Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Projektbericht im Rahmen des Projekts
„Parallelrechnerevaluation“
im Wintersemester 2013/2014

Hadoop/Hive-Benchmarks

Eingereicht von

Daniel Pajonzeck
1pajonze@informatik.uni-hamburg.de
Matrikel-Nr. 6334710

Stephan Succo
1succo@informatik.uni-hamburg.de
Matrikel-Nr. 6345119

Betreuer: Dr. Julian Kunkel
Marc Wiedemann

Hamburg, 28.04.2014

Inhalt

1. Einleitung

Seit Beginn der 2000er-Jahre finden Mehrprozessorsysteme (Parallelrechner) zunehmende Verbreitung und sind heute Bestandteil nahezu jedes modernen Rechnersystems. Die parallele Rechenarchitektur ist heute dominierend und deckt ein breites Spektrum an Einsatzmöglichkeiten, angefangen bei mobilen Endgeräten wie Smartphones oder Tablets bin hin zu Hochleistungsrechnern, ab.

Die Technik ermöglicht durch Parallelisierung von Recheneinheiten wie Prozessoren, Prozessorkernen oder ganzen Rechnersystemen, die Rechenleistung zu steigern und zu skalieren. Überdies ist sie nicht durch die physikalischen Grenzen eines klassischen Scale-Ups beschränkt. Im Zuge der Nutzbarmachung paralleler Systeme und vor dem Hintergrund stetig wachsender Datenmengen sowie Leistungsanforderungen an Rechnersysteme hielt Parallelismus auch in der Softwareentwicklung in Form von parallelen Programmen bzw. Algorithmen vielerorts Einzug. Insbesondere im Bereich des Hochleistungsrechnens (High Performance Computing, HPC) sind Parallelrechner längst De-facto-Standard und ermöglichen dort die Berechnung komplexer Probleme wie bspw. physikalische Berechnungen, Klimabestimmung oder Proteindesign in angemessener (akzeptabler) Zeit. Parallele Systeme und Programme sind jedoch häufig komplexer als Monoprozessorsysteme und sequentielle Programme, da sie einerseits synchronisiert werden müssen und andererseits über eine ungleich größere Anzahl von Komponenten verfügen können, was die Suche nach Fehlern oder Bottlenecks[1] erschweren und die Leistungsanalyse verkomplizieren kann.

Der vorliegende Projektbericht ist im Rahmen des Projekts „Parallelrechner-evaluation“ entstanden. Ziel dieses Projekts ist die Evaluation des parallelen Rechenframeworks und verteilten Dateisystems „Hadoop“ im Zusammenspiel mit dem Data Warehouse „Hive“ auf einem Entwicklungscluster. Beide Systeme wurden entwickelt, um große Datenmengen durch effizienten Parallelismus verarbeiten, d.h. berechnen, speichern und abrufen zu können. Die Evaluation soll u.a. folgende, zentrale Fragen beantworten:

- Wie gut skaliert Hive (Weak Scaling)?
- Welche Datenübertragungsraten liefert Hive?
- Wie groß ist der Leistungszuwachs, wenn alle Daten im Speicher liegen?

[1] Engstelle (Flaschenhals), die sich negativ auf die Leistung des gesamten Systems auswirkt

2. Grundlagen

Nachfolgend soll ein Überblick über die evaluierten Systeme gegeben werden. Eine besondere Bedeutung kommt dabei Hadoop zu, da das Data Warehouse Hive auf dessen Komponenten HDFS (verteiltes Dateisystem) und YARN (MapReduce-Implementation) aufsetzt, die in einer Master-Slave-Architektur realisiert sind.

2.1. Hadoop

Das Hadoop Projekt [Hadoop] von Apache entwickelt Open-Source-Software für zuverlässiges, skalierbares verteiltes Rechnen *(distributed computing)*.

Ursprünglich wurde Hadoop von Michael J. Cafarella und Doug Cutting [Caf14] als verteilte Plattform für eine quelloffene Suchmaschine[2] entwickelt und durch das proprietäre Google File System [GGL03] und MapReduce (vgl. 2.1.2) inspiriert. Seit 2008 ist Hadoop Top-Level-Projekt der Apache Foundation und wird heute von zahlreichen, großen Unternehmen wie bspw. Amazon, Adobe oder Facebook eingesetzt.[3] Darüber hinaus sind für Hadoop viele Erweiterungen verfügbar, u.a. die NoSQL-Datenbank HBase[4] und das Data-Warehouse Hive (vgl. 2.2).

Bei Hadoop handelt es sich um ein in Java geschriebenes Software-Framework, das die verteilte Verarbeitung großer Datenmengen auf Computerclustern mittels einfacher Programmiermodelle ermöglichen soll. Ein zentrales Merkmal ist seine hohe Skalierbarkeit von einzelnen bis hin zu tausenden Maschinen, von denen jede die Möglichkeit lokaler Daten-Berechnungen und –Speicherung bietet.

Die Hadoop-Bibliothek ist in der Lage, Fehler in der Anwendungsschicht selbst zu erkennen und zu behandeln, so dass Hochverfügbarkeit nicht zwangsläufig durch die zugrundeliegende Hardware gewährleistet werden muss *(cluster awareness)*. Dementsprechend schränken Teilausfälle des Systems nicht seine Verfügbarkeit ein und Rechnerknoten können - wie in Clustern üblich - aus fehleranfälligen, kostengünstigen handelsüblichen Komponenten bestehen *(commodity hardware)*. Streng genommen ist Hadoop jedoch erst seit der Version 2.0.0 ein hochverfügbarer Dienst, da der NameNode (vgl. 2.1.1.1) zuvor einen Single Point of Failure darstellte.

[2] http://nutch.apache.org
[3] http://wiki.apache.org/hadoop/PoweredBy/
[4] http://hbase.apache.org

Das Hadoop-Framework besteht aus folgenden Modulen:

- Hadoop Common: Gemeinsame Dienstprogramme zur Unterstützung der anderen Hadoop-Module.
- Hadoop Distributed File System *(HDFS)*: Ein verteiltes Dateisystem (vgl. 2.1.1).
- Hadoop YARN: Ein *(Sub-)*Framework für Job-Scheduling und Ressourcen-Management des Clusters.
- Hadoop MapReduce: Eine auf YARN basierende Implementierung des MapReduce-Algorithmus zur Verarbeitung großer Datenmengen (vgl. a. 2.1.2).

2.1.1. Hadoop Distributed File System (HDFS)

HDFS ist ein verteiltes Dateisystem und besitzt eine Master/Slave-Architektur: Ein NameNode resp. Master koordiniert die Distribution der Daten und eine Menge von DataNodes *(Slaves)* persistiert sie. Aus Anwendungssicht lassen sich Dateien wie in einem lokalen Dateisystem speichern und abrufen, intern werden sie jedoch in Blöcke aufgeteilt und physikalisch *(über mehrere DataNodes)* verteilt.

2.1.1.1. NameNode

Der NameNode *(„Namensknoten“)* ist das Herzstück des verteilten Dateisystems und organisiert die Verteilung der Daten über die Knoten. Ein solcher Knoten kann redundant in einem HDFS existieren und beinhaltet neben der Verzeichnisstruktur eine Zuordnungstabelle für die Daten und ihre *(physikalischen)* Speicherorte. Für jegliche Dateioperationen findet zunächst eine Kommunikation mit dem NameNode statt, der bei gültigen Anfragen an das Dateisystem eine Liste jener DataNodes (vgl. 2.1.1.2) zurückliefert, auf denen sich die betroffenen Daten befinden.

Das *„HDFS High Availability Feature“* ermöglicht seit der Hadoop-Version 2.0.0 eine Active/Passive-Konfiguration des NameNodes, d.h. es können zwei redundante NameNodes betrieben werden, von denen sich einer im Hot-Standby befindet und im Fehlerfalle die Arbeit des *(zuvor)* aktiven Knotens übernehmen kann. Der Failover kann mit zusätzlichen Komponenten auch automatisiert werden.[5]

[5] http://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-site/HDFSHighAvailabilityWithQJM.html

Aufgrund seiner essentiellen und in früheren Versionen einmaligen Rolle war der NameNode ein Single Point of Failure: Fiel er aus, war das Dateisystem nicht mehr verfügbar. In älteren Hadoop-Clustern gab es lediglich die Möglichkeit, Sicherungspunkte des Namensraumes durch einen sog. SecondaryNameNode anlegen zu lassen, hierbei handelt es sich jedoch nicht um eine echte Redundanz des NameNodes.[6]

2.1.1.2. DataNode

DataNodes speichern die eigentlichen Daten, sind üblicherweise vielfach in einem Hadoop-Cluster vorhanden und verbinden sich nach einmaliger Konfiguration automatisch mit dem NameNode. Anwendungen können direkt auf DataNodes zugreifen, nachdem der NameNode den Speichort der Daten bereitgestellt hat.

Ist *„Block Replication“* aktiviert, kommunizieren die DataNodes auch untereinander und replizieren Datei-Blöcke auf verschiedene Knoten, um bspw. die Verfügbarkeit der Daten zu erhöhen. Ein DataNode benötigt daher keine RAID-Konfiguration o.ä., da HDFS diese Funktionalität ohnehin bereitstellt. Die Anzahl der gewünschten Kopien ist konfigurierbar und per Default 3, für unsere Evaluation benötigen wir jedoch keine Replikation und verwenden somit den Faktor 1 (vgl. a 5.1).[7]

2.1.2. MapReduce

MapReduce [DeGh04] ist ein von Google im Jahre 2004 eingeführtes Programmiermodell für parallele Berechnungen großer Datenmengen auf Computerclustern. Das Verfahren ist durch die Funktionen höherer Ordnung *map*[8] und *reduce*[9] der funktionalen Programmierung inspiriert worden und ermöglicht eine effiziente nebenläufige Berechnung großer verteilter Datenmengen in Clustern. Funktionale Sprachen können besonders gut parallelisiert werden, da ihre Funktionen keine Seiteneffekte aufweisen und die Reihenfolge ihrer Ausführung somit unerheblich ist.

Üblicherweise ist MapReduce Bestandteil eines Frameworks, welches darüber hinaus als fundamentalen Bestandteil ein verteiltes Dateisystem für die Ein-/Ausgabe der Daten aufweist. Apaches Hadoop (vgl. 2.1) mit seinem verteilten Dateisystem HDFS ist heute die verbreitetste Open-Source-Implementation von MapReduce.

---

[6] Für Absatz 1 und 3, vgl. http://wiki.apache.org/hadoop/NameNode
[7] Für Punkt 2.1.1.2 vgl. http://wiki.apache.org/hadoop/DataNode
[8] map bildet eine Funktion auf alle Elemente einer Liste ab
[9] reduce aggregiert (faltet) eine Liste

Anfragen in Hive (vgl. 2.2) werden intern zu MapReduce-Aufgaben *(Jobs)* kompiliert und als solche von Hadoop ausgeführt.

Das Verfahren läuft nebenläufig in zwei Phasen ab. Zunächst müssen beide Funktionen durch den Anwender oder eine Anwendung *(bspw. Hive-Compiler)* bestimmt und so die Logik des zu berechnenden Algorithmus definiert werden, während technische Aspekte, wie die Verteilung/Speicherung der Daten, weitgehend durch das Framework *(Dateisystem)* übernommen werden.

In der ersten Phase werden die Eingabedaten als Paare aus einem Schlüssel und einer Menge von Werten *(z.B. Liste oder Vektor)* auf verschiedene Prozesse verteilt, die dann nebenläufig eine *map*-Funktion auf alle Elemente ihrer Menge anwenden und lokal eine Liste mit Zwischenergebnissen, bestehend aus Schlüssel-Wert-Paaren, ablegen. Anschließend wird parallel auf jedes Zwischenergebnis eine *reduce*-Funktion angewendet, die die darin enthaltenen Paare akkumuliert und auf einen Ausgabewert reduziert.

Hadoops MapReduce-Engine besteht aus zwei Komponenten und ist wie HDFS in einer Master/Slave-Architektur realisiert: Einem JobTracker-Service *(Master)*, der MapReduce-Jobs entgegennimmt, und TaskTrackern *(Slaves)*, die Map- und Reduce-Prozesse ausführen. Mit Einführung von Hadoop 2.0.0 erfuhr diese Systematik eine Überarbeitung (vgl. 2.1.2.3), das grundlegende MapReduce-Konzept blieb jedoch weitgehend unverändert [Yarn]. Die nachfolgende Darlegung soll die allgemeine Arbeitsweise des MapReduce-Verfahrens in einem Hadoop-Cluster erläutern. Punkt 2.1.2.3 stellt einige spezifische Änderungen der neuen Version 2.0.0 *(„YARN“)* vor.

2.1.2.1. JobTracker

Der JobTracker ist ein Dienst, der MapReduce-Aufgaben einer Anwendung entgegennimmt und an spezielle Knoten *(TaskTracker)* im Cluster delegiert. Er fungiert als Koordinator des MapReduce-Verfahrens. Da ihm wie auch dem NameNode des HDFS eine zentrale Rolle inhärent ist, sollte er im besten Falle auf einem separaten Knoten des Clusters betrieben werden, um kapazitative Engpässe zu vermeiden *(Bottleneck)*.

Erhält der JobTracker eine neue Aufgabe, kommuniziert er zunächst mit dem NameNode, um die benötigten Daten zu lokalisieren. Anschließend werden verfügbare TaskTracker-Knoten (vgl. 2.1.2.2) ermittelt und jene ausgewählt, die sich *(idealerweise)* in Datennähe befinden, d.h. entweder selbst die Daten vorhalten oder sich zumindest im selben Rack wie der Knoten mit den benötigten Daten befinden. An die so ausgewählten Knoten wird dann die eigentliche Arbeit in Form von Map- bzw. Reduce-Aufgaben delegiert.

Der JobTracker überwacht den gesamten Prozess mittels Heartbeats und kann z.B. bei Ausfall eines TaskTrackers die jeweilige Aufgabe an einen anderen Knoten neu vergeben. Überdies ist er in der Lage, unzuverlässige Knoten[10] für zukünftige Aufgaben zu sperren. Ist die Aufgabe abgeschlossen, aktualisiert der JobTracker-Service seinen Status, der durch die Anwendung abgefragt werden kann.[11]

2.1.2.2. TaskTracker

TaskTracker sind Knoten, die Map- und Reduce-Aufgaben ausführen. Sie besitzen ein begrenztes Kontingent zuteilbarer Aufgaben *(sog. „Slots“)*, welches durch Hadoop standardmäßig anhand des für MapReduce auf der jeweiligen Maschine reservierten Arbeitsspeichers festgelegt wird, bei Bedarf jedoch individualisiert werden kann. Üblicherweise befinden sich TaskTracker auf DataNodes, da eine größtmögliche Nähe zu den Daten kostspielige Umkopiervorgänge und Netzwerkkommunikation vermeidet.

Für jede erhaltene Aufgabe wird eine separate JVM[12] erzeugt, um im Fehlerfalle einen Absturz des Systems zu vermeiden. Der TaskTracker überwacht seine JVM-Prozesse durch Erfassung ihrer Exit-Codes und informiert den JobTracker *(ResourceManager)* nach Abschluss der Aufgabe – unabhängig von deren Erfolg oder Misserfolg – über das Ergebnis. Zusätzlich werden periodisch Heartbeats an den JobTracker gesandt, um ihn über die Erreichbarkeit und die Anzahl verfügbarer Slots zu unterrichten.[13]

[10] Knoten mit inadäquater Verfügbarkeit, hohen Antwortzeiten, Fehlern etc.
[11] Für Punkt 2.1.2.1 vgl. http://wiki.apache.org/hadoop/JobTracker
[12] Java Virtual Maschine
[13] Für Punkt 2.1.2.2 vgl. http://wiki.apache.org/hadoop/TaskTracker

2.1.2.3. YARN

Wie oben bereits erwähnt, wurden die Hauptfunktionen des JobTrackers *(Ressourcen- und Job-Cycle-Management)* mit der Einführung von MapReduce 2.0 *(YARN)* in getrennte Komponenten aufgeteilt, um die MapReduce-spezifische Logik von der Ressourcenverwaltung zu entkoppeln und so eine gemeinsame, dynamische Ressourcennutzung mit anderen parallelen Frameworks zu ermöglichen. Seit der Hadoop-Version 2.0.0 übernehmen „ResourceManager“, „ApplicationMaster“ und „HistoryServer“ die Funktionalität des obsoleten JobTrackers.[14]

ResourceManager

Der ResourceManager ist ein YARN-Service, der die globale Zuteilung von Rechenressourcen an sog. Applications *(MapReduce-Jobs bzw. Anwendungen)* verwaltet und plant. Der Service nimmt Anwendungen eines Clients entgegen und bringt sie auf dem Cluster zur Ausführung.

ApplicationMaster

Als Application wird ein einzelner MapReduce-Job oder ein gerichteter azyklischer Graph mehrerer solcher Jobs bezeichnet. Jede Anwendung wird von einem Master gekapselt, der als ihr Scheduler und Koordinator fungiert.

Der ApplicationMaster übernimmt die Ressourcenverhandlung mit dem ResourceManager, führt Jobs aus und überwacht ihren Status und Fortschritt. Ein ApplicationMaster terminiert, sobald seine Application terminiert.

HistoryServer

Der HistoryServer ist ein Daemon, der Informationen über fertiggestellte MapReduce-Jobs speichert und bereitstellt.

NodeManager

Im Zuge der YARN-Entwicklung erfuhr auch der TaskTracker eine *(terminologische)* Anpassung und wird fortan als „NodeManager“ bezeichnet. Analog zum TaskTracker ist der NodeManager für die knotenlokale Ressourcenverwaltung und die Ausführung von Containern zuständig, in denen sich Applications befinden.

[14] Für die Absätze unter Punkt 2.1.2.3, vgl. http://hadoop.apache.org/docs/r2.2.0/ und [Yarn]

2.2. Hive

Hive [Hive] ist ein Data Warehouse für das Hadoop-Framework und erleichtert die Verwaltung, Aggregation und Analyse großer Datenmengen auf verteilten Speichern *(HDFS)*. Hive wurde ursprünglich von Facebook *(für Hadoop)* entwickelt und veröffentlicht, ist heute jedoch wie Hadoop selbst ein Top-Level-Projekt der Apache Software Foundation.

Hive bietet die Möglichkeit, verteilte Daten zu strukturieren und Metadaten auf Rohdaten *(Dateien)* abzubilden, die dann mittels HiveQL, einer auf SQL basierenden Anfragesprache, abgefragt werden können. HiveQL-Abfragen werden durch Indizierung beschleunigt, von einem Compiler in MapReduce Jobs umgesetzt und anschließend an Hadoop zur Ausführung übergeben.

Das Data Warehouse wurde nicht für OLTP[15] entwickelt und bietet keine Echtzeit-Queries. Es findet idealerweise bei Abfragen auf großen Mengen unveränderlicher Daten Verwendung: Da Hadoop MapReduce-Jobs als Stapel verarbeitet *(batch processing)*, haben derartige Abfragen in Hive selbst bei kleinen Datensätzen typischerweise eine hohe Latenz von bis zu einigen Minuten und weisen durch das zugrundeliegende Job-Scheduling einen gewissen Overhead auf.

Mit Hive sollen insbesondere die Datenanalyse und das Data Mining auf einem Hadoop-Cluster vereinfacht werden, da der Anwender keine MapReduce-Anfragen selbst formulieren können muss und stattdessen auf vertraute SQL-Syntax zurückgreifen kann. HiveQL setzt den SQL-92-Standard jedoch nicht vollständig um. Ferner besitzt Hive eine Schnittstelle für individuelle MapReduce-Funktionen, um komplexe *(möglicherweise nicht durch HiveQL abgedeckte)* Anfragen auszuführen zu können.[16]

[15] Online Transaction Processing
[16] Für Punkt 2.2 vgl. https://cwiki.apache.org/confluence/display/Hive/Tutorial

2.3. Presto

Presto ist eine interaktive SQL-Engine für Hadoop, wurde von Facebook entwickelt und im November 2013 auf GitHub als Open Source-Projekt veröffentlicht. Es erlaubt analytische Anfragen auf Datenquellen aller Größen von Gigabyte bis Petabyte. Facebook gibt dabei an, Presto sei 10x schneller als die Kombination Hive/MapReduce.[17]

Presto ist ein verteiltes System und besteht aus einem Koordinator und mehreren s.g. Workern, die jeweils einen Node im Cluster repräsentieren. Um eine Anfrage auf die Daten abzusetzen, verbindet sich der Client mit dem Koordinator. Presto bietet hierfür einen eigenen CLI-Client an. Die Anfrage wird vom Koordinator entgegengenommen, in Teile zerlegt, analysiert, das Scheduling geplant und letztendlich die Aufgaben an die Worker-Nodes verteilt.[18]

3. Projektplanung und -ablauf

Nach einem allgemeinen Einführungstermin und einer ersten Besprechung mit unseren Projektbetreuern erstellen wir zunächst einen Projektplan (siehe Anlage A). Der zeitliche Umfang dieses Projektes ist auf ein Semester begrenzt und wurde zunächst in Projektphasen eingeteilt. Hier bot sich eine Gliederung in Recherche, Installation und Evaluation an.

3.1. Recherche

Die Recherche umfasste das theoretische „Vertrautmachen“ mit den zu benutzenden Systemen Hadoop, Hive und Presto, deren Kopplung sowie mit dem Entwicklungscluster des Deutschen Klimarechenzentrums. Zudem war es nötig einen geeigneten Datengenerator für die Benchmarks zu finden. Ein Download von bereits existierenden Datenbanken kam nicht in Frage, da die meisten Quellen nur einige Tausend Datensätze beinhalteten und somit zu klein waren, um damit aussagekräftige Ergebnisse zu erzielen. Auf der anderen Seite wäre für passendere, größere Datensätze ein Download von einigen hundert Gigabytes nötig gewesen, den wir möglichst vermeiden wollten. Überdies sollte sich später zeigen, dass bei der Wahl des Generators auch die Queries von Bedeutung sind (vgl. 5.2).

