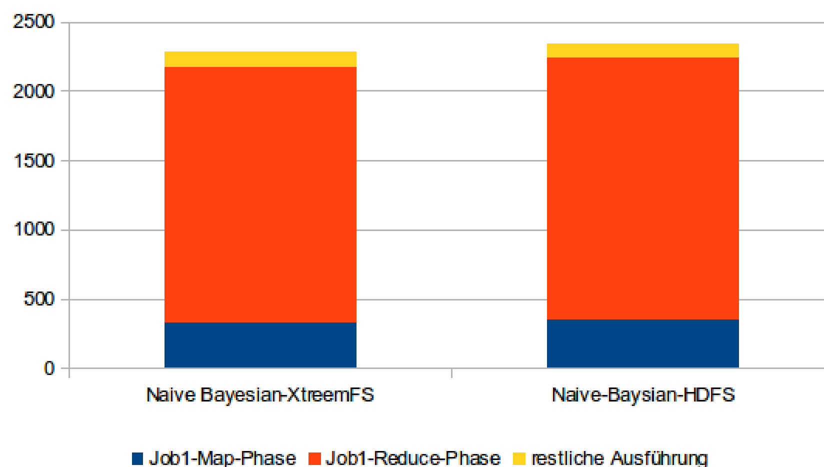

Abbildung 12: Ausführungszeiten der Naive Bayesian Anwendung

Wie in 4.1 beschrieben, besteht die Naive Bayesian Implementierung aus vier hintereinander ausgeführten MR-Jobs, welche bis auf den ersten leseintensiv

sind. Der erste MR-Job ist rechenintensiv. In der Map-Phase des ersten MR-Jobs wird eine große Menge an Zwischenergebnissen erzeugt, da diese aber in das lokale Dateisytem der Knoten geschrieben werden, beeinflusst dies nicht die Ausführungszeiten unter XtreemFS und HDFS. Wie in Abbildung 12 deutlich wird, nimmt der erste MR-Job den Großteil der Gesamtausführungszeit ein. Die restlichen MR-Jobs bestehen je aus einer Map- und Reduce-Task mit kleinen Ein- und Ausgabedaten und einer geringe Ausführungszeit, welche sich unter XtreemFS und HDFS wenig voneinander unterscheidet.
Die Ausführungszeiten des ersten MR-Jobs unterscheiden sich unter XtreemFS und HDFS wenig voneinander. Dies folgt aus rechenintensiven Berechnung während der Reduce-Phase, die unter XtreemFS und HDFS ähnlich schnell ausgeführt werden und der geringen Eingabemenge, durch die der höhere Lesedurchsatz in HDFS nur einen geringen Einfluss hat. Für größere Eingabemengen kann die Ausführungszeit der Map-Phase stärker variieren.

