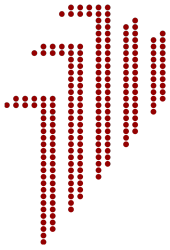


h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt
- Fachbereich Informatik -

Benchmarking von NoSQL-Datenbanksystemen

Abschlussarbeit zur Erlangung des akademischen Grades
Master of Science (M.Sc.)

vorgelegt von
Holger Wegert

Referentin: Prof. Dr. Uta Störl
Korreferentin: Prof. Dr. Inge Schestag

Ausgabedatum: 01.10.2014
Abgabedatum: 01.04.2015

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 1. April 2015

Zusammenfassung

NoSQL-Datenbanksysteme gewinnen im Kontext Big Data geprägter Problemstellungen kontinuierlich an Bedeutung. Einerseits steigt die Vielfalt speziell entwickelter Systeme, die eine dem Anwendungsfall gegenüber optimierte Einsatzfähigkeit ermöglicht. Andererseits entwickeln sich bereits etablierte NoSQL-Datenbanksysteme stetig weiter und erweitern mit optionalen Funktionalitäten das jeweilige Leistungsspektrum.

Eine für die Auswahl und den nachfolgenden Betrieb interessante Fragestellung adressiert die Auswirkungen auf das Leistungsvermögen infolge unterschiedlicher Systemkonfigurationen. Aus diesem Aspekt heraus wurde im Rahmen dieser Arbeit untersucht, inwiefern sich verschiedene Konfigurationsparameter bei den NoSQL-Datenbanksystemen Couchbase 3.0.1 und Apache Cassandra 2.1.2 auswirken. Der Verzicht auf eine direkte Gegenüberstellung der Datenbanksysteme ermöglicht eine den Systemen entsprechende Evaluierung, ohne dass einzelne Konfigurationsparameter gesetzt werden müssten, welche eine vermeintliche Vergleichbarkeit herstellen.

Für das Benchmarking der Systeme wurde Thumbtack Technologies Variante des Yahoo! Cloud Serving Benchmarks (YCSB) weiterentwickelt. Die durchgeführten Modifikationen umfassen zum einen funktionale Erweiterungen, wie bspw. den Ausbau der Mutli-Client Unterstützung. Zum anderen konnten bestehende Schwächen und Fehler identifiziert und behoben werden, welche andernfalls zu Verfälschungen der Messergebnisse führen.

Die Evaluierung der Datenbanksysteme hat gezeigt, dass die unterschiedlichen Konfigurationsmöglichkeiten eine Verwendung der Systeme bei verschiedenartigen Anforderungen grundsätzlich ermöglichen. Je nach Konfiguration wirkt sich deren Anpassung unterschiedlich stark auf das Leistungsvermögen der Systeme aus. Als besonders ausschlaggebend haben sich die unterschiedlichen Konfigurationsmöglichkeiten bzgl. der Dauerhaftigkeit und der Konsistenz herausgestellt. Beispielsweise führt bei beiden Datenbanksystemen ein garantiert dauerhaftes Schreiben zu einer Durchsatzverminderung von mehr als 97 %.

Abstract

NoSQL database systems gain in the context of Big Data affected problems continuously in importance. On the one hand increases the diversity of specially designed systems, which allow an optimized utilizability for each application. On the other hand, already established NoSQL database systems develop steadily and expand with optional functionalities their range of services.

One for the selection and subsequent operation interesting question addresses the effects of different system configurations on the performance of the system. Within the scope of this master thesis this aspect was taken into account and it was examined to what extend various configuration parameters in the NoSQL database systems Couchbase 3.0.1 and Apache Cassandra 2.1.2 impact. The forgoing of a direct comparison of the database systems allows an appropriate evaluation of the single systems without the definition of individual configuration parameters, which would establish an alleged comparability.

To establish an appropriate benchmark for the systems, Thumbtack Technologies version of Yahoo! Cloud Serving Benchmark (YCSB) was further developed. The realized modifications include on the one hand functional enhancements, such as the expansion of the multi-client support. On the other hand could existing weaknesses and errors be identified and corrected which otherwise would lead to mistakes in the measurement results.

The evaluation of the data base systems has shown that the different configuration possibilities in principle make the use of the systems at various requirements possible. Depending on the configuration its modulation affects to different extents the performance of the systems. It turned out that particularly decisive are different possibilities of configuration regarding the durability and consistency. For example, for both tested database systems an assured durable writing leads to a reduction in throughput of more than 97 %.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

1. Einführung

1.1. Motivation und Ziel der Arbeit

Als IT-Dienstleister mit einer ausgeprägten Expertise im Bereich relationaler Datenbanksysteme interessiert sich die ORDIX AG stets für aufkommende Trends und Technologien im gesamten Datenbankumfeld. Einen über die letzten Jahre anhaltenden Trend stellen NoSQL-Datenbanksysteme dar. Insbesondere im Kontext von Big Data geprägten Problemstellungen gewannen NoSQL-Datenbanksysteme durch Eigenschaften wie der horizontalen Skalierung an Bedeutung.

Ein wesentlicher Aspekt, der bei der Auswahl eines derartigen Systems berücksichtigt werden muss, ist, dass die meisten Systeme für einen bestimmten Bereich von Anwendungsszenarien konzipiert wurden. Entsprechend wurden bereits bei der Architektur grundlegende Designentscheidungen getroffen, die das Verhalten der unterschiedlichen Systeme entsprechend charakterisieren. Neben Unterschieden bei der Performanz wirken sich diese wesentlich auf die Dauerhaftigkeit, Verfügbarkeit und Konsistenz von Daten aus.

Infolge der zunehmenden produktiven Verwendung und der kontinuierlichen Weiterentwicklung verfügen die Systeme mittlerweile über eine Vielzahl an Konfigurationsparametern, die eine individuelle Anpassung an die Bedürfnisse des Kunden erlauben.

Im Gegensatz zu der Mehrzahl an durchgeführten Benchmarks von NoSQL-Datenbanksystemen (bspw. [Ava15, GGK+14, GP14]), soll diese Arbeit nicht die Leistungsfähigkeit verschiedener Systeme im direkten Vergleich untersuchen. Vielmehr sollen die Leistungsunterschiede verschiedenartiger Betriebskonfigurationen für einzelne NoSQL-Datenbanksysteme getrennt voneinander untersucht werden. Durch den Verzicht auf den direkten Vergleich können die Systeme grundlegend entsprechend ihrer Fähigkeiten evaluiert werden, ohne dass dabei gleichzeitig zusätzliche Einstellungen vorgenommen werden müssen, die lediglich eine mehr oder weniger gute Vergleichbarkeit zwischen den heterogenen Systemen herstellen sollen. Die somit praxisnäheren Ergebnisse sollen zukünftig Mitarbeiter der ORDIX AG bei der fundierten Technologieberatung Ihrer Kunden unterstützen. Gegenstand der konkreten Untersuchungen sind die populären NoSQL-Datenbanksysteme Couchbase und Apache Cassandra.

1.2. Vorstellung der ORDIX AG

Seit der Gründung im Jahr 1990 hat sich die ORDIX AG als zuverlässiger Partner im Bereich der IT-Dienstleistungen etabliert. Ausgehend von den sechs Standorten Paderborn, Wiesbaden, Köln, Münster, Gersthofen (Augsburg) und Essen unterstützt die ORDIX AG zusammen mit den beiden Tochterunternehmen coniatos und Object Systems ihre Kunden bei der Realisierung ihrer Projekte. [ORD15c]

Das angebotene Leistungsspektrum umfasst Dienstleistungen aus den Bereichen des Projektmanagements, der Entwicklung, des Service und der Schulung [ORD15b]. Kern des Beratungsgeschäfts bilden hierbei Applikations- & Middleware-Technologien, Datenbanken und Unix/Windows-Betriebssysteme. Kernkompetenzen bestehen diesbezüglich vor allem rundum Fragestellungen der Performanz, Hochverfügbarkeit, Virtualisierung und des Backup & Recovery. [ORD15a]

Die Unternehmenskultur ist maßgeblich durch die Philosophie des Wissenstransfers geprägt. Auf der einen Seite spiegelt sich diese in einer kontinuierlichen Aus- und Weiterbildung der eigenen Mitarbeiter wider. Auf der anderen Seite wird das gewonnene Know-how stets mit den Kunden geteilt. Neben dem direkten Austausch im Projektalltag erfolgt dies zusätzlich im Rahmen angebotener Seminare, diverser Vorträge[1] und in Form eines eigenen IT-Magazins namens ORDIX® news. [ORD15c]

1.3. Aufbau der Arbeit

Zu Beginn dieser Arbeit werden in Kapitel 2 die Grundlagen der beiden NoSQL-Datenbanksysteme Couchbase und Apache Cassandra erläutert. Basierend auf der in Kapitel 3 folgenden Betrachtung verschiedener Benchmarking-Frameworks wird für die Evaluierung der Datenbanksysteme eine konkrete Implementierung ausgewählt. Anschließend werden sowohl die daran erfolgten Modifikationen, als auch das Benchmarking-Umfeld und der -Ablauf dargestellt. In Kapitel 4 werden die Resultate der durchgeführten Benchmark-Messungen beschrieben. Abschließend erfolgt in Kapitel 5 eine Zusammenfassung der Ergebnisse dieser Arbeit und eine Beschreibung möglicher aufbauender Untersuchungsaspekte in einem Ausblick.

[1] Beispielsweise bei der alljährlichen Fachkonferenz der DOAG (Deutsche ORACLE-Anwendergruppe).

2. Grundlagen

Auf eine allgemeine Einführung in das Themenumfeld der NoSQL-Technologien wird in anbetracht der gegeben Spezialisierung dieser Arbeit verzichtet. Für eine grundlegende Einführung in diesen Bereich eignet sich das Studium bestehender Literatur (bspw. [SF12, MK13]). Stattdessen liegt der Fokus in diesem Kapitel bei der Einführung in die NoSQL-Datenbankensysteme Couchbase und Apache Cassandra. Aufgrund derer Komplexität beschränkt sich deren Beschreibung auf jene Aspekte, die speziell im Rahmen dieser Arbeit relevant oder dem Bilden eines Gesamtverständnisses dienlich sind.

Eine Betrachtung geeigneter Benchmarks zur Evaluierung von NoSQL-Datenbanksystemen erfolgt dagegen im Zusammenhang mit der Auswahl einer konkreten Implementierung in Kapitel 3.1.

2.1. Couchbase

Unter den NoSQL-Datenbanken gehört Couchbase zu den Document Stores, bietet darüber hinaus aber auch die Funktionalität eines einfachen Key-Value Stores. In seiner derzeitigen Form konzentriert sich Couchbase vor allem auf die Unterstützung interaktiver Webanwendungen. In diesem Zusammenhang werden vor allem die Aspekte der Skalierung, Verfügbarkeit und der Performanz fokussiert. [Cou14, S. 8]

Innerhalb eines Couchbase Clusters verfügen sämtliche Nodes über die gleiche Funktionalität und bearbeiten gleichermaßen die identischen Aufgaben [Cou14, S. 84]. In Kombination mit Verteilung der Daten und der ebenfalls unterstützten Replikationsfähigkeit wird sichergestellt, dass das Datenbanksystem auch beim Ausfall einzelner Nodes vollständig funktionsfähig bleibt.

Für den Zugriff auf Daten des Clusters, sendet ein Client seine Anfrage direkt an den zuständigen Node. Die Information darüber, welche Nodes für welche Keys zuständig sind, wird beim initialen Verbindungsaufbau ausgetauscht. [Cou14, S. 94]

Couchbase wird als Open Source Projekt[2] maßgeblich durch Mitarbeiter der gleichnamigen Couchbase Inc. entwickelt. Diese bietet, neben Support- und Trainingsdienstleistungen, ebenfalls eine kommerzielle Version von Couchbase an.

[2] `https://github.com/couchbase`, zuletzt besucht am 22.03.2015.

2.1.1. Buckets

Innerhalb eines Couchbase Clusters erfolgt die Organisation der Daten mit Hilfe von sogenannten Buckets. Hierbei handelt es sich um virtuelle Container, welche die eigentlichen Key-Value Datensätze verwalten. Je nach Bedarf kann es sich beim Value um eine einfache Zeichenkette oder um ein JSON Dokument handeln. Derzeit verfügt Couchbase über zwei verschiedene Bucket Typen. [Cou14, S. 89]

Beim Memcached Bucket handelt es sich um einen In-Memory Cache zur Zwischenspeicherung von Key-Value Paaren. Infolge der fehlenden Persistierungs- und Replikationsunterstützung ist hierbei keine Dauerhaftigkeit garantiert. Zudem ist die Größe des Values auf ein Megabyte begrenzt. Verwendet wird diese Variante primär für den parallelen Einsatz mit einer separaten relationalen Datenbank. Durch das Zwischenspeichern häufig benötigter Daten, soll hierbei die Anzahl erforderlicher Anfragen an das relationale Datenbanksystem reduziert werden. [Cou14, S. 89]

Im Vergleich hierzu unterstützt der bei Couchbase standardmäßige "Couchbase Bucket" die Persistierung und Replikation von Daten. Des Weiteren können die Daten zwischen den Nodes mittels einer Rebalancing Funktionalität neu verteilt werden. Dies stellt sicher, dass sämtliche Nodes auch gleichmäßig ausgelastet werden können, sofern neue Nodes nachträglich dem Cluster hinzugefügt oder bestehende entfernt werden. Bei diesem Bucket Typ ist die Value-Größe auf 20 Megabyte begrenzt. [Cou14, S. 89]

Die Verteilung der Daten eines Buckets über die Nodes erfolgt mit Hilfe von vBuckets. Diese werden eindeutig einem Node zugeordnet und verwalten jeweils einen festen Wertebereich von Keys bzw. Document IDs. Greift ein Client auf Daten des Clusters zu, berechnete er mittels eines Hashing-Algorithmus für den betroffenen Key den zugehörigen vBucket. Anschließend wird unter der Verwendung einer Mapping-Tabelle ermittelt, welcher konkrete Node diesen verwaltet (siehe Abbildung 1). Daraufhin sendet der Client seine Anfrage direkt an den zuständigen Node. [Cou14, S. 94]

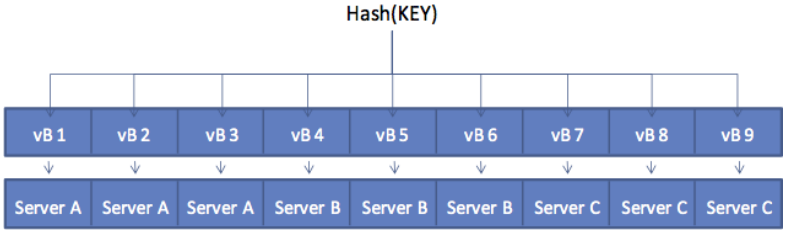


Abbildung 1: Couchbase: vBuckets [Cou14, S. 94]

2.1.2. Caching & Storage

Für die Verarbeitung der Client-Anfragen werden sämtliche der dafür erforderliche Daten aus einer internen Caching-Struktur gelesen oder in ihr gespeichert. Für jeden Bucket wird individuell ein separater Teil des für Couchbase verfügbaren Hauptspeichers zugewiesen.

Die Persistierung von neuen bzw. geänderten Daten erfolgt in der Regel asynchron. Hierzu werden die Daten einerseits in einer Disk Write Queue zwischengespeichert. Andererseits wird dem Client unmittelbar die erfolgreiche Verarbeitung der Schreibanfrage bestätigt. [Cou14, S. 97, 190] Infolgedessen kann eine Vielzahl von Dokumenten in einem zusammenhängenden sequentiellen Schreibvorgang auf dem Sekundärspeicher persistiert werden. Dieses Vorgehen erzielt einen höheren Durchsatz, als wenn die Persistierung für einzelne Bestätigungsrückmeldungen unterbrochen werden müsste. In der Konsequenz bedeutet dieses Vorgehen jedoch auch, dass sämtliche Daten, welche sich noch in der Disk Write Queue befinden, bei einem Serverausfall verloren gehen[3]. Gegenüber diesem Standardverhalten erlaubt die, von Couchbase zur Verfügung gestellte, API jedoch auch explizit auf die Durchführung der Persitierung zu warten.

Für Anwendungsszenarien bei welchen Couchbase eine größere Menge an Daten verwalten soll, als der interne Cache aufnehmen kann, existieren zwei unterschiedlichen Verfahren zur Verwaltung der im Cache gehaltenen Daten. Diese unterscheiden sich maßgeblich im Umfang der Daten, die bei zunehmender Auslastung des Caches entfernt werden. [Cou14, S. 104]

Bei der standardmäßigen Value Ejection werden lediglich die eigentlichen Values bzw. Dokumente aus dem Cache entfernt. Sämtliche Keys und die dazugehörigen Metadaten bleiben dagegen im Hauptspeicher erhalten. Im Gegensatz dazu werden bei der Full Ejection sowohl die Values, als auch deren Keys und die betreffenden Metadaten aus dem Cache entfernt. [Cou14, S. 104]

Unabhängig vom konkreten Typ wird der Eviction-Prozess Schwellwert basiert ausgeführt. Erreicht die Auslastung des Caches 85 % (*mem_high_wat*) werden solange Daten aus dem Cache entfernt, bis die Auslastung auf 75 % (*mem_low_wat*) sinkt (siehe folgende Abbildung 2). Die Auswahl der Daten erfolgt auf Basis der Not recently used Strategie. Entsprechend werden bevorzugt Datensätze entfernt,

[3] Sofern keine Replikation erfolgt

auf die in jüngster Vergangenheit weniger häufig zugegriffen wurde. Hierbei wird zudem sichergestellt, dass nur bereits persistierte Daten aus dem Cache entfernt werden. [Cou14, S. 104 - 105]

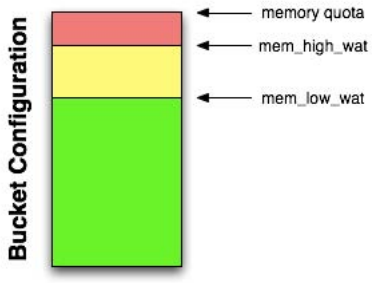


Abbildung 2: Couchbase: Cache Eviction Schwellwerte [Cou14, S. 105]

2.1.3. Compaction

Mit zunehmender Anzahl an durchgeführten schreibenden Operationen ist es erforderlich, die zugrundeliegenden Datenbankdateien eines Buckets zu reorganisieren bzw. zu verdichten. Diese Maßnahme ist notwendig, da bei der Persistierung von Daten keine Update-in-Place Strategie verwendet wird. Stattdessen werden die Datenbankdateien fortwährend ergänzt, wodurch deren Größe und Fragmentierung langfristig steigt. [Cou13]

Dem entgegen wirkt der Compaction Prozess. Hierbei werden neue Datenbankdateien erstellt, in die nur aktuelle Daten übernommen werden. Sobald die Comapction abgeschlossen ist, werden die neuen Datenbankdateien als aktuell betrachtet und entsprechend bei neuen Änderungen fortgeschrieben. Abschließend werden die veralteten Datenbankdateien gelöscht. Die nachfolgende Abbildung 3 illustriert die letztendliche Auswirkung. [Cou14, S. 141]

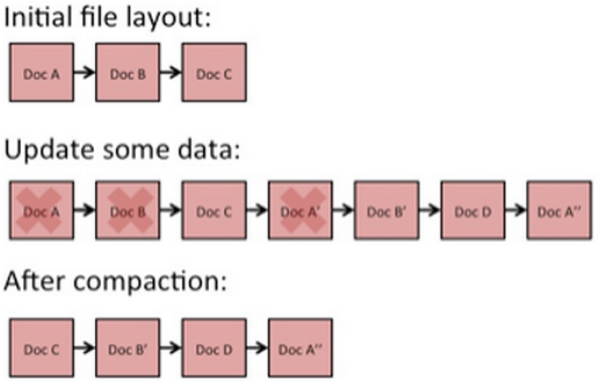


Abbildung 3: Couchbase: Compaction [Cou13]

In der Regel wird der Compaction Prozess automatisiert ausgeführt. Zum einen kann dies Schwellwert basiert anhand des Fragmentierungsgrades eines Buckets erfolgen. Zum anderen aber auch zeitgesteuert, wobei ein konkretes Zeitfenster definiert werden kann, auf welches sich die Compaction-Ausführung beschränkt. [Cou14, S. 142 - 143]

2.1.4. Replikation

Couchbase unterstützt individuell pro Bucket von einer einfachen bis zu einer dreifachen Replikation der Daten. Jeder Node eines Clusters hält dabei sowohl aktive Daten als auch Replikate von anderen Nodes. Sobald eine schreibende Anfrage eintrifft und deren Änderung im Cache übernommen ist, wird den entsprechend weiteren Nodes eine Kopie des geänderten Datensatzes zugesandt (siehe Abbildung 4). [Cou14, S. 139]

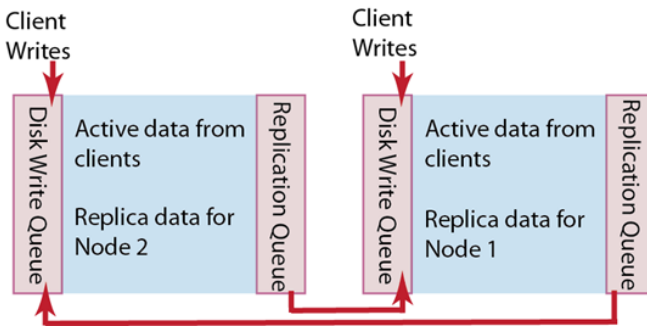


Abbildung 4: Couchbase: Replikation [Cou14, S. 139]

In der Regel erfolgt die Replikation asynchron zur schreibenden Client-Anfrage. Je nach Bedarf kann der Client, mit einer entsprechenden Parametrisierung der Anfrage, ein abweichendes Verhalten bewirken. Hierbei kann zunächst definiert werden, wie vielen Nodes die Kopie erfolgreich zugesandt bekommen haben müssen, bevor der Client die Anfrage bestätigt bekommt. Zusätzlich kann hierfür definiert werden, wie viele der Nodes den entsprechenden Datensatz auch auf dem Sekundärspeicher persistiert haben müssen.

2.2. Apache Cassandra

Cassandra wurde ursprünglich von Facebook [LM10] mit dem Fokus auf Zuverlässigkeit und hochgradige Skalierung als verteiltes Datenbanksystem entwickelt. Aktuell wird (Apache) Cassandra unter der Leitung der Apache Software Foundation weiterentwickelt. Der größte Teil der veröffentlichten Codeänderungen erfolgt dabei durch Mitarbeiter von DataStax Inc., welche mit DataStax Enterprise eine kommerzielle Version von Cassandra anbietet.

Jeder Node eines Clusters besitzt grundsätzlich die gleiche Funktionalität und bearbeitet gleichermaßen die identischen Aufgaben. Infolge des somit vermiedenen Single Point of Failures bleibt die Verfügbarkeit selbst bei Ausfällen einzelner Nodes erhalten. Für den Zugriff auf Daten des Clusters, sendet ein Client seine Anfrage an einen (beliebigen) Node. Dieser übernimmt daraufhin die Koordination zur Verarbeitung der Anfrage. Sofern hierfür Daten von weiteren Nodes benötigt werden, vollführt dieser in der Rolle des sogenannten Coordinator Nodes den notwendigen Austausch und sendet dem Client das Resultat zurück. [Dat15b, S. 11]

Cassandra zählt zu den zeilenorientierten Datenbanksystemen. Konzeptionell erfolgt die Organisation der Datensätze über Keyspaces[4], welche wiederum Tabellen beinhalten. Tabellen selbst untergliedern die darin gespeicherten Datensätze dagegen mit Spalten. Üblicherweise erfolgt der Zugriff auf die Daten des Cluster mittels Cassandras eigener Anfragesprache CQL[5]. [Dat15b, S. 10 - 11]

In den nachfolgenden Unterkapiteln folgte eine Beschreibung des Verhaltens bei der Verarbeitung schreibender und lesender Anfragen. Zudem werden verschiedene Konfigurationsmöglichkeiten hinsichtlich der Compaction, Komprimierung, Replikation und des Write-Ahead Loggings erläutert.

2.2.1. Verarbeitung schreibender Anfragen

Im Rahmen der Verarbeitung schreibender Anfragen vollführt Cassandra eine Reihe verschiedener Arbeitsschritte (siehe Abbildung 5). Mit dem Eintreffen einer schreibenden Operation erfolgt zunächst ein Eintrag im Commit Log, wodurch die Dauerhaftigkeit der Datenänderung sichergestellt werden soll. Anschließend wird die Änderung in einer sogenannten Memtable zwischengespeichert. Je nach konfiguriertem Schwellwert wird die im Hauptspeicher befindliche Memtable früher

[4] Vergleichbar mit dem Begriff der Datenbank aus dem Kontext relationaler Datenbanksysteme.
[5] Cassandra Query Language (CQL)

oder später in Form einer Sorted String Table (SSTable) auf den Sekundärspeicher geschrieben. Hierbei gilt zu beachten, dass einmal geschriebene SSTables nicht wieder verändert werden und nachfolgende Datenänderungen in weiteren SSTables persistiert werden. Bei jeder Neuanlage einer SSTable wird zusätzlich ein Partition Index und ein Bloom Filter angelegt. Der Partition Index beinhaltet eine Liste der in der SSTable enthaltenen Keys und deren Startpositionen innerhalb der Datei. Ein Auszug dieses Indizes wird in Form der sogenannten Partition Summary im Hauptspeicher gehalten. [Dat15b, S. 62 - 63] Mit Hilfe des Bloom Filters kann berechnet werden, wie hoch die Wahrscheinlichkeit ist, dass die jeweilige SSTable den gesuchten Datensatz enthält. Infolgedessen können unnötige Lesezugriffe vermieden werden. [Dat15b, S. 68]

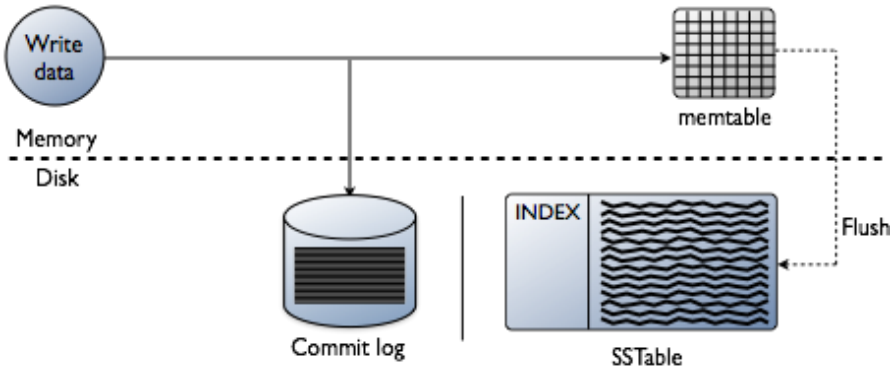


Abbildung 5: Cassandra: Verarbeitung schreibender Anfragen [Dat15b, S. 63]

Die Organisation der entsprechenden Datenbankdateien erfolgt, indem für sämtlich erstellte Keyspaces ein eigenes Verzeichnis angelegt wird. Unterhalb des jeweiligen Keyspace Verzeichnisses werden wiederum separate Ordner für die jeweils angelegten Tabellen erzeugt, welche ihrerseits die eigentlichen Datenbankdateien beinhalten. [Dat15b, S. 62].