[17] https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920
[18] http://prestodb.io/overview.html

3.2. Installation und Konfiguration

Nach der Informationsbeschaffung haben wir uns mit der Installation beschäftigt. Zunächst war es nötig, Hadoop zu installieren, da Hive *(und Presto)* auf die Hadoop-Infrastruktur aufbaut. Anfängliche Tests wie das Erstellen von Ordnern oder Laden von lokalen Daten in das Dateisystem liefen zu diesem Zeitpunkt bereits erfolgreich. Zum Testen der Funktionalität des MapReduce-Algorithmus‘ und um die Möglichkeiten von Hadoop besser einschätzen zu können, haben wir ein kleines Programm in Anlehnung an [Amr12] geschrieben, welches das Vorkommen verschiedener Wörter in Dateien *(die sich im verteilten Dateisystem HDFS befinden)* zählen und auswerten kann (siehe Anlage B). Auch diese Prüfung verlief positiv und somit war der Hadoop-Cluster bereit zur Erweiterung mit Hive.

Die Erstellung einer endgültigen Konfiguration war jedoch eine Herausforderung, da wir die aktuellste Version des Hadoop-Frameworks eingesetzt haben, der Großteil der Informationsquellen im Internet sich jedoch auf ältere Versionen bezieht und die Änderungen zwischen den verschiedenen Hadoop-Versionen teilweise nicht unerheblich sind. Dies wird auch bei der Betrachtung des offiziellen Hadoop-Wikis erkennbar, da selbst dort veraltete Hinweise und Beschreibungen anzutreffen sind.

Die Installation und Konfiguration haben wir zunächst in einer virtuellen Umgebung mittels Virtualisierungssoftware umgesetzt, um auf dem Entwicklungscluster des DKRZ keine überflüssige Ressourcennutzung zu beanspruchen. Die nötigen Einzelschritte, um den Cluster aufzusetzen, haben wir notiert und daraus ein Konfigurationsscript (vgl. 4.1) zur Automatisierung der Installation angefertigt, welches inkrementell[19] entwickelt wurde und so eine unterschätzte Größe angenommen hat. Dieser Aufwand hat sich jedoch bezahlt gemacht, da es im Anschluss sehr einfach und effektiv war, einen kompletten Cluster auszurollen.

Nachdem das interaktive Anlegen von Datenbanken, Schemata, sowie das Einfügen und Ausführen von HiveQL-Queries funktionierten, erachteten wir die Installation von Hive und seine Verknüpfung mit Hadoop in der virtuellen Umgebung als erfolgreich und abgeschlossen.

[19] Zunächst für Hadoop und anschließend für Hive und Presto

Erste Versuche, die Programme in gleicher Weise auf dem Live-System zu installieren, führten jedoch zu einigen unerwarteten Fehlern, so dass nach wiederholter Analyse der Logfiles des Hadoop-Frameworks kleinere Änderungen der Konfiguration notwendig waren, die auch in das Konfigurationsscript einflossen.

Die Presto-Installation verlief mithilfe des Scripts ebenfalls erfolgreich, es gab jedoch Probleme bei der Ausführung der Presto-Worker, die den Cluster insgesamt nicht sehr stabil auftreten ließen. Die Fehlersuche gestaltete sich schwierig, da die Quellen von Facebook lediglich die Installation und Benutzung *(grob)* beschreiben, im Fehlerfalle aber keine Hilfestellung anbieten. Ebenso problematisch war der Rückgriff auf Quellen Dritter, da die Verbreitung von Presto bis dato nur sehr gering erscheint. Der Vergleich von Hive und Presto wäre sicherlich sehr interessant gewesen, aufgrund der zahlreichen Probleme haben wir uns letztendlich jedoch entschlossen, die geplanten Benchmarks mit Presto zurückzustellen und zunächst nicht weiter zu verfolgen, um die Evaluation nicht (zeitlich) zu gefährden.

Ungeachtet dessen liefen Hadoop und Hive nun fehlerfrei auf jenem System, auf dem auch die Benchmarks durchgeführt werden sollten, so dass wir begannen, Testdaten zu generieren. Nach ausgiebiger Recherche und vergeblichem Testen gefundener Datengeneratoren (vgl. 4.2), haben wir uns für [HTMLgen] entschieden, der nach einigen Anpassungen lauffähig war. Insgesamt mussten ~307 GiB Daten generiert werden. Den dafür benötigten Zeitaufwand von einigen Tagen hatten wir jedoch deutlich niedriger eingeschätzt, so dass wir die Funktionsweise des quelloffenen Generators untersuchten. Es stellte sich heraus, dass einzelne Wörter mit Random generiert werden und die schlechte Performanz u.a. auf das Sammeln von Entropie zurückzuführen ist.

Ferner war es problematisch, die Generierung direkt in unserem Home-Verzeichnis durchzuführen, da der Datendurchsatz *(temporär)* sehr gering war. Die Erzeugung in /tmp auf Worker-Nodes der compute-Partition des Clusters mit anschließendem Kopieren in das Home-Verzeichnis war erheblich schneller.

Dies ist darauf zurückzuführen, dass die großen Dateien für die Benchmarks aus vielen kleinen erzeugt wurden und das /home-Verzeichnis via NFS[20] auf einem entfernten Server eingebunden ist.

[20] Network File System (NFS); Netzwerkprotokoll zum entfernten Zugriff auf Dateien

3.3. Evaluation

Da nun alle Datensätze vorhanden und die Systeme lauffähig waren, konnten wir mit dem Benchmarking der Query-Performance beginnen. Die Phase der Evaluation, welche neben der Durchführung der eigentlichen Benchmarks auch ihre Analyse und die Anfertigung der vorliegenden Dokumentation beinhaltete, gestaltete sich am umfangreichsten.

Auf dem Cluster läuft slurm[21] zur Vergabe von Ressourcen. Da wir nicht die einzigen Nutzer des Clusters waren, hätten wir auch hier mehr Zeit als die ursprünglich angenommenen 3 Tage einplanen sollen, da der Cluster zu diesem Zeitpunkt sehr gefragt war und unsere Jobs mit einer niedrigeren Priorität in die Warteschlange aufgenommen wurden.

Das Messen der Query-Ausführungszeit und die Aufzeichnung der Hardware-Performance erfolgte erneut mithilfe eines Scripts (vgl. 4.3), welches einige hundert Dateien zur Analyse erzeugte. Nach Begutachtung der ersten Ergebnisse stellten wir jedoch fest, dass der Skalierungsgrad nicht wie gewünscht ausfiel und stark von unseren Erwartungen abwich. Daraufhin begannen wir abermals, die Konfigurationsmöglichkeiten von Hadoop und Hive zu studieren. Es stellte sich heraus, dass der Fehler in der Anbindung von Hive an den YARN-Service (vgl. 2.1.2.3) lag. Nach seiner Beseitigung und einigen Tests, konnten wir die Fehlerfreiheit der Anbindung feststellen und entschieden uns, die Messreihe noch einmal komplett zu wiederholen, um „korrektere“ Werte zu erhalten.

Da wir in unserer (optimistischen) Projektplanung nur kleinere Puffer berücksichtigt hatten, mussten wir auch hier im Nachhinein feststellen, dass es ratsam gewesen wäre, Zeit für mindestens eine weitere Testreihe (sowie Queue-Wartezeiten auf dem Cluster) einzukalkulieren.

Nachdem die insgesamt 108 Messungen mit verschiedenen Cluster-Konfigurationen und verschiedenen Speicherorten des HDFS' durchgeführt waren, begannen wir im Rahmen der Projektdokumentation auch mit der Analyse der Ergebnisse.

[21] Simple Linux Utility for Resource Management

Zunächst wurden die Ausführungszeiten der Queries in einer Tabelle aggregiert, Durchschnittswerte gebildet und in einem Diagramm visualisiert. Auf gleiche Weise erzeugen wir mithilfe der Log-Files von vmstat auch die Darstellungen der CPU-Auslastung im Verlauf der Ausführungszeit, deren Anzahl sich auf 621 Dateien belief. Da sich diese Dateien nicht ohne vorherige Bereinigung (zum Erhalt der Rohdaten) in ein Tabellenkalkulationsprogramm importieren lassen, soll an dieser hervorgehoben werden, dass dieser Arbeitsschritt bei der Projektplanung als nicht unerheblicher Punkt eingeplant werden sollte. Bedauerlicherweise hatten wir diesen Aspekt gar nicht weiter berücksichtigt.

Mit Abschluss der Analyse beendeten wir auch dieses Projekt. Dem Bericht wurden alle erstellen Scripte als Anlagen angefügt (siehe 8). Der Verlauf des Projektes war durch monatliche Besprechungstermine und zwei Präsentationen der (Teil-)Ergebnisse gekennzeichnet, die durch regelmäßige Email-Korrespondenz mit unseren Betreuern ergänzt wurden.

4. Durchführung (Scripte)

Nach anfänglicher Recherche und Informationsbeschaffung begannen wir in der nächsten Phase unseres Projekts, die Installation und Konfiguration von Hadoop, Hive und Presto zunächst auf einem virtualisierten Cluster vorzunehmen, um Erfahrungen mit den Systemen sammeln zu können und nicht unnötig wertvolle Rechenzeit des Live-Systems beanspruchen zu müssen. In einer Zwischenbesprechung schlug uns Herr Dr. Kunkel vor, ein Konfigurationsscript zu erstellen, wodurch die Bereitstellung der Systeme zu einem späteren Zeitpunkt automatisiert und somit vereinfacht würde. Wir griffen diese Idee auf und erstellen ein Shell-Script zur Installation und Konfiguration von Hadoop, Hive und Presto. Darüber hinaus boten sich Scripte auch zur Datengenerierung und für die eigentlichen Benchmarks an, um einige (sich z.T. oft wiederholende) Arbeitsvorgänge zu vereinfachen. Nachfolgend sollen diese Scripte und ihre wichtigsten Funktionen näherer erläutert werden. Die Details zu unserer Cluster-Konfiguration sind in Abschnitt 5.1 zu finden.

4.1. Konfigurationsscript

Das Konfigurationsscript ist dieser Dokumentation als Anlage beigelegt (siehe Anlage C) und enthält eine Vielzahl nützlicher Funktionen. Für uns hat es sich rückblickend betrachtet unbedingt bewährt, da wir den Cluster diverse Male neu aufsetzen mussten, bis das System wie gewünscht arbeitete. Vor diesem Hintergrund wurde das Script allmählich um Funktionen erweitert, die die Einrichtung und Konfiguration des Clusters erleichterten. Mit den sondierenden Funktionen

```
./node-manager check network
./node-manager check ssh
```

lässt sich die Erreichbarkeit der Server und die Möglichkeit des entfernten Zugriffs überprüfen. Weitere Informationen wie die IP-Adresse oder der Ort der entfernten Java-Verzeichnisse können mit

```
./node-manager show cluster
```

ermittelt werden. Zum Testen verschiedener Konfigurationen des Clusters war es nötig, die Konfigurationsdateien auf die Nodes zu verteilen. Für diese Funktion kann das Script mit den Parametern

```
./node-manager put LOCAL_FILE REMOTE_TARGET [NODE]
```

gerufen werden. Die lokale Datei `LOCAL_FILE` wird dabei mit dem Ziel `REMOTE_TARGET` auf alle Nodes gepusht, sofern der vierte Parameter `NODE` nicht angegeben wurde. Wird dieser angegeben, wird die Konfigurationsdatei nur auf diesen Node übertragen. Eine weitere hilfreiche Funktion ist die Ausführung eines Befehls auf allen Nodes. Der entsprechende Befehl dafür lautet:

```
./node-manager exe COMMAND [NODE]
```

Auch hier ist es wieder möglich, nur einen Node anzugeben.

4.1.1. Hadoop-Funktionen

Als der Hadoop-Cluster erstmalig richtig konfiguriert und lauffähig war, wurden die Installationsschritte notiert und die Konfigurationen gesichert. Darauf aufbauend wurde dann die Funktion

```
./node-manager hadoop install|uninstall|start|stop|restart
```

implementiert, um den Cluster im Live-System effizient ausrollen zu können. Sie ermöglicht es, einen ganzen Hadoop-Cluster mit einem Befehl aufzusetzen und zu starten.

Die Konfiguration erfolgt mit Hilfe der Datei `node-manager.conf`. Neben allgemeinen Einstellungen können hier die Aufgaben des Hadoop-Clusters an die einzelnen Nodes delegiert werden. Dazu gehören vor allem die Angabe des NameNodes und des ResourceManagers sowie die Zuordnung der DataNodes. Weitere Zuordnungen wie die Angabe des Installationsortes auf den Nodes, sowie jene für das HDFS und die Ablage des NameNodes, sind essenziell. Ferner muss die Aufgabe des HistoryServers zugeteilt und die Replikation eingestellt werden.

Das Aufsetzen beinhaltet einen kurzen Verfügbarkeitstest der Server, einen SSH-Login-Test und überprüft, ob bereits eine Hadoop-Installation vorhanden ist. Sind die Tests erfolgreich, werden im Anschluss die Quelldateien heruntergeladen, insofern sie noch nicht existieren. Nach erfolgreichem Download werden die Installationsdateien auf alle Nodes übertragen, dort entpackt und weitere Verzeichnisse für das HDFS und den NameNode angelegt.

Konfigurationsdateien werden bei jeder Installation dynamisch erzeugt, so dass diese nicht verändert werden müssen. Die Konfiguration erfolgt lediglich über die o.g. Datei `node-manager.conf`. Die Generierung erfolgt mit Hilfe von Templates, die sich im Ordner `templates` des Scripts befinden (sofern es keine ausgelesenen Systeminformationen wie z.B. `JAVA_HOME` sind). Spezielle Platzhalter werden mit Werten der conf-Datei substituiert, so dass jeder Node in Abhängigkeit seiner Aufgabe eigene Dateien erhält. Der letzte Schritt zur Lauffähigkeit des Hadoop-Clusters ist die Formatierung des NameNodes. Der Start des Clusters kann mit

```
./node-manager hadoop start
```

erfolgen.

4.1.2. Hive-Funktionen

Die Installation von Hive erfolgt nach einem ähnlichem Schema: Wird keine existierende Installation gefunden, werden die Quelldateien heruntergeladen und zum in der conf-Datei definierten Node übertragen, entpackt und die Konfiguration erzeugt. Die Interaktion mit Hive kann direkt auf dem Node, auf dem es installiert wurde, erfolgen. Der Thrift Hive Server lässt sich mit folgendem Befehl starten:

```
./node-manager hive start
```

Es handelt sich dabei um einen optionalen Dienst, der es Remote-Clients ermöglicht, Anfragen auszuführen. Dieser Dienst ist jedoch u.a. für Presto essenziell.

4.1.3. Presto-Funktionen

Presto wird ebenfalls analog ausgerollt. Konfigurationsdateien werden auch hier dynamisch erzeugt und anhand der Rollen (Presto Coordinator/Worker) unterschiedlich verteilt und angepasst.

Auch hier kann das Konfigurationsscript (wie üblich) als zweiten Parameter `install/uninstall/start/stop` annehmen. Eine zusätzliche Funktion bietet sich mit

```
./node-manager presto status
```

Sie gibt eine Übersicht der Verfügbarkeit der Worker-Nodes und des Koordinators. Die Aufgaben werden auch hier mit Hilfe der Konfigurationsdatei `node-manager.conf` an die Nodes delegiert. Weitere Konfigurationsmöglichkeiten sind in der Datei durch kurze Kommentare über den Variablen beschrieben.

4.2. Datengenerierung

Auf der Suche nach einem geeigneten Datengenerator stellten wir fest, dass fast alle verfügbaren *(Java-basierten)* Tools nur mit älteren Hadoop-Versionen kompatibel sind. Daher haben wir uns entschieden einen Datengenerator zu benutzen, der die Daten nicht direkt in das HDFS schreibt, sondern in das lokale Dateisystem. Dies hatte einen weiteren Vorteil, da der Speicherort des HDFS nach `/tmp` bzw. `/dev/shm` zeigte. So wäre es nicht möglich gewesen, mit den gleichen Daten nach einem Systemneustart oder nach einem Neuaufsetzen des Hadoop-Clusters weiterzuarbeiten. Die Daten hätten jedes Mal neu generiert werden müssen, was zu unnötiger Rechenzeit geführt hätte. Die Daten aus dem lokalen Dateisystem können mit Hadoop durch folgenden Befehl in das HDFS geladen werden:

```
${HADOOP_PREFIX}/bin/hdfs dfs -put <LOCAL_FILE> <HDFS_LOCATION>
```

Alternativ lassen sich die Daten aber auch direkt mit Hive in HiveQL-Syntax in die Datenbank-Tabelle laden, die einen Ort im HDFS repräsentiert:

```
LOAD DATA LOCAL INPATH <LOCAL_FILE> INTO TABLE <TABLE_NAME>;
```

Unsere Testdaten und –anfragen sind jenen von Pavlo et al. entlehnt und stehen auf deren Website [PPR11] zur Verfügung. Für die Erzeugung der Testdaten verwendeten wir denselben Generator [HTMLgen], der jedoch erst Shell-kompatibel gemacht und an die Hostname-Konvention des Testclusters angepasst werden musste. Weitere Änderungen am Makefile sowie weitere kleine Anpassungen waren nötig, um den Datengenerator in unserem Cluster einsetzen zu können. Die vorgenommen Änderungen sind als Patch in Anlage D zu finden und können mit

```
patch –p0 < generator.patch
```

angewandt werden.

Die so erzeugen Datensätze modellieren Traffic-Protokolldateien eines HTTP-Servers und bestehen aus zwei Dateien *(Tabellen)*: Die erste Datei enthält ein Pageranking, dessen Einträge sich aus einer URL, einem Rang und einer Aufenthaltsdauer zusammensetzen und durch eine Pipe *(„ | “)* getrennt sind. Die zweite Datei stellt das eigentliche Protokoll dar und enthält je Eintrag 9 Attribute.

Die Tabellenschemata für die Daten sind folgende:

```
CREATE TABLE Rankings (
          pageURL VARCHAR(100) PRIMARY KEY,
          pageRank INT,
          avgDuration INT
       );

CREATE TABLE UserVisits (
          sourceIP VARCHAR(16),
          destURL VARCHAR(100),
          visitDate DATE,
          adRevenue FLOAT,
          userAgent VARCHAR(64),
          countryCode VARCHAR(3),
          languageCode VARCHAR(6),
          searchWord VARCHAR(32),
          duration INT
       );
```

Die genauen Datenmengen der verwendeten Testdaten sind Tabelle 1 zu entnehmen. Ihre geringfügigen Abweichungen *(< 0,4%)* von exakten Zweierpotenzen begründen sich in dem Umstand, dass der Generator lediglich eine bestimmte Anzahl Datensätze erzeugen kann, die jedoch minimale Größenunterschiede aufweisen können.

Dies sollte jedoch keinen Einfluss auf die Messergebnisse haben, da ohnehin stets die gleichen Testdaten Verwendung finden und das Verhältnis somit für alle Messungen gewahrt bleibt.

Da die Skalierbarkeit mittels Weak-Scaling evaluiert werden soll und dabei Cluster-Konfigurationen mit 1, 2, 4 und 8 DataNode(s) verwendet werden (vgl. 5.1), wachsen die Datensätze analog. Die Wahl der Dateigröße von ~4 GiB als Ausgangsbasis begründet sich in der limitierten Speicherkapazität des Shared Memory (vgl. 5.3.1, Abs. 3). Mit den Datensätzen von ~20 GiB *(und seinen Vielfachten)* soll in einer weiteren Evaluationsreihe näherungsweise BigData simuliert werden (vgl. 5.5.1).

Bezeichnung	Bytes	MiB	GiB	Datensätze in Mio.	
				insgesamt	je Node
4 GB	4279226642	4080,99	3,9853	33	33
8 GB	8557623303	8161,19	7,9699	66	
16 GB	17135072228	16341,3	15,9583	132	
32 GB	34297285026	32708,4	31,9418	264	
20 GB	21427660388	20435,0	19,9561	165	165
40 GB	42883813502	40897,2	39,9387	330	
80 GB	85767047709	81793,8	79,8768	660	
160 GB	171535260837	163589,0	159,7550	1320	

Tabelle 1: Anzahl und Größe der generierten Datensätze

4.3. Benchmarkscript

Da die HiveQL-Queries alle mehrmals ausgeführt werden mussten, um statistisch verwertbare Ergebnisse zu erhalten, bot sich auch hier ein Script an. Nach der Erstellung des Metastores und Laden der Daten, erfolgte die Ausführung der Queries. Vor jeder Abfrage wurde jeweils die Aufzeichnung der Hardware-Aktivitäten mit `vmstat` auf allen Nodes begonnen und mit

```
echo 3 > /proc/sys/vm/drop_caches
```

der Page-Cache, Dentries und Inodes geleert. `stdout` und `stderr` *(für die Ergebnisse und Statusinformationen)* wurden nach `/tmp` umgeleitet, da eine normale Ausgabe zu langsam ist und das Messergebnis zu sehr beeinflussen würde. Der COUNT, JOIN und GROUP BY (vgl. 5.2) wurden jeweils drei Mal ausgeführt. Nach Beendigung der Messung wurden die Messdaten aller Nodes im Home-Verzeichnis aggregiert. Das Script ist in Anlage E zu finden.

5. Leistungsanalyse

Für die Leistungsanalyse verwenden wir die Methode des Weak Scalings, d.h. die Problemgröße wird konstant zur eingesetzten Rechenleistung (hier Anzahl der verwendeten Knoten) gehalten. Konkret bedeutet dies, dass bei einer Verdoppelung der Datengröße auch die Anzahl der verwendeten DataNodes verdoppelt werden muss. Idealerweise sollte sich dabei die Abfragedauer identischer Queries innerhalb einer Testreihe nicht verändern, in diesem Fall würde man von einem sehr guten Weak Scaling bzw. hervorragender Skalierbarkeit des Systems sprechen.

Bevor die einzelnen Evaluationsreihen näher erläutert werden, folgt eine Skizzierung der Rahmenbedingungen sowie eine Darlegung der verwendeten Abfragen (Queries).