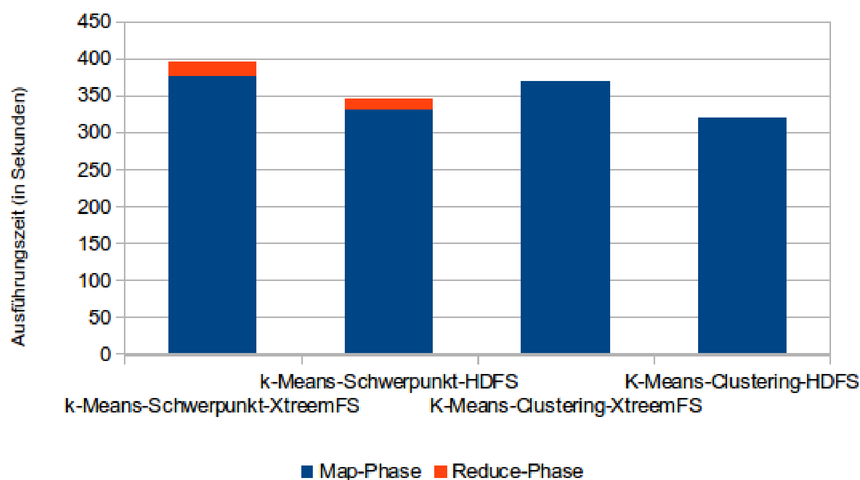

Abbildung 13: Ausführungszeiten der k-Means Anwendung

In Abbildung 13 sind die Ausführungszeiten der Map- und Reduce-Phase der k-Means Anwendung unter XtreemFS und Hadoop dargestellt.
Die Berechnung eines Clusterschwerpunkts in der k-Means Anwendung ist lese- und rechenintensiv. Insgesamt wurden fünf Clusterschwerpunkte von je einem MR-Job berechnet. Für den Vergleich wird sich nur auf einen dieser MR-Jobs bezogen.
Der erste Teil der k-Means Anwendung, also das Berechnen der Clusterschwerpunkte, wurde aufgrund des höheren Lesedurchsatzes unter HDFS schneller ausgeführt. Obwohl die Reduce-Tasks unter XtreemFS schneller

ausgeführt wurden, ist die Ausführungszeit der Reduce-Phase unter XtreemFS höher.
Der zweite Teil der Anwendung, also das Zuordnen der Testdaten in eines der berechneten Cluster, ist lese- und schreibintensiv und besteht ausschließlich aus Map-Tasks. Diese lesen die Testdaten ein und schreiben das Ergebnis zurück. Auch hier ist aufgrund des höheren Lesedurchsatzes die Ausführungszeit unter HDFS geringer.

5 Fazit

Anhand der Ergebnisse aus Abschnitt 4 lässt sich schlussfolgern, dass es möglich ist Hadoop mit XtreemFS als verteiltes Dateisystem zu nutzen, ohne einen größeren Verlust der Leistungsfähigkeit von Hadoop. Die Ausführungszeiten der getesteten Hadoop Anwendungen unter HDFS und XtreemFS unterscheiden sich je nach Anwendung wenig voneinander, wobei leseintensive Anwendungen unter HDFS schneller ausgeführt wurden. Die Ausführungszeit von rechen- und schreibintensiven Anwendungen unter XtreemFS und HDFS variierten in Abhängigkeit der konkreten Anwendung. Hier lies sich kein eindeutiges Verhalten feststellen.
In Abschnitt 3 wurde der Einfluss der Verteilung und Replikation in XtreemFS auf die Leistung von Hadoop untersucht und eine Konfiguration für XtreemFS festgelegt. Dabei wurde ein Blockgröße von 64 MB für XtreemFS festgelegt und eine für den Einsatz mit Hadoop optimierte OSD Selection Policy entwickelt. Des weiteren wurde durch das Hinzufügen von Lese- und Schreibpuffern in die Hadoop Schnittstelle von XtreemFS eine Optimierung an dieser vorgenommen.
Aufgrund der asynchrone Replikation in XtreemFS und der synchronen Replikation in HDFS wurde für den Vergleich auf die Replikation verzichtet. Für einen zukünftigen Vergleich könnte eine synchrone Replikation in XtreemFS für Nutzung mit Hadoop implementiert werden. So könnte, wie in HDFS der Schreibvorgang erst abgeschlossen werden, wenn alle vollen Replikate vollständig erzeugt wurden.
Eine weitere Optimierungsmöglichkeit wäre bei Verteilung der Blöcke in der in Abschnitt 3.1.1 definierten OSD Selection Policy möglich. Wie beschrieben werden die Dateien unabhängig von der Lage des Klienten nur auf einem OSD geschrieben und so nicht über mehrere OSDs verteilt. Dieses Verhalten bringt nur einen Vorteil wenn sich der Klient lokal auf einem OSD befindet, für nicht lokale Klienten wäre ein Verhalten wie das in HDFS zu bevorzugen. Dateien, die von nicht lokalen Klienten geschrieben werden, sollten zufällig über alle OSDs des Clusters verteilt werden. So ist eine gleichmäßigere Verteilung der Blöcke im Cluster möglich.
Die in Abschnitt 1.1 beschrieben Anwendungsfälle lassen sich umsetzen, ohne die Ausführungszeit der Hadoop Anwendungen negativ zu beeinflussen. Durch die POSIX-Kompatibilität von XtreemFS lassen sich die Daten direkt in XtreemFS durch beliebige Anwendung erzeugen und weiterverarbeiten. Das in HDFS zwingende Kopieren der Ein- und Ausgabedateien in diesen Anwendungsfällen ist unter XtreemFS nicht nötig und die Anwendungen können zeitnah zur Erzeugung der Dateien ausgeführt werden. Betrachtet man nun zusätzlich zur Ausführungszeit der Hadoop Anwendung, die Zeit, die für das Kopieren der Ein- und Ausgabedaten in HDFS nötig ist, kann sich häufig eine schnellere Gesamtausführungszeit unter XtreemFS ergeben. XtreemFS bietet durch die flexiblere Gestaltung der Verteilung zudem die