Mit voranschreitender Menge an geschriebenen Datensätzen wächst kontinuierlich die Anzahl der erforderlichen SSTables und folglich auch der dafür benötigte Speicherbedarf. Um letztendlich die Menge der zugrundeliegenden SSTables zu reduzieren, wird die sogenannte Compaction durchgeführt. Basierend auf den bestehenden SSTables wird eine komplett neue SSTable erstellt, woraufhin die veralteten SSTables gelöscht werden. Während der Datenübernahme in die neue SSTable wird jedoch stets nur die aktuellste Version einen Datensatzes übernommen und ggf. als gelöschte Datensätze übersprungen. Die nachfolgende Abbildung 6 illustriert das beschriebene Verhalten. [Dat15b, S. 64]

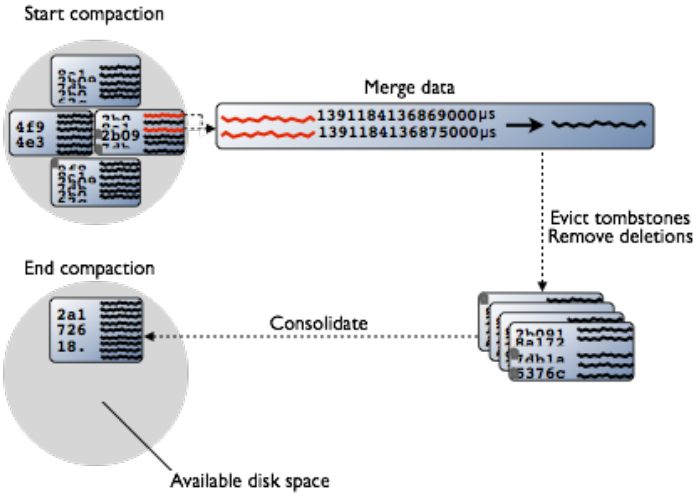


Abbildung 6: Cassandra: Compaction [Dat15b, S. 64]

2.2.2. Verarbeitung lesender Anfragen

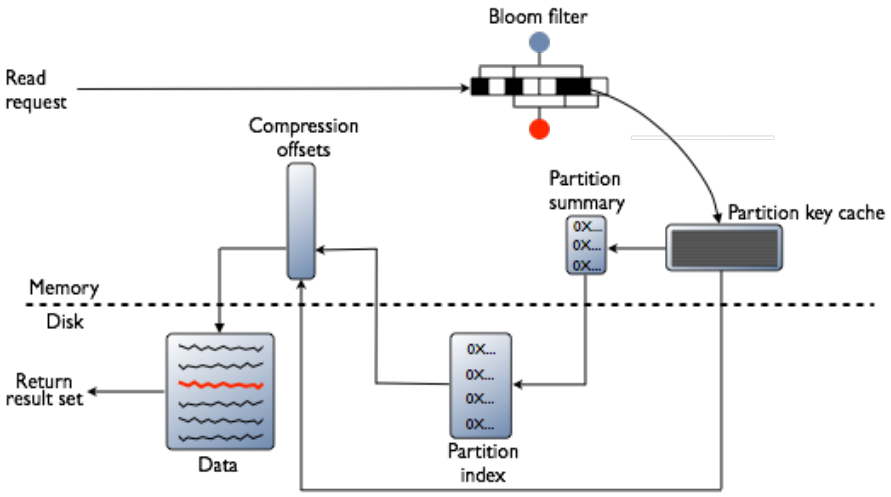


Abbildung 7: Cassandra: Verarbeitung lesender Anfragen [Dat15b, S. 69]

Wie anhand von Abbildung 7 dargestellt, wird mit dem Eintreffen einer lesenden Anfrage zunächst bei sämtlichen Bloom Filtern überprüft, wie hoch die Wahrscheinlichkeit ist, dass die jeweils zugehörige SSTable gesuchte Daten enthält. Sofern hierbei nicht ausgeschlossen wird, dass relevante Daten durch eine SSTable gehalten werden, erfolgt eine Prüfung im Partition Key Cache[6]. [Dat15b, S. 68]

Wird ein entsprechender Eintrag gefunden, erfolgt das Nachschlagen des notwendige Offset in der Compression Offset Map. Anschließend kann gezielt in der SSTable

[6]Der Partition Key Cache hält Einträge aus den jeweiligen Partition Indizes zu den zugrundeliegenden SSTables von einer Tabelle [Dat15b, S. 110].

gelesen werden. Wird stattdessen kein Eintrag im Partition Key Cache gefunden, findet dagegen mit Hilfe der Partition Summarys eine Suche nach dem zuständigem Indexeintrag auf dem Sekundärspeicher statt. Nachdem dieser gefunden und gelesen wurde, wird wiederum in der Compression Offset Map der notwendige Offset nachgeschlagen. Abschließend werden die angeforderten Daten aus der SSTable gelesen. [Dat15b, S. 68]

Während des Lesens muss zusätzlich berücksichtigt werden, dass die neusten Daten evtl. noch nicht persistiert wurden und sich ausschließlich im Hauptspeicher in der Memtable befinden. Des Weiteren können in den verschiedenen SSTables verschiedene Daten zu einem angeforderten Datensatz vorhanden sein, sodass deren Teile noch vor dem Zurücksenden an den Client zusammengeführt werden müssen (siehe Abbildung 8). [Dat15b, S. 68, 70]

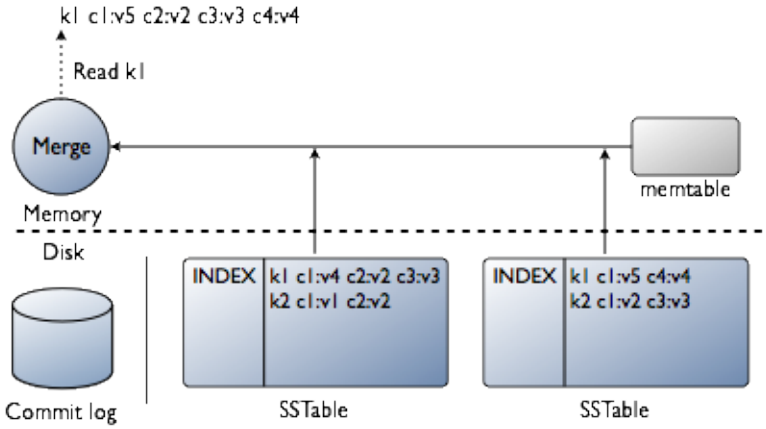


Abbildung 8: Cassandra: Verarbeitung lesender Anfragen II [Dat14]

Bezüglich des hier beschriebenen standardmäßigen Verhalten beim Lesen von Daten ist hervorzuheben, dass Cassandra keine Inhaltsdaten in eigenen Cache-Strukturen zur Beschleunigung der Abfragen vorhält. Stattdessen werden die Inhalte indirekt über den betriebssystemeigenen Page Cache im Hauptspeicher vorgehalten [Dat15b, S. 116]. Es bleibt jedoch zu erwähnen, dass Cassandra über eine separate Row Cache Implementierung verfügt, welche standardmäßig jedoch nicht verwendet wird [Dat15b, S. 70, 110].

2.2.3. Compaction

Wie bereits in Kapitel 2.2.1 beschrieben, ist es bei Cassandra angesichts der zugrundeliegenden Append-Only-Strategie beim Persistieren von Daten erforderlich, dass mit voranschreitender Befüllung und Änderung von Daten einer Tabelle, deren zugehörigen SSTables bereinigt und zusammengefasst werden.

Hierfür besitzt Cassandra bereits drei unterschiedliche Strategien, wobei bei Bedarf auch eigene Implementierungen verwendet werden können. In der Default-Konfiguration wird die sogenannte Size Tiered Strategie verwendet. Sofern hierbei eine Mindestmenge von relativ gleichgroßen SSTables vorhanden ist, werden diese zu einer neuen SSTable zusammengefasst. [Dat15a, S. 58]

Mittels der folgenden Abbildung 9 ist dieses Verhalten bei wachsender Anzahl von SSTables schematisch dargestellt. Jeder grüne Balken symbolisiert dabei eine SSTable, wobei die Höhe des Balken die Größe der SSTable widerspiegelt. Der Pfeil zeigt die Veränderung infolge der Compaction. Über den horizontalen Verlauf wird dargestellt, wie sich die Menge der vorhandenen SSTables verändert. [Ell11]

Abbildung 9: Cassandra: Size Tiered Compaction [Ell11]

Bei der Leveled Strategie werden prinzipiell kleinere SSTables[7] persistiert. Diese werden in verschiedenen Level gruppiert, wobei jedes Level um das zehnfache größer ist als das vorherige. Hierbei wird sichergestellt, dass innerhalb eines Levels jeweils nur eine SSTable Daten zu einem bestimmten Key enthält. Dies hat den Vorteil, dass lesende Abfragen beschleunigt werden können, da nur noch pro Level eine SSTable relevante Daten für einen bestimmten Key enthält. [Dat15a, S. 58]

Das entsprechende Verhalten bei wachsender Anzahl von SSTables ist in Abbildung 10 skizziert. Jede Farbe stellt dabei ein anderes Level dar (Grün Level 0, Blau Level 1, ...). [Ell11]

[7] Standardmäßig beträgt die Größe einer SSTable 5 MB, kann jedoch je nach Bedarf angepasst werden [Dat15a, S. 58].

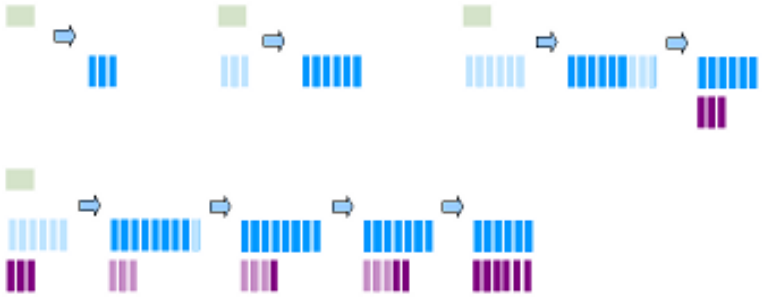


Abbildung 10: Cassandra: Leveled Compaction [Ell11]

Bei der dritten implementierten Strategie handelt es sich um die sogennante Date Tiered Strategie. Hierbei werden sämtlichen Daten in einer SSTable zusammengefasst, die innerhalb des gleichen Zeitfensters erfasst wurden. Standardmäßig beträgt dieses vier Stunden, kann aber je nach Bedarf angepasst werden. [Dat15a, S. 58]

Die Konfiguration der zu verwendenden Compaction Strategie erfolgt auf Tabellen-Ebene.

2.2.4. Komprimierung

Cassandra verfügt bereits über drei unterschiedliche Verfahren zur Komprimierung der auf dem Sekundärspeicher persistierten SSTables. In der Default-Konfiguration wird die LZ4 Kompression verwendet [Dat15a, S. 56]. Alternativ stehen noch Implementierungen des Snappy und des Deflate Komprimierungsalgorithmus zur Verfügung. Gemäß der Dokumentation [Dat15a, S. 58 - 59] erfolgt bei LZ4 die Dekomprimierung am schnellsten, gefolgt von Snappy und abschließend von Deflate. Mit sinkender Performance steigt jedoch die Komprimierungseffektivität der Algorithmen. Allerdings kann die erhöhte Kompression von Deflate und Snappy deren verminderte Geschwindigkeit nicht ausgleichen.

Die Konfiguration des zu verwendenden Kompressionsverfahrens erfolgt auf Tabellen-Ebene.

2.2.5. Write-Ahead Logging

Wie bereits in Kapitel 2.2.1 erwähnt, erfolgt bei der Verarbeitung schreibender Operationen neben der Zwischenspeicherung der Daten in der Memtable ein Eintrag im Commit Log. Dies soll sicherstellen, dass im Falle eines Serverabsturzes keine noch nicht persistierten Daten dauerhaft verloren gehen. Besondere Relevanz

besitzt hierbei die verwendete Strategie zur Persistierung des zugrundeliegenden Commit Logs. [Dat15a, S. 72]

Beim standardmäßigen Periodic Sync erfolgt die Speicherung des Commit Logs aus dem Hauptspeicher auf den Sekundärspeicher alle 10 Sekunden[8]. Hierbei gilt zu beachten, dass dem Client, bei der Ausführung von schreibenden Operationen unmittelbar nach dem Eintrag in die Memtable und in den Commit Log, ein erfolgreiches Schreiben bestätigt wird. [Dat15a, S. 86] Folglich können in dieser Konfiguration mehr oder weniger Daten, je nach Größe des zugrundeliegenden Intervalls, dauerhaft auf einem Node verloren gehen.

Demgegenüber erfolgt beim Batch Sync keine Bestätigung an den Client, solange nicht der entsprechende Commit Log dauerhaft auf dem Sekundärspeicher gespeichert ist [Dat15a, S. 86]. Der standardmäßige Intervall beträgt bei dieser Konfiguration 50 Millisekunden [Apa15]. Folglich ist bei dieser Variante ein Datenverlust infolge eines Serverabsturzes[9] ausgeschlossen.

Die Konfiguration der zu verwendenden Persistierungsstrategie erfolgt auf Node-Ebene. Für jeden Keyspace kann jedoch wiederum auch einzeln die Verwendung des Commit Logs deaktiviert werden.

2.2.6. Replikation

Die in Cassandra implementierte Replikationsfunktionalität dient maßgeblich zur Gewährleistung der Fehlertoleranz und Zuverlässigkeit. Infolge des Aspektes, dass bei dieser keine Unterscheidung zwischen der Master und Slave bzw. Primary und Secondary Instanz eines Datensatzes erfolgt, wird in der Terminologie von Cassandra jede Instanz als Replikat bezeichnet. Dies erfolgt selbst dann, wenn nach klassischem Verständnis keine Replikation stattfindet, sondern im gesamten Cluster nur eine einzige (originale) Instanz eines Datensatzes gehalten wird. [Dat15b, S. 12, 14, 16]

Die letztendlich verwendete Anzahl von Instanzen eines Datensatzes wird auf Ebene des Keyspaces mittels des Replikationsfaktors konfiguriert. Ein Replikationsfaktor von eins bedeutet, dass nur eine einzelne Instanz des Datensatzes im Cluster gehalten wird und folglich keine Kopie erzeugt wird. Erst mit höherem Re-

[8] Persistierungsintervall kann je nach Bedarf angepasst werden.
[9] Sofern hierbei keine Beschädigung des Sekundärspeichers oder dessen Dateisystems erfolgt.

plikationsfaktor werden mehrere Instanzen des identischen Datensatzes im Cluster gehalten. [Dat15b, S. 16]

Während des Zugriffs auf die Daten einer Tabelle kann Client-seitig über die Konfiguration eines Konsistenz-Levels individuell bei jedem Zugriff festgelegt werden, wie viele der verfügbaren Replika-Nodes die Anfrage beantwortet haben müssen, bevor der Coordinator Node dem Client entsprechend antwortet. Bei schreibenden Anfragen wird jedem Replika-Node diese auch immer weitergeleitet. Dagegen stellt sich bei lesenden Anfrage ein differenziertes Verhalten dar. Während bei den Konsistenz-Level QUORUM und All immer alle Replika-Nodes angefragt werden, erfolgt bei den Konsistenz-Level ONE, TWO und THREE lediglich eine Anfrage bei der entsprechenden Anzahl an Replika-Nodes. [Dat15b, S. 73 - 77]

Die nachfolgende Tabelle 1 zeigt diesbezüglich eine Auflistung der im Rahmen dieser Arbeit betrachteten Konsistenz-Level.

Konsistenz-Level	**Beschreibung**
ALL	Lesende und schreibende Anfragen erfolgen an alle Replika-Nodes. Das Ergebnis von sämtlichen Replika-Nodes wird abgewartet, bevor eine Antwort an den Client erfolgt.
ONE	Lesende Anfragen erfolgen bei einem Replika-Node, schreibende dagegen bei allen. Antwort eines Replika-Nodes wird abgewartet, bevor dem Client geantwortet wird.
TWO	Lesende Anfragen erfolgen bei zwei Replika-Nodes, schreibende dagegen bei allen. Antwort von zwei Replika-Nodes wird abgewartet, bevor dem Client geantwortet wird.
THREE	Lesende Anfragen erfolgen bei drei Replika-Nodes, schreibende dagegen bei allen. Antwort von drei Replika-Nodes wird abgewartet, bevor dem Client geantwortet wird.
QUORUM	Lesenden und schreibende Anfragen erfolgen an alle Replika-Nodes. Das Ergebnis von der Mehrheit aller Replika-Nodes wird abgewartet, bevor eine Antwort an den Client erfolgt.

Tabelle 1: Cassandra: Konsistenz-Level [Dat15b, S. 73 - 76]

Es gilt zu beachten, dass es sich hierbei lediglich um eine Teilmenge der verfügbaren Konfigurationen handelt und insbesondere für den Betrieb mit mehr als einem Data Center weitere Konsistenz-Level existieren.

Zu der Behebung von ggf. bestehenden Inkonsistenzen wird bei der Verarbeitung von lesenden Anfragen ein Read-Repair durchgeführt. Hierbei sendet der Coordinator Node die Anfrage, unabhängig vom Konsistenz-Level, an sämtlich betroffene Replika-Nodes. Während dem Client, je nach Konsistenz-Level geantwortet wird, erfolgt die eigentliche Read-Repair Verarbeitung im Hintergrund. Sofern dabei eine Inkonsistenz festgestellt wird, setzt sich der Datensatz mit dem aktuellsten Zeitstempel durch. Mit einer standardmäßigen Wahrscheinlichkeit von 10 % erfolgt dieser Vorgang zusätzlich bei sämtlichen Konsistenz-Leveln, welche nicht per se eine Antwort von allen Replika-Nodes anfordern. [Dat15b, S. 77][Dat15a, S. 56]

Mittels der folgenden Abbildung 11 ist beispielhaft dargestellt, wie ein Client den Datensatz A unter Verwendung des Konsistenz-Levels Quorum liest. Der zugrundeliegende Replikationsfaktor beträgt hierbei drei.

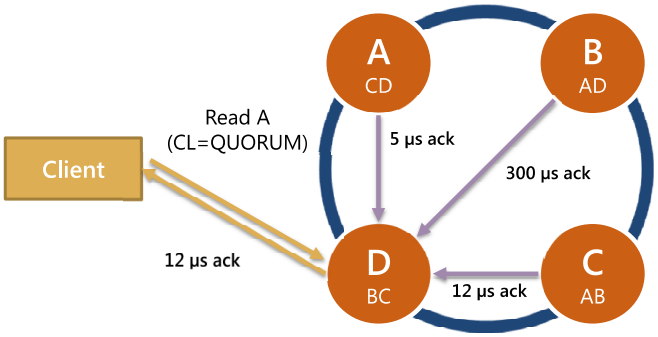


Abbildung 11: Cassandra: Lesender Zugriff unter Verwendung des Konsistenz-Levels Quorum [Til14, S. 24]

3. Benchmark-Vorbereitung

Dieses Kapitel beschreibt die getroffenen Vorbereitungen zum Benchmarking der NoSQL-Datenbanksysteme. Zu Beginn erfolgt eine Betrachtung verschiedener Benchmark Frameworks und darauf basierend eine Auswahl einer konkreten Implementierung. Im Anschluss daran werden die durchgeführten Modifikationen an dem gewählten Framework erläutert. Abschießend erfolgt eine Darstellung der Benchmark-Umgebung und des Benchmark-Ablaufs.

3.1. Benchmark Frameworks

Bedingt durch den allgemeinen Hype um Big Data Thematiken erfahren gleichzeitig auch NoSQL-Technologien immer größere Aufmerksamkeit. Dieser Umstand ist darauf zurückzuführen, dass NoSQL-Datenbanksysteme auch maßgeblich zur Realisierung von Big Data Lösungen beitragen. So wie die verschiedenen Technologien kombiniert werden um Big Data Lösungen zu realisieren, werden sie auch beim Benchmarking berücksichtigt. Häufig stehen jedoch speziell im Kontext von Big Data orientierten Benchmarks analytische und verarbeitende Methoden im Vordergrund. Beispielsweise existieren mit GridMix3 [Cha12], PigMix[10], HiBench [HHD+10] und TPCx-HS [Tra15] eine Reihe von MapReduce Benchmarks, welche sich jedoch maßgeblich auf die Evaluierung von Apache Hadoop und dessen Erweiterungen konzentrieren. Dementsprechend bleiben die alternativen MapReduce Implementierungen von NoSQL-Datenbanksystemen, wie bspw. der von MongoDB, weitestgehend unberücksichtigt. Neben derart fokussierten Benchmarks existieren noch Benchmark Suiten, welche Sammlungen verschiedener Einzelbenchmarks bilden. Ein entsprechender Repräsentant ist der BigDataBench [WZL+14], welcher 19 unterschiedliche Benchmarks beinhaltet und somit das Benchmarking von verschiedenen Anwendungstypen, Datentypen, Anwendungsszenarien und Software Stacks ermöglicht. Hierbei ist jedoch festzustellen, dass der NoSQL relevante Benchmark-Teil auf einem bereits bestehenden Benchmark Framework[11] basiert und somit auch eigenständig verwendet werden kann.

Bezüglich der expliziten Evaluierung von NoSQL-Datenbanksystemen konnten insbesondere die vier unterschiedlichen Benchmark Frameworks YCSB [CST+10], BG [BG13], YorkMDE [SWK+14] und Primeball [FAC+14] als potentiell geeignet iden-

[10] `https://cwiki.apache.org/confluence/display/PIG/PigMix`, zuletzt besucht am 02.03.2015.

[11] BigDataBench verwendet für seine (NoSQL-)Datenbank orientierten Tests das YCSB Framework.

tifiziert werden. Sämtliche der vier Benchmarks fokussieren den Einsatzbereich von Online Services. Analytische Aspekte oder Faktoren, welche bspw. die Batch-Verarbeitung betreffen, bleiben dagegen weitestgehend unberücksichtigt. Besonders hervorzuheben ist zudem, dass die Benchmarks BG, YorkMDE und Primeball nicht auf vollständig generischen Tests basieren, sondern die (NoSQL-)Datenbanksysteme innerhalb eines konkreten Anwendungsszenarios evaluieren.

Im Folgenden werden die wesentlichen Aspekte der drei Benchmark Frameworks BG, YorkMDE und YCSB beschrieben. In Ermangelung einer lauffähigen Implementierung des Primeball Benchmarks wird auf seine weitere Beschreibung verzichtet. Anschließend erfolgt eine kurze Bewertung der vorgestellten Benchmarks und die Auswahl eines Frameworks zur Verwendung im Rahmen dieser Arbeit.

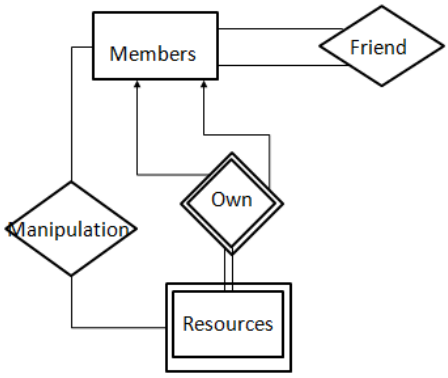
3.1.1. BG

Der von Barahmand und Ghandeharizadeh [BG13] vorgestellte BG Benchmark fokussiert das Benchmarking unterschiedlicher Datenbanksysteme im Anwendungskontext sozialer Netzwerke. In diesem Zusammenhang wird das typische Nutzerverhalten mit Hilfe einer Reihe von vordefinierten Nutzerprofilen simuliert, welche sich durch unterschiedliche Nutzerinteraktionen auszeichnen. Eines der Ziele ist hierbei, das Leistungsvermögen von Datenbanksystemen hinsichtlich verschiedener physischer Datenmodelle zu evaluieren. Ein wesentlicher Aspekt liegt bei diesen Untersuchungen in der Detektion von unvorhergesehenen Ergebnissen. Dabei handelt es sich entweder um ungültige, veraltete oder inkonsistente Daten, die vom Datenbanksystem an den Client zurückgeliefert werden. [BG13]

Beim BG Framework wird bezüglich der bereitgestellten Metriken zwischen dem Social Action Rating (SoAR) und dem Socialites Rating unterschieden. Die Grundlage beider Metriken bildet ein parametrierbares Service Level Agreement (SLA). Dieses definiert, wie viel Prozent der durchgeführten Operationen mit einer bestimmten maximalen Latenz erfolgen müssen und wie groß der Anteil der Operationen sein darf, welcher innerhalb einer fest definierten Zeit in unvorhergesehenen Ergebnissen resultiert. Die SoAR Metrik beschreibt den höchsten Durchsatz an durchgeführten Operationen pro Sekunde, welcher unter Einhaltung des SLA maximal erzielt wird. Das Socialites Rating beschreibt dagegen die höchste Anzahl an Threads die unter Einhaltung der SLA parallel verwendet werden kann. [BG13]

3.1.1.1. Datenmodell

Zentraler Ausgangspunkt einer jeden Evaluierung auf Basis dieses Frameworks bildet das vorgegebene konzeptionelle Datenmodell (siehe Abbildung 12(a)).