5.1. Testumgebung

Die Durchführung unserer Hadoop/Hive-Benchmarks erfolge am Arbeitsbereich *Wissenschaftliches Rechnen* der Universität Hamburg. Dieser stellte uns Rechenzeit und notwendige Ressourcen auf seinem Cluster für den Lehrbetrieb zur Verfügung. Dieser befindet sich im Deutschen Klimarechenzentrum (DKRZ) und besteht aus zehn mittels Gigabit-Ethernet[22] vernetzten Intel-Knoten. Die Hostnames der Knoten lauten west[1-10]. Jeder dieser Knoten weist folgende Hard- bzw. Softwarekonfiguration auf:

CPU:	Intel Xeon Processor X5650
	2.66 GHz, 64 Bit, 6 Cores, 12 Threads
Main Memory:	12 GB DDR3, 1333 MHz
Shared Memory:	~ 5 GB
Harddisk:	250 GB Seagate (ST3250318AS), SATA II, max. 125 MB/s
Operating System:	Ubuntu v12.04.4 (64-Bit)
Java-Version:	1.7.0_51

[22] Der Durchsatz jedes Knotens kann theoretisch maximal 117MiB/s betragen.

Die aktuelle „Stable-Version“ des Hadoop Frameworks ist 2.2.0. Sie wurde auf jedem Knoten, der Teil der Cluster-Konfiguration ist, installiert. Der Cluster besteht immer aus zwei Mastern, einem NameNode und einem ResourceManager (vgl. 2.1.2.3), und n Slaves. Apache beschreibt so einen „typischen“[23] Hadoop-Cluster und empfiehlt beide Master jeweils exklusiv auf einem Node zu installieren. Die Anzahl der DataNodes/NodeManager (Slaves) variiert je nach Benchmark. Hieraus ergibt sich auch die im Folgenden verwendete Bezeichnung „n+2“, wobei mit „+2“ die zwei Master-Nodes bezeichnet seien. Für alle Benchmarks wurde die folgende Cluster-Konfiguration verwendet:

NameNode:	west1
ResourceManager:	west2
DataNode(s):	west[3-10]

Ferner wurde die Replikation der Datenblöcke im HDFS deaktiviert, da sie sich einerseits negativ auf die I/O-Performance des Systems auswirken könnte und andererseits eine untergeordnete Rolle im HPC-Bereich spielt (Redundanz wird i.d.R. nicht benötigt). Hierfür muss der Wert der Variable `HADOOP_REPLICATION` in der Config-Datei des Konfigurationsscriptes auf 1 gesetzt werden.

Im Gegensatz zu Hadoop wird Hive nur auf einem Knoten installiert, da Hive die MapReduce-Aufgaben an Hadoop delegiert. Die Installation muss auf einem Knoten erfolgen, auf dem Hadoop lauffähig ist, da Hive Hadoop zur Ausführung benötigt. Die aktuellste stabile Version zum Projektstart war Hive 0.12, mit welcher auch die Benchmarks durchgeführt wurden. Am 21.04.2014 wurde jedoch Hive 0.13 von der Apache Hive Community veröffentlicht. Sie verspricht effizientere Berechnungen und eine erhöhte Query-Performance.[24]

Presto nutzt zur Ausführung kein MapReduce, sondern eine eigene Abfrage- und Ausführungs-Engine. Um Aufgaben verteilen zu können ist es nötig Presto auf mehreren Knoten zu installieren. Da ein Presto-Cluster aus einem Koordinator und einer variablen Anzahl Workern besteht, wäre auch hier eine Bezeichnung wie n+1 (Worker-Nodes + Koordinator) denkbar gewesen.

[23] https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/ClusterSetup.html
[24] http://hortonworks.com/blog/announcing-apache-hive-0-13-completion-stinger-initiative/

5.2. Selection Queries

Bei der Auswahl geeigneter Benchmark-Queries zogen wir zunächst naheliegende Aspekte in Betracht: Unterschiedliche Komplexität, Bezug zu realen Anwendungsszenarien sowie ihre mögliche Beanspruchung verschiedener Komponenten des Clusters (z.B. I/O, Prozessor, Netzwerk). Im Zuge unserer Recherche bezüglich geeigneter Datengeneratoren (vgl. 4.2) stellten wir jedoch fest, dass nicht jeder Generator geeignete Datensätze produzierte, um bspw. Verbundoperationen darauf durchzuführen, so dass die Auswahl des Generators letztendlich auch jene der Queries mitbestimmte. Die für unsere Benchmarks verwendeten Abfragen JOIN (vgl. 5.2.2) und GROUP BY (vgl. 5.2.3) sind Pavlo et al. [PPR09] entlehnt, da sie o.g. Kriterien erfüllen und überdies dazu passende Datensätze generiert werden können.

Zunächst sah unser Testplan auch den ursprünglichen „MapReduce Task“ von Dean und Ghemawat [DeGh04] vor, bei dem es sich um eine Mustersuche mit Wildcards handelt. Bedauerlicherweise wäre hierfür ein anderer Datenbestand nötig gewesen, dessen Generierung aufgrund von Inkompatibilitäten erfolglos blieb (vgl. 4.2).

Nachfolgend werden die in unseren Evaluationsreihen final verwendeten Queries COUNT, JOIN und GROUP BY näher erläutert.

5.2.1. COUNT (Anzahl)

COUNT stellt die einfachste Operation unserer verwendeten Benchmark-Queries dar. Die Abfrage ermittelt lediglich die Anzahl aller Datensätze in der Tabelle „UserVisits“. Um dies zu bewerkstelligen müssen alle DataNodes in der Map-Phase über ihre lokalen Tabellen (Dateien) iterieren. Anschließend werden die ermittelten Teilergebnisse addiert und somit auf ein Endergebnis reduziert.

Der Query soll die Bestimmung des maximalen Durchsatzes ermöglichen, da keine komplexen Berechnungen notwendig sind und sich die Kommunikation mit anderen Knoten auf ein Minimum beschränken sollte. Die verwendete HiveQL-Abfrage ist folgende:

```
SELECT COUNT(*)
  FROM UserVisits;
```

5.2.2. JOIN (Verbund)

Eine Verbundoperation ist ein typisches Anwendungsszenario, da sich erfahrungsgemäß in den wenigsten Fällen alle benötigten Daten in nur einer Tabelle bzw. Datei befinden. Der JOIN-Query soll einen derartigen Fall modellieren und überdies eine komplexe (rechenintensive) Aufgabe darstellen, die das System stärker belastet.

Dabei müssen zwei verschiedene Datensätze verarbeitet (gefiltert, aufsummiert und vereint) werden, um anschließend die richtigen Paare aus „Rankings“ und „UserVisits“ zu finden, die den Werten von „pageURL“ und „destinationURL“ entsprechen.

Aufgrund seiner Komplexität und der Notwendigkeit des Austauschs von Zwischenergebnissen zwischen den Knoten (Netzwerkkommunikation), erwarten wir für den JOIN-Query einen deutlich geringeren Durchsatz als für den vergleichsweise einfachen COUNT. Die für die Benchmarks verwendete Abfrage hat folgende HiveQL-Syntax (in Anlehnung an [PPR09]):

```
  SELECT AVG(pageRank) AS avgPageRank,
         SUM(adRevenue) AS totalRevenue
    FROM Rankings AS R, UserVisits AS UV
    JOIN UV ON (R.pageURL = UV.destinationURL)
   WHERE UV.visitDate
         BETWEEN Date('2000-01-15') AND Date('2000-01-22')
GROUP BY UV.sourceIPaddr;
```

5.2.3. GROUP BY (Aggregation)

Die Aggregation mit GROUP BY modelliert eine analytische Anfrage auf einer Read-Only-Tabelle, wie sie bspw. beim Data Mining verwendet wird. Die Aufgabe erfordert, dass zunächst die Attribute „sourceIPaddr“ und „adRevenue“ in der UserVisits-Tabelle gefiltert werden. Der gesamte Umsatz (adRevenue) muss für jede „sourceIPaddr“ berechnet und die lokalen Gruppen anschließend zusammengeführt werden, wodurch der Austausch von Zwischenergebnissen zu einem hervorstechenden Aspekt dieser Abfrage wird. Die verwendete HiveQL-Syntax ist folgende (in Anlehnung an [PPR09]):

```
  SELECT sourceIPaddr, SUM(adRevenue)
    FROM UserVisits
GROUP BY sourceIPaddr;
```

5.3. Evaluationsreihe 1: 4 GiB/Node (Harddisk)

Cluster-Konfiguration:	1 NameNode, 1 ResourceManager, [1, 2, 4, 8] DataNodes
Messungen insgesamt:	36 (3×3 Queries $\times$ 4 Konfigurationen)
Speicherort der Daten:	/tmp (Harddisk, exklusiv)
Datengröße je DataNode:	~4 GiB (3,98 GiB)
Datensätze je DataNode:	33 Mio.

5.3.1. Zielsetzung

Herkömmliche Festplatten stellen bereits seit einigen Jahren einen Flaschenhals in Rechnersystem dar und erreichen aktuell meist Datenübertragungsraten von ungefähr 100MB/s. Vor dem Hintergrund sich rasch entwickelnder Hardware stieg ihre E/A-Leistung vergleichsweise langsam und steht heute oftmals nicht mehr in Relation zu ihrer Speicherkapazität und dem Leistungspotenzial anderer Komponenten wie Prozessor, Speicher oder Bussystem.

Die Industrie ist seit einigen Jahren bestrebt, diesem Problem mit der Entwicklung von elektronischen Festspeichern (SSDs) entgegenzuwirken. Neben ihrer gegenüber Festplatten deutlich geringeren Speicherkapazität sind Solid State Drives in der Anschaffung bis dato jedoch auch erheblich teurer als ihre Geschwister, was insbesondere dem Konzept der kostengünstigen „commodity hardware“ (vgl. 2.1) für Rechnercluster widerspricht.

Die Knoten unseres Entwicklungsclusters verfügen jeweils über eine herkömmliche 250 GB Festplatte, die maximal 125 MB/s (ca. 119 MiB/s) liefern kann (vgl. 5.1). Mit dieser Evaluationsreihe soll eine Vergleichsbasis für die folgende geschaffen werden, um zu ermitteln, ob die Festplatten einen Flaschenhals darstellen und so die unter Punkt 1 aufgeführte Frage zu klären, wie der Leistungszuwachs ausfällt, wenn alle Daten im *(erheblich)* schnelleren Speicher liegen. Hieraus ergibt sich auch die Wahl der Datengröße von 4 GiB je Node, da im Shared Memory nur eine Kapazität von ca. 5 GB vorhanden ist (vgl. 5.1) und die Messungen in beiden Fällen auf den gleichen Daten durchgeführt werden sollen, um unverfälschte Ergebnisse zu erhalten. „/tmp“ ist der Name des Pfades, unter dem die Festplatte *(exklusiv)* in das System eingebunden ist und sei nachfolgend synonym für selbige verstanden. Analog bedeutet „shm“ Shared Memory, dessen Pfad „/dev/shm“ lautet. Der Einfachheit halber wurden die Bezeichnungen an dieser Stelle übernommen.

5.3.2. Ergebnisse

Nodes	COUNT		JOIN		GROUP BY	
	Sek.	MiB/s	Sek.	MiB/s	Sek.	MiB/s
1+2	116,6	35,1	146,3	27,9	177,9	23,0
2+2	139,9	58,4	169,1	48,3	187,2	43,6
4+2	148,3	110,3	175,4	93,2	213,8	76,5
8+2	182,7	179,0	206,1	158,7	237,9	137,5

Tabelle 2: Ergebnisse für 4 GiB/Node in tmp; Durchschnitt aus 3 Messungen je Query (gerundet)

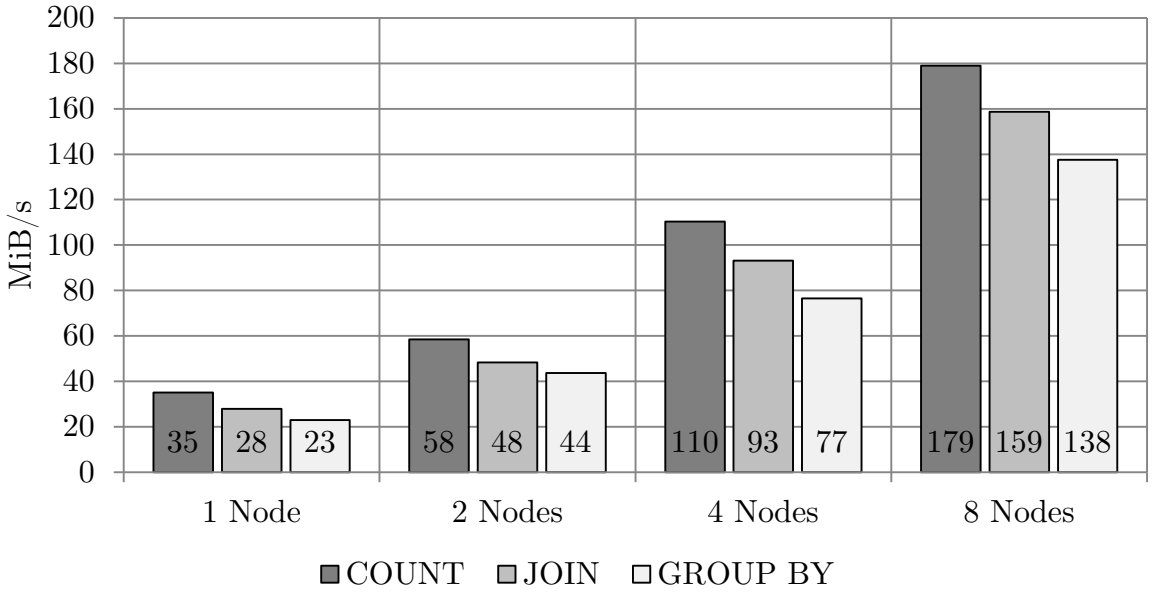

Diagramm 1: Datenübertragungsrate aller Queries bei 4 GiB/Node in tmp

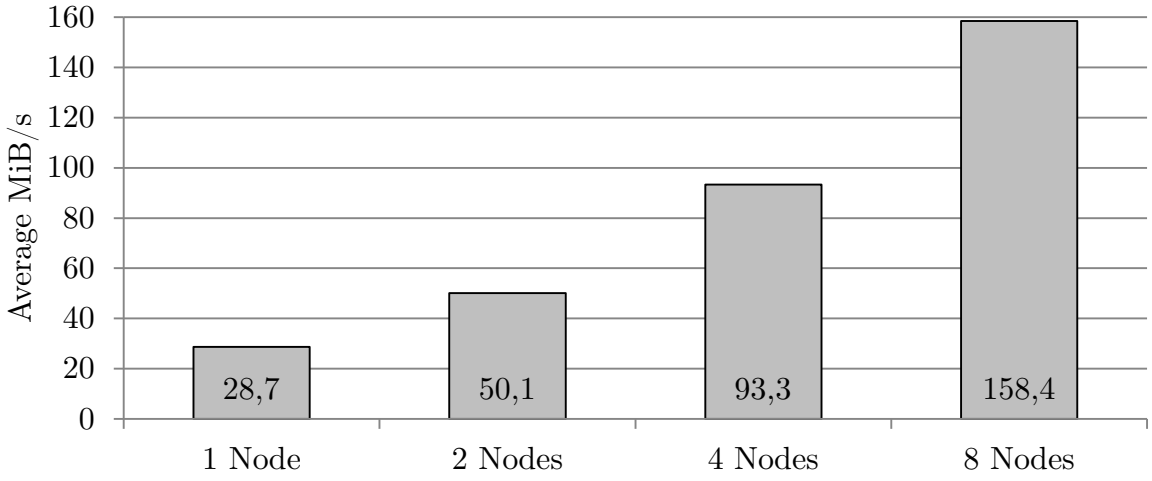

Diagramm 2: Durchschnittliche Datenübertragungsrate aller Queries bei 4 GiB/Node in tmp

5.3.2.1. COUNT (4 GiB/Node in tmp)

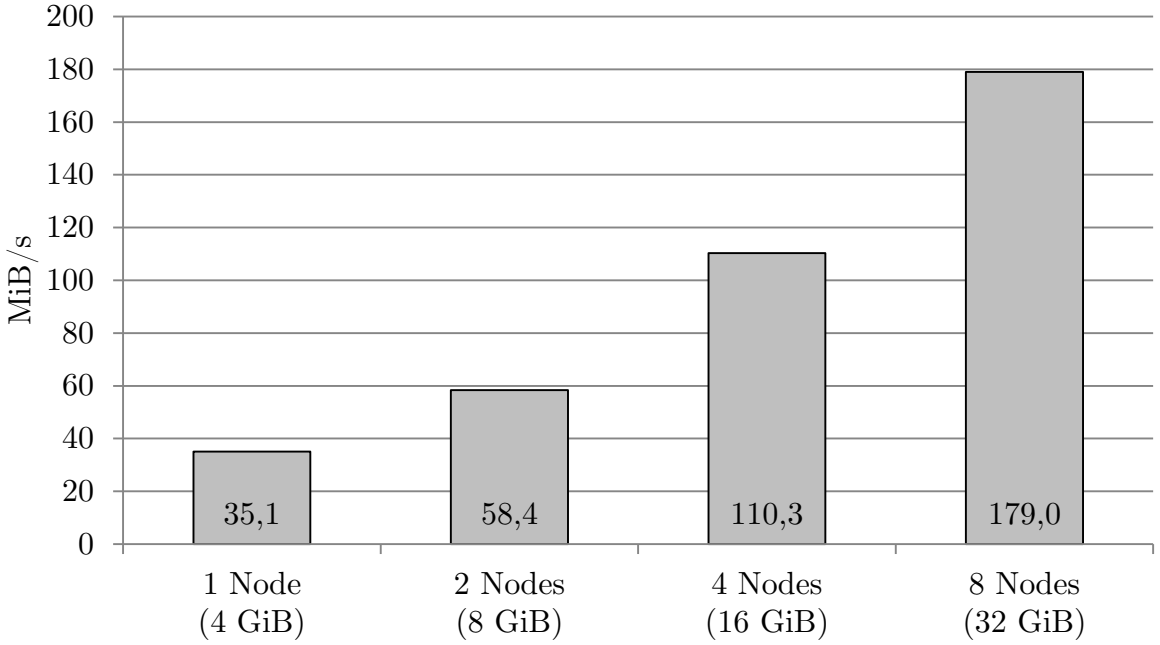

Diagramm 3: Datenübertragungsrate in Mebibytes für COUNT bei 4 GiB/Node in tmp

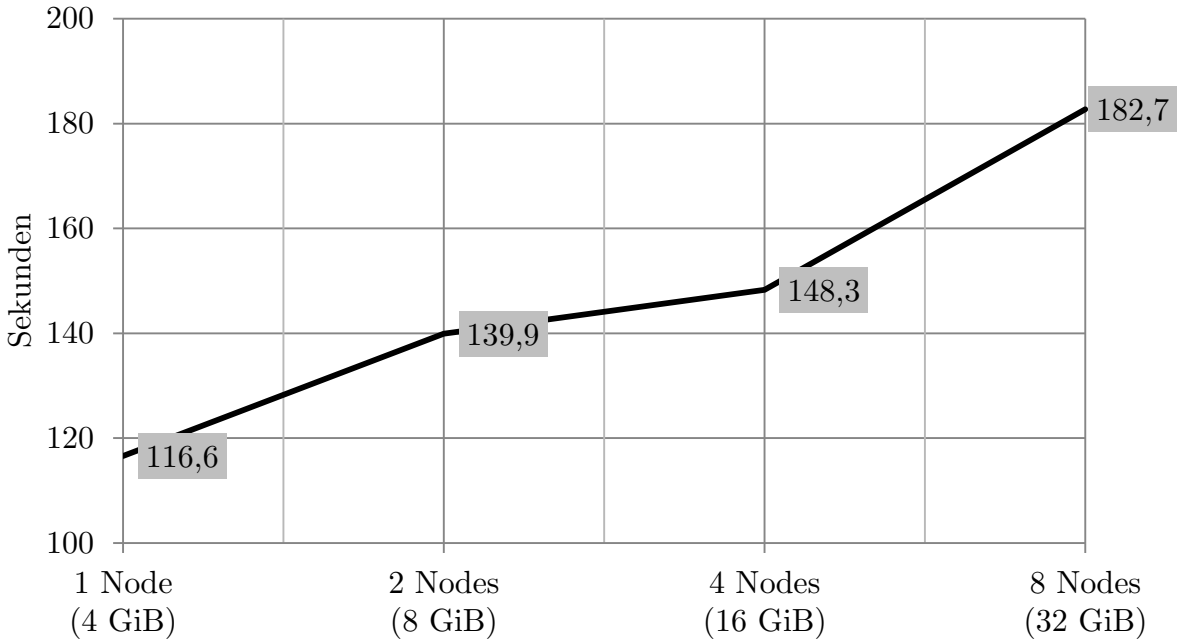

Diagramm 4: Abfragedauer in Sekunden für COUNT bei 4 GiB/Node in tmp

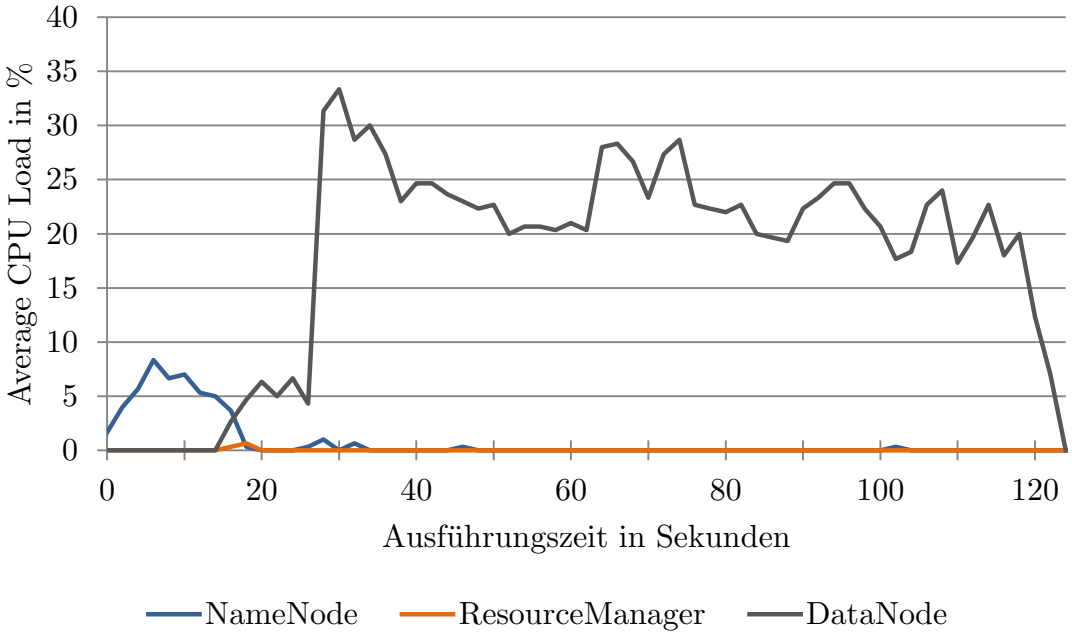

Diagramm 5: Durchschnittliche CPU-Auslastung des Clusters für COUNT bei 4 GiB in tmp auf einem DataNode (1+2)

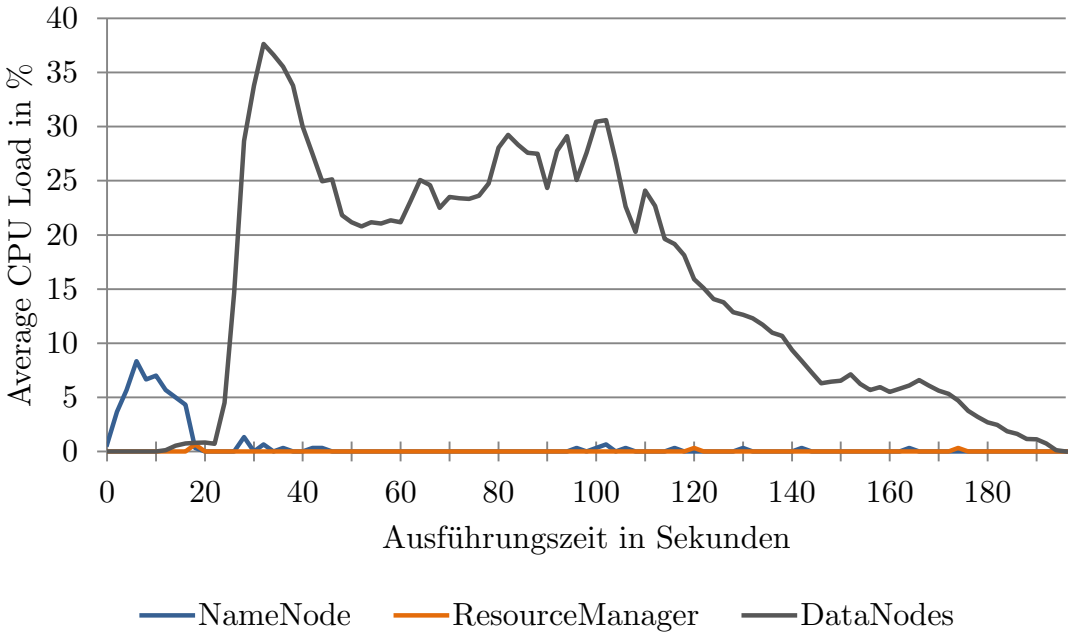

Diagramm 6: Durchschnittliche CPU-Auslastung des Clusters für COUNT bei 4 GiB in tmp je DataNode und insgesamt 8 DataNodes (8+2)

Die DataNodes-Linie in Diagramm 6 zeigt die durchschnittliche Auslastung aller eingesetzten DataNodes aus 3 Messungen. In den folgenden CPU-Diagrammen für 8 DataNodes ist die Darstellung analog.