Möglichkeit die Dateien besser für eine gegebene Infrastruktur und Anwendung zu speichern. So können Daten auf Knoten in unterschiedlichen Datenzentren oder Knoten in heterogenen Systemen durch die Wahl von entsprechenden OSD Selection Policies besser platziert werden, was die Ausführung von Hadoop Anwendungen effizienter macht.
Zusammenfassend lässt sich sagen, dass es für Hadoop möglich ist ein spezialisiertes Dateisystem wie HDFS durch ein allgemein nutzbares Dateisystem wie XtreemFS auszutauschen, ohne größere Leistungseinbußen für Verarbeitung großer Datenmengen hinnehmen zu müssen.

Literatur

[1] Hadoop filesystem interface. http://hadoop.apache.org/docs/current/api/org/apache/hadoop/fs/FileSystem.html. Stand: 5.09.2013.

[2] Mahout k-means. http://cwiki.apache.org/MAHOUT/k-means.html.

[3] Mahout naive bayasien. http://cwiki.apache.org/MAHOUT/naivebayes.html.

[4] Smartfrog webseite. http://wiki.smartfrog.org/wiki/display/sf/SmartFrog+Home. Stand: 20.08.2013.

[5] Wikipedia dump. http://en.wikipedia.org/wiki/index.php?curid=68321.

[6] IEEE standard for information technology- portable operating system interface (POSIX) base specifications, issue 7. *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, pages c1–3826, 2008.

[7] Dhruba Borthakur. HDFS architecture guide. http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html. Stand: 23.April.2013.

[8] Dhruba Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

[9] P. Castagna. Having fun with pagerank and mapreduce. http://static.last.fm/johan/huguk-20090414/paolo_castagna-pagerank.pdf.

[10] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM SIGCOMM '04 Conference*, Portland, Oregon, August 2004.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[12] Apache Software Foundation. Apache hadoop project. http://hadoop.apache.org/.

[13] Apache Software Foundation. Apache mahout website. http://mahout.apache.org/.

[14] The Apache Software Foundation. HDFS user guide. http://hadoop.apache.org/docs/stable/hdfs_user_guide.html. Stand: 11.Juli.2013.

[15] H. Haselgrove. Using the wikipedia page-to-page link database. http://users.on.net/~henry/home/wikipedia.htm.

[16] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. *Data Engineering Workshops, 22nd International Conference on*, 0:41–51, 2010.

[17] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesús Malo, Jonathan Martí, and Eugenio Cesario. The XtreemFS architecture – a case for object-based file systems in Grids. *Concurr. Comput. : Pract. Exper.*, 20.

[18] Rohit Khare and Doug Cutting. Nutch: A flexible and scalable open-source web search engine. Technical report, 2004.

[19] Bjorn Kolbeck, Mikael Hogqvist, Jan Stender, and Felix Hupfeld. Flease - lease coordination without a lock server. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 978–988, Washington, DC, USA, 2011. IEEE Computer Society.

[20] Björn Kolbeck, Jan Stender, Michael Berlin, Matthias Noack, Paul Seiferth, Felix Langner, NEC HPC Europe, Felix Hupfeld, and Juan Gonzales. XtreemFS installation and user guide. http://www.xtreemfs.org/xtfs-guide-1.4/index.html, 2011. Stand: 27.Juni.2013.