(a) BG's konzeptionelles Datenmodell

```
Members:{
    "userid" : ""
    "username": ""
    "pw": ""
    "firstname": ""
    "lastname": ""
    "gender" : ""
    "dob" : ""
    "jdate" : ""
    "ldate" : ""
    "address": ""
    "email" : ""
    "tel" : ""
    "imageid":""
    "thumbnailid":""
    "pendingFriends":[]
    "confirmedFriends":[]
    }
```

```
Resources:{
    "rid" : ""
    "creatorid" : ""
    "walluserid": ""
    "type" : ""
    "body": ""
    "doc": ""
    "manipulation":{
        "mid": ""
        "modifierid": ""
        "type": ""
        "content": ""
        " timestamp": ""
    }
}
```

```
GridFSImages.Files.: {
    "id" : ""
    "length":""
    "chunkSize" :""
    "uploadDate" : ""
    "md5" : ""
}

GridFSImages.Chunks: {
    "id"
    "files_id" :
     "n" :
    "data"
}
```

(b) Bsp. für ein dokumentenbasiertes Datenmodell

```
Members(userid,username,pw,lastname,gender,dob,jdate,ldate,address,email,  profileImage, thumbnail)
Friends(userid1,userid2,status)
Resources(rid, creatorid, wallUserid, body, doc)
Manipulation(mid, modifierid, rid, resourcecreatorid, timestamp,type,content)
```

(c) Bsp. für ein relationales Datenmodell

Abbildung 12: BG: Datenmodellierung [BG13]

Dabei obliegt es dem Entwickler/Anwender in welcher konkreten Form das konzeptionelle Datenmodell für das jeweilige Datenbanksystem realisiert und evaluiert wird. Entsprechende Beispiele können für ein dokumentenbasiertes und relationales Datenmodell den Abbildungen 12(b) und 12(c) entnommen werden. Zudem ist es möglich für das gleiche Datenbanksystem verschiedene Modellierungsansätze zu evaluieren, sodass bspw. untersucht werden kann, wie sich die Verwendung von eingebetteten Strukturen gegenüber deren expliziter Vermeidung verhält. [BG13]

Bedingt durch die jeweils individuelle Modellierung der Datenstruktur ist es entsprechend erforderlich, die vom Framework vorgesehenen Nutzerinteraktionen spezifisch zu implementieren. Insgesamt haben Barahmand et al. elf typische Operationen sozialer Netzwerke[12] identifiziert und für den Benchmark standardisiert. Zu diesen typischen Operationen zählt unter anderem, sich das eigene Profil anzuschauen, seine Freunde aufzulisten, eine Freundschaftsanfrage zu versenden bzw.

[12]Basierend auf der Analyse von Facebook, Google+, Twitter, LinkedIn, YouTube, FourSquare, Delicious, Academia.edu und Reddit.com.

eine eingehende Freundschaftsanfrage entweder zu akzeptieren oder zurückzuweisen. [BG13]

Um das Anwenderverhalten sozialer Netzwerke über einen definierten Zeitraum hinweg nachzuahmen, besitzt das BG Framework acht unterschiedliche Session-Profile. Für jedes dieser Session-Profile ist eine eigene Abfolge von Operationen definiert, wobei diese nicht unmittelbar nacheinander ausgeführt, sondern durch Wartezeiten unterbrochen werden. Startpunkt einer jeden Session ist die Betrachtung des eigenen Benutzerprofils und das anschließende Betrachten der aktuellsten Posts anderer Mitglieder auf der eigenen Seite. Je nach zugrundeliegendem Session-Profil endet die Aktivität eines Nutzers hier bereits[13] oder wird durch eine Reihe weitere Aktionen fortgeführt. So wird beispielsweise bei der Session *ThawFrdshipSession* anschließend die eigene Freundesliste angefordert, ein Freund zufällig ausgewählt, die Freundschaft beendet und anschließend erneut die eigene Freundesliste angefordert. [BG13]

3.1.1.2. Architektur

Das BG Framework besteht im Wesentlichen aus den drei Komponenten *BG Visualisation Deck*, *BGCoord* und dem *BGClient*. Das *BG Visualisation Deck* stellt die Schnittstelle zum Anwender dar und ermöglicht die Konfiguration von Benchmarks und deren Monitoring. Die zentrale Koordinierungskomponente *BGCoord* ist für die Ausführung der *BGClients* zuständig und empfängt während der Benchmark-Ausführung kontinuierlich Metriken von den *BGClients*. Entsprechende Metriken werden Client-übergreifend aggregiert und anschließend an das *BG Visualisation Deck* weitergegeben. Die vom *BGCoord* gestarteten *BGClients* dienen maßgeblich der Workload-Generierung und Protokollierung erhobener Metriken. Zusätzlich werden Sie für vorbereitende Maßnahmen, wie dem Einrichten der Datenbank genutzt. Während der Benchmark-Durchführung wird eine definierbare Anzahl von Threads verwendet, um den Workload im erforderlichen Maße generieren zu können. Jeder *BGClient* arbeitet dabei unabhängig von den anderen, sodass die gesamte Menge an *BGClients* weitreichend[14] skaliert werden kann. [BG13]

[13] Hierbei handelt es sich um die erste Session mit der Bezeichnung *ViewSelfProfileSession*.

[14] Bedingt durch den erforderlichen Netzwerkverkehr zwischen dem *BGCoord* und sämtlichen *BGClients* wird die Skalierung limitiert. Zusätzlich schränkt die begrenzte CPU Kapazität des *BGCoords* die Verarbeitung eingehender Nachrichten weiterhin ein.

3.1.2. YorkMDE

Der von Seyyed M. Shah et al. [SWK+14] vorgestellte Benchmark[15] wurde zur Evaluierung verschiedener Datenbanksysteme im Kontext der Persistierung von Model Driven Engineering (MDE) Datenmodellen entwickelt. Seinen Ursprung besitzt dieser Benchmark im aktuellen Forschungsschwerpunkt, der Skalierungsaspekte bei MDE Projekten untersucht. Erforscht wird hierbei wie zukünftig noch mehr Modellierer an noch größeren MDE Projekten gemeinsam arbeiten können. In diesem Zusammenhang hat sich das bisher verwendete XML Metadata Interchange (XMI) Format zum Austausch und zur Persistierung der Modelle als weniger vorteilhaft erwiesen. In der Konsequenz haben Seyyed M. Shah et al. den im folgenden beschriebenen Benchmark entwickelt, um zu untersuchen, inwiefern sich verschiedene (NoSQL-)Datenbanksysteme zur Persistierung der Modelle im Vergleich eignen. Einerseits wird hierbei gemessen wie viel Prozessorzeit notwendig ist um ein bestehendes MDE Modell in der Zieldatenbank zu persitieren und andererseits wie viel Zeit erforderlich ist um über das Modell Client-seitig mittels lesender Operationen zu traversieren. Zusätzlich wird erhoben, wie viel Sekundärspeicher bei den verschiedenen Datenbanksystemen zur Persistierung der Modelle benötigt wird. Neben der Verwendung von Modellen, die im Rahmen realer Projekte erstellt wurden, können diese auch synthetisch auf Basis bereits existierenden Source Codes, per Reverse Engineering, generiert werden. [SWK+14]

3.1.2.1. Architektur

Im Kern besteht das Framework aus zwei elementaren Schichten (siehe Abbildung 13). Auf der einen Seite ist die Persistenzschicht dafür zuständig, bereits bestehende MDE Modelle einzulesen. Bedingt durch die Standardisierung der entsprechenden Schnittstellen kann die Menge der unterstützten Modellformate beliebig erweitert werden. Derzeit werden jedoch lediglich Eclipse Modeling Framework (EMF) Modelle unterstützt. Auf der anderen Seite existiert zur Interaktion mit den verschiedenen

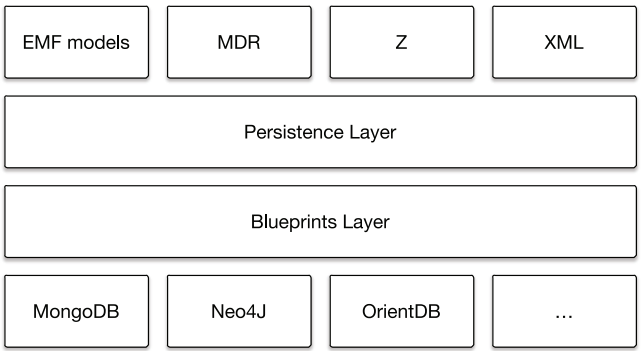


Abbildung 13: YorkMDE: Architektur [SWK+14]

[15] `https://bitbucket.org/yorkmde/benchmarks/overview`, zuletzt besucht am 15.01.2015.

Datenbanksystemen die sogenannte Blueprints-Schicht. Diese Schicht basiert auf der namensgebenden Java Graph API Blueprints[16], welche intern eine einheitliche Schnittstelle gegenüber einer beliebigen Menge von Datenbanksystemen bereitstellt. Die Verwendung dieser extern gepflegten Bibliothek bringt den Vorteil, dass sobald ein neuer Blueprints kompatibler Datenbankadapter implementiert wird, dieser auch automatisch für den Benchmark verwendet werden kann. [SWK+14]

3.1.2.2. Datenmodell

Als internes Datenmodell wird das Property Graph Model[17] verwendet. Das Mapping der MDE Modelle erfolgt, indem die Objekte als Eckpunkte des Graphen und die Objekteigenschaften jeweils als Eigenschaften der Eckpunkte abgebildet werden. Bestehende Beziehungen zwischen den Objekten werden mittels Kanten dargestellt. Die Abbildung 14 zeigt beispielhaft wie ein einfaches Modell (Abb. 14(a)) auf den Property Graphen (Abb. 14(b)) abgebildet wird. Das Mapping des Property Graphen auf das zugrundeliegende Modell der entsprechenden Zieldatenbank erfolgt in Abhängigkeit der jeweilige Datenbankadapterimplementierung. Diesbezüglich sind ebenfalls in Abbildung 14 beispielbezogene Mappings für einen Triple Store, ein Document Store und für eine relationale Datenbank dargestellt. [SWK+14]

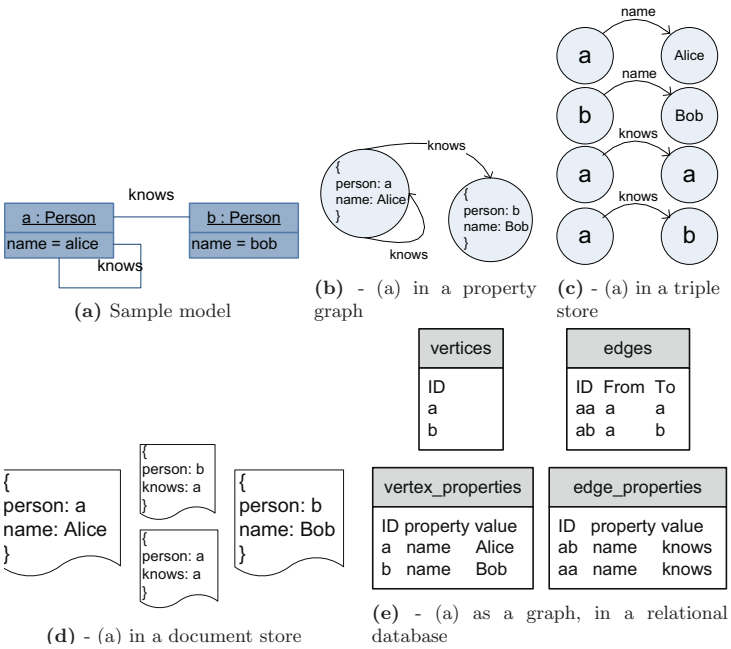


Abbildung 14: YorkMDE: Datenmodellierung [SWK+14]

[16] `https://github.com/tinkerpop/blueprints/wiki`, zuletzt besucht am 15.01.2015.

[17] `https://github.com/tinkerpop/blueprints/wiki/Property-Graph-Model`, zuletzt besucht am 15.01.2015.

3.1.3. YCSB

Bei dem von Brain F. Cooper et al. [CST+10] veröffentlichtem Yahoo! Cloud Serving Benchmark (YCSB)[18] handelt es sich um ein Java-basiertes Open Source Framework zum Leistungsvergleich verschiedener Datenbanksysteme. Primäres Ziel dieses Benchmarks ist es unterschiedliche Systeme bzgl. online lesender und schreibender Zugriffe zu vergleichen. Aspekte der Batch und analytischen Verarbeitung werden dabei explizit außer Acht gelassen. Im Fokus der Messungen steht die zur Ausführung einer Operation erforderliche Latenz und der durchschnittlich erreichbare Durchsatz an Operationen pro Sekunde. Insgesamt verfolgt YCSB einen sehr generischen Ansatz, der weitestgehend jedwedem komplexen Anwendungskontext entsagt. Hierzu wurde das Datenmodell auf eine einzige Tabelle beschränkt, wobei die Anzahl der Felder und deren einheitliche Inhaltsgröße frei definierbar ist. [CST+10]

3.1.3.1. Workloads

Zur Konfiguration verschiedener Testfälle können separate Workloads erstellt werden. Hierbei kann sowohl das Häufigkeitsverhältnis für die Aufrufe zwischen den Operationsarten, die Größe der Datensätze, als auch der zu verwendende Verteilungsalgorithmus zur Auswahl von Datensätzen definiert werden. [CST+10]

Insgesamt werden dabei die fünf Operationsarten Insert, Update, Delete, Read und Scan unterstützt. Bezüglich der lesenden Operationen kann festgelegt werden, ob jeweils sämtliche Felder eines Datensatzes gelesen werden oder nur ein einzelnes zufälliges Feld. Vergleichbar hierzu kann für die Update-Operation definiert werden, ob alle Felder oder nur ein einzelnes zufällig gewähltes Feld überschrieben werden soll. [CST+10]

Für den Zugriff von Operationen auf bestehende Datensätze existieren eine Reihe verschiedenster implementierter Verteilungsstrategien. Beispielsweise besitzt jeder Datensatz bei Uniform-Verteilung die gleiche Auswahlwahrscheinlichkeit, wodurch schlussendlich auf alle Datensätze vergleichsweise gleich häufig zugegriffen wird. Unter Verwendung der implementierten Zipfian-Verteilung wird auf den Großteil der Datensätze verhältnismäßig selten zurückgegriffen, auf einige wenige Datensätze jedoch besonders häufig. Des Weiteren existiert eine Variante der Zipfian-Verteilung (Latest), welche bewirkt, dass die zuletzt eingefügten Datensätze auch

[18] `https://github.com/brianfrankcooper/YCSB`, zuletzt besucht am 12.01.2015

jene mit der höchsten Auswahlwahrscheinlichkeit sind. Anhand der nachfolgenden Abbildung 15 ist erkennbar, wie sich die Zugriffsverhalten über die Menge der eingefügten Datensätze für die drei beispielhaft vorgestellten Strategien darstellt. [CST+10]

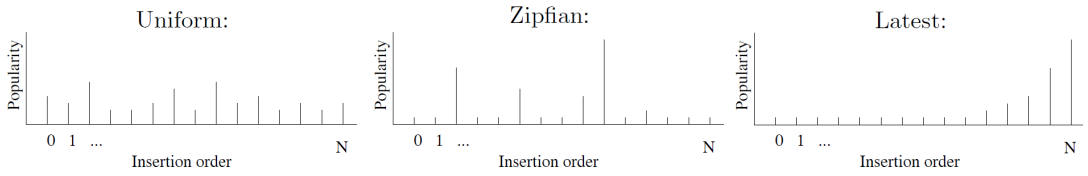


Abbildung 15: YCSB: Beispiele zu Verteilungsstrategien [CST+10]

Standardmäßig sind bereits fünf verschiedene Workloads vordefiniert (siehe Tabelle 2). Zudem ist es möglich, ohne Anpassung des Source Codes, jederzeit neue Workloads entsprechend der eigenen Anforderungen hinzuzufügen.

Workload	**Verwendete Operationen**	**Verteilung**
A - Update heavy	Read: 50%, Update:50%	Zipfian
B - Read heavy	Read: 95%, Update: 5%	Zipfian
C - Read only	Read: 100%	Zipfian
D - Read latest	Read: 95%, Insert: 5%	Latest
E - Short ranges	Scan: 95%, Insert: 5%	Zipfian/Uniform[19]

Tabelle 2: YCSB: Übersicht vorkonfigurierter Workloads [CST+10]

3.1.3.2. Architektur

Im Kern besteht der YCSB Client aus vier unterschiedlichen Komponenten (siehe Abbildung 16). Der *Workload Executor* sorgt dafür, dass der Workload entsprechend der Konfiguration ausgeführt wird. Hierzu wird eine konfigurierbare Menge an *Threads* erzeugt, welche einerseits dazu genutzt wird, die entsprechende Datenbank mit initia-

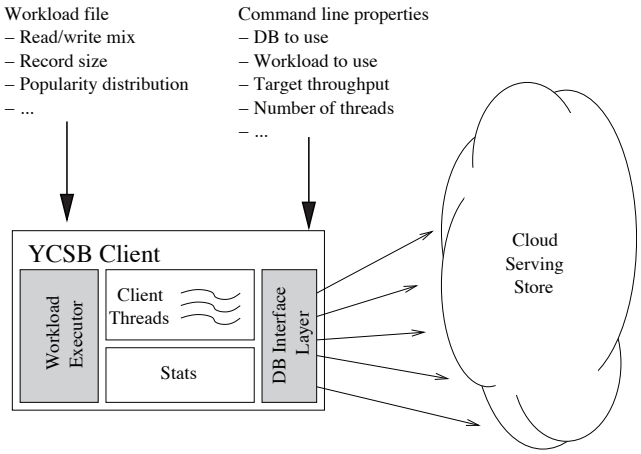


Abbildung 16: YCSB: Architektur [CST+10]

[19] Zipfian zur Auswahl des ersten Datensatzes und Uniform zur Auswahl der abzurufenden Datensatzanzahl.

len Datensätzen zu befüllen. Andererseits werden die *Threads* während der Workload-Ausführung dazu benötigt, um mit einem einzigem Client eine Vielzahl von Operationen gleichzeitig gegen das Datenbanksystem absetzen zu können. Sämtliche *Threads* rufen zur Interaktion mit dem Datenbanksystem die Methoden des *DB Interface Layers* auf. Durch die Verwendung dieses generischen Interfaces ist es möglich beliebige Datenbanksysteme anzubinden. Während der Workload-Ausführung wird der Durchsatz an durchgeführten Operationen und die erforderliche Latenz pro Operation gemessen und an das Statistikmodul (*Stats*) übergeben. Nach Beendigung des Workloads aggregiert dieses die gemessenen Werte und gibt die Ergebnisse aus. Im Wesentlichen wird zusätzlich noch zwischen zwei unterschiedlichen Messtypen unterschieden. Während die erste Variante ein Histogramm über die gemessenen Latenzwerte erzeugt, protokolliert die zweite Variante den Zeitverlauf der gemessenen Durchsatz- und Latenzwerte. [CST+10]

Insgesamt zeichnet sich die zugrundeliegende Architektur durch eine ausgeprägte Erweiterbarkeit aus. So ist es bereits möglich, eine Vielzahl verschiedener Benchmark-Konfigurationen ohne Anpassungen des Source Codes bereitzustellen. Sofern bisher nicht berücksichtigte Funktionalitäten, wie etwa das Anbinden neuer Datenbanksysteme, realisiert werden sollen, erlauben es die generischen Schnittstellen den YCSB Client mit verhältnismäßig geringem Aufwand um neue Funktionalitäten zu erweitern.

3.1.3.3. YCSB Varianten und Erweiterungen

Neben der originalen YCSB Version existieren eine Vielzahl weiterer Varianten, welche das Original um neue Funktionalitäten erweitern und/oder bestehende Fehler beheben. Allein auf GitHub.com existieren derzeit mehr als 534 Forks[20]. Im Folgenden werden die wesentlichen Aspekte dreier verschiedener YCSB Varianten in Form einer Kurzbeschreibung vorgestellt. Ausgewählt wurden hierfür YCSB++, YCSB+T und die Variante der Firma Thumbtack Technology Inc., da diese ihre durchgeführten Modifikationen im Rahmen von Veröffentlichungen dokumentierten und anhand ebenfalls präsentierter Messergebnisse deren grundsätzliche Funktionsfähigkeit unter Beweis gestellt haben.

Thumbtack YCSB

Die Firma Thumbtack Technology Inc. hat YCSB um einige neue Funktiona-

[20] Am 02.03.2015 existierten insgesamt 534 Forks. Sofern Entwickler ihrer YCSB Varianten als eigenständige Projekte bei GitHub hochgeladen haben, werden diese bei der Menge der existierenden Forks nicht berücksichtigt.

litäten[21] für ihre eigenen Benchmark-Untersuchungen [NE13a] erweitert. Zu einer der wichtigsten Neuerungen zählt die Verbesserung der Multi-Client Unterstützung, sodass mehrere YCSB Clients parallel verwendet werden können. Hierzu wurden Python/Fabric[22] Skripte erstellt, welche den Start mehrerer YCSB Clients auf unterschiedlichen Computern synchronisieren[23] und deren Aufrufe derartig parametrisieren, dass es bspw. beim initialen Befüllen der Datenbank zu keinen mehrfach verwendeten Keys kommt. Zudem existieren weitere Skripte, die die Ergebnisse von den einzelnen Client Computern abrufen und aggregieren können. Neben solch YCSB bezogenen Funktionalitäten existieren zudem auch Skripte, die das Management der Benchmark-Umgebung erleichtern. So ist es bspw. möglich, zentral die entsprechenden Datenbanksysteme auf allen hinterlegten Server Nodes zu beenden und zu starten. Zusätzlich wurde YCSB im Wesentlichen um eine Warmup-Phase, eine Reconnect-Funktionalität und um die Möglichkeit zur Wiederholung fehlgeschlagener Operationen ergänzt. Darüber hinaus wurden unter anderem die Datenbankadapter für Couchbase, MongoDB und Aerospike aktualisiert und um neue Parametrisierungen erweitert. Eine weiterreichende Beschreibung der erfolgten Modifikationen kann der Projektdokumentation[24] entnommen werden.

YCSB++

Beim von Swapnil Patil et al. [PPR+11] veröffentlichten YCSB++[25] handelt es sich um eine YCSB Variante der Carnegie Mellon University. Für ihre Untersuchungen der Datenbanksysteme (NSA/Apache) Accumulo und Apache HBase erweiterten sie YCSB unter anderem ebenfalls um eine Multi-Client Funktionalität. Im Gegensatz zu Thumbtacks Ansatz verwendet YCSB++ hierzu den Apache ZooKeeper als Regulierungskomponente. Dieser stellt sicher, dass erst mit dem Benchmark begonnen wird, sobald sämtliche Clients ihre Bereitschaft signalisiert haben. Als weitere Neuerung ist ein Konsistenz-Benchmark hinzugekommenen. Dieser misst die Zeit, die zwischen dem Einfügen eines Datensatzes durch den einen Client und dem Sichtbarwerden für einen anderen Client erforderlich ist. Des Weiteren wurde die Unterstützung von Server-seitigen Filtern für lesende Operationen, das Bulk Loading mittels Hadoop, das Table pre-splitting sowie die Unterstützung verschiedener Zugriffskontrollmechanismen ergänzt.

[21] `https://github.com/thumbtack-technology/ycsb`, zuletzt besucht am 13.01.2015.
[22] `http://www.fabfile.org/`, zuletzt besucht am 10.12.2014.
[23] Synchronisierung erfolgt durch das gleichzeitige Starten auf Basis einer gemeinsamen Startzeit.
[24] `https://github.com/thumbtack-technology/ycsb/blob/master/IMPROVEMENTS.txt`, zuletzt besucht am 13.01.2015.
[25] `http://www.pdl.cmu.edu/ycsb++/`, zuletzt besucht am 13.01.2015.

YCSB+T

Der von Akon Dey et al. [DFNR14] vorgestellte YCSB+T[26] Benchmark erweitert das originale YCSB um zwei wesentliche Aspekte. Zum einen wird eine Transaktionsunterstützung hinzugefügt und zusätzlich die interne Messung derart erweitert, dass der durch die Transaktionen entstehende Overhead quantifiziert werden kann. Zum anderen wurde ein Workload-Szenario zur Konsistenzevaluierung erstellt. Bei diesem wird die Datenbank mit einer festen Anzahl von Datensätzen initial befüllt. Jeder Datensatz entspricht dabei einem Konto, welches zu Beginn 1.000 $ führt. Während der Workload-Durchführung werden verschiedene Beträge zwischen den Konten hin und her gebucht, sodass zum Ende jedes Konto wiederum durchschnittlich 1.000 $ besitzen sollte. Anhand der ggf. aufgetretenen Differenz wird in Verbindung mit der Anzahl durchgeführter Transaktionen ein sogenannter Anomaly Score berechnet.