5.3.2.2. JOIN (4 GiB/Node in tmp)

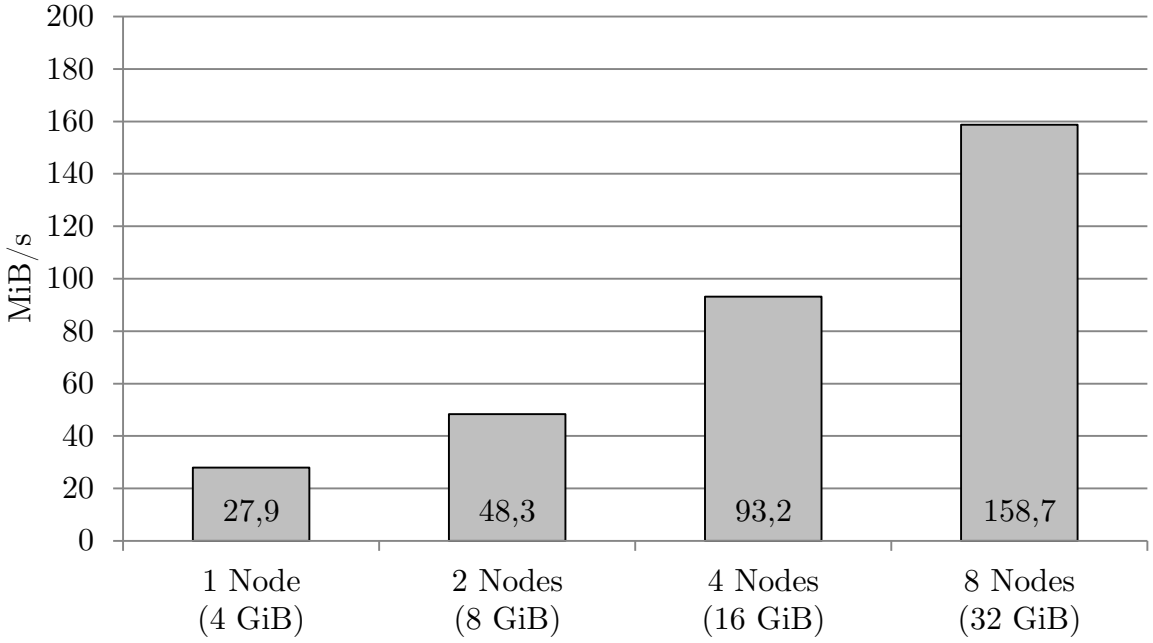

Diagramm 7: Datenübertragungsrate in Mebibytes für JOIN bei 4 GiB/Node in tmp

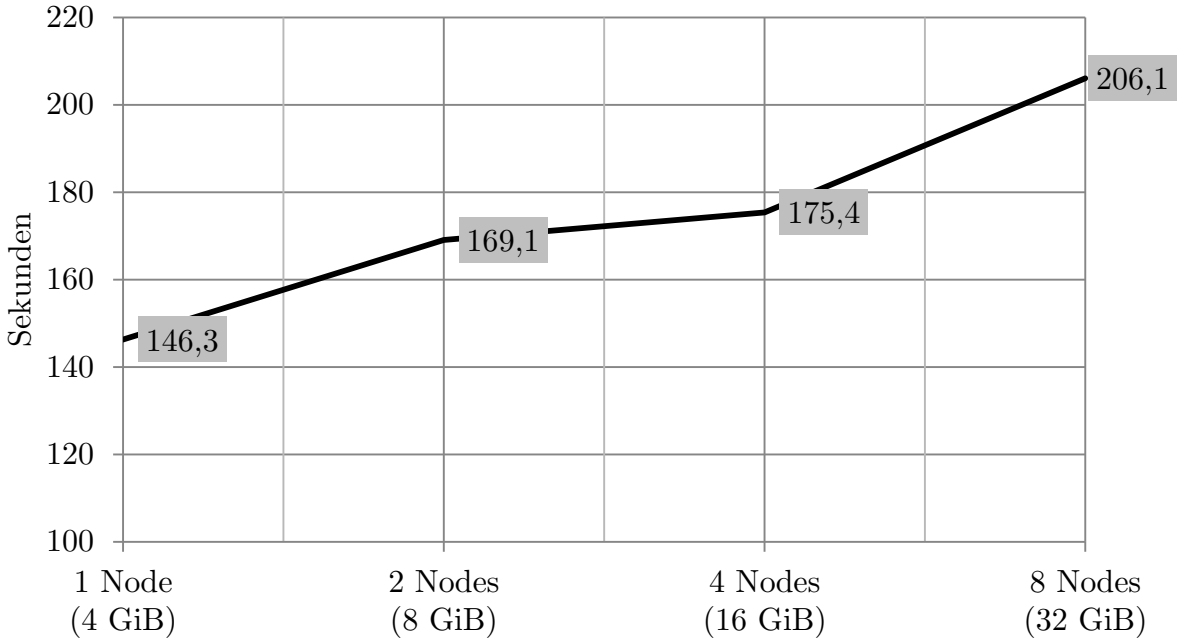

Diagramm 8: Abfragedauer in Sekunden für JOIN bei 4 GiB/Node in tmp

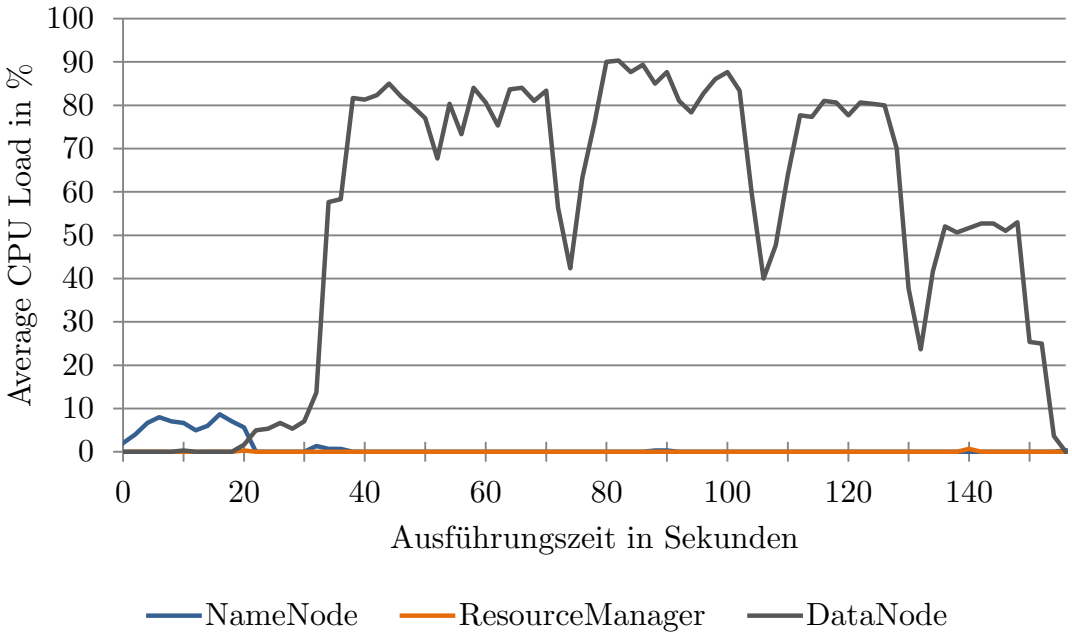

Diagramm 9: Durchschnittliche CPU-Auslastung des Clusters für JOIN bei 4 GiB in tmp auf einem DataNode (1+2)

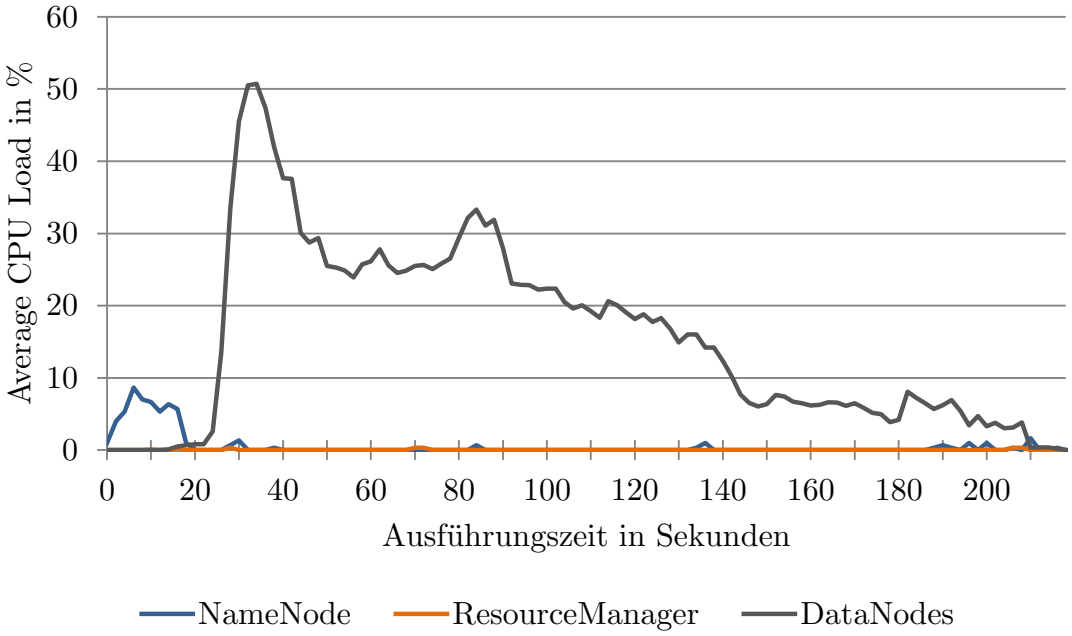

Diagramm 10: Durchschnittliche CPU-Auslastung des Clusters für JOIN bei 4 GiB in tmp je DataNode und insgesamt 8 DataNodes (8+2)

5.3.2.3. GROUP BY (4 GiB/Node in tmp)

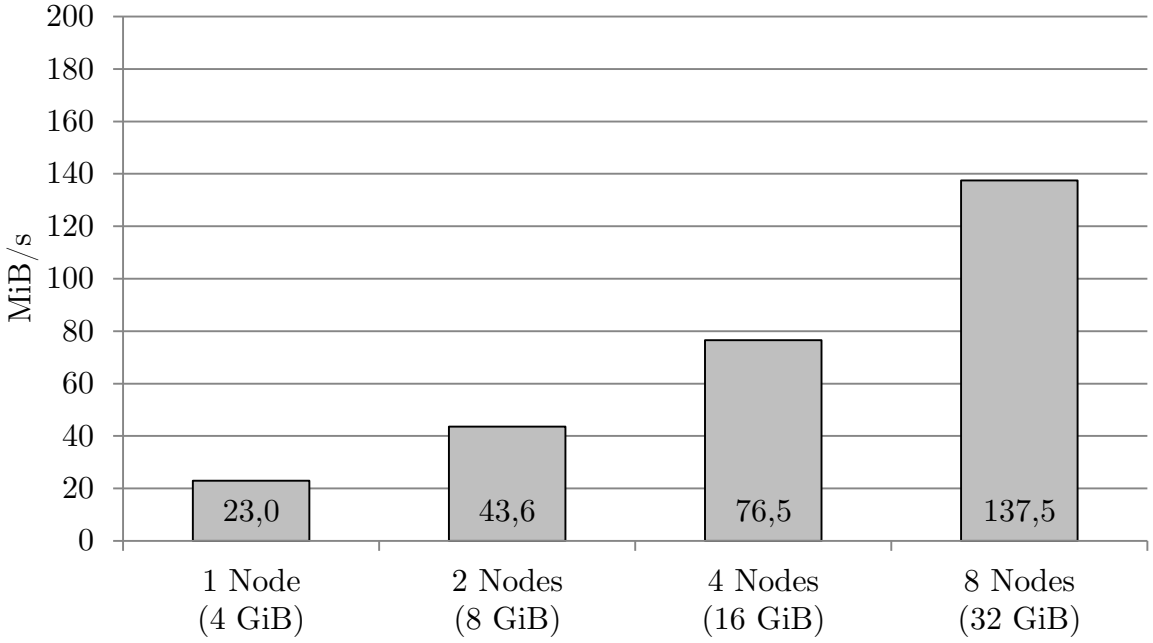

Diagramm 11: Datenübertragungsrate in Mebibytes für GROUP BY bei 4 GiB/Node in tmp

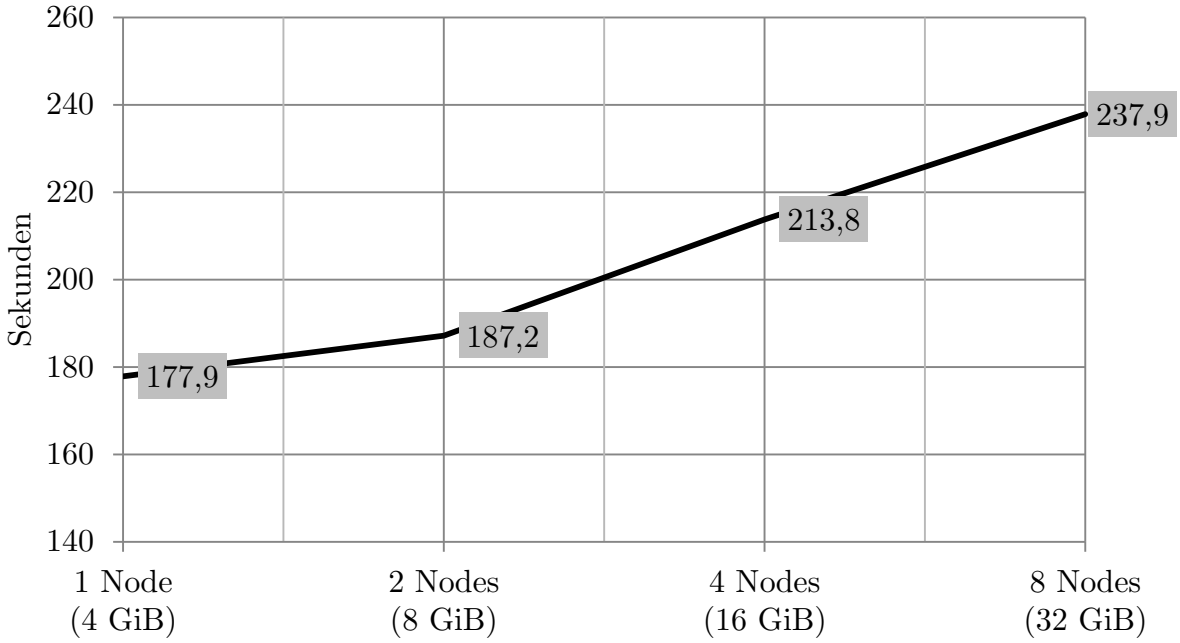

Diagramm 12: Abfragedauer in Sekunden für GROUP BY bei 4 GiB/Node in tmp

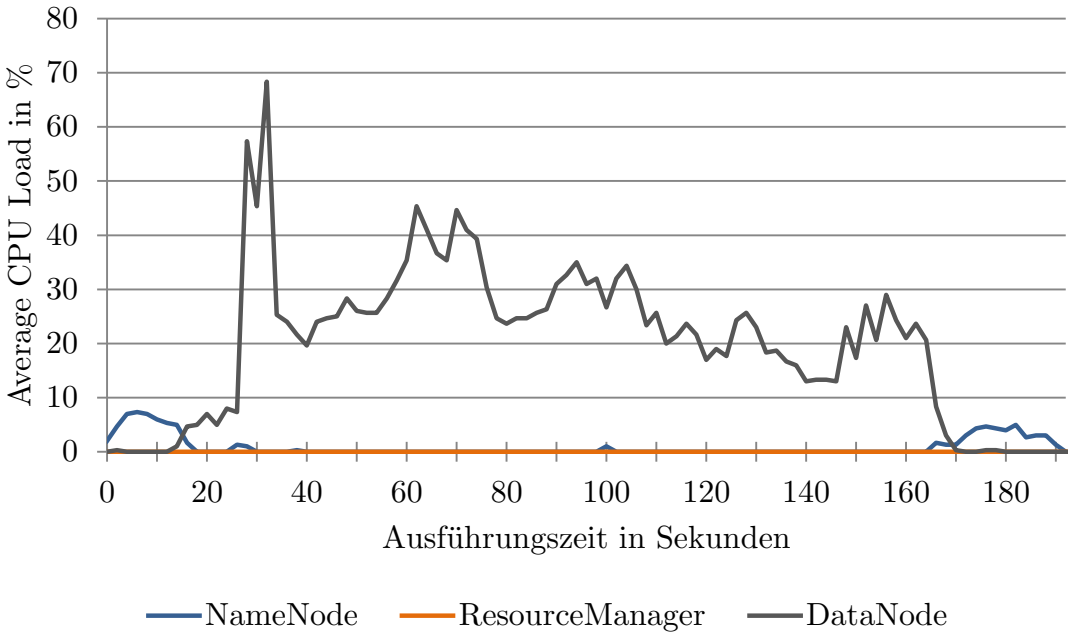

Diagramm 13: Durchschnittliche CPU-Auslastung des Clusters für GROUP BY bei 4 GiB in tmp auf einem DataNode (1+2)

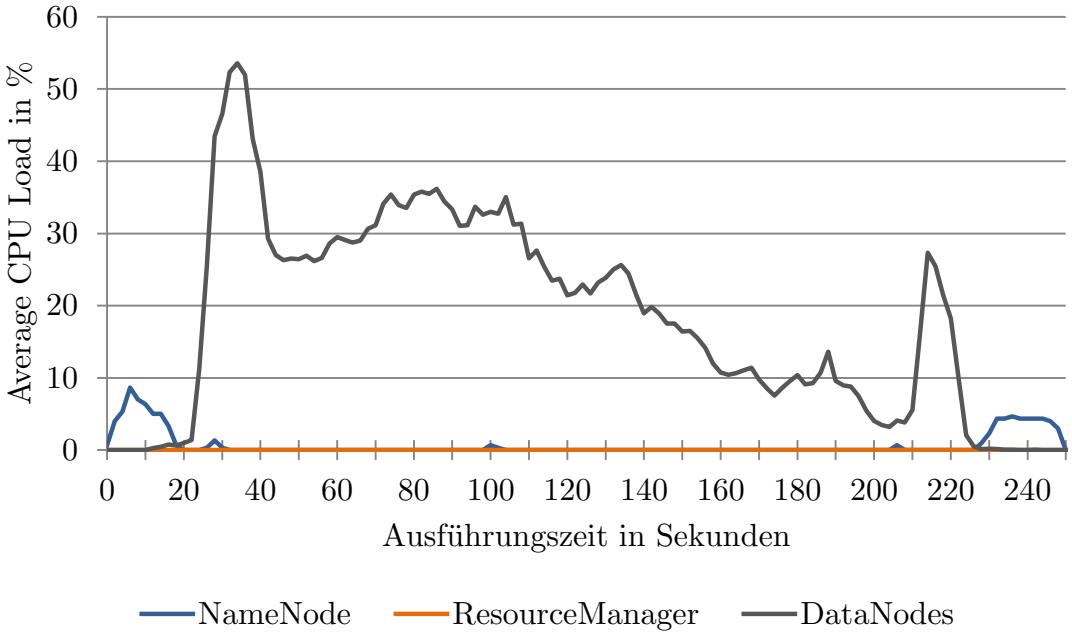

Diagramm 14: Durchschnittliche CPU-Auslastung des Clusters für GROUP BY bei 4 GiB in tmp je DataNode und insgesamt 8 DataNodes (8+2)

5.4. Evaluationsreihe 2: 4 GiB/Node (In-memory)

Cluster-Konfiguration:	1 NameNode, 1 ResourceManager, [1, 2, 4, 8] DataNodes
Messungen insgesamt:	36 (3 × 3 Queries × 4 Konfigurationen)
Speicherort der Daten:	/dev/shm (Shared Memory)
Datengröße je DataNode:	~4 GiB (3,98 GiB)
Datensätze je DataNode:	33 Mio.