3.1.4. Bewertung und Auswahl eines Benchmark Frameworks

Das BG Framework zeichnet sich vor allem durch eine ausgeprägte Praxisorientierung aus. Maßgeblich hierfür ist das vorgegebene Anwendungsszenario, welches mit seinen Session-Profilen bereits einen Betriebsrahmen auf Basis real existierender sozialer Netzwerke vordefiniert. Vorteilhaft für die Evaluierung verschiedenartiger Datenbanksysteme ist, dass es dem Anwender/Entwickler überlassen bleibt, in welcher konkreten Form das Anwendungsszenario mit dem jeweiligen Datenbanksystem realisiert wird. Somit unterstützt diese Eigenschaft die in dieser Arbeit vorgesehene Evaluierung verschiedener Betriebskonfigurationen einzelner Datenbanksysteme. Ein weiterer Aspekt, der die ausgeprägte Praxisorientierung des BG Frameworks unterstreicht, ist die Verwendung SLA basierter Metriken. In Ermangelung eines vollständig definierten Anwendungsfalls gestaltet sich jedoch gerade dieser Aspekt als problematisch bei der Festlegung der erforderlichen SLA Parameter. Entsprechend unsicher bleibt es, ob eine Parametrisierung identifiziert werden kann, die für sämtliche der zu untersuchenden Betriebskonfigurationen gleichermaßen[27] geeignet und sinnvoll ist. Beispielsweise wird es kompliziert die maximale Latenz einheitlich so zu definieren, dass sie bei der Evaluierung unterschiedlicher Replikationskonfigurationen gleichermaßen zu aussagekräftigen Messergebnissen führt. In der Folge wird somit deutlich, dass das BG Framework ausschließlich dazu geeignet ist, zu evaluieren inwiefern sich ein bestimmtes Datenbanksystem unter Einhaltung einer zuvor festgelegten SLA im Anwendungssze-

[26] `https://github.com/akon-dey/YCSB`, zuletzt besucht am 13.01.2015.

[27] Eine einheitliche SLA Parametrisierung ist für jedes Datenbanksystem erforderlich, um die Vergleichbarkeit zwischen den Messergebnissen zu gewährleisten.

nario eines sozialen Netzwerkes eignet. Dementsprechend wird das BG Framework für die grundlegende Untersuchung verschiedener Betriebskonfigurationen einzelner Datenbanksysteme als ungeeignet betrachtet und für die Verwendung im Rahmen dieser Arbeit fortan nicht weiter in Erwägung gezogen.

Vergleichbar zum BG Benchmark handelt es sich bei YorkMDE ebenfalls um einen anwendungsorientierten Benchmark. Besonders hervorzuheben ist, dass die zugrundeliegenden Daten nicht wie bei den Benchmarks YCSB und BG zur Laufzeit generiert werden, sondern bestehende MDE Modelle verwendet werden. Dies hat den Vorteil, dass jedes Datenbanksystem auf Grundlage der identischen Datenbasis evaluiert wird und je nach Vorhandensein auch Echt- bzw. Produktivdaten verwendet werden können. Somit ist es möglich entsprechende Benchmarks in einem deutlich realitätsnäheren Szenario zu evaluieren. Bedingt durch die Verwendung der Open Source Frameworks Blueprints können prinzipiell eine Reihe bestehender Datenbankadapter[28] verwendet werden. Für die Datenbanksysteme Cassandra und Couchbase existieren jedoch noch keine offiziellen Adapter, sodass hierfür eigene Implementierungen erforderlich wären. Für die Verwendung des YorkMDE Benchmarks bzgl. der grundlegenden Evaluierung von NoSQL-Datenbanksystemen existiert zudem ein weiterer Nachteil, welcher infolge der MDE spezifischen Fokussierung auftritt. Bedingt durch eine ausschließliche Betrachtung der Persistierung und Traversierung der MDE Modelle bleiben weitere Aspekte wie bspw. die Aktualisierung bestehender Daten weitestgehend unbeachtet. Des Weiteren existiert auch keine Möglichkeit einen Anwendungsfall zu untersuchen, welcher verschiedene Operationsarten gleichzeitig vorsieht. Insgesamt wird somit deutlich, dass auch der YorkMDE Benchmark aufgrund seiner strikten Anwendungsfokussierung nicht in geeigneter Weise für die geplanten Untersuchungen verwendet werden kann.

Im Gegensatz zu den beiden bereits diskutierten Benchmarks BG und YorkMDE basiert YCSB auf keinem konkreten Anwendungsszenario. Die ausschließliche Verwendung einer einzigen Tabelle und die Beschränkung auf die Basisoperationen (read, update, insert, scan) spiegelt auf der einen Seite keinen typischen Praxisbetrieb einer (NoSQL-)Datenbank (mehr) wider. Auf der anderen Seite erlaubt die Verwendung eines derart einfachen Basisszenarios jedoch die Betrachtung eines bei "sämtlichen" Datenbanken existierenden Funktionsumfangs. Infolgedessen wird zu mindestens eine rudimentäre Vergleichbarkeit für die unterschiedlichsten

[28] Vor allem existieren Datenbankadapterimplementierungen für Graphdatenbanken wie z.B. Neo4j, OrientDB oder ArrangoDB. Darüber hinaus existieren noch weitere Implementierungen für andere Datenbanksysteme wie Accumulo, MongoDB oder Oracle NoSQL.

Datenbanksysteme geschaffen. Auch für die im Rahmen dieser Arbeit grundlegende Evaluierung einzelner NoSQL-Datenbanksysteme erscheint die Verwendung von YCSB als grundsätzlich geeignet. Entsprechend kann ermittelt werden, inwiefern sich einzelne Konfigurationsparameter im Vergleich auf die Leistungsfähigkeit eines Datenbanksystems auswirken. Ein weiteres Argument, welches für die Verwendung von YCSB spricht, ist dessen hoher Verbreitungsgrad. Derzeit scheint das YCSB Framework als eine Art de-facto Standard für das Benchmarking von NoSQL-Datenbanksysteme zu dienen. Dies zeigt sich vor allem durch die bereits häufige Verwendung[29] des Frameworks und seiner vielzähligen Varianten. Somit wird davon ausgegangen, dass das YCSB Framework bereits eine gewisse Reife besitzt und entsprechend gut für das produktive Benchmarking geeignet ist. Statt der originalen YCSB Version wird Thumbtacks Variante für das Benchmarking verwendet. Ausschlaggebend hierfür ist zum einen die unterstützte Multi-Client Funktionalität, mit deren Hilfe sichergestellt werden soll, dass die Datenbankcluster auch ausreichend ausgelastet werden können. Zum anderen ist davon auszugehen, dass die ergänzten Benchmark-Automatisierungsfunktionalitäten einerseits die Anwendbarkeit vereinfachen und andererseits dadurch gleichzeitig das Fehlerpotential senken.

3.2. Modifikationen an Thumbtacks YCSB Variante

Das nachfolgende Kapitel beschreibt die im Rahmen dieser Arbeit durchgeführten Modifikationen an Thumbtacks YCSB Variante. Neben der Dokumentation der Änderungen zur Wahrung der Reproduzierbarkeit der Arbeitsergebnisse soll dieses Kapitel dem Leser zusätzlich das breite Spektrum an notwendigen Erweiterungen und erforderlichen Fehlerbehebungen aufzeigen, welche nach Ansicht des Autors notwendig sind, um ein adäquates Benchmarking von NoSQL-Datenbanksystemen zu gewährleisten.

3.2.1. Modifikationen und Erweiterungen

Dieses Kapitel beschreibt die durchgeführten Änderungen an Thumbtacks YCSB Variante. Hierbei wird insbesondere dargestellt, weshalb diese notwendig sind und inwiefern sich diese auf das Benchmarking mit (Thumbtacks) YCSB auswirken.

[29] Siehe bspw. [Ava15, Alt14, GGK+14, GP14, Dat13, DG13, NE13a, NE13b, RGVS+12, PPR+11].

3.2.1.1. Neue Datenbankadapter

Im Rahmen dieser Arbeit wurden insgesamt drei neue Datenbankadapter für das YCSB Framework realisiert. Zwei davon für Couchbase und einer für Cassandra. Diese Maßnahme ist erforderlich, da sowohl die beim originalen YCSB Framework vorliegenden Adapter, als auch die von Thumbtack bereitgestellten Adapter noch auf veralteten Softwarebibliotheken basieren.

Beide der für Couchbase implementierten Adapter basieren in großen Teilen auf der Codebasis des ursprünglich von Thumbtack erzeugten Adapters[30]. Die Erstellung von zwei unterschiedlichen Datenbankadaptern wurde aufgrund der Feststellung eines schwerwiegenden Fehlers innerhalb der derzeit aktuellen Couchbase Java SDK Version 2.0.1 notwenig. Der Fehler macht eine Verwendung des ursprünglichen Datenbankadapters zur Durchführung der Benchmarks unmöglich. Eine ausführliche Beschreibung des in Bezug zur Timeout-Behandlung stehenden Fehlers kann im Anhang dem Kapitel A.1.2 entnommen werden. Zur letztendlichen Verwendung wurde ein zweiter Adapter in Verbindung mit der derzeit aktuellsten Version des vorherigen Major-Releases (Version 1.4.5) erstellt.

Der für Cassandra erforderliche neue Datenbankadapter wurde unter Verwendung der derzeit aktuellsten Version (2.1.4) von DataStax's Java Client Driver[31] implementiert. Im Vergleich zu einigen alternativen Java Client Drivern, wie z.B. Easy Cassandra[32], verwendet der DataStax Client Driver nicht mehr Apaches Thrift Framework[33], sondern kommuniziert mit den Cassandra Servern direkt auf Basis von Cassandra's eigenem Binärprotokoll. Des Weiteren unterstützt DataStax's Client Driver Cassandra's eigene Anfragesprache CQL in der Version 3.

Bezüglich sämtlich implementierter Datenbankadapter ist anzumerken, dass jeder YCSB (Workload) Thread eine eigene Datenbankverbindung herstellt und während der Benchmark-Ausführung verwendet.

[30] Verwendet das Couchbase Java Client SDK 1.1.0.
[31] `https://github.com/datastax/java-driver`, zuletzt besucht am 27.02.2015.
[32] `https://github.com/otaviojava/Easy-Cassandra`, zuletzt besucht am 27.02.2015.
[33] `https://thrift.apache.org/`, zuletzt besucht am 27.02.2015.

3.2.1.2. Datenbankverbindung-Reconnect

Die von Thumbtack implementierte Reconnect Funktionalität unterstützt den automatischen Ab- und Aufbau einer Datenbankverbindung beim Unterschreiten einer definierbaren durchschnittlichen Durchsatzgeschwindigkeit. Problematisch gestaltet sich dabei deren Konfiguration, da im Rahmen verschiedener Benchmark-Szenarien Situationen auftreten können, in denen sehr hohe Latenzen entstehen. Je nach erforderlicher Latenz kann es somit zu nur wenigen durchführbaren Operationen pro Sekunde oder gar wenigen Operationen pro Minute kommen. Infolgedessen führt eine ggf. ungeeignete Konfiguration zu einer dauerhaften Wiederholung von Verbindungsreinitialisierungen. Abgesehen von dieser Wiederholungsproblematik wird der für den Verbindungsneuaufbau benötigte Zeitraum nicht innerhalb der Messungsstatistiken berücksichtigt. Je nach zugrundeliegender Konfiguration führt dies zu mehr oder weniger stark ausgeprägten Verfälschungen der Messergebnisse. Zur Vermeidung derartiger Messverzerrungen wurde die Reconnect Funktionalität aus dem Thumbtack Code wieder entfernt.

3.2.1.3. Synchrone YCSB Clientstarts

Bedingt durch die Verwendung mehrerer YCSB Client Nodes ist eine separate Koordination zur gleichzeitigen Ausführung der YCSB Client Software notwendig. Selbst wenn auf jedem Client die Messungen zu einer im Vorfeld bestimmten Client-übergreifenden Systemzeit ausgeführt werden, bedeutet es nicht, dass alle Clients tatsächlich gleichzeitig starten, da die Systemzeiten der Clients nicht per se synchron gehalten werden. Die Folge wäre, dass am Anfang und am Ende der gesamten Messung weniger YCSB Clients Anfragen gegen das Datenbanksystem stellen als zur Mitte der gesamten Laufzeit. Eine entsprechendes Verhalten wird beispielhaft mittels der Abbildung 17 skizziert.

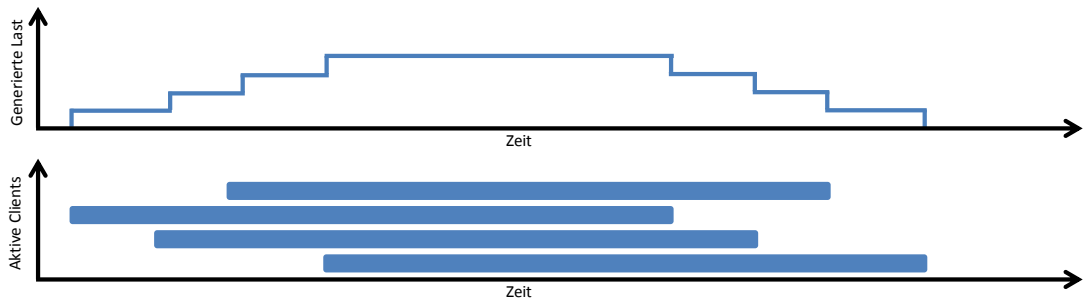
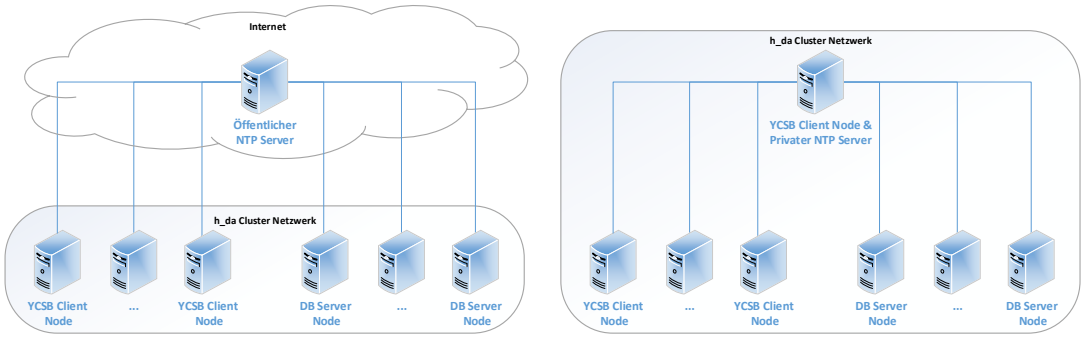


Abbildung 17: Benchmark-Vorbereitung: Asynchrone YCSB Client-Ausführung

Der gleichzeitige Start von YCSB Clients auf unterschiedlichen Nodes wird durch Thumbtacks Python/Fabric Erweiterungen mittels der Verwendung des linuxeige-

nen Job-Schedulings realisiert. Damit dieser Mechanismus ordnungsgemäß funktioniert, müssen die Uhrzeiten auf den Nodes demzufolge synchron gehalten werden. Innerhalb der gegebenen Erweiterungen existiert hierfür jedoch keine Funktionalität.

Im Folgenden werden zwei auf dem Network Time Protocol (NTP) basierende Varianten unterschieden. Bei der ersten Variante synchronisiert jeder Cluster Node seine Uhrzeit gegen einen öffentlichen NTP Server. Bedingt durch die fehlende Kontrolle über jene externen NTP Server bestünde bei dieser Variante zu mindestens theoretisch die Gefahr, dass unterschiedliche Uhrzeiten an einzelne Cluster Nodes verteilt werden könnten oder aufgrund von Problemen mit der Internetverbindung ggf. keine Synchronisierung durchführen könnten. Die zweite Variante sieht dagegen einen eigenen NTP Server Dienst auf einem der Cluster Nodes vor, mit dem sich allen weiteren Nodes synchronisieren. Eine Synchronisierung des lokalen NTP Servers mit einem öffentlichen NTP Server wäre technisch einfach realisierbar, ist jedoch nicht notwendig. Relevant für die synchrone Ausführung ist lediglich, dass alle weiteren Nodes des Clusters dieselbe Uhrzeit besitzen. Inwiefern die Uhrzeit realweltlich korrekt ist, besitzt dagegen keine Relevanz.



(a) NTP Variante 1 - Öffentlicher NTP Server (b) NTP Variante 2 - Privater NTP Server

Abbildung 18: Benchmark-Vorbereitung: Gegenüberstellung zweier NTP Varianten

Aufgrund der gebotenen Unabhängigkeit und der damit verbundenen erhöhten Fehlertoleranz wird die Variante 2 präferiert. Entsprechend wurde auf einem der Client Nodes ein zusätzlicher NTP Server Dienst installiert. Die Synchronisierung aller anderen Nodes erfolgt mittels Python/Fabric einmalig vor dem Messen einer Benchmark-Konfiguration. Somit bleibt sichergestellt, dass alle Nodes eine nahezu identische Zeit verwenden und während der Benchmark-Durchführung kein zusätzlicher Overhead entsteht.

3.2.1.4. Warmup-Funktionalität

Die von Thumbtack zusätzlich implementierte Warmup-Funktionalität verwendet die gleiche Anzahl separater Threads, wie die darauffolgenden Workload-Durchführung. Diese bauen eine eigenständige Datenbankverbindung auf und führen entweder eine fest definierte Anzahl von Read-Operationen oder Read-Operationen über eine festgelegte Dauer hinweg durch. Nach Absolvierung der Read-Operationen beenden die Threads ihre Datenbankverbindung und terminieren sich selbst. Für die eigentliche Workload-Durchführung werden anschließend neue Threads mit eigenen Datenbankverbindungen erstellt. Die Abbildung 19 illustriert die beschriebene Funktionalität mit dem Fokus auf die dabei für das Datenbanksystem generierte Last.

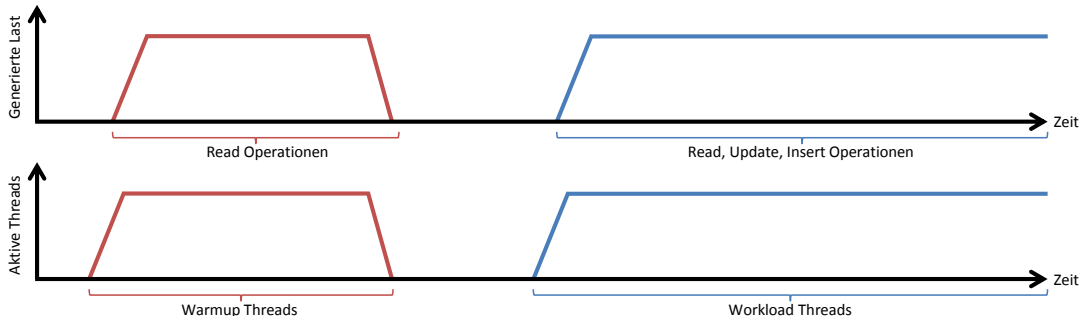


Abbildung 19: Benchmark-Vorbereitung: Thumbtacks Warmup Implementierung

Dabei erscheint es fragwürdig, inwiefern eine rein auf Read-Operationen basierte Warmup-Phase die Benchmark-Umgebung hinreichend auf verschiedene Workloads vorbereitet. Gerade in Anbetracht der meist "unbekannten" internen Datenbankmechanismen bleibt zum einen offen, ob eine derartige Warmup-Phase beispielsweise ein Insert-Only Workload in vollem Umfang vorbereitet. Zum anderen bleibt offen, wie sich die separaten Verbindungen/Sessions auf die Vorbereitung der Datenbanksysteme auswirken.

In Folge der festgestellten Problempunkte wurde die bestehende Implementierung entfernt und durch eine neue ersetzt. Anhand der folgenden Abbildung 20 lässt sich das veränderte Warmup-Verfahren erkennen.

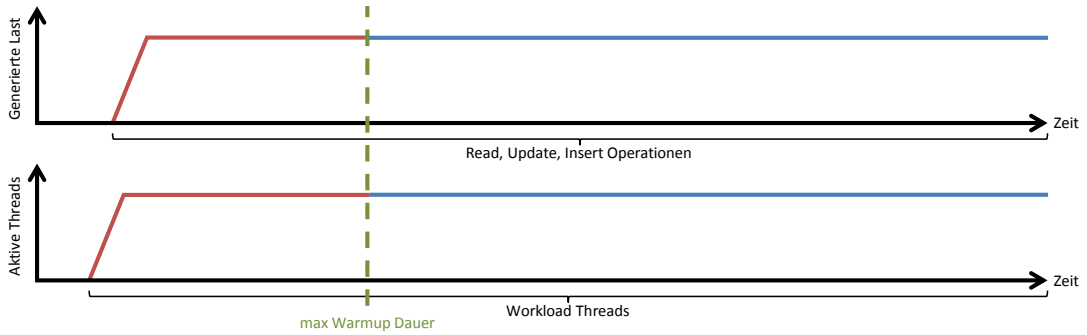


Abbildung 20: Benchmark-Vorbereitung: Neue Warmup Implementierung

Die Warmup-Phase ist nunmehr Bestandteil der eigentlichen Benchmark-Phase, wodurch beide Phasen sowohl die gleichen Threads, als auch dieselben Datenbankverbindungen benutzen. Entsprechend der Konfiguration werden dabei die ersten x Millisekunden für die Warmup-Phase verwendet. In der Konsequenz bedeutet dies, dass der Workload bereits ausgeführt wird, seine Messergebnisse jedoch erst nach Ablauf der Warmup-Phase protokolliert werden. Dies hat die zusätzlichen Vorteile, dass sämtliche Operationsarten Bestandteil der Warmup-Phase sein können und die anfangs benötigte Zeit zur Stabilisierung der Messung ausgeblendet wird.

Im Zuge dieser Erweiterung wurde außerdem der Thread-übergreifende Zeitmessungsbeginn synchronisiert. Während bei der originalen Implementierung jeder Thread seinen eigenen Startzeitpunkt ermittelte, erfolgt dies fortan einmalig vor der Initialisierung und gilt gleichermaßen für sämtliche Threads. Anhand der Abbildungen 21 und 22 lässt sich das veränderte Messungsverhalten erkennen.

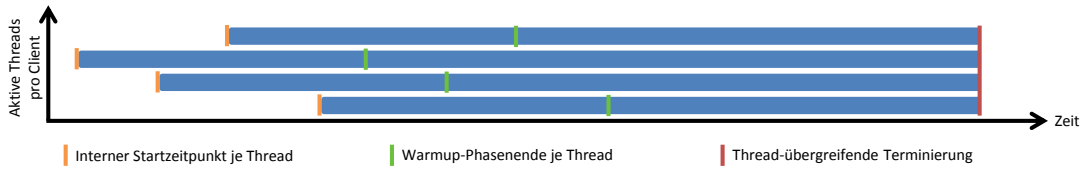


Abbildung 21: Benchmark-Vorbereitung: Zeitmessungsbeginn je Thread

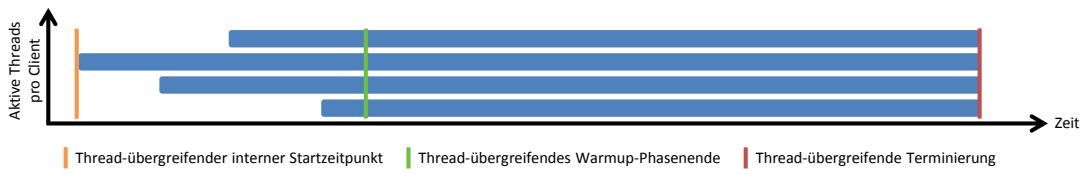


Abbildung 22: Benchmark-Vorbereitung: Zeitmessungsbeginn Thread-übergreifend

Als Resultat dieser Anpassung erfolgt die Benchmark-Phase bei allen Threads nahezu simultan über den gleichen Zeitraum. Allerdings variiert die Dauer der Warumup-Phase von Thread zu Thread. Ggf. daraus resultierende negative Auswirkungen können jedoch durch eine entsprechend lang gewählte Warmup-Phase umgangen werden.

3.2.1.5. Unterstützung von Insert-Operationen im Multi-Client Betrieb

Bereits in der originalen YCSB Version ist eine Verwendung von Insert-Operationen mit mehreren Clients lediglich zur initialen Befüllung vorgesehen. Während der Workload-Durchführung resultiert die Nutzung mehrerer Clients in redundant verwendeten Keys und folglich in fehlgeschlagenen Insert-Operationen.

Zur zukünftigen Unterstützung dieser Funktionalität wurden sowohl Thumbtacks Python/Fabric Skripte als auch der YCSB Client an sich erweitert. Wesentliche Änderungen erfolgten dabei hinsichtlich der Key-Verwaltung. Global betrachtet wird fortan bei einer Menge von x unterschiedlichen Clients der genutzte Key-Wertebereich auf $x + 1$ Abschnitte aufgeteilt (siehe Abbildung 23).

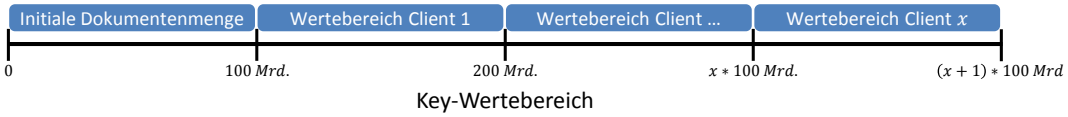


Abbildung 23: Benchmark-Vorbereitung: Aufteilung Key-Wertebereich

Der erste Abschnitt ist dabei für die initiale Befüllung der Datenbank vorgesehen und wird gleichermaßen von allen Clients genutzt[34]. Jeder weitere Abschnitt wird ausschließlich durch einen einzigen Client verwendet, wodurch Konflikte zwischen den Clients ausgeschlossen werden. Demnach besitzt der Wertebereich eines jeden[35] Clients eine Lücke innerhalb seines Wertebereiches, auf welchen er nicht zurückgreift. Um in diesem Zusammenhang weitreichende Anpassungen der zugrundeliegenden Funktionen zu vermeiden, erfolgt die Wertebereichsaufteilung zwischen der Key-Generierung und der Weitergabe an die Datenbankadapter. Dementsprechend findet die Aufteilung des Key-Wertebereichs für die intern verwendeten Key-Generatoren und Verteilungsalgorithmen transparent statt. Die nachfolgende Abbildung 24 illustriert den Unterschicht zwischen dem intern verwaltetem Key-

[34] Es gilt zu beachten, dass der im Rahmen dieser Arbeit verwendete UniformGenerator standardmäßig für die Operationen Read, Update und Delete auf den Wertebreich der initialen Datenmenge limitiert ist.