5.4.1. Zielsetzung

Mit Ausnahme des Speicherorts der Daten, die hier direkt im Speicher (shm) und nicht auf einer Festplatte gehalten werden, ist Reihe 2 identisch mit der ersten Messreihe (vgl. 5.3). Zu diesem Zweck wurde das Root-Verzeichnis des HDFS auf den Pfad /dev/shm umgestellt.

Die Knoten des Clusters sind mit 12 GB DDR3 RAM bestückt, der effektiv mit 1333 MHz taktet und anteilig als Shared Memory zur Verfügung steht (vgl. 5.1). Durch seine Dual Channel Konfiguration ergibt sich für den Speicher die folgende theoretische, maximale Datenübertragungsrate:

```
(166 MHz × 64-Bit × 8 × 2)/8 = 21248 MB/s = 19,78 GiB/s
```

Gegenüber den verwendeten Festplatten verfügt der Speicher somit über eine um den Faktor 170 größere Datenübertragungsrate, die ferner auch die Netzwerkkapazitäten von 117 MiB/s je Node sowie jegliche Erwartungswerte an die Messungen weit überschreitet. Demzufolge ist sichergestellt, dass die E/A-Leistung des Speichermediums keinen Einfluss auf diese Evaluationsreihe hat. Deshalb sind auch höhere Datenübertragungsraten als in Reihe 1 zu erwarten, aus deren Vergleich ein Leistungszuwachs erkennbar werden sollte.

5.4.2. Ergebnisse

Nodes	COUNT		JOIN		GROUP BY	
	Sek.	MiB/s	Sek.	MiB/s	Sek.	MiB/s
1+2	38,0	107,5	135,6	30,1	139,4	29,3
2+2	42,7	191,7	83,0	98,4	129,6	63,0
4+2	41,8	391,6	88,2	185,4	137,0	119,4
8+2	59,0	556,2	88,8	368,7	144,0	227,1

Tabelle 3: Ergebnisse für 4 GiB/Node in shm; Durchschnitt aus 3 Messungen je Query (gerundet)

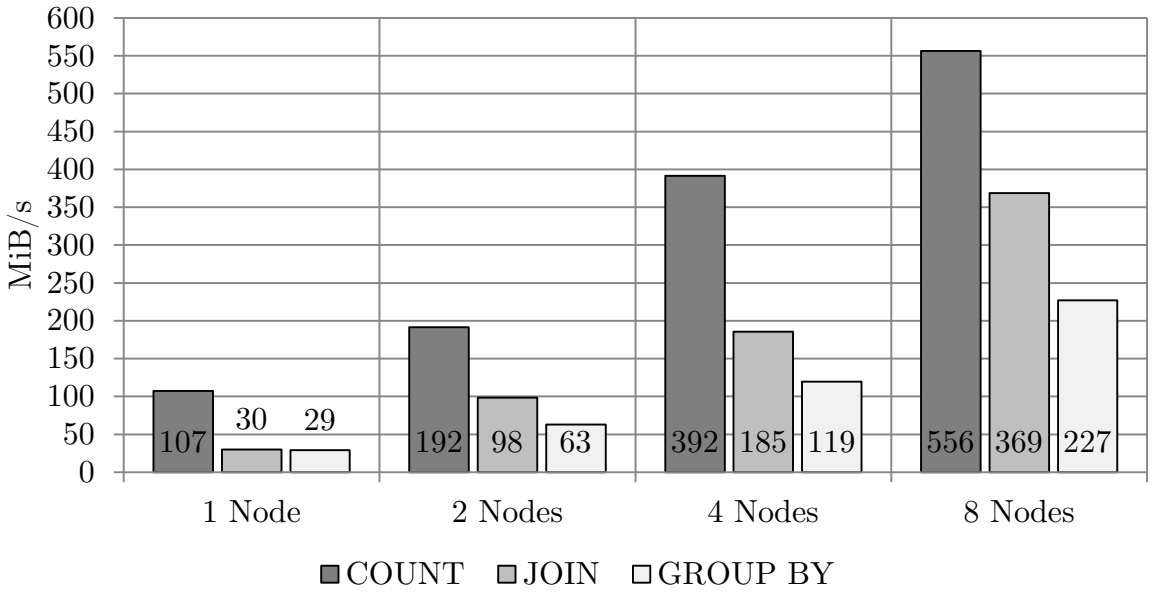

Diagramm 15: Datenübertragungsrate aller Queries bei 4 GiB/Node in shm

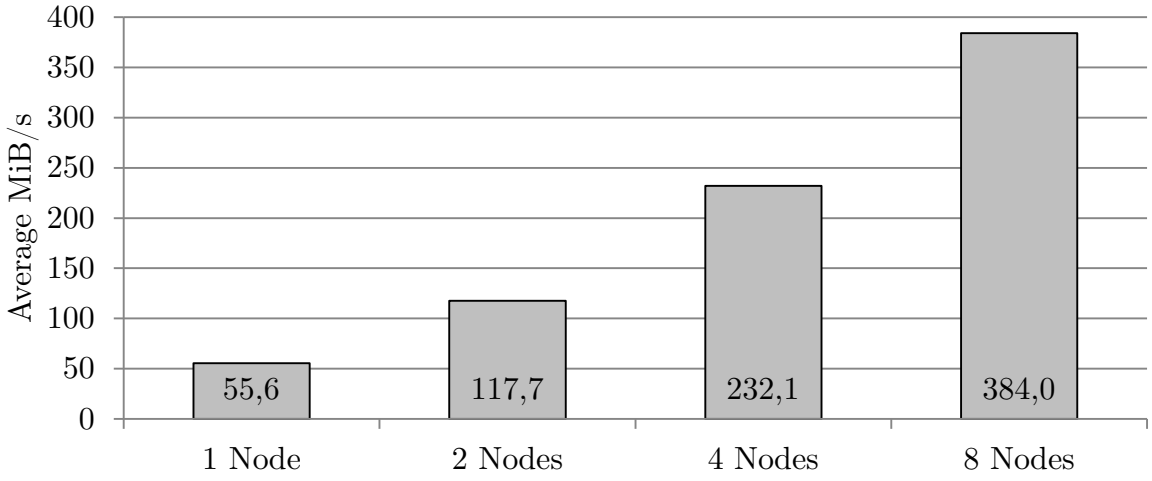

Diagramm 16: Durchschnittliche Datenübertragungsrate aller Queries bei 4 GiB/Node in shm

5.4.2.1. COUNT (4 GiB/Node in shm)

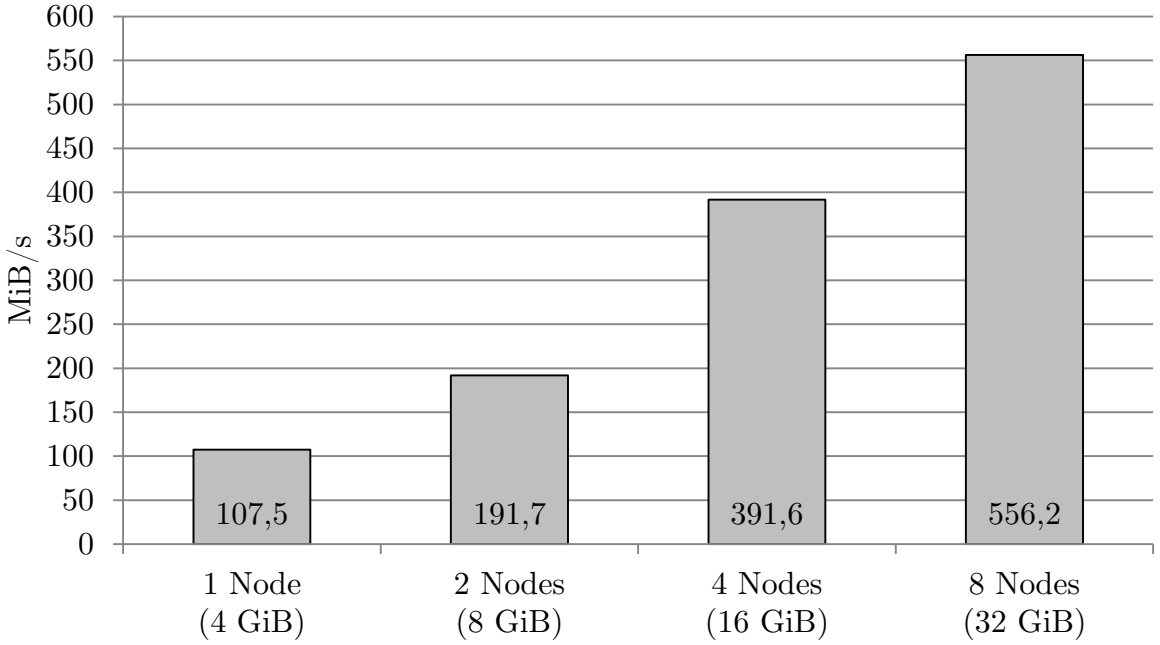

Diagramm 17: Datenübertragungsrate in Mebibytes für COUNT bei 4 GiB/Node in shm

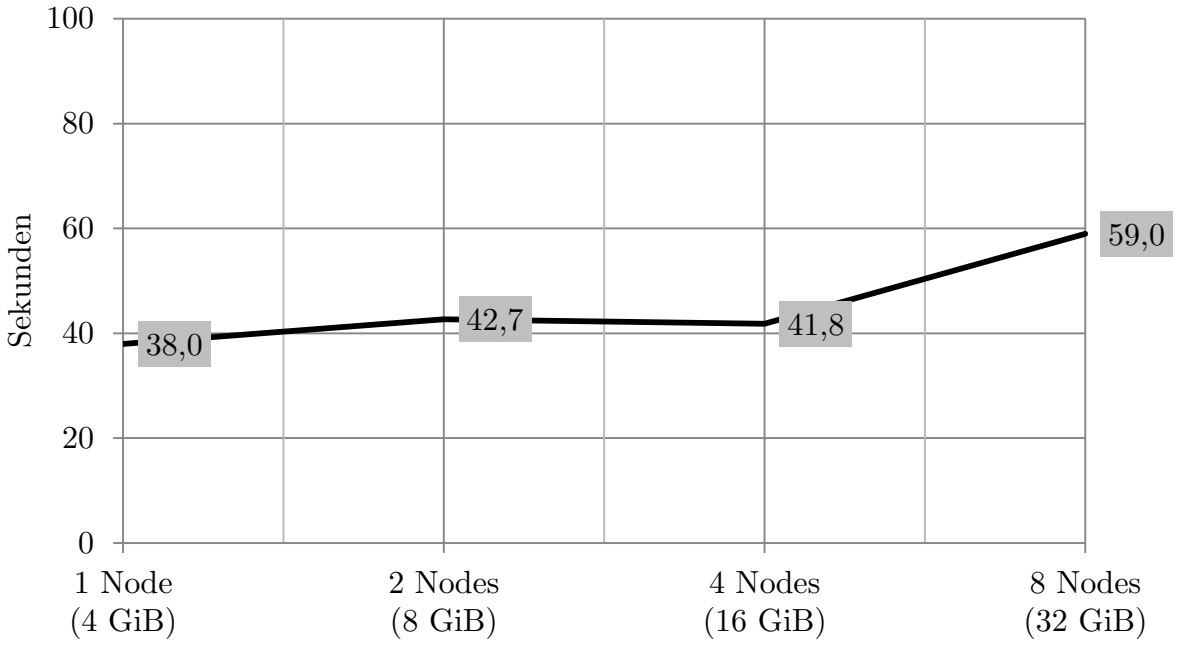

Diagramm 18: Abfragedauer in Sekunden für COUNT bei 4 GiB/Node in shm

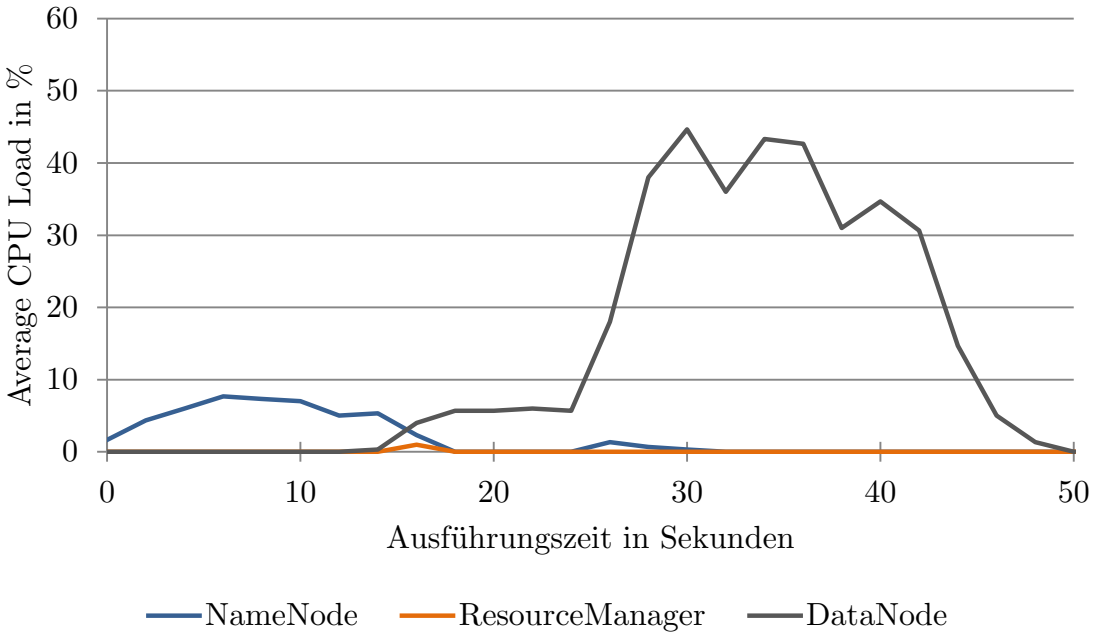

Diagramm 19: Durchschnittliche CPU-Auslastung des Clusters für COUNT bei 4 GiB in shm auf einem DataNode (1+2)

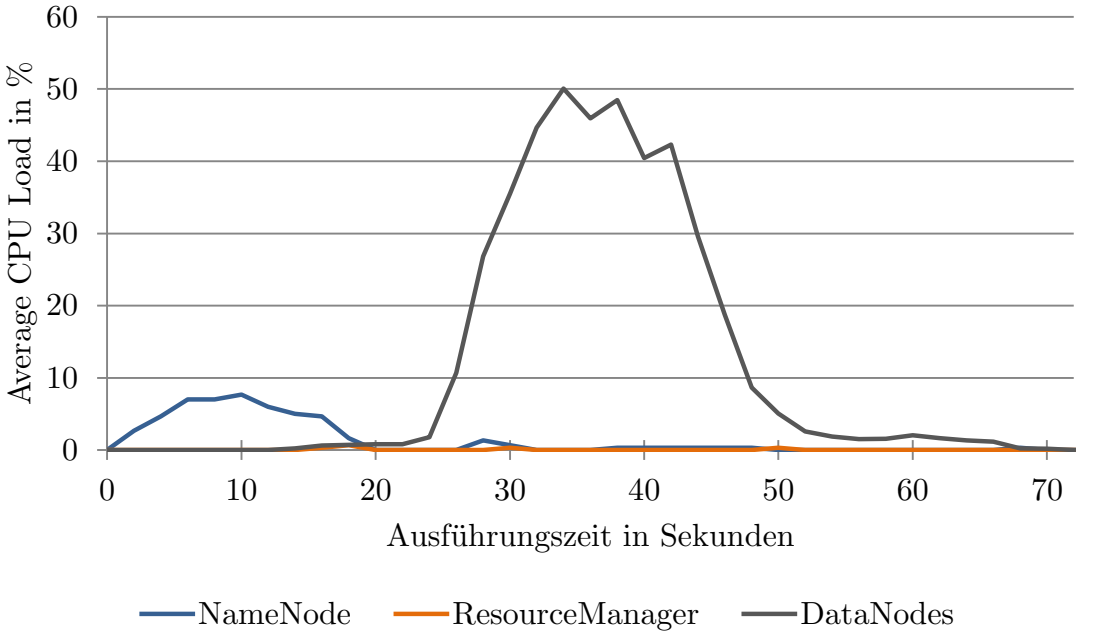

Diagramm 20: Durchschnittliche CPU-Auslastung des Clusters für COUNT bei 4 GiB in shm je DataNode und insgesamt 8 DataNodes (8+2)

5.4.2.2. JOIN (4 GiB/Node in shm)

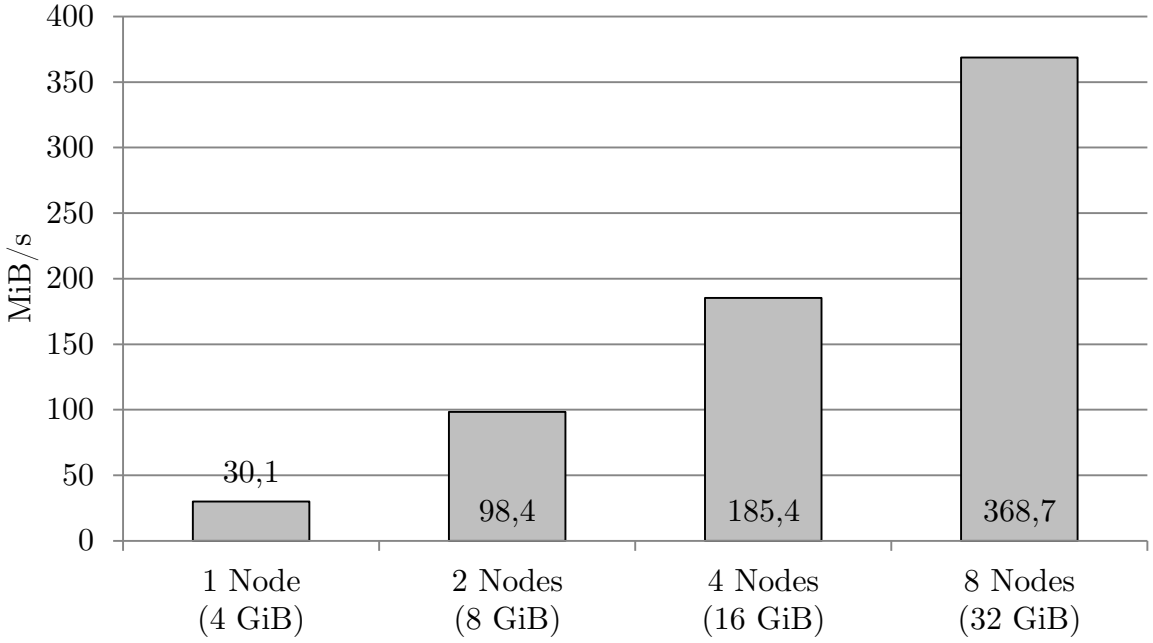

Diagramm 21: Datenübertragungsrate in Mebibytes für JOIN bei 4 GiB/Node in shm

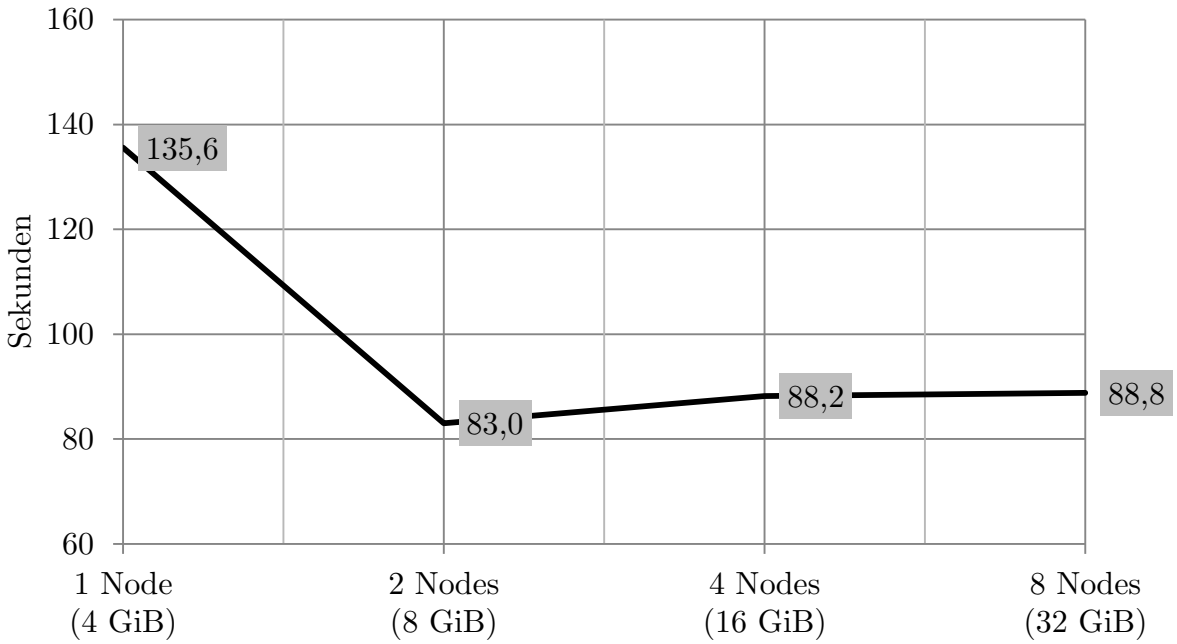

Diagramm 22: Abfragedauer in Sekunden für JOIN bei 4 GiB/Node in shm

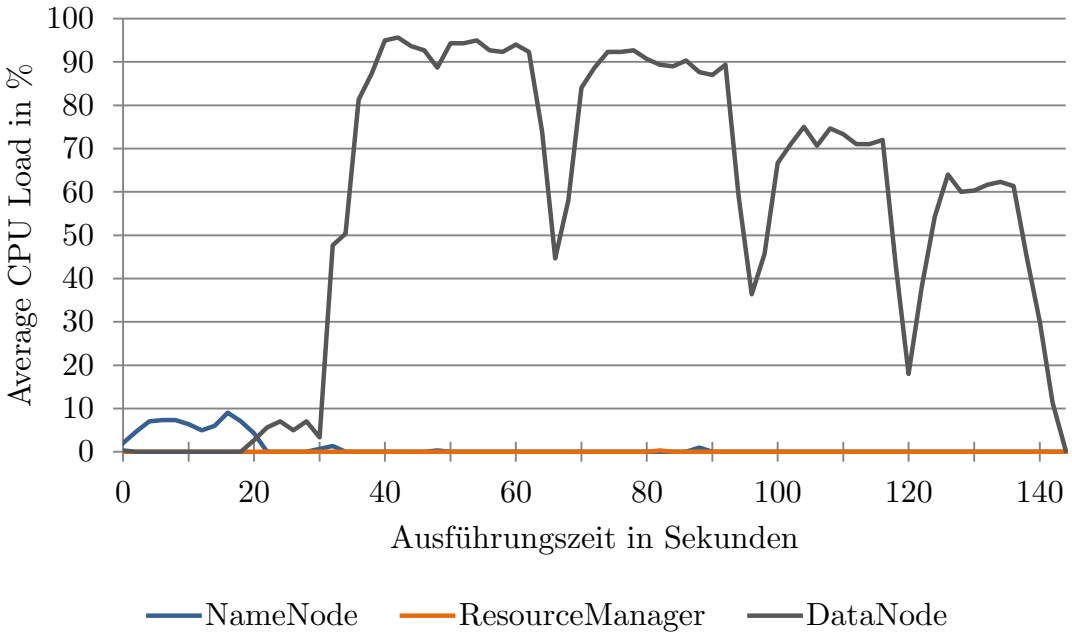

Diagramm 23: Durchschnittliche CPU-Auslastung des Clusters für JOIN bei 4 GiB in shm auf einem DataNode (1+2)

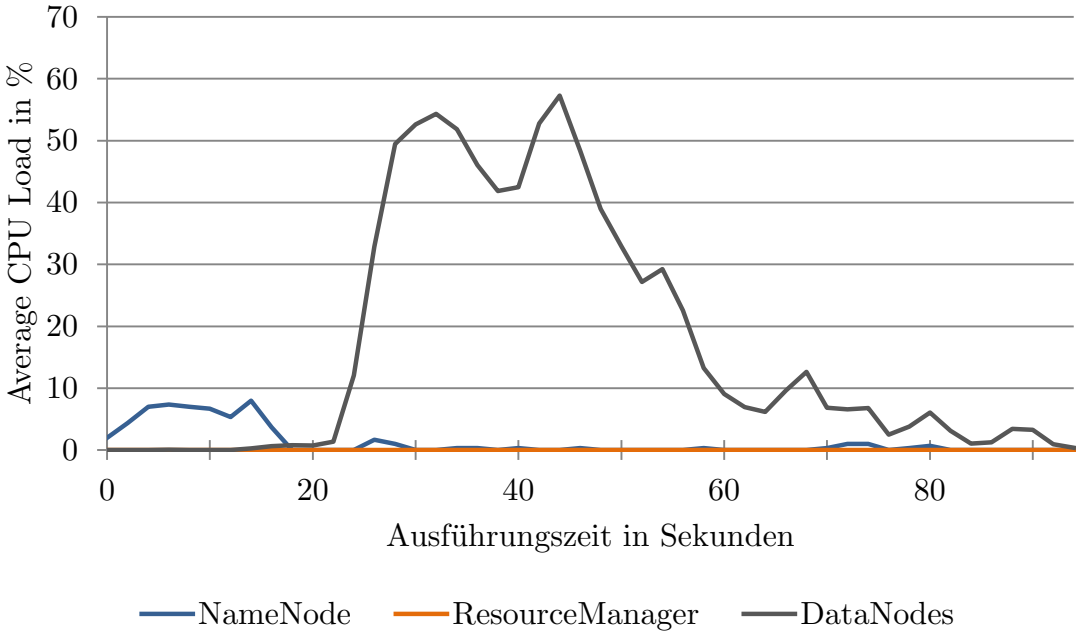

Diagramm 24: Durchschnittliche CPU-Auslastung des Clusters für JOIN bei 4 GiB in shm je DataNode und insgesamt 8 DataNodes (8+2)

5.4.2.3. GROUP BY (4 GiB/Node in shm)

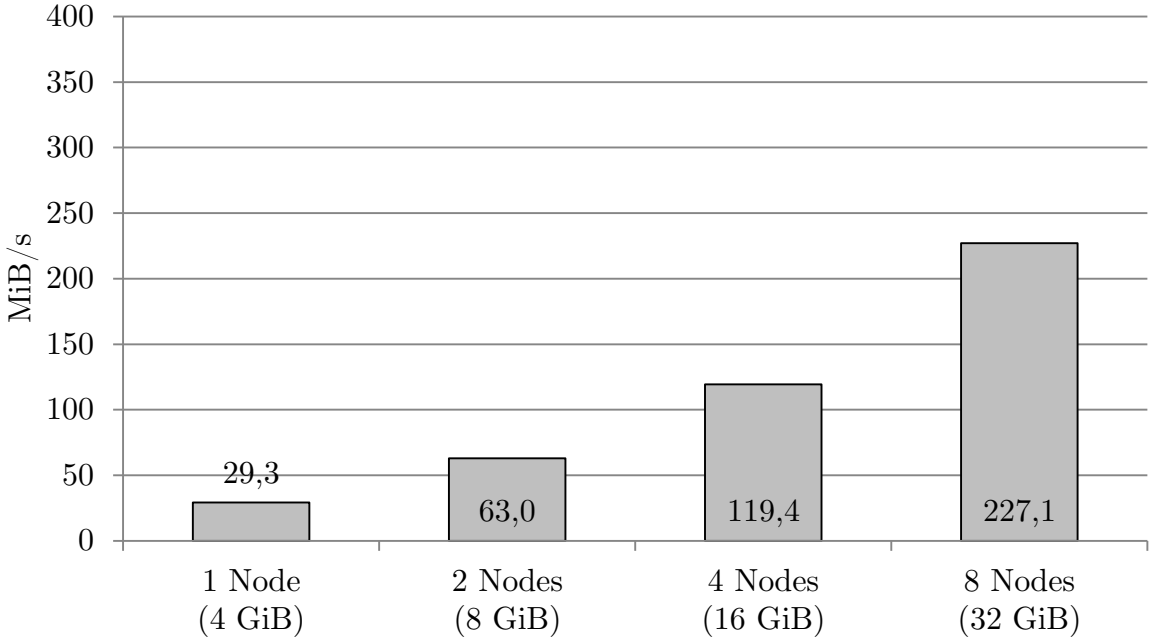

Diagramm 25: Datenübertragungsrate in Mebibytes für GROUP BY bei 4 GiB/Node in shm

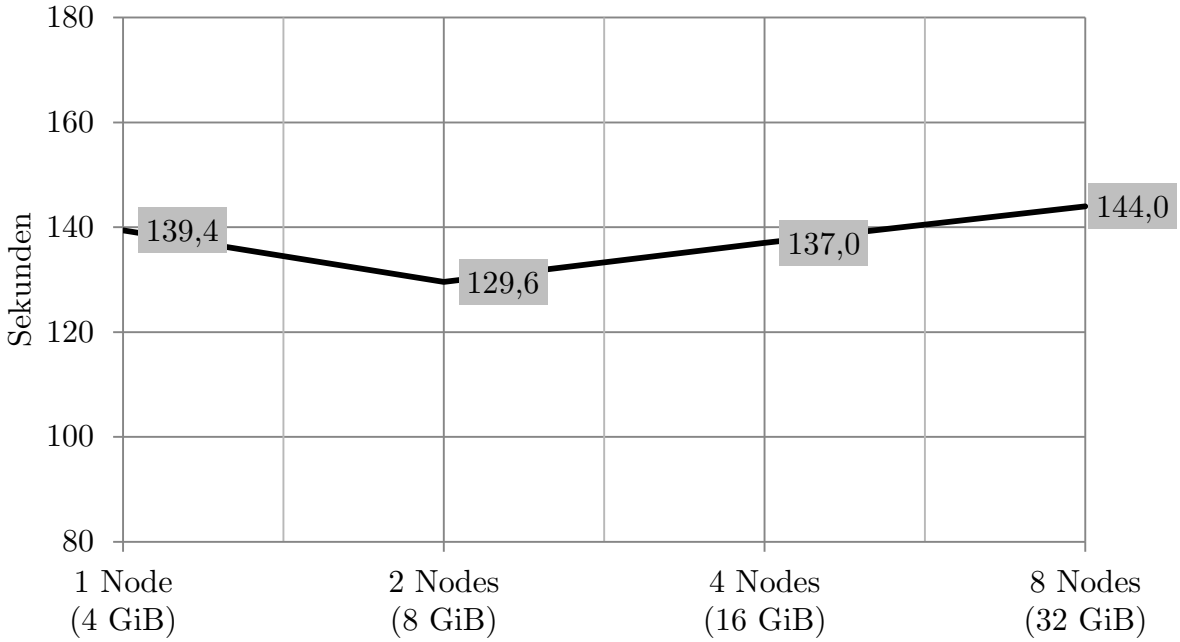

Diagramm 26: Abfragedauer in Sekunden für GROUP BY bei 4 GiB/Node in shm

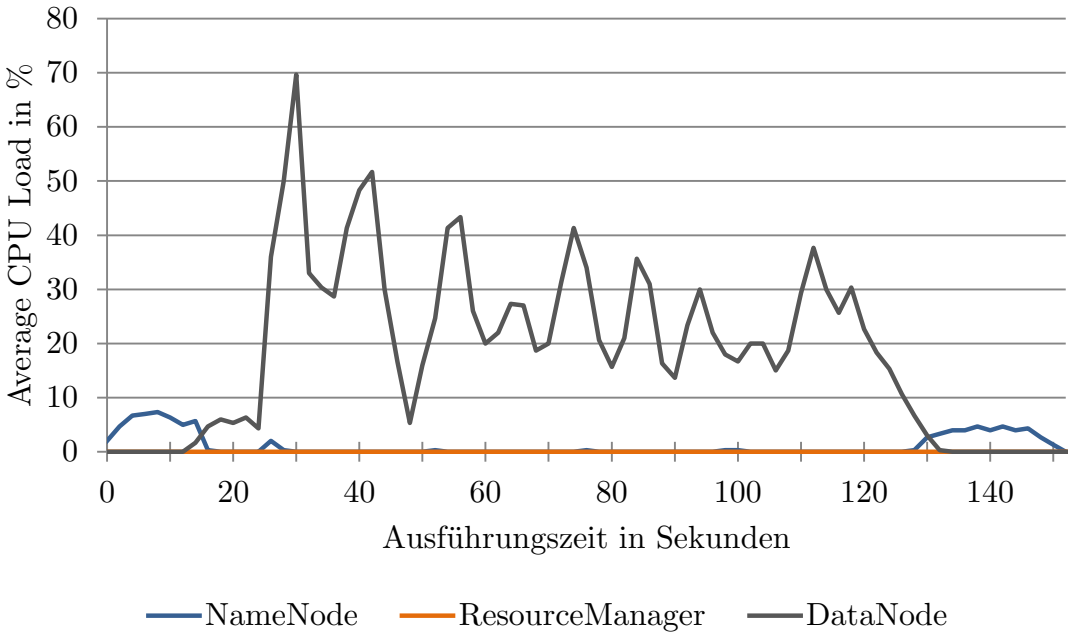

Diagramm 27: Durchschnittliche CPU-Auslastung des Clusters für GROUP BY bei 4 GiB in shm auf einem DataNode (1+2)

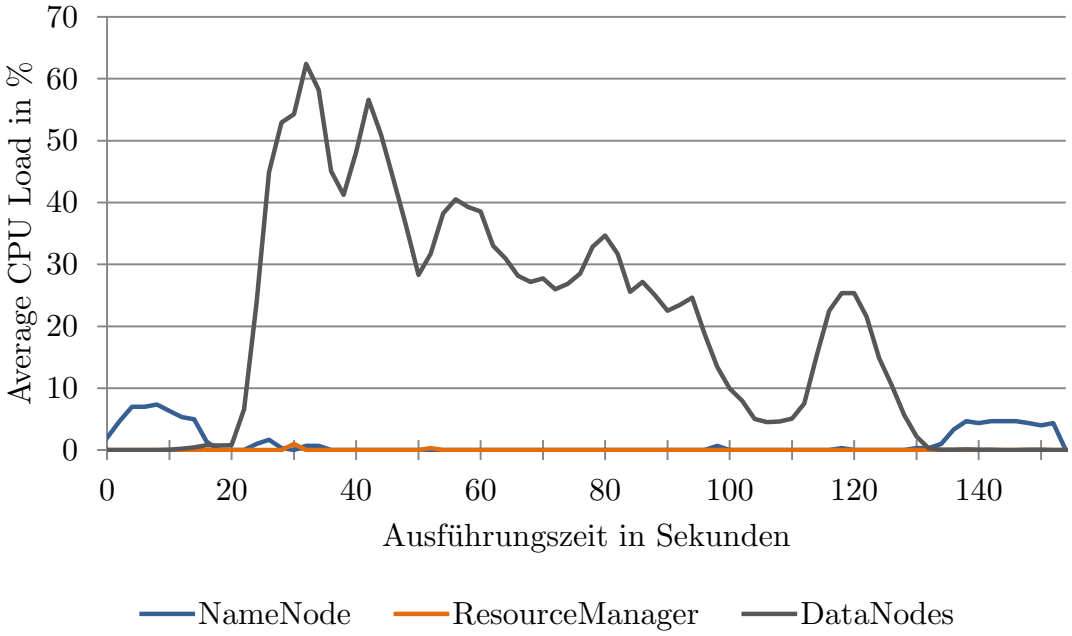

Diagramm 28: Durchschnittliche CPU-Auslastung des Clusters für GROUP BY bei 4 GiB in shm je DataNode und insgesamt 8 DataNodes (8+2)

5.5. Evaluationsreihe 3: 20 GiB/Node (Harddisk)

Cluster-Konfiguration:	1 NameNode, 1 ResourceManager, [1, 2, 4, 8] DataNodes
Messungen insgesamt:	36 (3×3 Queries $\times$ 4 Konfigurationen)
Speicherort der Daten:	/tmp (Harddisk, exklusiv)
Datengröße je DataNode:	~20 GiB (19,95 GiB)
Datensätze je DataNode:	165 Mio.

5.5.1. Zielsetzung

Da die Speicherkapazität (RAM) eine vergleichsweise limitierte Ressource ist, war die verwendete Datengröße in den vorherigen Testreihen ebenfalls auf 4 GiB/Node begrenzt. Da Hadoop und Hive explizit für die Verarbeitung großer Datenmengen konzipiert wurden, hat diese Evaluationsreihe das Ziel, unter Verwendung von größeren Datensätzen eine realitätsnahe Aussage über die Leistung der analysierten Systeme zu treffen.

„Große Datenmengen“ ist unbestreitbar eine ungenaue Formulierung, die auch in entsprechender Literatur vielerorts anzutreffen ist und je nach Anwendungsfall stark variieren kann. Um die Abfragedauer der Queries überschaubar und im Zeitlimit des Job-Schedulers zu halten, entschieden wir uns für 20 GiB je Node, was 165 Mio. Datensätzen entspricht. Ferner sollte bei der Wahl geeigneter Datengrößen auch der für das Generieren der Daten benötigte Zeitaufwand bedacht werden, der sich bei diese Testreihe bereits auf einige Stunden für den kleinsten Datensatz von 20 GiB belief.

Neben seinem Bezug zu praktischen Anwendungsfällen ist an diesem Benchmark auch der Vergleich mit den Ergebnissen der ersten Evaluationsreihe (vgl. 5.3) interessant, da beide über identische Rahmenbedingungen verfügen und sich lediglich in ihren Datengrößen unterscheiden.

5.5.2. Ergebnisse

Nodes	COUNT		JOIN		GROUP BY	
	Sek.	MiB/s	Sek.	MiB/s	Sek.	MiB/s
1+2	647,6	31,6	681,9	30,0	890,4	23,0
2+2	552,2	74,1	593,7	68,9	846,1	48,4
4+2	649,6	126,0	701,7	116,6	914,3	89,7
8+2	843,9	193,9	830,6	197,0	1049,8	155,9

Tabelle 4: Ergebnisse für 20 GiB/Node in tmp; Durchschnitt aus 3 Messungen je Query (gerundet)

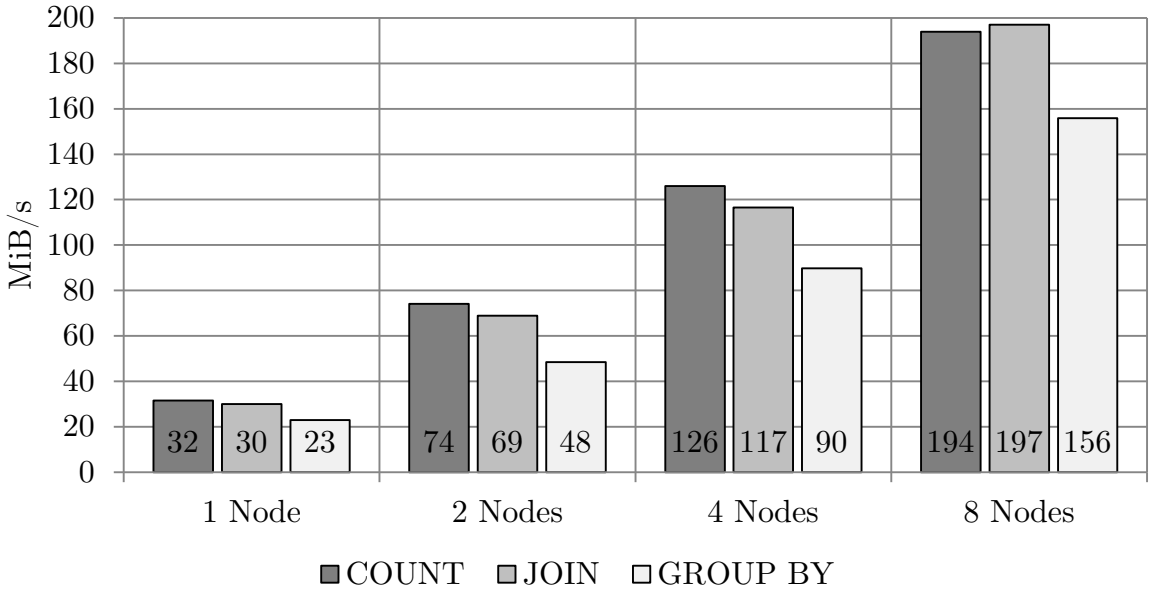

Diagramm 29: Datenübertragungsrate aller Queries bei 20 GiB/Node in tmp

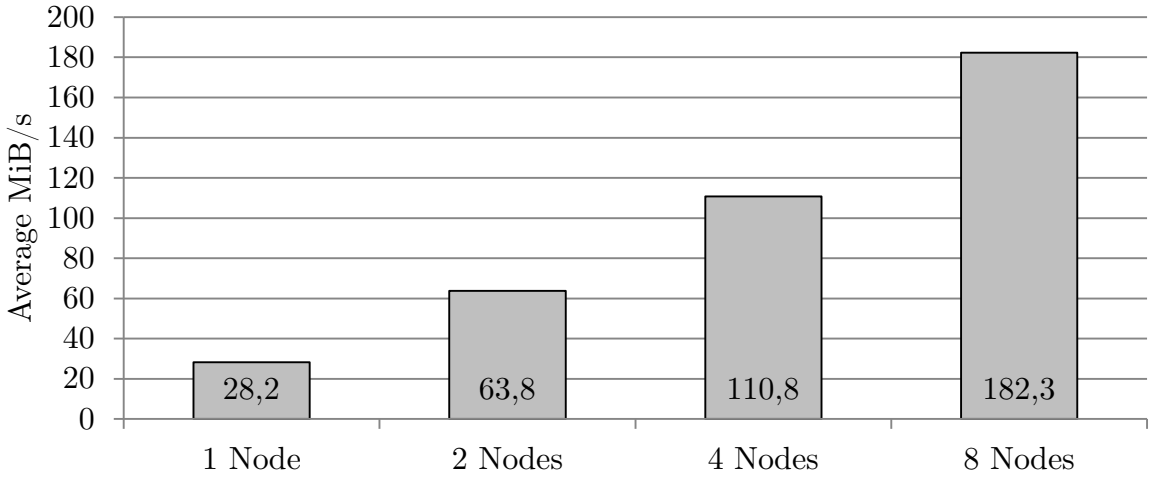

Diagramm 30: Durchschnittliche Datenübertragungsrate aller Queries bei 20 GiB/Node in tmp