[35] Ausnahme bildet hierbei der erste Client.

Wertebereich und dem Wertebereich der Keys, welche an die Datenbankadapter weitergegebenen werden.

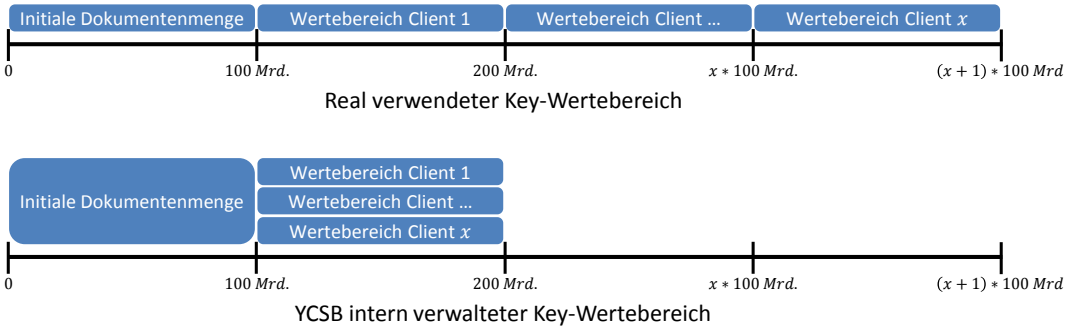


Abbildung 24: Benchmark-Vorbereitung: Aufteilung Key-Wertebereich II

Die einzelnen Abschnitte sind standardmäßig mit einer Größe von 100 Milliarden Keys gezielt überdimensioniert, sodass es im Rahmen einer Benchmark-Ausführung zu keiner Bereichsüberschreitung kommen sollte. Selbst bei einer Benchmark-Laufzeit von 24 Stunden müsste ein Client durchschnittlich mehr als 1.157.407 Insert-Operationen pro Sekunde durchführen, bis der Wertebereich eines anderen Clients verletzt würde. Keines der in dieser Arbeit untersuchten NoSQL-Datenbanksysteme erreichte auch nur annähernd einen solch hohen Durchsatz. Entsprechend kann die gewählte Abschnittsgröße als geeignet betrachtet werden.

3.2.1.6. Erweiterung der Aufteilung von Operationen im Multi-Client-Betrieb

Die innerhalb einer Workload-Konfiguration festgelegte Menge an durchzuführenden Operationen wird mittels der zugrundeliegenden Python/Fabric Skripte auf die Client Nodes gleichermaßen aufgeteilt. Innerhalb der YCSB Client Software wird diese Menge wiederum auf die Anzahl der konfigurierten Threads verteilt. Für beide Divisionen ist es jedoch erforderlich, dass deren Ergebnis jeweilig ganzzahlig ist. Andernfalls werden nicht alle Operationen durchgeführt. Insofern werden bspw. bei vier Client Nodes mit jeweils 16 Threads und einer Menge von 100 durchzuführenden Operationen nur 64 Operationen schlussendlich durchgeführt.

Der Vorteil dieser bestehenden Implementierung besteht darin, dass sämtliche Threads Client-übergreifend stets die exakt identische Anzahl von Operationen durchführen. Hinderlich wirkt sich diese Implementierung jedoch auf Benchmark-Konfigurationen aus, welche eine beliebige Anzahl von initial eingefügten Datensätzen erfordern. Hierzu zählt unter anderem die Evaluierung verschiedener

Datensatzgrößen[36]. Zur besseren Unterstützung dieser Anforderung wurde innerhalb der YCSB Client Software die Aufteilung der Operationen auf Threads derart verändert, dass auch nicht ganzzahlig durch die Anzahl der Threads dividierbare Mengen nahezu gleichmäßig aufgeteilt werden. Die Differenz an durchzuführenden Operationen beträgt dabei zwischen den Threads maximal eins.

3.2.1.7. Benchmark-Automatisierung

Thumbtacks YCSB Variante besitzt bereits einzelne Mechanismen um das Deployment der YCSB Client Software zu übernehmen, deren Protokolldateien herunterzuladen und dessen Messergebnisse zu aggregieren. Basierend auf der hierbei verwendeten Toolbasis von Python und dessen Erweiterung Fabric, wurden neue Mechanismen ergänzt, um den Benchmark-Ablauf weitestgehend zu automatisieren. Die dafür erstellten Skripte erlauben eine sequenzielle Messung verschiedener YCSB Workloads, welche beliebig häufig wiederholt werden können. In der Konsequenz ist es hierdurch möglich, beliebig viele unterschiedliche Benchmark-Konfigurationen vollautomatisiert[37] nacheinander zu benchmarken. Eine konkrete Darstellung des Benchmark-Ablaufs erfolgt in Kapitel 3.4 (Seite 52). Zur Strukturierung der gewonnen Benchmark-Ergebnisse werden die während der Benchmark-Durchführung erhobenen Ausführungsprotokolle und Messergebnisse in einer separaten Ordnerstruktur automatisch archiviert.

Insgesamt vereinfacht die erweiterte Benchmark-Automatisierung maßgeblich die Ausführung der Benchmarks und erhöht zugleich die Übersichtlichkeit über bereits durchgeführte Messungen. Zudem wird die Fehleranfälligkeit aufgrund der einheitlichen Vorgehensweise vermindert. Dementsprechend ermöglicht erst diese Erweiterung der Benchmark-Automatisierung die Durchführung der Vielzahl an vorgesehenen Messungen mit Thumbtacks YCSB Variante in der für diese Arbeit zur Verfügung stehenden Zeit.

3.2.2. Fehlerbehebungen

Es folgt eine Darstellung der als fehlerhaft identifizierten Funktionalitäten innerhalb von Thumbtacks YCSB Variante. Hierbei werden die Auswirkungen der Fehler inklusive der zur Behebung erforderlichen Maßnahmen beschrieben. Bedingt

[36] Siehe bspw. Kapitel 4.1.3.

[37] Voraussetzung hierfür ist das sämtliche Konfigurationsparameter Client-seitig gesetzt werden können. Hierzu zählen bspw. die Menge der zu verwendenden Threads, die Anzahl der Datensätze, die Größe und Anzahl der Felder sowie datenbankadapterspezfische Parameter wie das Verwenden von Prepared Statements.

durch die Tatsache, dass einige der Fehler erst während der Messungsdurchführung festgestellt werden konnten, wird zudem explizit darauf hingewiesen, insofern die Evaluierung einzelner NoSQL-Systeme durch einen Fehler betroffen bleibt. Dies ist zum Teil erforderlich, da die Wiederholung größerer Mengen bereits durchgeführter Messungen aufgrund der zeitlichen Begrenzung dieser Arbeit nicht möglich ist.

3.2.2.1. Fehlerhafte Aggregation von YCSB Client Ergebnissen

Nach der Durchführung eines Benchmarks und dem anschließendem Herunterladen der Protokolldateien von den einzelnen Clients können mit Hilfe des Python Skripts merge.py die Statistiken aller Clients zusammengefasst werden. Hierzu durchsucht das Skript die einzelnen Protokolldateien, puffert jeweils die wichtigsten Kennzahlen und aggregiert diese abschließend clientübergreifend. Aufgrund sporadisch auftretender Störungen beim Schreiben in die Protokolldatei kommt es teilweise zu falsch übernommenen Latenzwerten. Zur Verdeutlichung dieses Problems erfolgt dessen Darstellung anhand eines Beispiels.

Innerhalb einer jeden Protokolldatei (pro YCSB Client) sucht das Python Skript mittels regulärer Ausdrücke unter anderem nach dem im Listing 1 dargestellten Eintrag zur durchschnittlichen gemessenen Latenz. Derartig zusammenfassende Einträge befinden sich am Ende einer Protokolldatei.

```
1 ...
2 [INSERT], AverageLatency(us), 1332.8895487006196
3 ...
```

Listing 1: Gesuchter Eintrag am Ende einer Protokolldatei

Neben solch zusammenfassenden Einträgen bestehen die Protokolldateien zudem aus Einträgen, welche das Verhalten über die gesamte Laufzeit des Benchmarks protokollieren (siehe Listing 2).

```
1 ...
2 [INSERT], 199000, 1207.756836588852, 104950.0
3 510 sec: 20107611 operations; 97511 current ops/sec; [INSERT
     AverageLatency(us)=1107.27]
4 ...
```

Listing 2: Korrekte Einträge am Ende einer Protokolldatei

Als problematisch in diesem Zusammenhang erweist sich das Phänomen, dass die normalerweise in separaten Zeilen persistierten Einträge sporadisch in eine einzige Zeile geschrieben werden (siehe nachfolgendes Listing 3). Dies führt dazu, dass die Bedingungen der regulären Ausdrücke auch in dieser Zeile fälschlicherweise erfüllt sind, wodurch sämtliche der gefundenen Werte aufsummiert werden. Im Kontext des Beispiels bedeutet dies eine durchschnittliche Latenz von $\sim$ 2540 μs[38] statt der korrekten $\sim$ 1332 μs und somit eine deutliche Verfälschung des Ergebnisses.

```
1 ...
2 [INSERT], 199000, 1207.756836588852, 104950.0 510 sec: 20107611
     operations; 97511 current ops/sec; [INSERT AverageLatency(us)
     =1107.27]
3 ...
```

Listing 3: Fehlerhafter Eintrag am Ende einer Protokolldatei

Zur Behebung dieses Fehlers wurde einerseits die Protokollierung innerhalb des YCSB Clients als auch das entsprechende Python Skript verändert. Der für die im Listing 2 Zeile 3 verantwortliche Status-Thread[39] gibt seine Einträge standardmäßig in zwei separate Dateien aus. Da eine derartige redundante Protokollierung nicht erforderlich ist, wurde die Ausgabe in die betroffene Protokolldatei entfernt. Des Weiteren wurde das Python Skript um einen zusätzlichen regulären Ausdruck[40] ergänzt, sodass auch fehlerhafte Protokolldateien korrekt ausgewertet werden können.

3.2.2.2. Fehlerhafte Protokollierung von Messergebnissen unter Verwendung des TimeSeries Parameters

Jede Workload-Durchführung wird von YCSB mit einem Protokoll dokumentiert. Dieses enthält unter anderem Angaben bezüglich der durchschnittlich verarbeiteten Operationen und der durchschnittlich erforderten Latenz pro Sekunde. Am Ende des Protokolls erfolgt zudem eine Zusammenfassung, welche die Anzahl der durchgeführten Operationen pro Operationsart sowie die Anzahl der jeweils abgeschlossenen und fehlgeschlagenen Operationen enthält. Anhand des im Listing 4 dargestellten Protokollauszuges lässt sich erkennen, dass dieses Zählen jedoch nicht

[38] Die Summe von $\sim$ 2540 μs ergibt auf Basis der beiden Summanden $\sim$ 1332 μs aus Listing 1 und $\sim$ 1207 μs aus Listing 3.

[39] YCSB Thread erzeugt während der Benchmark-Ausführung alle zwei Sekunden einen Protokolleintrag bzgl. der durchschnittlich erfolgten Operationen pro Sekunde und entsprechender Latenzausgaben.

[40] Sofern die bestehenden regulären Ausdrücke erfüllt sind, darf die geprüfte Zeile nicht auch zusätzlich die Zeichenkette " sec: " enthalten.

immer fehlerfrei funktioniert. Insofern wurden im Beispiel insgesamt 5.000.000 Datensätze eingefügt, jedoch lediglich 4.999.830 als erfolgreich abgeschlossen und keine Inserts als fehlgeschlagen[41] registriert. Somit lässt sich anhand des Protokolls nicht zweifelsfrei feststellen, ob die verbleibenden 170 Inserts erfolgreich abgeschlossen wurden oder fehlgeschlagen sind.

```
1 ...
2 [INSERT], Operations, 5000000
3 ...
4 [INSERT], Return=0, 4999830
5 ...
6 [OVERALL], Reconnections, 0.0
7 [OVERALL], RunTime(ms), 299136.0
8 [OVERALL], Operations, 5000000.0
9 [OVERALL], Throughput(ops/sec), 16714.805305947797
```

Listing 4: Fehlerhaftes Zählen erfolgreicher und fehlgeschlagener Operationen

Der Ursprung dieses Fehlers liegt in einem unberücksichtigtem Nebenläufigkeitsproblem bei der Zwischenspeicherung der Messergebnisse. In der Konsequenz stört dieser Umstand die Nachvollziehbarkeit bzgl. einer fehlerfreien Kommunikation zwischen YCSB-Client und Datenbanksystem. Insbesondere bei der Analyse durchgeführter Benchmarks und im speziellen auch bei der Datenbankadapterentwicklung erweist sich dieser Fehler als besonders hinderlich. Direkte Auswirkungen auf die durchschnittlichen Durchsatz- und Latenzstatistiken sind jedoch an dieser Stelle nicht gegeben. Bedingt durch die frühzeitige Feststellung jenes Fehlers und dessen anschließender Behebung sind keine der in dieser Arbeit dokumentierten Messungen hiervon betroffen.

Ein weiteres Problem in Verbindung mit einer unbehandelten Nebenläufigkeitsproblematik betrifft einerseits die Feststellung der Latenz Maxima und Minima, sowie andererseits die Protokollierung der durchschnittlichen Latenz- und Durchsatzwerte über den Zeitraum der Messung. Dieses Problem wurde erst im Rahmen der Vorbereitungen für die Messungen mit Cassandra identifiziert, sodass die vorangegangenen Messungen für Couchbase bereits durchgeführt waren. Bedingt durch den begrenzten Zeitrahmen dieser Arbeit ist eine Wiederholung der Messungen nicht möglich. Aus diesem Anlass wird grundsätzlich auf eine Diskussion der gemessenen Latenz Maxima und Minima verzichtet. In Folge der unbekannten Auswirkung

[41] Protokollausgabe für die Insert-Menge mit Return=1 erfolgt lediglich bei einem Häufigkeitsvorkommen größer 0.

auf die begleitende Zeitreihenprotokollierung wird auf die Verwendung der Protokollergebnisse von der dafür zuständigen Java Klasse *OneMeasurementTimeSeries* verzichtet. Stattdessen wird auf die Ausgaben des Status-Threads zurückgegriffen. Dieser berechnet alle zwei Sekunden den durchschnittlichen Durchsatz pro Sekunde und gibt ihn unmittelbar aus.

Beide Fehler konnten schlussendlich in der originalen YCSB Klasse *OneMeasurementTimeSeries* mittels entsprechender Synchronisierung[42] der Thread-Zugriffe behoben werden.

3.2.2.3. Benchmark-Terminierung

Bereits die originale YCSB Version besitzt zwei Mechanismen zur Terminierung eines ausgeführten Benchmarks. Einerseits wird hierzu die Anzahl der durchzuführenden Operationen limitiert, wobei keine Unterscheidung zwischen den Operationsarten erfolgt. Andererseits existiert zudem die Möglichkeit eine maximale Ausführungsdauer zu definieren, nach deren Ablauf alle Threads beendet werden.

Die Terminierung mittels einer fest definierten Menge von durchzuführenden Operationen stellt sich jedoch als potentiell problematisch dar. Ursache hierfür ist die Verwendung mehrerer Threads zu deren Abarbeitung. Einerseits garantiert die gleichförmige Aufteilung sämtlicher durchzuführender Operationen auf alle Threads eine gleichmäßige Belastung der Threads. Andererseits führt dieses Verhalten zu unterschiedlichen Terminierungszeitpunkten zwischen den einzelnen Threads. Dies ist unter anderem darin begründet, dass zum einen die erforderliche Latenz von Operation zu Operation leicht schwankt und zum anderen, dass nicht garantiert ist, dass jeder Thread gleichmäßig von der CPU bearbeitet wird. Demzufolge nimmt die Anzahl der gleichzeitig an das Datenbanksystem gestellten Operationen gegen Ende des Benchmarks immer weiter ab und führt somit zu einer ungleichmäßigen Belastung des Datenbanksystems. Je nach zu evaluierender Benchmark-Konfiguration kann dies zu einer beschleunigten Bearbeitung der verbleibenden Operationen führen und somit das Messergebnis letztendlich verfälschen. Eine schematische Darstellung eines derartiges Verhaltens kann der Abbildung 25 entnommen werden.

[42] `https://github.com/brianfrankcooper/YCSB/pull/213/commits`, zuletzt besucht am 09.02.2015.

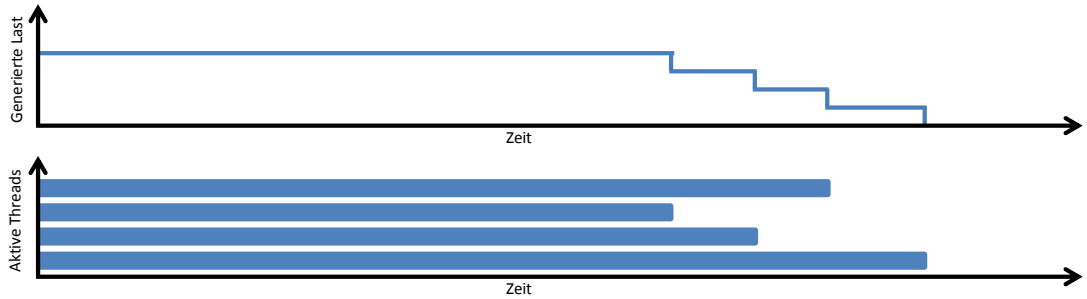


Abbildung 25: Benchmark-Vorbereitung: Terminierung auf Basis durchzuführender Operationen

Vor diesem Hintergrund erfolgt die Workload-Terminierung im Kontext dieser Arbeit ausschließlich auf Basis der maximalen Ausführungsdauer (siehe folgende Abbildung 26). Zu diesem Zweck wird die innerhalb der Benchmark-Konfigurationen definierte Anzahl durchzuführender Operationen mit 2.147.483.647 maximiert. In Folge der auf zehn Minuten festgelegten Benchmark-Dauer[43] wären somit durchschnittlich mehr als 3.579.139 Operationen pro Sekunde erforderlich, damit die Benchmark-Ausführung dennoch auf Basis der Anzahl durchgeführter Operationen vorzeitig beendet werden würde. Ein derart hoher Durchsatz kann jedoch aufgrund der gegebenen Testumgebung ausgeschlossen werden.

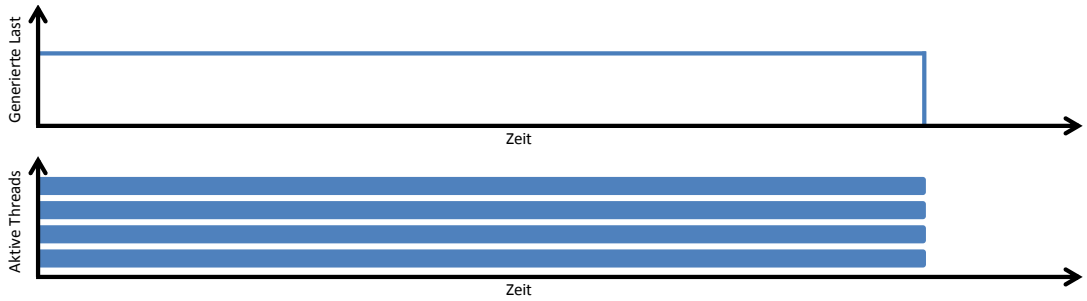


Abbildung 26: Benchmark-Vorbereitung: Terminierung auf Basis durchzuführender Operationen

Im Zuge der Auswertung von Messergebnissen verschiedener Cassandra-Konfigurationen wurde jedoch deutlich, dass die bestehende Realisierung der zeitbasierten Terminierung in Situationen mit hohen Operationsdurchführungslatenzen zu markant variierenden Benchmark-Laufzeiten führt. Bei der zugrundeliegenden Implementierung wird nach Ablauf der gewünschten Ausführungsdauer ein Kennzeichen gesetzt, sodass keiner der aktiven Threads mehr neue Operationen gegen das Datenbanksystem ausführt. Zu dieser Zeit noch aktive Operationen werden zunächst weiterhin ausgeführt. Es wird jedoch damit begonnen sämtliche der noch

[43] Die Benchmark-Dauer von zehn Minuten wird in Kapitel 3.4 erläutert.

verbleibenden Threads sequentiell nach einer jeweiligen Frist von zwei Sekunden von außen her abzubrechen. Somit wäre es bei einer entsprechend hohen Latenz möglich, dass bspw. bei einer Menge von 32 Threads der letzte Thread erst nach über 64 Sekunden abgebrochen werden würde. Dies würde im Extremfall bedeuten, dass der letzte Thread 62 Sekunden mehr Zeit hätte seine Operation zu beenden als der erste Thread. An besonderer Bedeutung gewinnt dieses Detail, weil ausschließlich nach der Ausführung einer Operation festgehalten wird, wie lange der jeweilige Thread schon mit der Ausführung des Benchmarks beschäftigt ist. Unter diesem Aspekt gilt es zu insbesondere zu beachten, dass die Berechnung des Thread-übergreifenden durchschnittlichen Durchsatzes auf Basis der maximal festgehaltenen Messdauer eines Threads erfolgt. In Bezug auf das vorangegangene Beispiel wäre es somit möglich, dass die tatsächliche Messdauer die maximal definierte Ausführungsdauer um bis zu 64 Sekunden übersteigt.

Konkret wurde in diesem Zusammenhang beim Benchmarking von Cassandra mit verschiedenen Thread-Mengen eine um bis zu ~ 10 Prozent höhere Messdauer als die maximal definierte Ausführungszeit benötigt. Neben derart variierenden Messlaufzeiten wirkt sich die bestehende Implementierung zusätzlich noch negativ auf die Benchmark-Umgebung aus. So entfällt durch das Abbrechen noch aktiver Threads die ordnungsgemäße Beendigung der jeweils geöffneten Datenbankverbindungen, sodass diese ggf. bis zum Eintreten eines entsprechenden Timeouts noch Server-seitig verwaltet werden.

Zur Behebung dieser Problematik wurde zum einen die Protokollierung der Messergebnisse dahingehend geändert, dass nach Ablauf der maximalen Ausführungsdauer keine Messwerte mehr aufgenommen werden und die Messdauer nicht auf Basis des Zeitpunkts der zuletzt durchgeführten Operation berechnet wird, sondern auf Basis des Zeitpunktes, an welchem die maximale Ausführungsdauer erreicht wurde. In Folge dieser Änderung ist es nicht mehr erforderlich, dass die noch verbleibenden Threads zeitig abgebrochen werden. Stattdessen wird nun im Normalfall jeder Thread seine noch laufende Operation ordnungsgemäß zu Ende führen und anschließend seine Datenbankverbindung beenden. Der Timeout zum Abbruch eines noch aktiven Threads wurde zudem von zwei Sekunden auf zehn Minuten erhöht, sodass dieser lediglich in Fehlersituationen zur Vermeidung von endlos laufenden Benchmarks zum Einsatz kommt[44].

[44] Eine Erhöhung des Timeouts wird erforderlich, sofern Benchmark-Konfigurationen getestet werden sollen, welche Operationsausführungslatenzen von mehr als 10 Minuten erfordern.

Die hier vorgestellte Problemlösung bleibt dem Benchmarking der Cassandra-Konfigurationen vorbehalten, da die zuvor beschriebene Problematik erst im Rahmen der Messungen verschiedener Cassandra-Konfigurationen und somit nach der Evaluierung von Couchbase festgestellt wurde. Sämtliche der bereits durchgeführten Cassandra-Messungen wurden entsprechend wiederholt. In Anbetracht der für diese Arbeit geltenden zeitlichen Begrenzung konnten die bereits durchgeführten Messungen für Couchbase nicht wiederholt werden. Es bleibt jedoch anzumerken, dass bzgl. der variierenden Messdauer die größte Abweichung beim Benchmarking bzgl. der Persistenzbestätigung[45] aufgetreten ist. Hierbei wurde die tatsächliche Messdauer gegenüber der angestrebten Messdauer um bis zu $\sim 4{,}4\ \%$ ($\sim 13{,}1$ Sekunden) überstiegen.

3.2.2.4. Generieren von JSON Dokumenten mit inkonsistenter Größe

Innerhalb des von Thumbtack bereitgestellten Datenbankadapters für Couchbase wurde eine Inkonsistenz bei der Generierung der erforderlichen JSON Dokumente festgestellt. Entsprechend der Workload-Konfiguration wird im YCSB "Kern" eine Menge an Feldern/Attributen mit vorgegebener Wertlänge generiert und an den Datenbankadapter übergeben. Aufgrund der zufällig generierten Zeichenfolgen sind dabei auch Anführungszeichen Bestandteil eines Wertes. Dies führt dazu, dass diese während der JSON Generierung mittels eines Backslashs maskiert werden (siehe Listing 5). Danach entspricht die Größe einzelner Attributwerte nicht mehr der zugrundeliegenden Konfiguration. Infolgedessen variiert letztendlich auch die Gesamtgröße der an das Datenbanksystem versendeten Dokumente. Dieser Umstand führt zum einen zu zusätzlichen Abweichungen bei der Wiederholung von Messungen und verhindert zum anderen eine korrekte Voraussage des erforderlichen Speicherbedarfs.

```
1 {
2 "f0": "!-,2;.\"7:&",
3 "f1": "9;-*(0%/;,",
4 ...
5 "f8": "'8<<\"51\"7+",
6 "f9": ".)9%5*<$&?"
7 }
```

Listing 5: Exemplarischer Auszug eines JSON Dokuments mit variabler Wertgröße

[45] Siehe Kapitel 4.1.9.2.