5.5.2.1. COUNT (20 GiB/Node in tmp)

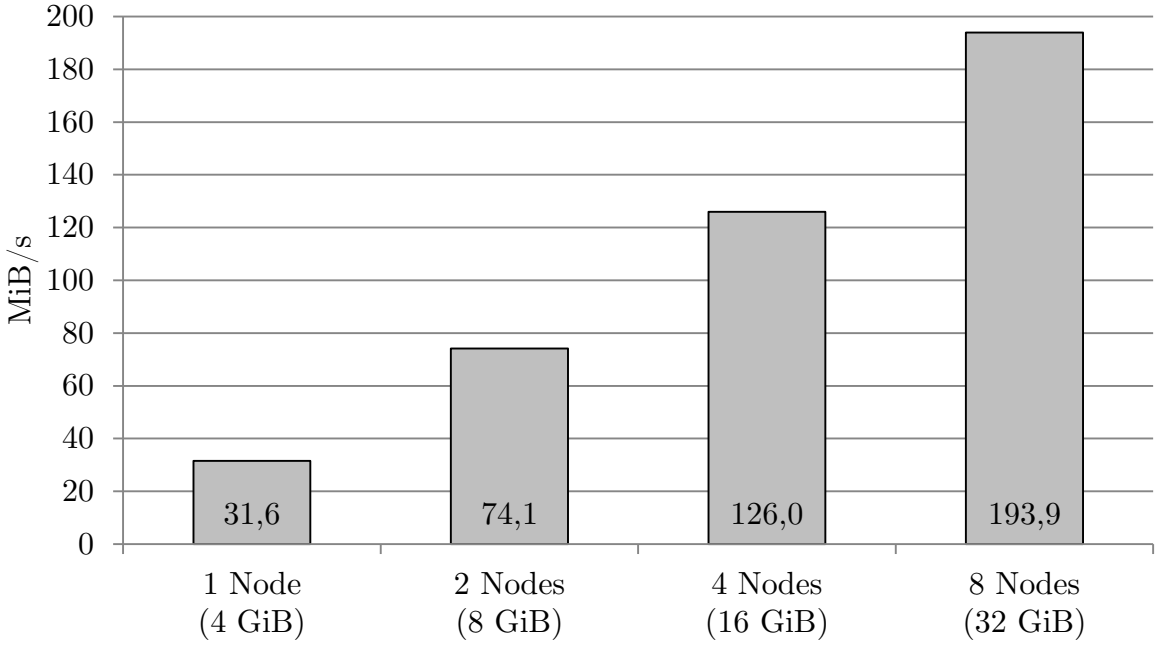

Diagramm 31: Datenübertragungsrate in Mebibytes für COUNT bei 20 GiB/Node in tmp

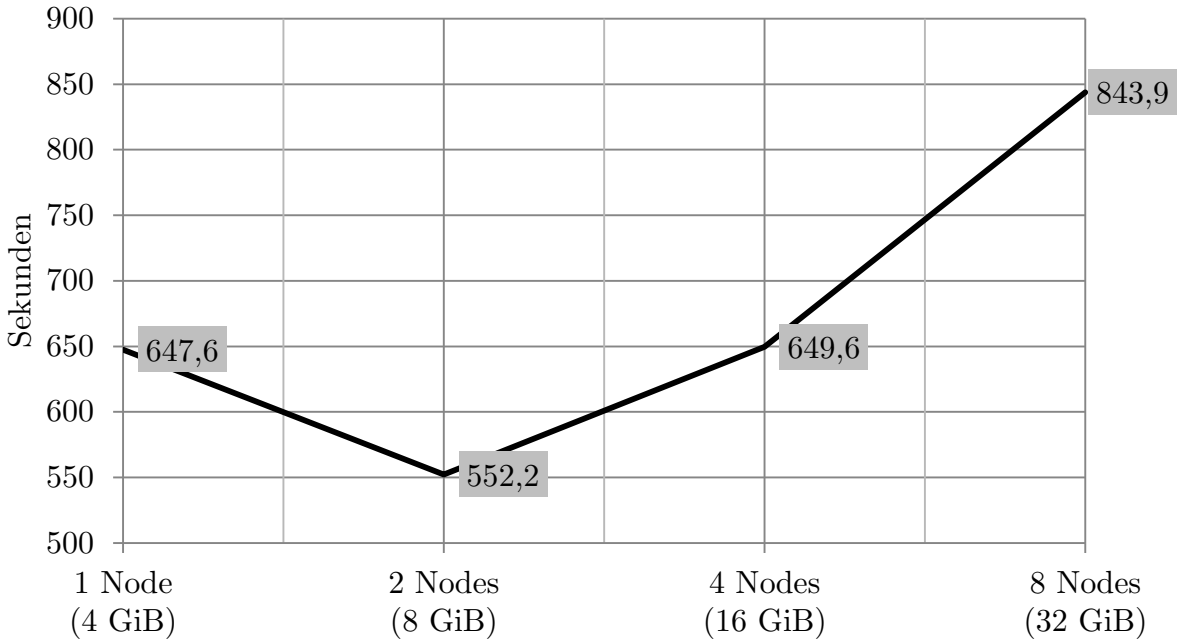

Diagramm 32: Abfragedauer in Sekunden für COUNT bei 20 GiB/Node in tmp

5.5.2.2. JOIN (20 GiB/Node in tmp)

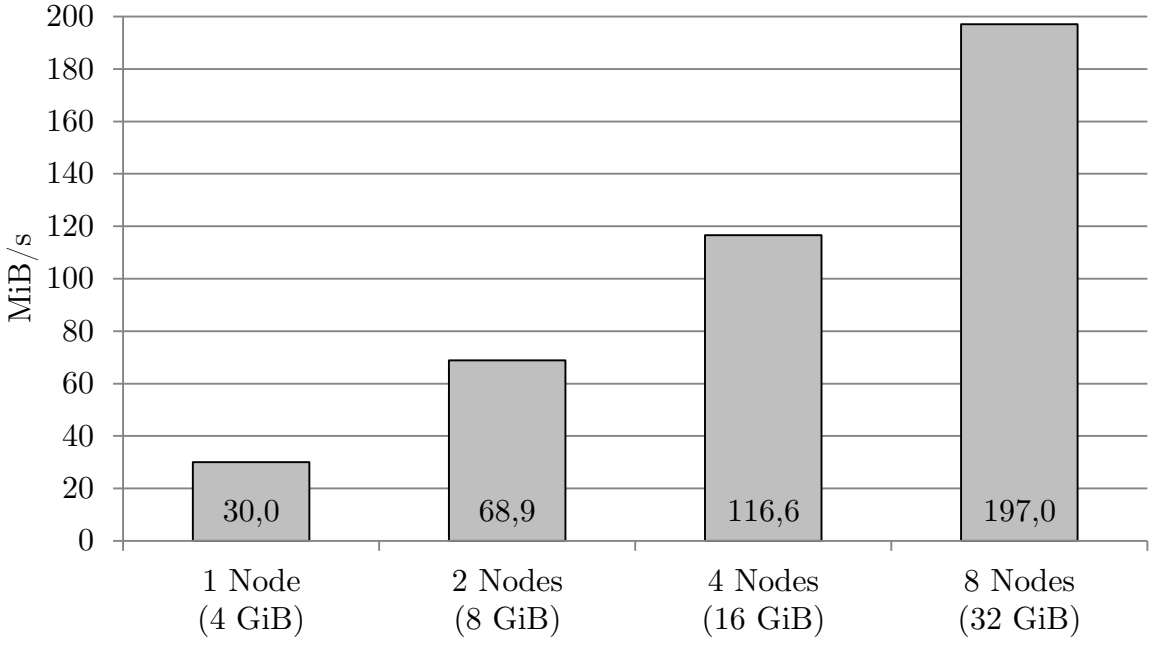

Diagramm 33: Datenübertragungsrate in Mebibytes für JOIN bei 20 GiB/Node in tmp

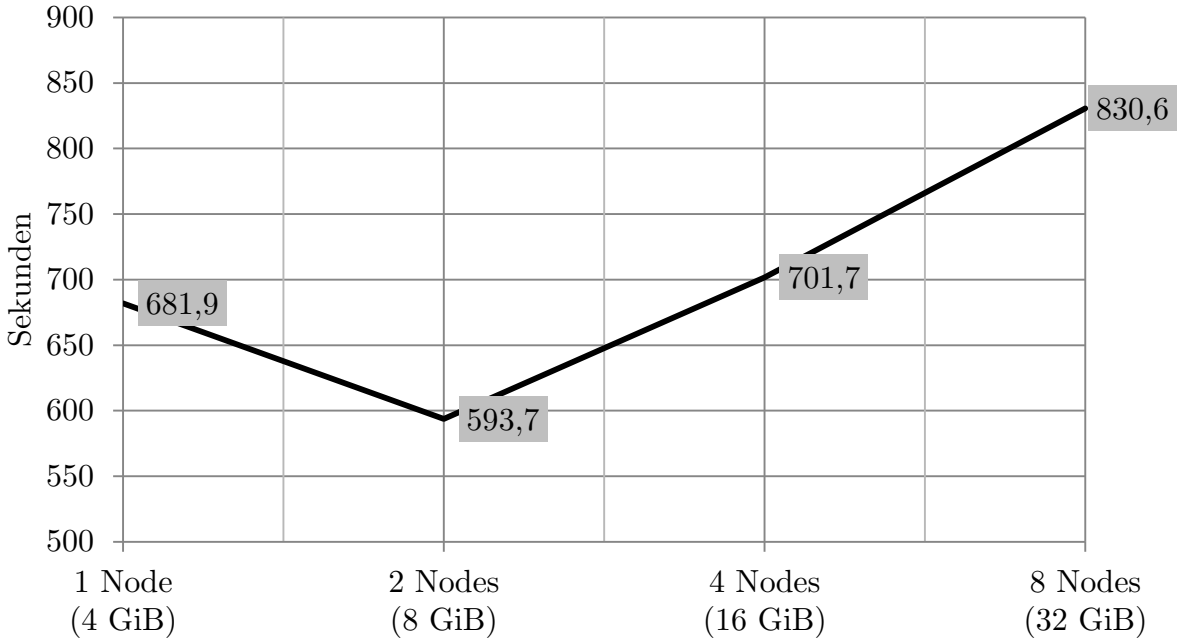

Diagramm 34: Abfragedauer in Sekunden für JOIN bei 20 GiB/Node in tmp

5.5.2.3. GROUP BY (20 GiB/Node in tmp)

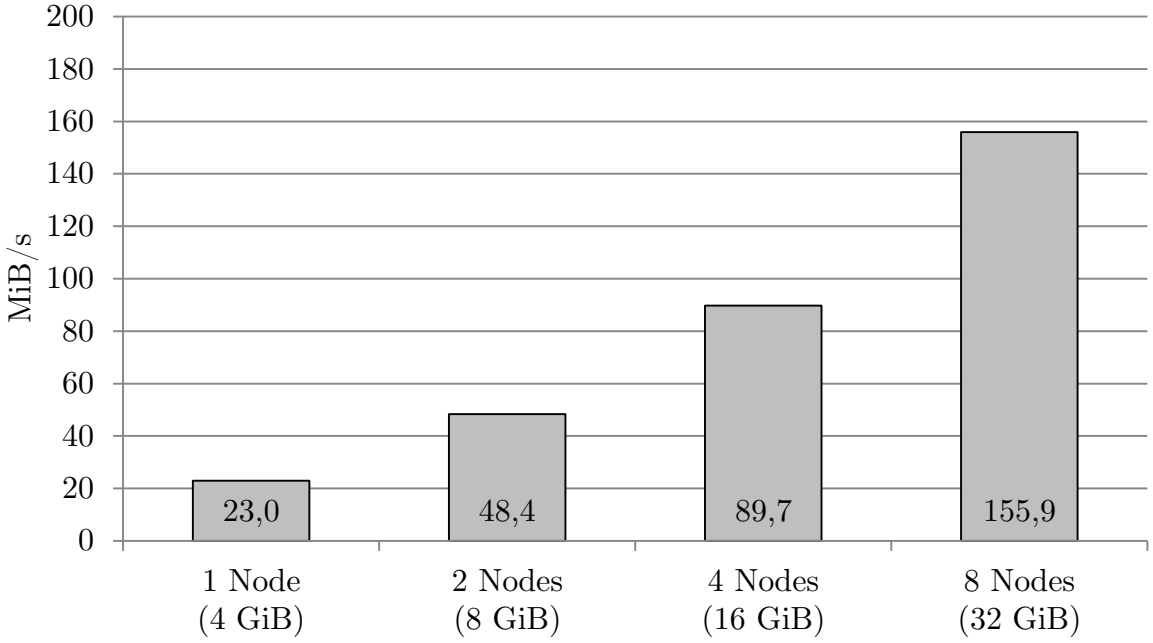

Diagramm 35: Datenübertragungsrate in Mebibytes für GROUP BY bei 20 GiB/Node in tmp

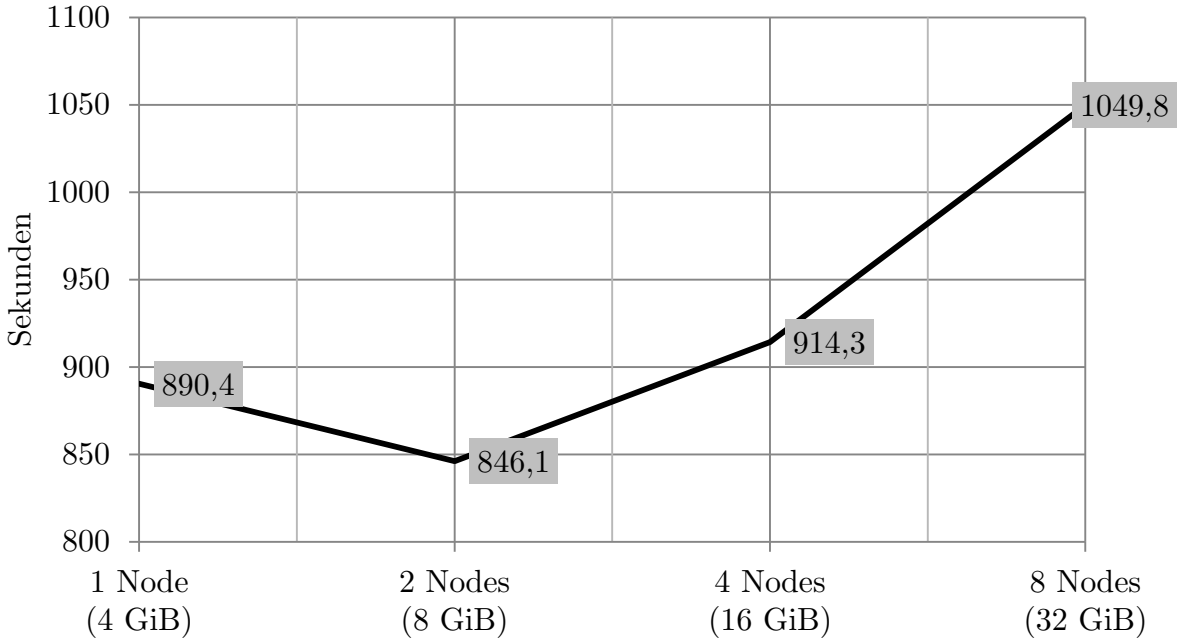

Diagramm 36: Abfragedauer in Sekunden für GROUP BY bei 20 GiB/Node in tmp

5.6. Interpretation der Messergebnisse

Nachfolgend werden die Ergebnisse der Evaluationsreihen interpretiert und in einen Zusammenhang mit den Kernfragen des Projekts gestellt.

5.6.1. Skalierbarkeit und Durchsatz

Bei der zweiten und dritten Evaluationsreihe ist eine auffällige Anomalie der Ausführungszeiten zu beobachten. Mit Ausnahme des COUNTs in Reihe 2 (vgl. Diagramm 18) benötigten alle Queries mit nur einem DataNode mehr Zeit, als die Abfrage auf zwei Knoten mit doppelter Datenmenge. Diese Diskrepanz gipfelt im JOIN der Reihe 2 (vgl. Diagramm 22), dessen Ausführung auf einem Node rund 63% mehr Zeit benötigt, als auf zwei oder mehr Knoten. Interessanterweise tritt dieses Phänomen nicht in der ersten Messreihe auf (vgl. u.a. Tabelle 2): Hier wächst die Abfragedauer aller Queries mit zunehmender Knotenanzahl *(Datengröße)* und weist einen linearen Trend auf.

Als Ursache für diese „Ausreißer“ kann ein Kommunikationsoverhead ausgeschlossen werden, da insbesondere bei der Verwendung nur eines DataNodes der Austausch von Zwischenergebnissen und somit die (Netzwerk-)Kommunikation mit anderen Knoten entfällt. Eine Untersuchung der Output-Files zeigte, dass Hive bei allen Konfigurationen mit mehr als einem DataNode zwei MapReduce-Jobs für einen JOIN erzeugt, die anschließend in Form von Map- und Reduce-Aufgaben an die Knoten verteilt werden. Dies ist jedoch nicht ungewöhnlich, da MapReduce keine separaten Datensätze verbinden kann und Hive den JOIN aus diesem Grund in mehrere Phasen als einzelne Jobs aufteilt. Zudem läuft der Prozess in der ersten Evaluationsreihe identisch ab, so dass dieser Aspekt ebenfalls als Ursache für die höhere Ausführungsdauer auf einem Node ausgeschlossen werden kann.

Eine Erklärung für die besonders extreme Ausprägung der Anomalie im Falle des JOINs bei 4 GiB in shm auf einem DataNode könnte die CPU-Auslastung des Knotens sein: Diagramm 23 zeigt eine Auslastung von >90% für etwa die Hälfte der Ausführungszeit, während die Konfiguration mit 8 DataNodes eine deutlich moderatere Auslastung von <60% aufweist (vgl. Diagramm 24). Da der JOIN in tmp jedoch eine vergleichbare Last erzeugt hat (vgl. Diagramm 9), kann dies nicht der einzige Grund für die Auffälligkeit sein.

Da die eingangs diskutierten Messergebnisse für die meisten Konfigurationen mit einem DataNode stark von den restlichen abweichen, könnten sie die Weak-Scaling-Effizienz verfälschen. Ein Vergleich unserer Ergebnisse mit den Experimenten von Pavlo et al. [PPR09] bestätigt außerdem unsere ursprüngliche Erwartung, dass sich die Ausführungsdauer auf einem DataNode unterhalb oder idealerweise in einer Reihe mit jener von Multiple-Slave-Konfigurationen anordnen müsste. Aus diesem Grund haben wir uns dazu entschieden, die Weak-Scaling-Effizienz in zwei Varianten zu bestimmen, um die tatsächlich ermittelten Ergebnisse nicht zu beschönigen oder vorzuenthalten.

Die Effizienz ist den Tabellen 5 und 6 zu entnehmen, wobei Tabelle 5 als pessimistische, möglicherweise fehlerhafte Variante aufgefasst werden sollte, die die Effizienz aller durchgeführten Messungen widerspiegelt und dafür die fragwürden Single-Node-Konfigurationen als Basis wählt. Tabelle 6 missachtet diese hingegen und zeigt das Weak-Scaling mit der 2+2-Konfiguration als Ausgangspunkt. Nach unserer Einschätzung hat Tabelle 6 die größere Aussagekraft über die Skalierung von Hive, da die o.g. Problematik möglicherweise auch auf einen Konfigurationsfehler zurückzuführen sein könnte.

Query	4 GiB/Node in tmp	4 GiB/Node in shm	20 GiB/Node in tmp
COUNT	0.64	0.65	0.77
JOIN	0.71	1.53	0.82
GROUP BY	0.65	0.97	0.85

Tabelle 5: Weak-Scaling-Effizienz auf Basis der 1+2-Konfiguration (gerundet)

Query	4 GiB/Node in tmp	4 GiB/Node in shm	20 GiB/Node in tmp
COUNT	0.77	0.73	0.65
JOIN	0.82	0.94	0.71
GROUP BY	0.79	0.90	0.80

Tabelle 6: Weak-Scaling-Effizienz auf Basis der 2+2-Konfiguration (gerundet)

Zur Berechnung der Effizienz wurde die Datenrate des größten Falles (8+2) durch die gewichtete Übertragungsleistung der kleinsten Konfiguration *(1+2 für Tabelle 5 bzw. 2+2 für Tabelle 6)* dividiert. Für den ersten COUNT in Tabelle 6 ergibt sich so bspw. folgende Rechnung:

```
179.0 MiBs / (58.4 MiBs * 4) = 0.766
```

Die in Tabelle 5 rot hervorgehobene Kennziffer ist ein weiteres hervorstechendes Indiz für die fragwürdigen Ausführungszeiten bei Verwendung eines einzelnen DataNodes. Eine Effizienz mit > 1.0 ist theoretisch nicht möglich, da die Datengröße konstant zur Knotenanzahl gehalten wurde und somit jeder Knoten bei gleicher Arbeit schneller geworden sein müsste. Im HPC-Bereich ist dies lediglich durch Superlinearität als Folge eines Caching-Effekts „möglich", den wir jedoch aufgrund der o.g. Konstanz der Problemgröße ausschließen können und der Unterschied ohnehin zu drastisch ausfällt. Im besten Falle sollte sich die Effizienz *(von unten)* 1.0 annähern, was einer verdoppelten Datenübertragungsrate bei Verdoppelung der Knotenanzahl und Datengröße entspräche.

Der COUNT skaliert wider Erwarten nur vergleichsweise durchschnittlich (0.64 bis 0.77), obwohl gerade diese Operation keinen nennenswerten Ressourcenbedarf haben sollte. Dies spiegelt sich auch in der Prozessorauslastung des Clusters wider, die sich je nach Konfiguration maximal zwischen 30 und 40 Prozent bewegt (vgl. Diagramme 5, 6). Für die In-memory-Konfiguration fällt die Auslastung etwa 10% größer aus (vgl. Diagramme 19, 20), was auf eine erheblich schnellere Datenbereitstellung zurückzuführen ist, so dass die Aufgabe *(mit entsprechend größerer Prozessorlast)* schneller abgearbeitet werden kann. Nachfolgendes Diagramm 37 stellt die Datenrate aller durchgeführten COUNTs gegenüber:

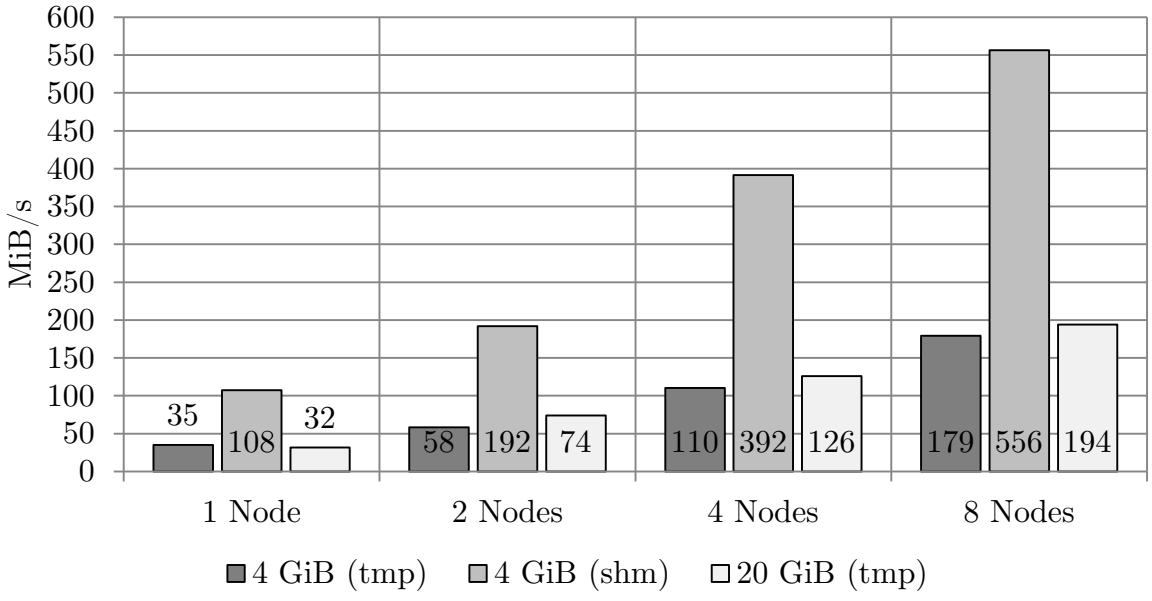

Diagramm 37: Datenübertragungsrate für COUNT; Alle Konfigurationen

Indes befand sich der Durchsatz bei der Abfrage von 4 bzw. 20 GiB auf einem ähnlichen Niveau, wie obigem Diagramm zu entnehmen ist. Die größte Datenübertragungsrate unserer Benchmarks (556 MiB/s) erreichte der COUNT auf 4 GiB in shm mit 8 Nodes. Auch war der COUNT verglichen mit den anderen Queries in jeder Konfiguration stets am schnellsten, so dass das Ziel, mithilfe dieser Abfrage den maximalen Durchsatz des Systems zu bestimmen, als erfolgreich angesehen werden kann. Demzufolge zeigt Diagramm 37 auch den größtmöglichen Durchsatz der getesteten Cluster-Konfigurationen.