Zur Vermeidung dieses Umstands wurde die JSON Dokumentgenerierung dahin gehend ergänzt, dass im Vorhinein Anführungszeichen durch Hochkommata ersetzt werden. Somit bleibt eine konstante Attributlänge garantiert.

3.2.2.5. Fehler beim Erzeugen eines Jobs zur Ausführung des YCSB-Clients

Für die synchrone Ausführung von mehreren YCSB Clients auf unterschiedlichen Nodes wird standardmäßig das Linux-eigene Job-Scheduling mittels der Thumbtack Python/Fabric Erweiterungen verwendet. Hierbei wird zunächst die aktuelle Uhrzeit ermittelt und auf eine volle Minute innerhalb der nächsten 120 Sekunden gerundet. Eine derartige Karenzzeit wird zur Verbreitung der einheitlichen Startzeit im verteilten System benötigt. In der ursprünglichen Implementierung wird dazu auf jedem der YCSB Clients, das Tool *at* unter Angabe der zukünftigen Startzeit (bestehend aus Stunden und Minuten) aufgerufen.

Zu einem Fehler kommt es, sobald die Uhrzeit während der Ausführung des Python Skriptes (bspw. 23:58) größer ist als die neu berechnete Zeit zum Start der YCSB Clients (bspw. 00:00). Dies ist der Fall, weil das *at* Tool in der verwendeten Parametrisierung davon ausgeht, dass die geplante Jobausführung in der Vergangenheit liegt und demnach nicht ausgeführt wird. Eine entsprechende Fehlermeldung kann dem im Anhang befindlichen Listing 8 entnommen werden. Zur Behebung dieser Fehlerproblematik wurde der Aufruf des Tools dahingehend verändert, dass fortan eine explizite Uhrzeit inklusive Datumsangabe übergeben wird.

3.2.3. Sonstige Hinweise

Dieses Kapitel beschreibt einige Besonderheiten, die es bei der Verwendung von (Thumbtacks) YCSB zu beachten gilt und inwiefern diese im Kontext dieser Arbeit berücksichtigt werden.

3.2.3.1. Berechnung der durchschnittlichen Key-Größe

Zur Vorbereitung einzelner Benchmark-Konfigurationen kann es erforderlich seien, den voraussichtlich benötigen Speicherbedarf zu berechnen. Hierzu wird unter anderem die durchschnittliche Größe der verwendeten Keys benötigt. Der von YCSB generierte Key setzt sich aus einem konstantem Präfix und einer gehashten fortlaufenden Zahl zusammen. Der zugrundeliegende Hashing-Algorithmus basiert der Codedokumentation [Kup12] zufolge auf der in der Wikipedia befindlichen Be-

schreibung[46] des Fowler–Noll–Vo Hashing-Algorithmus. Dieser erzeugt einen numerischen Hashwert mit variabler Länge, wodurch die gesamte Key-Größe variiert. Zur besseren Abschätzung der erwarteten durchschnittlichen Größe wurden für verschiedene Mengen fortlaufender Zahlen (von 0 beginnend) die entsprechenden Hashwerte generiert und deren durchschnittliche Größe berechnet.

Mengengröße	$\mathbf{10^0}$	$\mathbf{10^3}$	$\mathbf{10^6}$	$\mathbf{10^9}$	$\mathbf{10^{12}}$
∅ **Hashgröße in Byte**	19	$\sim$ 18,877	$\sim$ 18,879	$\sim$ 18,879	$\sim$ 18,879

Tabelle 3: Benchmark-Vorbereitung: Durchschnittliche FNV64bit Hashgrößen

In Folge dieser Berechnungen (siehe Tabelle 3) wird im weiteren Verlauf der Arbeit eine durchschnittliche Hash-Größe von 18,88 Byte angenommen.

3.2.3.2. Berechnung der durchschnittlichen Datensatzgröße

Die Berechnung des voraussichtlich benötigen Speicherbedarfs erfordert ebenfalls eine Berechnung der durchschnittlichen Größe der erwarteten Datensätze. Grundsätzlich werden Datensätze innerhalb von YCSB mit einer fest definierten Anzahl von Feldern mit jeweils konstanten Wertgrößen generiert und anschließend an die Datenbankadapter übergeben. Für die Berechnung der Dokumentengröße sind insbesondere die Eigenheiten der jeweiligen Datenbanksysteme zu beachten. Beispielsweise erfordern einige dokumentenbasierte Datenbanksysteme, wie etwa Couchbase, eine gesonderte Generierung von JSON Dokumenten. Neben dem dafür zusätzlichen Aufwand schlägt sich diese Gegebenheit ebenfalls in der Größe des letztendlichen Datensatzes bzw. Dokumentes nieder. Verursacht wird dies zum einen durch den JSON syntaktische Overhead und zum anderen durch die jeweils explizite Angabe der Feld bzw. Attributsnamen. Entsprechend benötigt bspw. ein für Couchbase[47] generiertes JSON Dokument in der Standardkonfiguration (10 Felder á 10 Byte Inhalt) statt der theoretischen 100 Byte insgesamt 181 Byte. Ein derartig erzeugtes Dokument kann dem nachfolgenden Listing 6 entnommen werden.

[46] `http://en.wikipedia.org/wiki/Fowler_Noll_Vo_hash`, zuletzt besucht am 30.12.2014.
[47] Unter Verwendung des neu erstellten Datenbankadapters.

```
{
    "f6": ",*9)26,4-,",
    "f7": ",>-.,;'1:$",
    "f8": "'8<<'51'7+",
    "f9": ".)9%5*<$&?",
    "f1": "9;-*(0%/;,",
    "f0": "!-,2;.'7:&",
    "f3": ":!93>43-&1",
    "f2": "9=0()&<71(",
    "f5": "+.>.*>5=2'",
    "f4": "? 4<.>,&)8"
}
```

Listing 6: JSON Dokument im Couchbase-Benchmark

Die hierbei dargestellten Leerzeichen und Zeilenumbrüche dienen lediglich der besseren Lesbarkeit und sind nicht Bestandteil des generierten Dokumentes.

3.2.3.3. Standardkonfiguration für YCSB Update-Operationen

In der Standardkonfiguration wird pro Update-Operation für ein einzelnes zufällig gewähltes Feld ein neuer Wert generiert und an den Datenbankadapter einzeln zur Übertragung übergeben. Besondere Relevanz besitzt diese Gegebenheit in Verbindung mit Datenbanksystemen, welche keine partielle Aktualisierung einzelner Felder eines Datensatzes unterstützten. Hierzu zählen vor allem dokumentenbasierte Systeme wie etwa Couchbase. Hier führt dieses Verhalten dazu, dass die einst initial eingefügten Dokumente mit bspw. jeweils 10 Feldern bei einer Update-Operation durch Dokumente mit nur jeweils einem Feld ersetzt werden.

Dies bedeutet, dass der erforderliche Speicherbedarf mit zunehmender Anzahl durchgeführter Update-Operationen abnimmt. Infolgedessen können mit fortschreitendem Benchmark-Verlauf immer mehr Dokumente im Hauptspeicher gehalten werden. Insbesondere bei gemischten Workloads kann dies zu einer deutlichen Beeinflussung der Messwerte von Read-Operationen führen. Hierbei könnte die verringerte Dokumentengröße einerseits zu einem schnelleren Server-seitigen Lesen und andererseits zu einer verringerten Netzwerk-Übertragungszeit führen.

Zur Vermeidung dieser Problematik wurde die Benchmark-Konfiguration für Couchbase insoweit abgeändert, als das für Update-Operationen immer Dokumente mit jeweils sämtlichen Attributen generiert werden, so dass die Dokumentengröße konstant bleibt.

3.2.3.4. Berücksichtigung fehlgeschlagener Operationen

Ein wesentlicher Aspekt bei der Auswertung der YCSB Protokolle sollte die Berücksichtigung der Menge an erfolgreich durchgeführten und fehlgeschlagenen Operationen seien. Die gesonderte Betrachtung dieser Kennzahlen ist erforderlich, da YCSB auf der einen Seite separat ermittelt wie viele Operationen mit welchem Rückgabe- bzw. Fehlercode abgeschlossen werden, auf der anderen Seite diesen Aspekt jedoch bei der Berechnung der durchschnittlichen Latenz und des durchschnittlichen Durchsatzes vernachlässigt.

An besonderer Relevanz gewinnt dieses Detail sofern man davon ausgeht, dass erfolgreiche und fehlgeschlagene Operationen meist unterschiedliche Latenzverhalten besitzen. Erfahrungsgemäß besitzen fehlgeschlagene Operationen häufig eine niedrigere Latenz als erfolgreich ausgeführte Operationen. Typisch in diesem Zusammenhang ist beispielsweise, dass sich viele Datenbanksysteme unmittelbar mit einer Fehlermeldung beim Client zurückmelden, sobald versucht wird ein Datensatz mit einem (Primary) Key einzufügen, für den bereits ein Datensatz hinterlegt ist. In der Folge entfällt der Aufwand zur Speicherung des Datensatzes, wodurch der Client je nach Datenbanksystem ggf. schneller eine Rückmeldung erhält. Eine vergleichbare Situation kann zudem beim Lesen von Datensätzen auf Basis nicht existenter (Primary) Keys zustande kommen. Entsprechend überträgt das Datenbanksystem lediglich die Information, dass zu dem gesuchten (Primary) Key kein Datensatz existiert. Somit entfällt auf der Serverseite das ggf. erforderliche Lesen des Datensatzes vom Sekundärspeicher und folglich auch der entsprechend höhere Aufwand zur Übertragung des Datensatzes über das Netzwerk.

Aufgrund dieser Gegebenheit wird im Rahmen der Ergebnisdiskussion explizit darauf hingewiesen, sofern bei den durchgeführten Messungen nennenswerte Mengen an fehlgeschlagenen Operationen aufgetreten sind. Zur Vermeidung von Fehlerquellen aufgrund einer unzureichenden Vorbereitung der Benchmark-Umgebung wird das Befüllen der Datenbank mit einer initialen Datenmenge mittels einer entsprechenden Parametrisierung durchgeführt, sodass fehlgeschlagene Insert-Operationen bis zu 1.000 mal mit einer Pause von einer Sekunde wiederholt werden. Somit bleibt sichergestellt, dass sämtliche angenommenen initial vorhandenen Datensätze auch tatsächlich während der Benchmark-Messung innerhalb der Datenbank zur Verfügung stehen.

3.2.3.5. Operations Timeout

Bei der Implementierung von Datenbankadaptern ist es in der Regel erforderlich, explizit den standardmäßig zulässigen Timeout zur Operationsdurchführung zu erhöhen. Beispielsweise besitzt das Couchbase Java Client SDK 1.5.4 einen standardmäßigen Timeout von 5 Sekunden. Sofern dieser Wert nicht erhöht wird, werden Operationen, die eine längere Bearbeitungsdauer benötigen, einerseits als fehlgeschlagen protokolliert, andererseits werden sie gleichermaßen bzgl. der Durchsatz- und Latenzstatistiken berücksichtigt[48]. In der Folge bedeutet dies, dass die gemessene Latenz niemals den konfigurierten Timeout-Wert übertreffen kann und somit zu einer Verfälschung der Messerergebnisse führt. Mit den folgenden Abbildungen 27 und 28 wird diese Problematik skizziert.

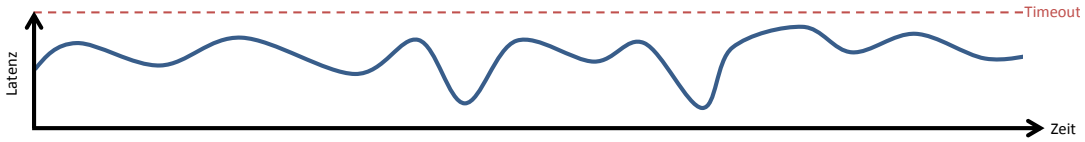


Abbildung 27: Benchmark-Vorbereitung: Nicht erreichbarer Timeout

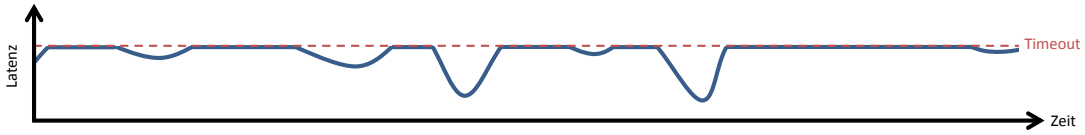


Abbildung 28: Benchmark-Vorbereitung: Niedriger Timeout

Zur Vermeidung dieser Problematik wurde ein Timeout von einer Stunde gleichermaßen für alle Datenbanksysteme definiert. Bedingt durch die gleichzeitig stets geringere Benchmark-Laufzeit wird eine derartige Beeinträchtigung ausgeschlossen.

[48] Je nach Implementierung des Datenbankadapters kann dieses Verhalten abweichen.

3.3. Benchmark-Umgebung

Die im Rahmen dieser Arbeit erfolgten Messungen wurden auf Basis der Infrastruktur des Big Data Clusters[49] der Hochschule Darmstadt durchgeführt. Insgesamt besteht das Cluster aus 28 (Dell PowerEdge C6220) Nodes, welche über zwei separate Dell PowerConnect 7048R-RA 48 GbE Switche redundant miteinander verbunden sind. Bezüglich der durchgeführten Benchmark-Messungen stand jedoch lediglich ein Switch zur Verfügung. Die redundante Netzwerkverbindung war dagegen grundsätzlich administrativen Aufgaben vorbehalten. Von den insgesamt 28 vorhanden Nodes standen neun identische Nodes (siehe Tabelle 4) exklusiv für diese Arbeit bereit.

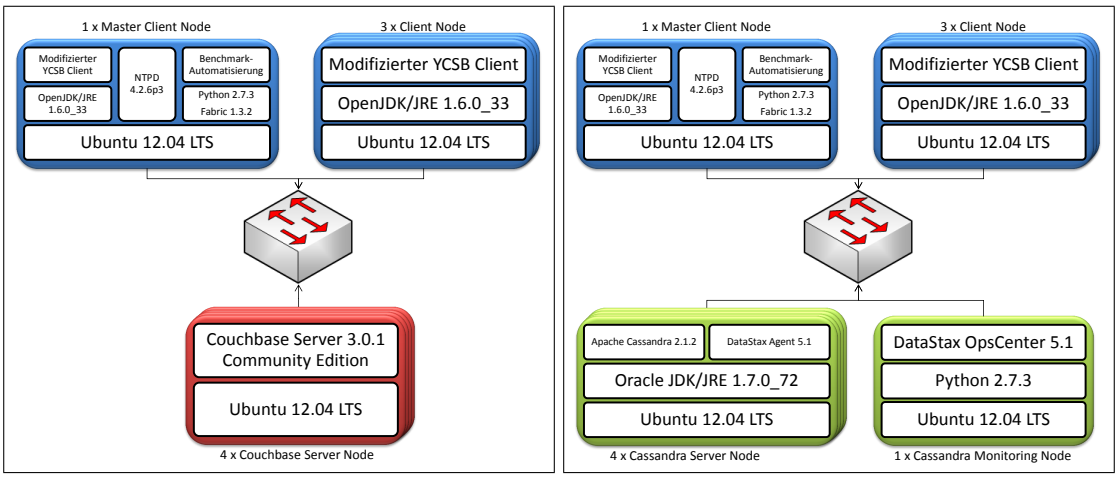
Komponente	**Typ**
CPU	2 x 4 Kerne á 2,4 GHz (2 x Intel(R) Xeon(R) CPU E5-2609 0)
Hauptspeicher	8 x 4 GB DDR3 1600 MHz ECC (8 x Samsung M393B5273DH0-CK0)
Sekundärspeicher	4 x 1 TB SATA mit 7.200 rpm (4 x Seagate Constellation ST91000640NS)
Netzwerk	2 x 1 Gbit/s Ethernet Adapter (2 x Intel Corporation I350 Gigabit Network Connection)

Tabelle 4: Benchmark-Umgebung: Server Hardwarespezifikation

Vier der Nodes wurden zur Ausführung der YCSB Clients und vier weitere für das jeweilige Cassandra bzw. Couchbase Cluster verwendet. Diesbezüglich ist anzumerken, dass jeweils nur die Datenbankinstanzen von Cassandra oder Couchbase gleichzeitig gestartet waren. Der verbleibende neunte Node wurde für das Monitoring des Cassandra Clusters mittels DataStaxs OpsCenter verwendet. Von den vier eingesetzten Client Nodes kommt dem Master Client Node eine besondere Rolle zu. Neben seiner normalen YCSB Client Funktionalität fungiert dieser Node zusätzlich als NTP Server und wird außerdem zur Steuerung und Kontrolle der Benchmark-Automatisierung verwendet. Diese funktionale Zusammenlegung war notwendig, da zu Beginn der Versuche erst acht Nodes exklusiv nutzbar waren. Erst später stand der neunte Node uneingeschränkt zur Verfügung. Im Rahmen verschiedener Testmessungen konnte hierdurch jedoch keine Einflussnahme auf die Ergebnisse festgestellt werden. Anhand der nachfolgenden Abbildung 29 ist die entsprechende funktionale Aufteilung der Nodes nochmals grafisch dargestellt. Zusätzlich gibt die

[49] `https://www.fbi.h-da.de/organisation/personen/stoerl-uta/big-data-cluster.html`, zuletzt besucht am 02.03.2015.

Abbildung Auskunft über die im Wesentlichen verwendeten Softwareprodukte und -versionen.



(a) Benchmark-Umgebung: Couchbase (b) Benchmark-Umgebung: Cassandra

Abbildung 29: Benchmark-Umgebung: Übersicht

Für das Benchmarking von Couchbase-Konfigurationen wurde die Swappiness gemäß der Couchbase Systemdokumentation [Cou14, S. 132, 133] auf 0 verringert. Diese Konfiguration bewirkt, dass der Kernel erst zum spätest möglichen Zeitpunkt mit der Auslagerung von Daten aus dem Hauptspeicher beginnt [MR15]. Des Weiteren wurde entsprechend der Empfehlung des Couchbase Senior Solution Engineers Kirk Kirkconnell [Kir14] die Transparent Huge Pages Funktionalität deaktiviert.

Für das Benchmarking von Cassandra-Konfigurationen wurde entsprechend der empfohlenen Produktionseinstellungen [Dat15b, S. 46 - 48] zum einen die Swapping Funktionalität vollständig deaktiviert. Zum anderen wurden der Kernel Parameter *vm.max_map_count* auf 131072 und die Grenzwerte der Benutzerressourcen für Cassandra (siehe Listing 7) erhöht.

```
1 cassandra - memlock unlimited
2 cassandra - nofile 100000
3 cassandra - nproc 32768
4 cassandra - as unlimited
```

Listing 7: Benchmark-Umgebung: Cassandra user resource limits [Dat15b, S. 46]

Neben derartig empfohlenen Konfigurationsanpassungen wurde zusätzlich beim Benchmarking von Cassandra-Konfigurationen der verfügbare Hauptspeicher auf 8 GB pro (Server) Node begrenzt. Diese Einschränkung erlaubt es mit Hilfe we-

niger Daten relevante Betriebskonfigurationen von Cassandra zu evaluieren. Diesbezüglich ist anzumerken, dass auch die reduzierte Menge von 8 GB Hauptspeicher der Herstellerempfehlung [Dat15b, S. 24] entspricht. Technisch realisiert wurde diese Beschränkung mit Hilfe des Kernel Parameters mem, welcher den sichtbaren bzw. zugreifbaren physischen Speicher limitiert [Lin15]. Um unter Linux die gewünschte Menge von 8 GB Hauptspeicher nutzen zu können, ist bei der gegebenen Systemkonfiguration eine Parametrisierung mit $mem = 10492M$ erforderlich.

3.4. Benchmark-Ablauf

Für die Evaluierung der unterschiedlichen Benchmark-Konfigurationen wurden vier eigene Workload-Profile (siehe Tabelle 5) definiert. Hierbei handelt es sich um drei Operationsarten reiner Workloads, welche eine fokussierte Betrachtung der Auswirkungen auf Read, Update oder Insert-Operationen erlauben. Beim vierten Workload werden lesende und schreibende Operationen gleichermaßen berücksichtigt, wobei sich die schreibenden Operationen gleichwertig in Updates und Inserts unterteilen. Somit erlaubt dieser Workload eine Untersuchung ggf. auftretender Wechselwirkungen zwischen den Operationsarten. Für sämtliche Workloads wurde bzgl. der Anfrageverteilung die Uniform-Verteilung gewählt. Somit wird sichergestellt, dass es bei relativ geringen Datenmengen nicht unbeabsichtigt zu reinen In-Memory Zugriffsszenarien kommt.

Workload-Profile	**Operationsarten**	**Anfrageverteilung**
Workload R (Read-only)	100 % Reads	Uniform
Workload U (Update-only)	100 % Updates	Uniform
Workload I (Insert-only)	100 % Inserts	Keine Relevanz
Workload M (Mixed)	50 % Reads, 25 % Updates, 25 % Inserts	Uniform

Tabelle 5: Benchmark-Ablauf: YCSB Workload-Profile

Der grundsätzliche Benchmark-Ablauf ist gleichermaßen für Couchbase und Cassandra in der nachfolgenden Abbildung 30 dargestellt.

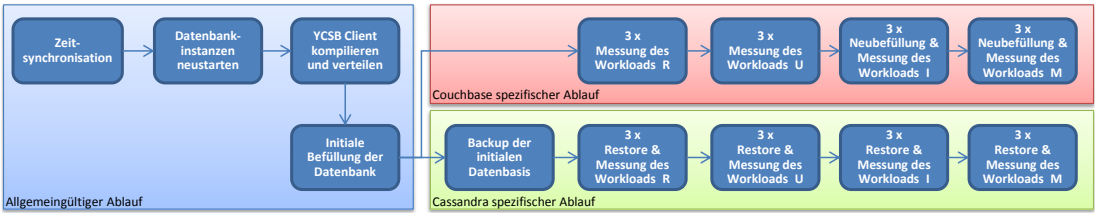


Abbildung 30: Benchmark-Ablauf: Schematische Ablaufdarstellung

Insgesamt betrachtet sind die durchgeführten Schritte sowohl für Couchbase als auch für Cassandra nahezu identisch. Entsprechend beginnt das Benchmarking einer jeden Konfiguration bei beiden Systemen stets mit der Zeitsynchronisierung von Client und Server Nodes mit dem Master Client Node. Anschließend werden die Datenbankinstanzen auf den Server Nodes neu gestartet. Des Weiteren wird auf dem Master Client Node der YCSB Client kompiliert und an sämtliche Client Nodes verteilt. Daraufhin wird die jeweilige Datenbank entsprechend der Benchmark-Konfiguration mit einer initialen Datenmenge befüllt. Dieser Schritt erfolgt bereits unter Verwendung der YCSB Clients. Nachdem dieser Vorgang abgeschlossen ist, werden grundsätzlich bei beiden Datenbanksystemen die vier Workloads mittels der YCSB Clients gemessen.

Identisch sind bei beiden Systemen auch die Reihenfolge, die Laufzeit und die Anzahl der durchgeführten Wiederholungen. So wird stets zuerst der Workload R, danach der Workload U, anschließend der Workload I und abschließend der Workload M gemessen. Die hier gewählte Reihenfolge[50] hat den Vorteil, dass sich die Anzahl der persistierten Datensätze erst bei den letzten beiden Workloads vergrößert[51]. Folglich muss somit erst bei den letzten beiden Workloads sichergestellt werden, dass sich lediglich die vorgegebene Anzahl an Datensätzen in der Datenbank befindet.

Während des Benchmarkings einer Couchbase Konfiguration wird hierzu vor jeder Messung der Workloads I und M die Datenbank mit einer neuen initialen Datenmenge befüllt. Aufgrund der niedrigeren Geschwindigkeit mit der Datensätze in Cassandra initial importiert werden können, ist diese Herangehensweise hierbei weniger geeignet. Stattdessen wird bei Cassandra von der initialen Datenmenge einmalig ein Backup erstellt, welches vor jeder einzelnen Messung wiederhergestellt wird.

[50] Die gewählte Reihenfolge folgt dem Prinzip der beim original YCSB [Coo10] vorgeschlagenen Ausführungsreihenfolge für die originalen YCSB Workloads.

[51] Bedingt durch die Ausführung von Insert-Operationen wächst die Datenmenge fortwährend an.

Jeder Workload wird insgesamt dreimal hintereinander ausgeführt, wobei jede Ausführung jeweils eine Laufzeit von zehn Minuten besitzt. Diese unterteilt sich in eine fünfminütige Warmup-Phase und eine effektive Messphase von weiteren fünf Minuten. Sowohl die hier gewählte Anzahl an durchgeführten Wiederholungen, als auch die Laufzeit einer einzelnen Workload-Messung stellen einen Kompromiss zwischen einer höchstmöglichen Messstabilität und möglichst niedrigen Gesamtlaufzeit dar.

Um neben anwachsenden Datenmengen weitere Beeinflussungen zwischen den Messungen zu vermeiden, wird einerseits vor jeder Messungsdurchführung der betriebssystemseitige Page Cache geleert. Andererseits werden datenbankspezifische Statistiken abgerufen[52], um festzustellen inwiefern die Verarbeitung vorangegangener Operationen abgeschlossen wurde.

Diesbezüglich wird für Couchbase bei jeder Server-Instanz überprüft, dass sämtliche Daten auf dem Sekundärspeicher persistiert und je nach Konfiguration auch repliziert wurden. Des Weiteren wird sichergestellt, dass noch laufende Compaction Vorgänge abgeschlossen werden, bevor mit der nächsten Messung begonnen wird.

Für Cassandra wird einerseits sichergestellt, dass sämtliche Server-Instanzen miteinander kommunizieren und sich gegenseitig ihren jeweiligen normalen Bereitschaftszustand mitgeteilt haben. Des Weiteren wird kontrolliert, dass keine Instanzen auf ausstehende Netzwerkantworten oder noch zu verarbeitende Netzwerkbefehle warten. Darüber hinaus wird überprüft, ob ggf. laufende interne Cassandra Prozesse abgearbeitet werden. Hierzu zählen bspw. Prozesse der Compaction und der Persistierung abgeschlossener Memtable's.

Bedingt durch verwendete Append-Only-Zugriffsstrategie von Couchbase und Cassandra beim Persisteren der Datensätze erfolgt für beide Systeme jeweils nach dem Befüllen mit den initialen Daten der explizite Aufruf der internen Compaction Funktionalität. Somit wird sichergestellt, dass zu Beginn einer jeden Messung die zugrundeliegenden Daten gleichermaßen durch die Compaction vorbereitet wurden.

[52] Die zur Überprüfung der Betriebsbereitschaft erstellten Python/Fabric Methoden können dem Anhang A.2 (Seite 122 - 123) entnommen werden.

4. Benchmark-Durchführung

In diesem Kapitel erfolgt zum einen die Beschreibung der im Rahmen dieser Arbeit betrachteten Benchmark-Konfigurationen. Zum anderen wird in Hinblick auf das Leistungsvermögen des jeweils zugrundeliegenden NoSQL-Datenbanksystems analysiert, inwiefern sich bestimmte Konfigurationen im Vergleich verhalten. Infolge des wechselseitigen Verhältnisses zwischen dem gemessenen Durchsatz und der Latenz, beschränkt sich die Darstellung der Messergebnisse weitestgehend auf den durchschnittlichen Durchsatz an durchgeführten Operationen pro Sekunde. Zu jeder verwendeten Messung können die durchschnittlichen absoluten Durchsatz- und Latenzwerte dem Anhang ab Seite 124 entnommen werden.

4.1. Couchbase

Im Rahmen dieses Kapitels wird beschrieben, inwiefern sich verschiedene Konfigurationen auf die Leistungsfähigkeit des Couchbase Community Servers 3.0.1 auswirken. Zunächst wird hierfür eine Default-Konfiguration festgelegt, welche den anschließenden Messungen zugrunde liegt. Daraufhin wird Couchbase hinsichtlich der folgenden acht Aspekte untersucht:

- Couchbase Java Client SDK
- Unterschiedliche Dokumentengrößen
- Unterschiedliche initiale Dokumentenmengen
- Auto-Compaction
- Couchbase Bucket vs. Memcached Bucket
- Cache Eviction
- Asynchrone Replikation
- Bestätigung schreibender Operationen

Abschließend erfolgt eine kurze Zusammenfassung der wichtigsten Ergebnisse.

4.1.1. Festlegung einer geeigneten Default-Konfiguration

Bevor mit der Messung verschiedener Benchmark-Konfigurationen begonnen werden kann, ist es erforderlich eine Default-Konfiguration festzulegen. Diese Maßnahme ist notwendig, damit weitestgehend eine Vergleichbarkeit zwischen den für

Couchbase beschriebenen Messungen gegeben ist. Sofern nicht Änderungen an den hier definierten Parametern im Speziellen evaluiert werden sollen, wird jede der im gesamten Kapitel 4.1 beschriebenen Messungen mit den hier festgelegten Parametern durchgeführt. Ausnahmen hiervon werden im Rahmen der Ergebnispräsentation dokumentiert.

Wie bereits in Kapitel 3.3 beschrieben, wird für die Default-Konfiguration standardmäßig eine Menge von acht Nodes verwendet. Diese teilt sich gleichmäßig in vier YCSB Client Nodes und vier Couchbase Server Nodes auf. Für die Persistierung von Couchbase selbst und der dazugehörigen Daten werden insgesamt drei Festplatten verwendet. Dabei wurde der Couchbase Community Server auf der gleichen Festplatte installiert wie das Betriebssystem. Couchbase dagegen speichert seine Dokumenten- und Indexdaten getrennt voneinander auf jeweils einer separaten Festplatte, welche exklusiv durch Couchbase verwendet werden.

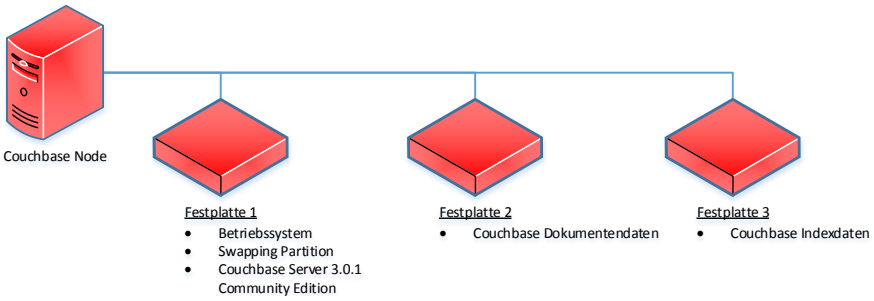


Abbildung 31: Couchbase: Physische Datenverteilung

Die Speicherverwendung durch Couchbase wird auf Seiten der Couchbase-Konfiguration auf 80% der insgesamt verfügbaren Hauptspeichermenge begrenzt und beträgt somit 25.708 MB pro Server Node und $\sim$ 100 GB für das gesamte Cluster. Während sämtlicher Messungen existiert zeitgleich ausschließlich ein einzelner Couchbase Bucket, sodass diesem der gesamte zugewiesene Hauptspeicher zur Verfügung steht. Zusätzlich wurde beim Couchbase Bucket die Flush Option aktiviert. Dies erlaubt es, nach der Ausführung verschiedener Benchmark-Messungen, den gesamten Bucket mit nur einem Befehl zurückzusetzen und somit alle seine Daten auf einmal zu entfernen. Des Weiteren wurde die Standardkonfiguration für die automatische Aktivierung der Compaction auf dem Default-Wert von 30% bzgl. der Datenbank- und View-Fragmentierung belassen. Eine Replikation erfolgt innerhalb der Default-Konfiguration hingegen nicht, deren Auswirkung wird jedoch in den Kapiteln 4.1.8 und 4.1.9.1 gesondert betrachtet. Entsprechend des Kapitels 3.2.3.3 wurde der YCSB Parameter *writeallfields* auf *true* gesetzt. Hier-

durch wird letztendlich verhindert, dass der erforderliche Speicherbedarf infolge von Update-Operationen sinkt.

4.1.1.1. Festlegung einer geeigneten Thread-Anzahl pro Client Node

Zu Beginn ist es erforderlich die Menge an parallel ausgeführten Threads pro YCSB-Client Node zu bestimmen. Dies ist notwendig, damit jeder Client Node in der Default-Konfiguration so viel Last wie möglich erzeugt, ohne dass Messergebnisse unbeabsichtigt negativ beeinträchtigt werden. Mit der hier ermittelten Anzahl an Threads pro Node werden alle folgenden Couchbase Benchmark-Konfigurationen getestet.

Zunächst wurden hierzu eine Reihe von Messungen mit einem einzelnen Client Node und einer Thread-Anzahl von eins bis 128 durchgeführt. Jeder Messung lag dabei eine initiale Menge von 10 Millionen Dokumenten zugrunde. Wie sich anhand der Abbildung 32 erkennen lässt, steigt der durchschnittliche Durchsatz an durchgeführten Operationen pro Sekunde mit zunehmender Anzahl an Threads, bis dieser bei 128 parallel ausgeführten Threads stagniert. Auffällig ist zudem, dass sich mit jeder Verdoppelung der Threads lediglich bis zur Thread-Menge 16 auch der durchschnittliche Durchsatz nahezu verdoppelt. Ab 32 Threads nimmt die Durchsatzsteigung mit jeder weiteren Verdoppelung deutlich ab.

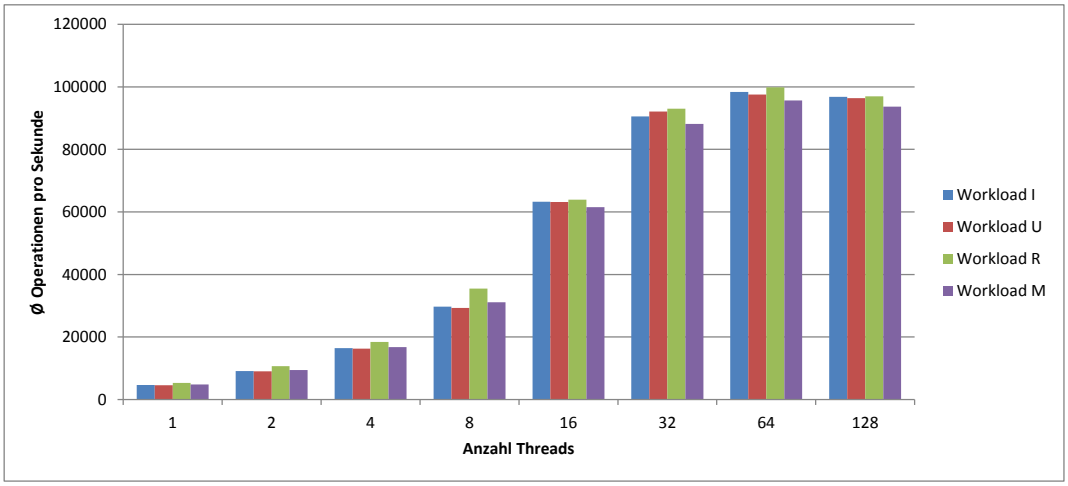


Abbildung 32: Couchbase: Threads pro Client - Durchsatz

Ungeachtet der Steigungsraten erreicht die Messung mit 64 Threads den insgesamt höchsten Durchsatz. Um die Konsistenz dieses Ergebnisses zu überprüfen wird zudem die durchschnittliche Latenz pro Operation untersucht. Je nach Auslastung des Couchbase Clusters sollte die Latenz mit zunehmender Thread-Anzahl ent-

weder konstant bleiben oder steigen. Zu erwarten ist, dass ein einzelner Client, unabhängig von der verwendeten Thread-Menge nicht ausreichend Last gegen das gesamte Couchbase Cluster generieren kann, sodass die Latenz annähernd konstant bleibt. Mittels Abbildung 33 folgt eine entsprechende Darstellung der gemessenen durchschnittlichen Latenzen pro Operation für einen einzelnen Client bei einem bis 128 parallel ausgeführten Threads.

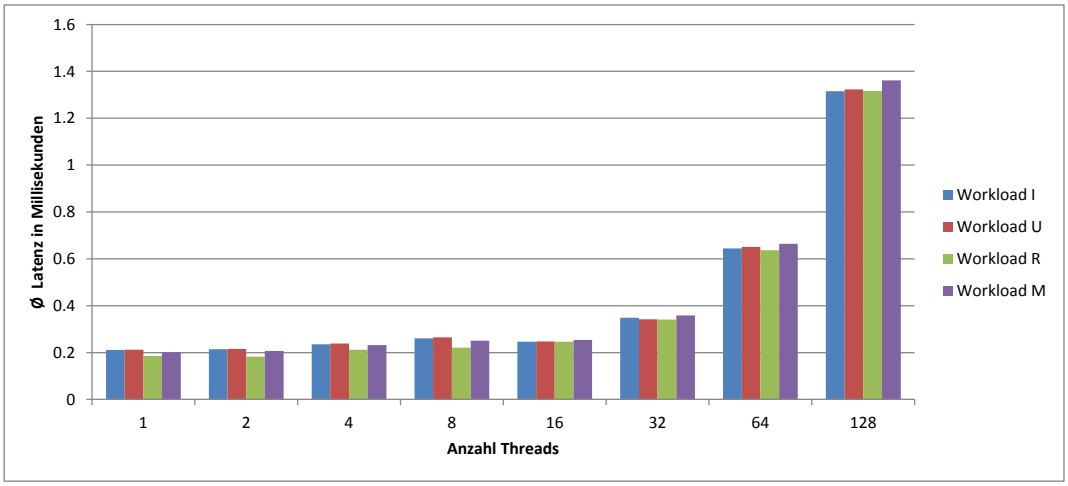


Abbildung 33: Couchbase: Threads pro Client - Latenz

Mit Blick auf Abbildung 33 fällt auf, dass die Latenz bis einschließlich 16 Threads nahezu stabil bleibt und ab 32 Threads stetig steigt, wodurch sich auch die in Abbildung 32 abnehmende Durchsatzsteigerung erklärt. Auf Basis der bisher ermittelten Erkenntnisse ist eine Festlegung der geeigneten Thread-Anzahl pro Client noch nicht zweifelsfrei möglich. Schließlich wäre es theoretisch möglich, dass entgegen der zuvor aufgestellten These, das Couchbase Cluster bereits ab 32 Threads hinsichtlich seiner Leistungsfähigkeit stagniert. Zur weiteren Überprüfung werden sämtliche möglichen Mengen von Clients[53] und Threads[54] kombiniert, sodass Client-übergreifend 16, 32 bzw. 64 Threads gegen das gesamte Cluster ausgeführt werden. Mit Abbildung 34 folgt die entsprechende Darstellung bzgl. des ermittelten durchschnittlichen Durchsatzes.

[53] Zugrundeliegende Clientmenge: {1, 2, 4}
[54] Zugrundeliegende Thread-Menge: {4, 8, 16, 32}

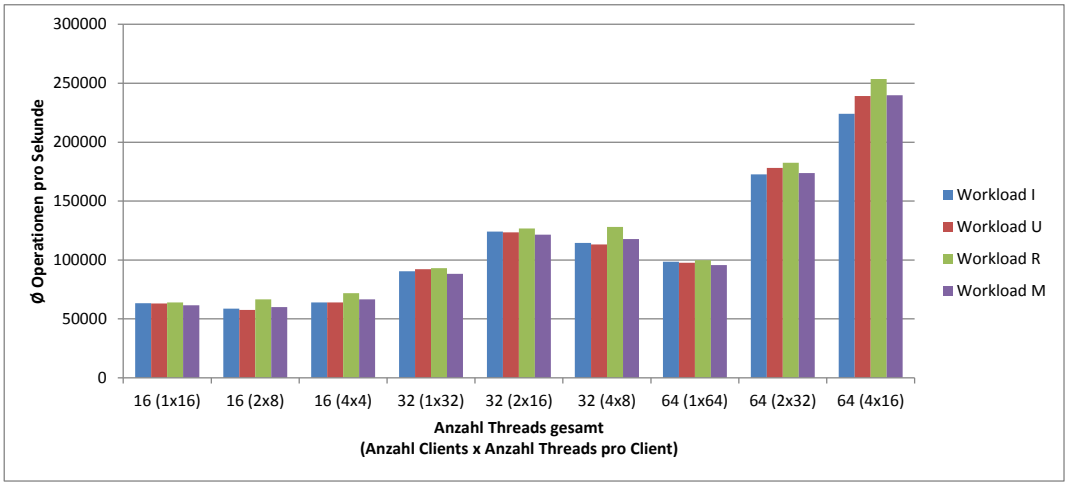


Abbildung 34: Couchbase: Client x Thread Kombinationen - Durchsatz

Während es bei insgesamt 16 Threads keinen wesentlichen Unterschied bzgl. der unterschiedlichen Kombinationsmöglichkeiten aus Clients und Threads zu erkennen gibt, zeigt sich dieser bei insgesamt 32 und 64 Threads umso deutlicher. Direkt ersichtlich wird, dass die Ausführung der Kombinationen 1 Client x 32 Threads und 1 Client x 64 Threads einen deutlich geringeren Durchsatz erreichen als ihre äquivalenten Konfigurationen mit doppelt sovielen Clients und der Hälfte an Threads. Infolgedessen erreicht die Kombination 4 Clients x 16 Threads einen bis zu 2,539 höheren Durchsatz[55] als die Kombination mit einem Client und 64 Threads, welche in den vorangegangenen Messungen mit nur einem Client den insgesamt höchsten Durchsatz erzielte (siehe Abbildung 32).

Mit der weiteren Betrachtung der zugehörigen durchschnittlichen Latenz (siehe Abbildung 35) wird zudem deutlich, dass die Latenz bei beliebig vielen Clients und mit bis zu 16 Threads vergleichsweise geringfügig ansteigt. Dagegen steigt die Latenz ab einer Menge von 32 Threads pro Client deutlich an.

[55] Vergleich erfolgt auf Basis des Workloads R.

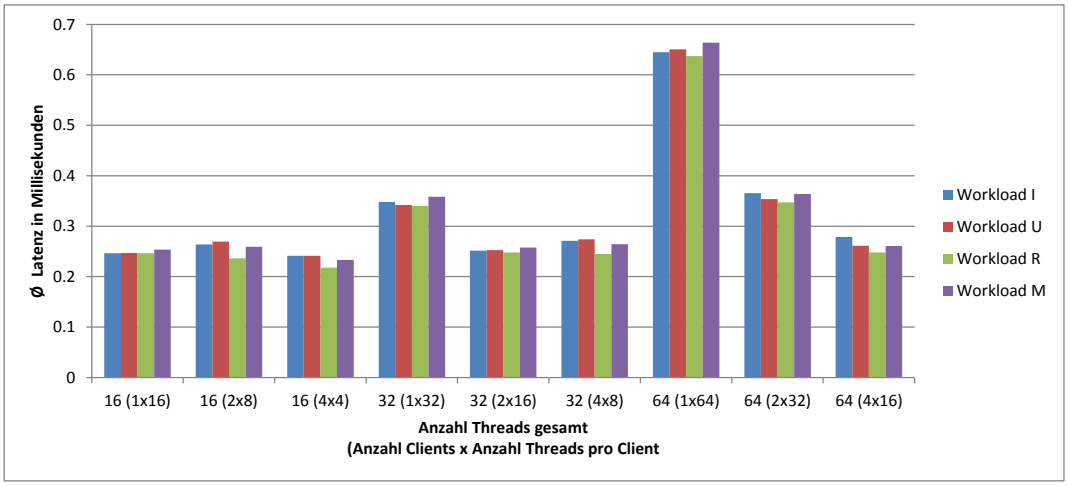


Abbildung 35: Couchbase: Client x Thread Kombinationen - Latenz

In Folge der durchgeführten Messungen lässt sich feststellen, dass die ab 32 Threads pro Client festgestellten erhöhten Latenzzeiten und verminderten Durchsatzsteigerungen nicht auf einen Flaschenhals seitens des Clusters, sondern auf begrenzte Verarbeitungskapazitäten auf Seite des Client Nodes zurückzuführen sind. Eine mögliche Ursache hierfür könnte in dem kontextwechselbedingten Overhead liegen, welcher insbesondere zur Geltung kommt, sobald mehr Threads unter hoher Last ausgeführt werden als Prozessorkerne zur Verfügung stehen. Hierdurch würde sich auch erklären, warum der beobachtete Effekt mit zunehmender Thread-Anzahl umso drastischer in Erscheinung tritt. Eine weiterführende Untersuchung, welche diese These bekräftigt, wird nicht durchgeführt.

Schlussendlich kann festgehalten werden, dass für die untersuchten Mengen von Threads pro Client eine parallele Ausführung von 16 Threads pro Client als optimal für die gegebene Hardwarekonfiguration erscheint. Es gilt jedoch zu beachten, dass es sich hierbei nicht zwangsläufig um die insgesamt "optimale" Menge an Threads handelt. Mit zusätzlichen Benchmarks könnte untersucht werden, ob eine Thread-Menge zwischen 16 und 32 noch höhere Durchsätze bei ähnlicher Latenz erlaubt. Hiervon wird jedoch abgesehen, da die Zielsetzung dieser Arbeit nicht darin besteht, die gegebene Hardware hinsichtlich der Performanz restlich auszuschöpfen.

4.1.1.2. Festlegung einer geeigneten initialen Dokumentenmenge

Neben der Anzahl an parallel ausgeführten Treads pro Client Node muss zudem eine Dokumentenmenge definiert werden, welche standardmäßig ins Couchbase Cluster geladen wird und somit einer jeden Benchmark-Durchführung zugrunde

liegt. Ziel ist es hierbei die Menge an Dokumenten so groß bzw. klein zu wählen, dass der Großteil aller Benchmark-Konfigurationen hinsichtlich der gesamten vier Workloads evaluiert werden kann, ohne dass dabei die empfohlenen Vorgaben seitens des Herstellers verletzt werden.

In Anbetracht der von Couchbase beschriebenen Dokumentation bezüglich der laufenden Überwachung und Wartung des Datenbanksystems [Cou14, S. 128 - 129] wird die standardmäßige Datenmenge so gewählt, dass sämtliche Dokumente im Cache gehalten werden können. Hintergrund hierfür ist die in der Dokumentation getätigte Beschreibung bzgl. des Cache-Auslagerungsverhaltens bei erhöhter Speichernutzung, dessen Verhalten maßgeblich von zwei Schwellwerten beeinflusst wird. Der Schwellwert *Low Water Mark* liegt bei 75 Prozent des gesamten, für Couchbase zur Verfügung stehenden, Hauptspeichers. Wird dieser Wert erreicht, erfolgt noch keine systemseitige Aktion, jedoch wird darauf hingewiesen, dass sich das Datenbanksystem bzgl. des Speicherverbrauchs in eine kritische Richtung entwickelt. Wird dagegen der Schwellwert *High Water Mark* (85 Prozent) erreicht fängt das System an, so lange Daten auszulagern[56] bis der Schwellwert *Low Water Mark* wieder unterschritten wird. Dieses Verhalten stellt aus Sicht der Datenbankarchitektur noch keine generelle Besonderheit dar, jedoch enthält die Dokumentation zusätzlich einen ausdrücklichen Hinweis darauf, dass die Systemadministration vor dem Erreichen der *High Water Mark* eingreifen sollte. [Cou14, S. 106]

In Folge dessen scheint Couchbase für Betriebsszenarien optimiert zu sein, in denen sämtliche Daten im Hauptspeicher gehalten werden können. Ebenfalls bekräftigt wird diese Annahme durch Präsentationsunterlagen von der Couchbase Connect 2014 [Mic14]. Hier gilt ein Datenbanksystem als "ungesund", sobald der verwendete Hauptspeicher den Schwellwert *High Water Mark* erreicht und/oder die systeminternen Metriken *Disk Reads per Sec* und *Cache Miss Ratio* ungleich 0 sind.

Zu der Bestimmung einer konkreten Dokumentenmenge wurde die von Couchbase zur Verfügung gestellte Formel [Cou14, S. 120 - 123] verwendet. Diese ermöglicht es, auf Basis einer vorgegebenen Dokumentenmenge, den erforderlichen Hauptspeicherbedarf und folglich die Anzahl benötigter Server Nodes zu berechnen. Als Resultat derer Verwendung konnte eine initiale Dokumentenmenge von 60 Millionen Dokumenten als weitestgehend geeignet bestimmt werden. Es folgt eine

[56] Aktive Dokumente und deren Replikate werden im Verhältnis 60:40 ausgelagert [Cou14, S. 106].

exemplarische[57] Darstellung der durchgeführten Berechnungen und der daraus resultierenden Konsequenzen.

Die Grundlage der folgenden Berechnungen bilden die anschließend beschriebenen Konstanten (siehe Tabelle 6) und Variablen (siehe Tabelle 7).

Konstante	**Beschreibung**	**Wert**
metadata_per_document	Von Couchbase intern benötigte Metadatenmenge pro Dokument beträgt 56 Byte.	56 Byte
headroom	Zusätzlicher Overhead an Metadaten, welchen Couchbase zusätzlich benötigt. Empfohlen werden 25% des Speichers bei SSDs und 30% bei herkömmlichen Festplatten.	0,3
high_water_mark	Schwellwert zur Auslagerung von Dokumenten aus dem Couchbase Cache. Standardmäßig liegt dieser bei 85%.	0,85

Tabelle 6: Couchbase: Übersicht notwendiger Konstanten zur Berechnung einer geeigneten initialen Dokumentenmenge [Cou14, S. 121]

Variable	**Beschreibung** [Cou14, S. 120 - 121]	**Wert**
documents_num	Anzahl erwarteter Dokumente	198.000.000
id_size	Durchschnittliche Key-Größe	32,88 Byte[58]
value_size	Durchschnittliche Dokumentengröße	181 Byte[59]
number_of_replicas	Anzahl der Replikate pro Dokument	0
working_set_percentage	Prozentualer Anteil der im Cache zu haltenden Daten. Entsprechend gültiger Wertebereich [0,1].	1,0
per_node_ram_quota	Zur Verfügung gestellter Hauptspeicher für einen Bucket pro Node.	25.708 MB

Tabelle 7: Couchbase: Übersicht notwendiger Variablen zur Berechnung einer geeigneten initialen Dokumentenmenge

[57] Die Berechnung wird ausschließlich für eine initiale Dokumentenmenge von 60 Millionen beschrieben, bei welcher zusätzlich noch die aus dem Workload I resultierenden Dokumente berücksichtigt werden.

[58] 32,88 Byte setzen sich aus einem 14 Byte großen Prefix und einem durchschnittlich 18,88 Byte großem Hashwert zusammen. Für weitere Erläuterungen siehe Kapitel 3.2.3.1.

[59] 181 Byte setzen sich aus 10 Werten mit jeweils 10 Byte Inhalt und einem zusätzlichen JSON spezifischem Overhead zusammen. Die Dokumentengröße wurde gleichermaßen auf YCSB Client Seite sowie mittels der von Couchbase zur Verfügung gestellten MapReduce-Funktion [Roo13] validiert.

Die Menge der erwarteten 198 Millionen Dokumente setzt sich zusammen aus der initialen Dokumentenmenge von 60 Millionen und einer erwarteten Menge von 138 Millionen, welche im Rahmen des Insert-Only Workloads in das Datenbanksystem geladen werden könnte. Zugrunde gelegt wird hierbei eine durchschnittliche Anzahl möglicher Inserts von 230.000[60] pro Sekunde, bei einer Laufzeit von zehn Minuten.