Ungeachtet dessen hatten wir beim COUNT mit einem etwas höheren Durchsatz gerechnet. Seine eher schlechte Effizienz könnte ein Anzeichen dafür sein, dass doch *(mehr)* Netzwerkkommunikation stattfindet, als ursprünglich angenommen. Berücksichtigt man auch den Java-Overhead, sind die Messwerte dennoch akzeptabel. Zudem erreichte die Abfrage auf 4 GiB in shm beinahe den Maximalwert von 117 MiB/s, der sich durch die Gigabit-Vernetzung der Knoten ergibt (vgl. 5.1). Möglicherweise stellte das Netzwerk hier sogar einen *(kleinen)* Bottleneck dar.

Die komplexen Operationen, die beim JOIN-Task durch das Verknüpfen über Schlüsselfelder verschiedenartiger Datensätzen einher gehen, spiegeln sich gegenüber dem „einfachen“ COUNT in einer deutlich erhöhren CPU-Auslastung wieder.

In den Diagrammen 9 und 23 der durchschnittlichen CPU-Auslastung des JOINs mit nur einem DataNode sind vier Lastphasen zu erkennen. Nach der Filterung der Datensätze nach Datumsbereich (Phase 1) und Verbund der Tabellen Rankings und UserVisits (Phase 2), erfolgt die Aufsummierung der adRevenue und die Durchschnittsbildung des pageRank (Phase 3). Die abschließende Reduce-Phase führt die Zwischenergebnisse zusammen. Zwischen den einzelnen Phasen ist eine niedrigere CPU-Auslastung zu erkennen. Dies verdeutlicht, dass eine Phase immer erst abgeschlossen sein muss, bevor die nächste beginnt.

Die hohe CPU-Auslastung von >80% mit nur einem DataNode (vgl. Diagramm 9) lässt darauf schließen, dass die gleiche Cluster-Konfiguration im Shared Memory nicht nennenswert performanter sein kann, was auch der berechnete Durchsatz bestätigt (vgl. Diagramm 38).

Die Diagramme 10 und 24 der 8+2 Konfiguration zeigen sowohl in tmp als auch in shm am Ende der Berechnung eine sehr niedrige Auslastung der Slaves. Diese Beobachtung ist auf sehr viel Kommunikationsbedarf beim Zusammenführen der Zwischenergebnisse zurückführbar.

Der JOIN skalierte *(mit Ausnahme der Single-Node-Problematik)* im Falle der 4 GiB-Testreihen gut (0.82 bzw. 0.94). Bei 20 GiB/Node reiht sich die Datenrate zwischen COUNT und GROUP BY ein. Interessanterweise bilden die Ergebnisse der Messreihen in tmp beinahe den Mittelwert aller Queries, was ein Vergleich der Diagramme 1 und 2 *(sowie 29 und 30)* bestätigt. Demnach liegt die Performance eines JOINs ungefähr im Mittelfeld der verwendeten Abfragen, eine Ausnahme bildet dabei jedoch die In-Memory-Konfiguration: Hier liegt der JOIN noch deutlich unter dem Durchschnitt, da dieser durch den sehr schnellen COUNT positiv beeinflusst wird.

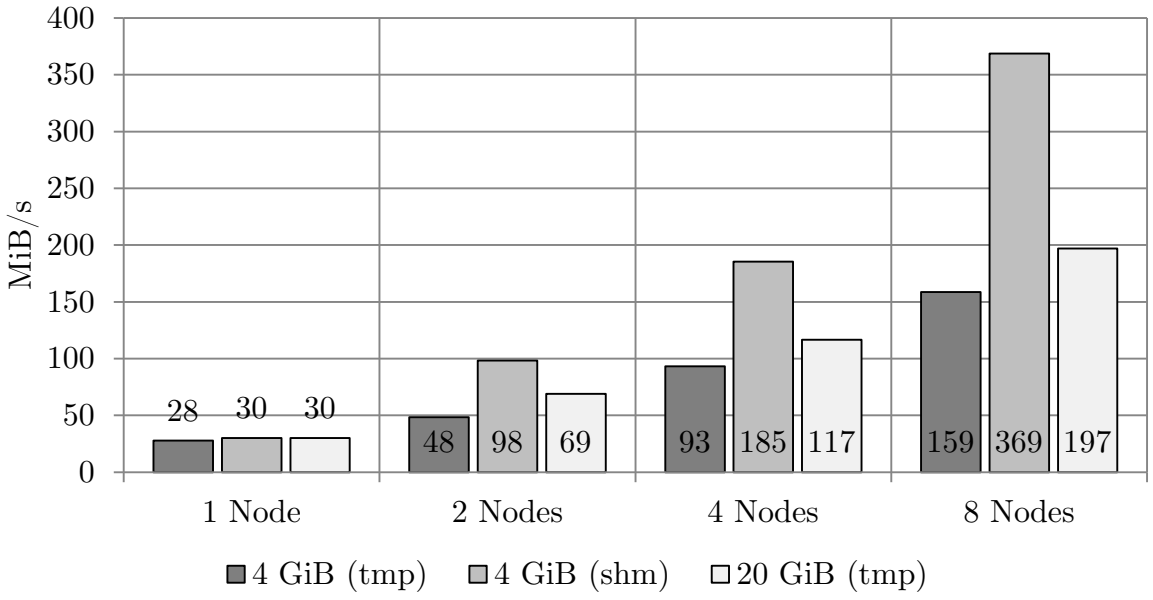

Diagramm 38: Datenübertragungsrate für JOIN; Alle Konfigurationen

Die Aggregation durch einen GROUP BY ist durch einen erhöhten Austausch von Zwischenergebnissen gekennzeichnet, die zwischen den Knoten notwendig ist, um ein endgültiges Ergebnis zu erhalten. Es war damit zu rechnen, dass das Netzwerk hierbei den entscheidenden Faktor der Performance bildet.

Die Diagramme für die GROUP BY-Query 13 und 14 *(tmp)*, sowie 27 und 28 *(shm)* zur CPU-Auslastung zeigen eine annähernd ausgeglichene Auslastung der DataNodes, denn hier finden sich keine verschiedenen Phasen, wie bspw. beim JOIN, wieder.

Da die Auslastung nur durchschnittlich ausfällt, ist nach unserer Einschätzung die o.g. Annahme der erhöhten Netzwerkkommunikation bestätigt. Ein Vergleich der Diagramme mit jenen der Auslastung der In-Memory-Konfiguration macht deutlich, dass die Bereitstellung der Daten durch die Festplatte hier nicht schnell genug erfolgt, da die Auslastung des Clusters geringfügig höher ist und so auch die Festplatte als wichtiger Einflussfaktor in Betracht gezogen werden muss. Dennoch nimmt das Netzwerk den größten Einfluss.

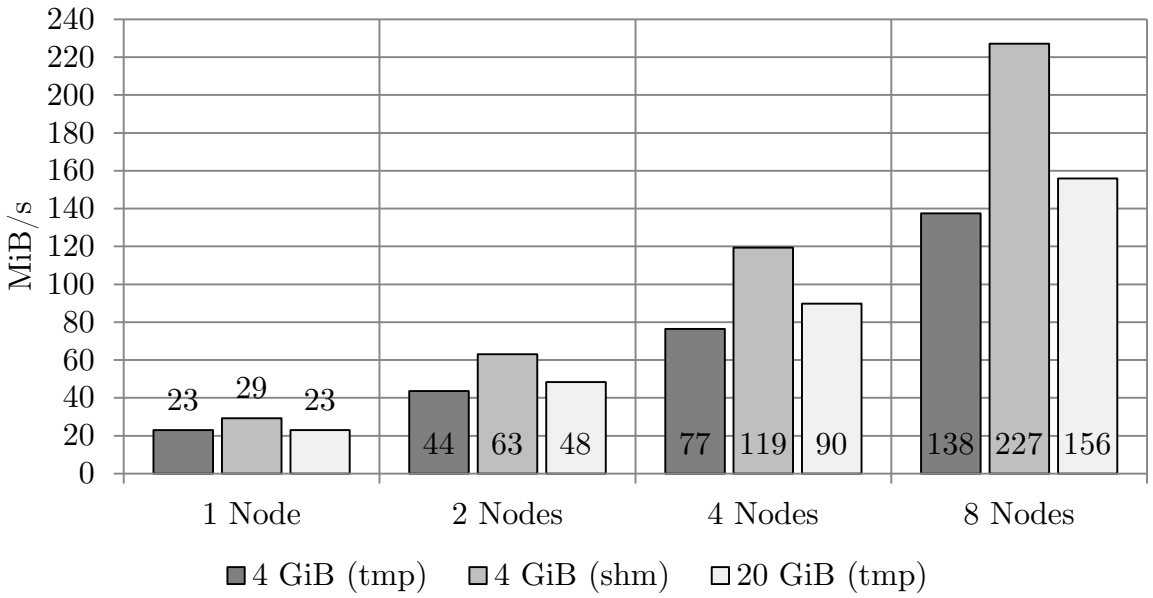

Diagramm 39: Datenübertragungsrate für GROUP BY; Alle Konfigurationen

Ein Vergleich der Messergebnisse unter Betrachtung der Datengrößen und der durchschnittlichen Datenrate aller Queries zeigt, dass letztere bei Abfragen auf größeren Datenmengen allgemein etwas höher ausfällt. Eine Diskrepanz ergibt sich hingegeben bei der Effizienz, die je nach gewählter Basis für größere Datenmengen *(pro Knoten)* unterschiedlich ausfällt *(steigend in Tabelle 5, fallend in Tabelle 6)*. Tabelle 7 ist die genaue Veränderung der Datenrate bei fünffach größerer Datenmenge (4 bzw. 20 GiB/Node) zu entnehmen.

Konfiguration	4 GiB in tmp MiB/s (∅)	20 GiB in tmp MiB/s (∅)	Differenz
1 Node (1+2)	28,7	28,2	−1,7%
2 Nodes (2+2)	50,1	63,8	+27,3%
4 Nodes (4+2)	93,3	110,8	+18,8%
8 Nodes (8+2)	158,4	182,3	+15,1%

Tabelle 7: Durchschnittliche Datenübertragungsrate aller Queries auf tmp; Alle Konfigurationen

Da die Differenz mit zunehmender Knotenanzahl kleiner wird, ist zu erwarten, dass sie bei Verwendung deutlich größerer Cluster-Konfigurationen *(z.B. 1000 Nodes)* nur minimal ausfallen wird. Für den praktischen Einsatz der Systeme bedeutete dies, dass die Größe der Datensätze *(zumindest in einem gewissen Rahmen)* unerheblich ist.

Dennoch ist der Unterschied unserer Ergebnisse beachtlich *(bis zu 27%)*, zumal wir erwartet hätten, dass kleinere Dateien in gleichen Konfigurationen schneller abgefragt werden können. Da die Auslastung der Knoten während unserer Benchmarks meist nicht ihr kapazitatives Limit erreicht hat, könnte die Ursache für die schnellere Abarbeitung größerer Datensätze darin liegen, dass jeder Knoten mehr Daten eigenständig verarbeiten kann, bevor Zwischenergebnisse ausgetauscht werden müssen und die eigentliche Reduktion anschließend nicht mehr sonderlich ins Gewicht fällt.

Die komplexeren Queries skalierten allgemein besser, ohne das System vollständig auszulasten. Möglicherweise rückt der Umfang der Berechnung dabei derart in den Vordergrund, dass das Netzwerk nur noch eine untergeordnete Rolle spielt.

Überraschenderweise ist den Diagrammen zur CPU-Auslastung zu entnehmen, dass die von Apache empfohlene Aufteilung des NameNodes und ResourceManagers auf zwei exklusive Knoten gar nicht notwendig zu sein scheint, da die Auslastung des NameNodes nur zu Beginn einer Abfrage *(beim Durchlaufen der Zuordnungstabelle)* leicht erhöht ist ($< 10\%$) und der ResourceManager über die gesamte Abfrage kaum nennenswert belastet ist ($< 2\%$). Möglicherweise kommt eine derartige Konfiguration erst dann zum Tragen, wenn viele Jobs parallel abgeschickt werden oder der Cluster aus deutlich mehr Nodes besteht. Für unsere Benchmarks hätten beide Dienste offensichtlich auch auf einem Knoten ausgeführt werden können, ohne die Hardware auszulasten.

5.6.2. Vergleich von tmp und shm

Erwartungsgemäß fällt die Datenübertragungsrate sehr viel höher aus, wenn alle Datensätze im Speicher liegen. Am deutlichsten wird der Zuwachs bei COUNT, der in der zweiten Evaluationsreihe durchschnittlich 225% schneller als in der ersten ausgeführt wurde (vgl. a. Diagramm 37). Beim JOIN beträgt die Durchsatzsteigerung im Schnitt 108% und der GROUP BY schaffte es auf einen *(immer noch beachtlichen)* Leistungszuwachs von durchschnittlich 56%.

Query	tmp MiB/s (Summe)	shm MiB/s (Summe)	Leistungszuwachs
COUNT	382,8	1247,0	+225,76%
JOIN	328,1	682,6	+108,04%
GROUP BY	280,6	438,8	+56,38%

Tabelle 8: Gegenüberstellung der akkumulierten Messergebnisse für Queries auf tmp und shm

Der Leistungszuwachs lässt sich darauf zurückführen, dass die Festplatten der Nodes einen Bottleneck bilden. Der enorme Zuwachs des COUNT lässt weiterhin darauf schließen, dass hier die Kommunikation über das Netzwerk geringfügiger ausfällt und so der Durchsatz des Speichermediums den größten Einflussfaktor der Performance bildet.

Im Gegensatz dazu findet bei einem GROUP BY ein erhöhter Austausch über das Netzwerk statt (vgl. 5.6.1). Der Leistungszuwachs fällt hier niedriger aus. Wäre auch hier ein höherer Zuwachs erwünscht, müsste der Durchsatz des Netzwerkes erhöht werden. Es lässt sich also verallgemeinern, dass die Wahl des Speichermediums umso wichtiger ist, desto weniger komplex die Abfrage ausfällt.

Ferner ist dem Versuch zu entnehmen, dass bei großem Leistungsbedarf - die entsprechenden Ressourcen und „passenden“ Datengrößen vorausgesetzt - eine In-Memory-Konfiguration des verteilten Dateisystems einer herkömmlichen (HDD-basierten) unbedingt vorzuziehen ist, da der Leistungszuwachs hier überragend ausfällt. Da Arbeitsspeicher jedoch oftmals keine günstige Komponente ist und die Datenmengen heute schnell weitaus größeren Umfang annehmen können, bleibt diese Konfiguration vermutlich eher eine Ausnahmelösung für einige wenige, spezifische Anwendungsfälle.

6. Fazit

Ziel dieses Projekts war die Evaluation von Hadoop und Hive auf einem Entwicklungscluster. In diesem Rahmen wurden die Systeme zunächst in einer virtuellen Umgebung installiert und konfiguriert. Darauf aufbauend wurde ein Installationsscript erstellt, mit dessen Hilfe ein Rollout dieser Systeme automatisiert werden kann. Im weiteren Verlauf wurden für einen Benchmark geeignete Datensätze und Abfragen erarbeitet, die in Form von „Evaluationsreihen“ festgelegt wurden. Abschließend folgten die Durchführung der Messreihen sowie die Analyse der Ergebnisse und die Ausgestaltung des vorliegenden Projektberichts.

Im Verlauf des Projekts konnten wir viele Erfahrungen mit Hadoop und Hive im Umfeld des verteilten Rechnens sammeln. Da wir zuvor nur im Rahmen der Vorlesung „Hochleistungsrechnen“ Kontakt mit Cluster-Architekturen und verteilten Systemen hatten, bot das Projekt „Parallelrechnerevaluation“ eine ideale Möglichkeit, sich näher und v.a. praktisch mit derartigen Systemen zu beschäftigen.

Im Nachhinein haben sich nicht nur der Umgang mit diesen System und ihre korrekte Konfiguration als fordernde Aufgaben herausgestellt, sondern auch die Planung des Projektablaufs und die Einhaltung gewisser Meilensteine. Wie wichtig zeitliche Puffer in einer solchen Planung sind, erfuhren wir bereits zu einem frühen Zeitpunkt, als sich herausstellte, dass viele Probeläufe nötig sein würden, bis die Einrichtung der Systeme abgeschlossen wäre. Im weiteren Verlauf des Projekts erfuhren wir diese „Lektion“ noch häufiger, u.a. beim Generieren der Daten und Durchführen der Messungen, so dass wir unseren ursprünglichen Zeitplan bald nicht mehr einhalten konnten. Rückblickend mangelte es uns sicherlich auch an Erfahrung mit den Systemen, um den Aufwand einer Evaluation richtig einschätzen zu können.

Nahezu alle Arbeitsschritte führten wir gemeinsam oder in enger Absprache durch, so dass die kollaborative Arbeit an dem Projekt kaum hätte besser laufen können, obgleich auch hier ein hohes Maß an Disziplin und Koordination erforderlich war. Auch die Kommunikation mit unseren Betreuern funktionierte sehr gut, die uns jederzeit mit Ratschlägen und hilfreichen Tipps zur Seite standen.

Um den zeitlichen Ablauf des Projekts nicht *(weiter)* zu gefährden, mussten wir bedauerlicherweise die Benchmarks mit Presto zurückstellen. Insbesondere hinsichtlich der Aussage, Presto sei in einigen Fällen bis zu zehn Mal schneller als Hive, wären die Ergebnisse sicher sehr spannend gewesen. An diesem Beispiel wird deutlich, wie wichtig eine transparente Dokumentation und gute Hilfestellung sein kann, wenn es um die Einführung neuer Systeme oder um eine Entscheidung hinsichtlich eines zu verwendenden Systems geht. Nach unseren Erfahrungen würden wir aktuell davon abraten, Presto in einer produktiven Umgebung zu verwenden, da die verfügbaren Ressourcen stark begrenzt sind und das System selbst noch sehr experimentell wirkt. Benchmarks mit der gerade erschienenen Hive-Version 0.13, die bessere Query-Performance verspricht, wären ebenfalls interessant gewesen.

Insgesamt erachten wir das Projekt als Erfolg, da die Systeme positiv skalieren, die gemessenen Ergebnisse *(näherungsweise)* mit unseren Erwartungen übereinstimmen und wir alle Evaluationsreihen durchführen konnten. Besonders gravierend sind die Unterschiede bei Verwendung des Speichers, die ferner verdeutlichen, welchen Einfluss ein schnelles Speichermedium auf die Performanz des gesamten Systems haben kann.

Die Arbeit am Projekt hat uns viel Freude bereitet und war hinsichtlich der theoretischen und praktischen Anteile sehr ausgewogen. Da nicht nur Parallelrechnern im Allgemeinen eine immer wichtigere Rolle zukommt, sondern auch Big Data Anwendungen und dessen Analyse (OLAP), wie eindrucksvoll an der schnellen Entwicklung und größer werdenden Verbreitung von Hadoop *(und Hive)* zu beobachten ist, werden unsere zukünftigen Projekte zweifelsfrei von unseren gewonnen Erfahrungen profitieren.

7. Referenzen

[Amr12] (2012, Sep.) Your first Hadoop Map-Reduce Job. [Online]. http://xamry.wordpress.com/2012/09/11/your-first-hadoop-map-reduce-job/

[Caf14] M. J. Cafarella. Biography. [Online]. http://web.eecs.umich.edu/~michjc/bio.html

[DeGh04] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Google, Inc. Paper, 2004.

[GGL03] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," Google, Inc. Paper, 2003.

[Hadoop] The Apache Software Foundation. (2014, Mar.) Apache Hadoop. [Online]. http://hadoop.apache.org

[Hive] The Apache Software Foundation. (2014) Apache Hive. [Online]. https://hive.apache.org

[HTMLgen] A. Pavlo, et al. mr-benchmarks - Revision 108: /datagen/htmlgen. [Online]. https://database.cs.brown.edu/svn/mr-benchmarks/datagen/htmlgen/

[PPR09] A. Pavlo, et al., "A Comparison of Approaches to Large-Scale Data Analysis," *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pp. 165-178, 2009.

[PPR11] A. Pavlo, et al. (2011, Aug.) A Comparison of Approaches to Large-Scale Data Analysis. [Online]. http://database.cs.brown.edu/projects/mapreduce-vs-dbms/

[Yarn] The Apache Software Foundation. Apache Hadoop NextGen MapReduce (YARN). [Online]. http://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-site/YARN.html

8. Anlagen