Es folgt die Berechnung der Anzahl benötigter Nodes für die veranschlagten 198 Millionen Dokumente nach der Couchbase Formel [Cou14, S. 120 - 123]. Hierfür wird zunächst die Anzahl erforderlicher Dokumentkopien festgelegt. Anschließend wird über die Größe des benötigten Speicherbereichs für Metadaten und Dokumente die Anzahl der erforderlichen Server Nodes berechnet.

$$
\begin{aligned}
\text{no\_of\_copies} &= 1 + \text{number\_of\_replicas} \\
&= 1 + 0 \\
&= \underline{1}
\end{aligned}
$$

$$
\begin{aligned}
\text{total\_metadata} &= (\text{documents\_num}) * \\
&\quad (\text{metadata\_per\_document} + \text{id\_size}) * \\
&\quad (\text{no\_of\_copies}) \\
&= 198.000.000 * (56 + 32{,}88) * 1 \\
&= \underline{17.598.240.000 \text{ Byte } (\approx 16{,}39 \text{ GB})}
\end{aligned}
$$

$$
\begin{aligned}
\text{total\_dataset} &= (\text{documents\_num}) * (\text{value\_size}) \\
&\quad (\text{no\_of\_copies}) \\
&= 198.000.000 * 181 * 1 \\
&= \underline{35.838.000.000 \text{ Byte } (\approx 33{,}38 \text{ GB})}
\end{aligned}
$$

$$
\begin{aligned}
\text{working\_set} &= \text{total\_dataset} * (\text{working\_set\_percentage}) \\
&= 35.838.000.000 * 1{,}0 \\
&= \underline{35.838.000.000 \text{ Byte } (\approx 33{,}38 \text{ GB})}
\end{aligned}
$$

[60] Auf zehntausend aufgerundete 224.098 Inserts pro Sekunde, gemessen für 4 Clients mit 16 Threads (siehe Kapitel 4.1.1.1).

$$
\begin{aligned}
\text{cluster\_ram\_quota\_required} &= (\text{total\_metadata} + \text{working\_set}) * \\
&\quad (1 + \text{headroom}) \,/\, (\text{high\_water\_mark}) \\
&= (17.598.240.000 + 35.838.000.000) * \\
&\quad (1 + 0{,}3) \,/\, 0{,}85 \\
&= 53.436.240.000 * 1{,}3 \,/\, 0{,}85 \\
&= \underline{81.726.014.117{,}64706 \text{ Byte } (\approx 76{,}11 \text{ GB})}
\end{aligned}
$$

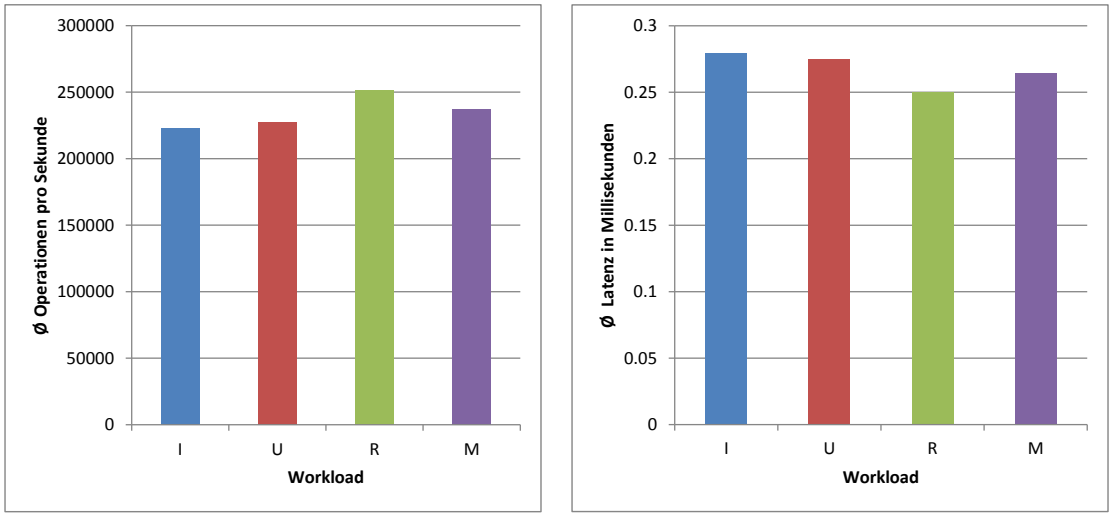
$$
\begin{aligned}
\text{number\_of\_ nodes} &= \text{cluster\_ram\_quota\_required} \,/ \\
&\quad \text{per\_node\_ram\_quota} \\
&= 81.726.014.117{,}64706 \,/\, 26.956.791.808 \\
&= \underline{\underline{3{,}031741117405267}}
\end{aligned}
$$

Entsprechend der Berechnung werden nicht mehr als die vier standardmäßig verwendeten Server Nodes benötigt, um sämtliche Dokumente vollständig im Hauptspeicher halten zu können. Stattdessen existiert noch ein Puffer, sodass sichergestellt ist, dass nicht bereits die Default-Konfiguration an den empfohlenen Systemgrenzen operiert. Somit wird sichergestellt, dass ein Großteil der geplanten Konfigurationen mit sämtlichen vier Workloads im Rahmen der Herstellerempfehlungen evaluiert werden kann. Beispielsweise werden bei einer größeren initialen Dokumentenmenge von 120 Millionen Dokumenten und dem anschließend durchgeführten Workload I voraussichtlich $\sim$ 3,95 Server Nodes benötigt. Demnach werden bei dieser Evaluationskonfiguration alle verfügbaren Server Nodes nahezu vollständig ausgelastet.

Als grundsätzlich problematisch stellt sich das Messen von Konfigurationen in Verbindung mit eingeschalteter Replikation dar. Insofern führt bereits eine einfache Replikation dazu, dass sich der erforderliche Speicherbedarf verdoppelt und in der Konsequenz $\sim$ 6,06 Server Nodes erforderlich sind. Selbst eine leere initiale Dokumentenmenge würde bei einfacher Replikation noch $\sim$ 4,22 Server Nodes erfordern und somit die Anzahl verwendeter Server Nodes übersteigen. Bei dreifacher Replikation benötigt bereits die ausgewählte initiale Dokumentenmenge von 60 Millionen $\sim$ 3,67 Server Nodes, ohne dabei Dokumente aus dem Workload I zu berücksichtigen. Dies zeigt deutlich, dass es zumindest unter Einhaltung der Herstellerempfehlungen nicht möglich ist, jeden der vier Workloads in allen Systemkonfigurationen zu evaluieren.

4.1.1.3. Messergebnisse zur Default-Konfiguration

Innerhalb dieses Kapitels wird die Leistungsfähigkeit von Couchbase hinsichtlich der Default-Konfiguration analysiert. Entsprechend der vorangegangen Untersuchungen wird dafür eine initiale Datenbasis von 60 Millionen Dokumenten und eine Verwendung von vier YCSB Client Nodes mit jeweils 16 Threads zugrunde gelegt.



(a) Durchschnittlicher Durchsatz (b) Durchschnittliche Latenz

Abbildung 36: Couchbase: Messergebnisse zur Default-Konfiguration

Anhand der Abbildung 36(a) wird deutlich, dass der größte Durchsatz mit $\sim$ 251 Tausend Read-Operationen pro Sekunde erzielt wird. Der gemischte Workload M erreicht dabei den zweithöchsten Durchsatz mit $\sim$ 237 Tausend Operationen. Dagegen erreichen die beiden ausschließlich schreibenden Workloads U und I mit $\sim$ 227 Tausend Operationen und $\sim$ 223 Tausend Operationen pro Sekunde den geringsten Durchsatz. Beim direkten Vergleich mit Workload R sinkt der Durchsatz bei Workload M um $\sim$ 5,85 %, bei Workload U um $\sim$ 9,61 % und bei Workload I um $\sim$ 11,28 %.

In Hinblick auf die gemessenen Latenzwerte zeigt sich ein vergleichbares Bild (siehe Abbildung 36(b)). Dabei benötigt Workload R die geringste durchschnittliche Latenz mit $\sim$ 0,25 ms und Workload I mit $\sim$ 0,28 ms die höchste Latenz.

Insgesamt bleibt festzustellen, dass Read-Operationen bzgl. der getesteten Default-Konfiguration schneller sind als schreibende Operationen. Unter Berücksichtigung des 50%igen Anteils an Read-Operationen bei Workload M erklärt dies zudem den

insgesamt zweit höchsten Durchsatz von Workload M. Des Weiteren wird deutlich, dass Insert-Operationen geringfügig langsamer sind als Update-Operationen.

4.1.2. Couchbase Java Client SDK

Dieses Kapitel untersucht die Auswirkungen bei der Verwendung verschiedener Operationen mittels des Couchbase Java Client Software Development Kits (SDKs). In diesem Kontext ist interessant, dass der Couchbase Java Client seine grundlegenden Funktionalitäten von der Memcached Client Implementierung erbt und lediglich um Couchbase spezifische Funktionalitäten ergänzt. Insofern wird beispielsweise beim Hinzufügen eines neuen Datensatzes mittels "add(...)" ohne Angabe von Parametern bzgl. der Persistierung und/oder der Replikation direkt eine Memcached Client Methode aufgerufen. Wird dagegen einer der Parameter verwendet, so wird eine Couchbase Client Methode aufgerufen, welche in letzter Konsequenz jedoch auch auf die Memcached Client Methoden zurückgreift. Gemäß Denis Nelubin [Nel13] existierten zu mindestens bei der von Thumbtack 2013 verwendeten SDK Version 1.1.0[61] deutliche Leistungsunterschiede bei der Verwendung dieser unterschiedlich überladenen Schreiboperationen.

Als Resultat einer eigens durchgeführten Codeanalyse der SDK Versionen 1.1.0 und der im Rahmen dieser Arbeit verwendeten Version 1.4.5[62] ist davon auszugehen, dass im Gegensatz zur Version 1.1.0 in der Version 1.4.5 maximal geringfügige Laufzeitunterschiede feststellbar seien sollten. Zur Validierung der getroffenen Annahme wurden die drei relevanten Workloads I, U und M mit 4 unterschiedlichen Parametrisierungen getestet. In der ersten Parametrisierung erfolgt keine Angabe bzgl. der Replikation und Persistierung, sodass direkt die Memcached Implementierung verwendet wird. Die zweite und dritte Parametrisierungsvariante enthält jeweils ausschließlich die Angabe bzgl. der Persistierung oder der Replikation. Die vierte und letzte Variante übergibt wiederum beide Parameter. Durch die Verwendung der entsprechenden *Zero* Parameter wird erreicht, dass alle vier Methodenaufrufe von der Bedeutung her identisch agieren sollten.

Anhand der nachfolgenden Abbildungen 37 und 38 lässt sich deutlich erkennen, dass die unterschiedlichen Methoden hinsichtlich des Durchsatzes und der Latenz zu fast identisch Ergebnissen kommen. Als Konsequenz der gemessenen Differenzen

[61] `https://github.com/couchbase/couchbase-java-client/blob/1.1.0/src/main/java/com/couchbase/client/CouchbaseClient.java`, zuletzt besucht am 26.12.2014.

[62] `https://github.com/couchbase/couchbase-java-client/blob/1.4.5/src/main/java/com/couchbase/client/CouchbaseClient.java`, zuletzt besucht am 26.12.2014.

von weniger als ein Prozent wird deren Leistungsverhalten als übereinstimmend gewertet.

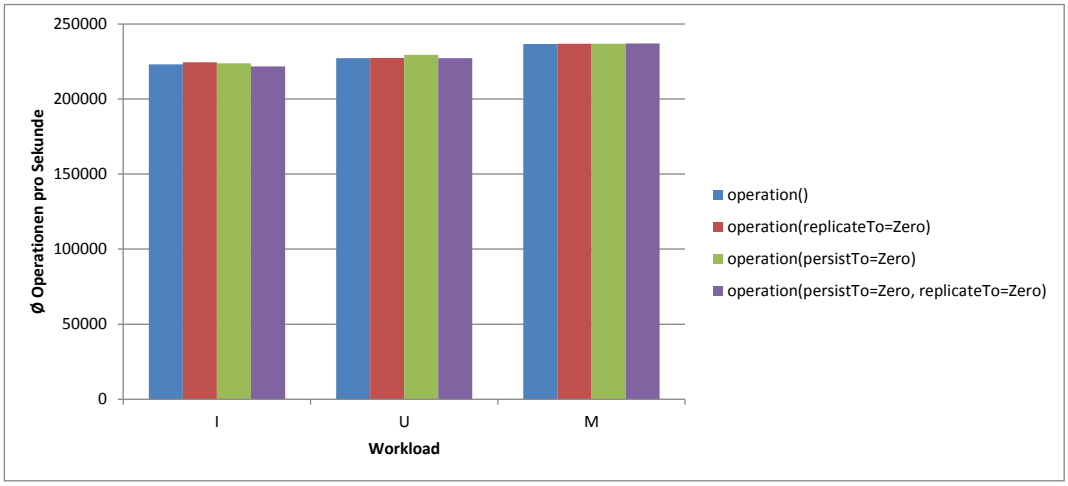


Abbildung 37: Couchbase: Java Client SDK 1.4.5 - Durchsatz

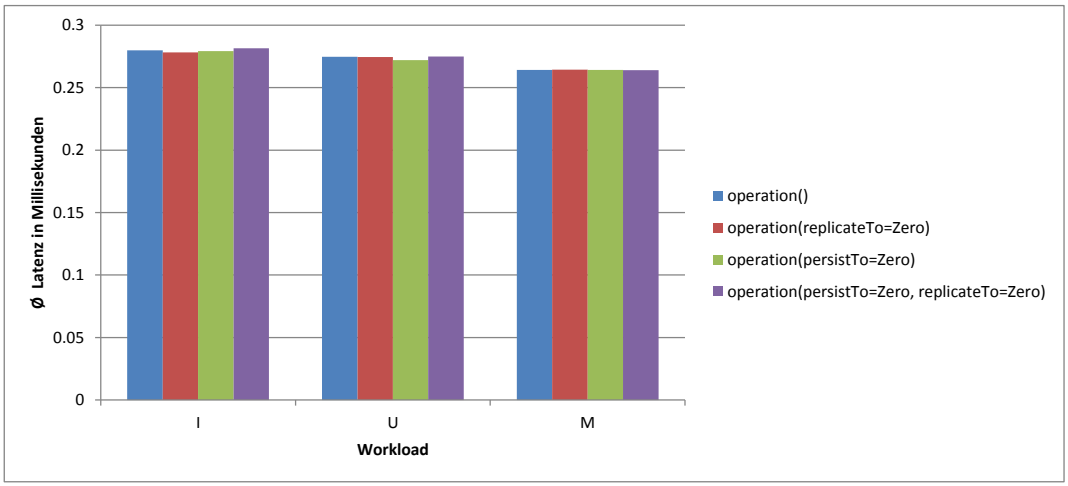


Abbildung 38: Couchbase: Java Client SDK 1.4.5 - Latenz

In der Folge lässt sich somit festhalten, dass eine differenzierte Verwendung der Methoden, wie es bspw. innerhalb Thumbtacks Couchbase Datenbankadapter Implementierung erfolgt, zu mindestens unter Verwendung der SDK Version 1.4.5 nicht erforderlich ist.

4.1.3. Unterschiedliche Dokumentengrößen

Dieses Kapitel untersucht das Verhalten von Couchbase in Bezug auf die Verwendung unterschiedlicher Dokumentengrößen. Ausgehend von der standardmäßigen Größe werden fünf weitere Messungen durchgeführt, bei denen sich die Dokumentengröße jeweils verzehnfacht. Bedingt durch den JSON spezifischen Overhead (81 Byte) ist eine exakte Skalierung der Dokumentgrößen unter Verwendung der von YCSB gegebenen Konfigurationsmöglichkeiten nicht möglich. Die Anzahl der initial geladenen Dokumente wurde derart gewählt, dass der Speicherbedarf (inklusive Metadaten) nahezu konstant bleibt. Eine Übersicht der zugrundeliegenden Konfigurationen kann der Tabelle 9 entnommen werden.

Faktor	**1**	**10**	**100**	**1.000**	**10.000**	**100.000**
Feldgröße in Byte	10	173	1.802 ≈ 1,76 KB	18.092 ≈ 17,7 KB	180.992 ≈ 177 KB	1.809.992 ≈ 1,73 MB
Dokumentengröße in Byte	181	1.811	18.101 ≈ 17,7 KB	181.001 ≈ 176,8 KB	1.810.001 ≈ 1,73 MB	18.100.001 ≈ 17,3 MB
Anzahl Dokumente	60.000.000	8.523.100	890.212	89.420	8.944	896
Speicherbedarf in GB[63]	~ 23,1 GB	~ 23,1 GB	~ 23,1 GB	~ 23,1 GB	~ 23,1 GB	~ 23,1 GB
Abweichung	-	~ 0,0004 %	~ 0,0003 %	~ 0,0016 %	~ 0,0207 %	~ 0,1537 %

Tabelle 9: Couchbase: Übersicht evaluierter Dokumentengrößen

Anhand der gewonnen Messergebnisse (siehe Abbildung 39) lässt sich erkennen, dass die Auswirkungen aufgrund unterschiedlicher Dokumentengrößen bei allen vier Workloads in ähnlicher Weise erfolgen.

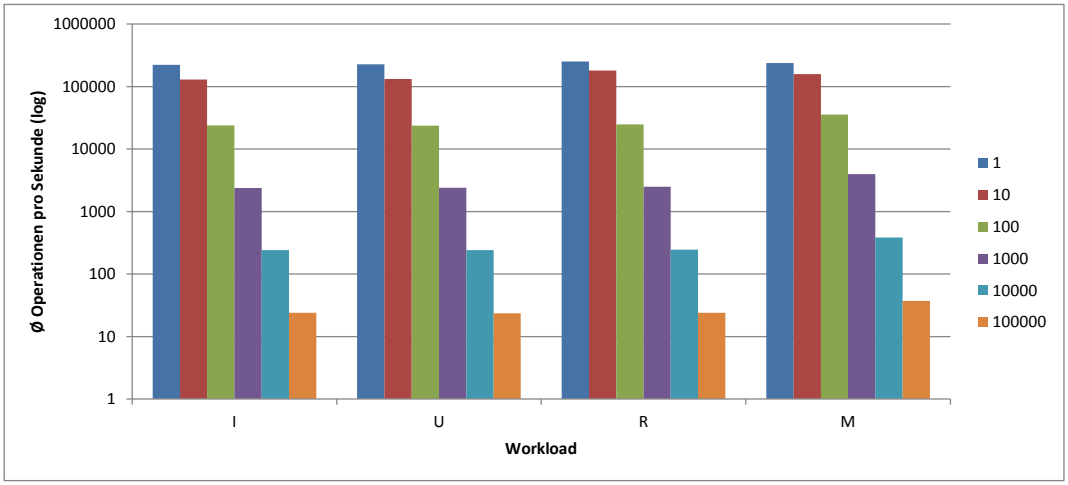


Abbildung 39: Couchbase: Unterschiedliche Dokumentengrößen - Durchsatz

[63]Die Berechnung des Speicherbedarfs erfolgt gemäß der in Kapitel 4.1.1.2 vorgestellten Formel.

Während die erste Verzehnfachung zu einer Reduktion des Durchsatzes um ~ 42 % bei den ausschließlich schreibenden Workloads führt, sinkt der Durchsatz beim Workload R um ~ 28 % und beim Workload M um ~ 33 %. Bei den Workloads I, U und R fällt der Durchsatz ab der Vergrößerung um den Faktor 100 auf ~ 10 % des ursprünglichen Durchsatzes (Faktor 1) und auf ~ 16 % bei Workload M. Ab der Vergrößerung um den Faktor 1000 sinkt der Durchsatz jedoch gleichermaßen bei allen Workloads nahezu umgekehrt proportional mit jeder weiteren Verzehnfachung der Dokumentengröße.

Im Zuge der Analyse gilt zudem zu beachten, dass Couchbase während der Ausführung von Workload I Dokumente ab der Vergrößerung um den Faktor 10 auslagert. Bei der Ausführung von Workload M werden dagegen Dokumente ab einer Vergrößerung um den Faktor 100 ausgelagert. Inwiefern sich diese Gegebenheit neben der variierenden Dokumentengröße ggf. zusätzlich auf das Messergebnis auswirkt, kann unter Betrachtung der hier vorgenommenen Messungen nicht beantwortet werden.

Zudem bleibt festzustellen, dass bei der Ausführung des Workloads I, ab einer Vergrößerung der Dokumente um den Faktor 100, nicht mehr alle Insert-Operationen erfolgreich durchgeführt werden konnten. Ein derartiges Verhalten konnte bisher noch bei keiner anderen Benchmarkkonfiguration beobachtet werden. Die nachfolgende Tabelle 10 gibt einen Überblick über den jeweiligen Anteil fehlgeschlagener Operationen bzgl. der insgesamt durchgeführten Inserts.

Faktor	**1**	**10**	**100**	**1.000**	**10.000**	**100.000**
Anteil fehlgeschlagener Operationen	0 %	0 %	$\sim 6{,}55$ %	$\sim 6{,}07$ %	$\sim 2{,}42$ %	$\sim 5{,}60$ %

Tabelle 10: Couchbase: Übersicht prozentualer Anteil fehlgeschlagener Insert-Operationen

Unter dem Aspekt, dass bei der zugrundeliegenden Konfiguration keine Persistierung oder Replikation der Dokumente explizit bestätigt wird, ist zu vermuten, dass das Fehlschlagen derart geringer Mengen zu keiner wesentlichen Beeinflussung der Messergebnisse führt. Maßgeblich für diese Annahme ist, dass die Übertragungsdauer für die einzufügenden Dokumente unabhängig von der letztendlichen Verarbeitung innerhalb von Couchbase sein wird.

4.1.4. Unterschiedliche initiale Dokumentenmengen

Dieses Kapitel untersucht, inwiefern die vier zugrundeliegenden Workloads durch verschieden große initiale Dokumentenmengen beeinflusst werden. Hierzu werden insgesamt fünf unterschiedliche Dokumentenmengen zwischen 10 Millionen und 120 Millionen evaluiert. Anhand der Tabelle 11 lässt sich erkennen, dass die Dokumentenmengen derart gewählt wurden, sodass entsprechend der in Kapitel 4.1.1.2 (Seite 60) verwendeten Formel keine Auslagerung von Dokumenten erforderlich ist.

	10 Mio.	**30 Mio.**	**60 Mio.**	**90 Mio.**	**120 Mio.**
Speicherbedarf ohne Workload I	~ 3,8 GB	~ 11,5 GB	~ 23,1 GB	~ 34,6 GB	~ 46,1 GB
Speicherbedarf mit Workload I	~ 56,9 GB	~ 64,6 GB	~ 76,1 GB	~ 87,6 GB	~ 99,1 GB
Benötigte Nodes ohne Workload I	~ 0,15	~ 0,45	~ 0,92	~ 1,38	~ 1,83
Benötigte Nodes mit Workload I	~ 2,27	~ 2,57	~ 3,03	~ 3,49	~ 3,95

Tabelle 11: Couchbase: Ressourcenbedarf bei unterschiedlich großen initialen Dokumentenmengen

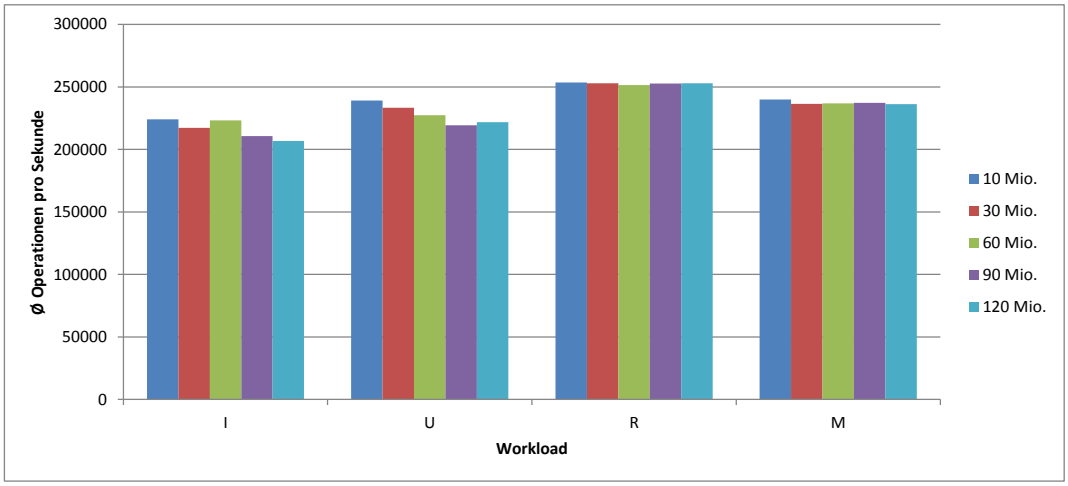


Abbildung 40: Couchbase: Unterschiedliche Dokumentenmengen - Durchsatz

Mit Betrachtung der gemessenen Durchsatzwerte (siehe Abbildung 40) fällt zuerst ein unregelmäßiges Verhalten bei den Workloads I und U auf. Hierbei bleibt nicht nachvollziehbar, warum der Durchsatz vereinzelt geringer ist, als bei kleineren und größeren Dokumentenmengen. Insgesamt ist bei beiden ausschließlich schreibenden Workloads jedoch ein tendenziell fallender Durchsatz bei zunehmender Größe

der initialen Dokumentenmenge zu beobachten. Beim direkten Vergleich der Ergebnisse für 10 Mio. und 120 Mio. Dokumente sinkt der Durchsatz um $\sim 7{,}74$ % beim Workload I und um $\sim 7{,}25$ % beim Workload U.

Der Durchsatz beim Workload R bleibt nahezu konstant und weicht im Ergebnis um weniger als ein Prozent zwischen den Konfigurationen voneinander ab. Ebenso nur geringfügig betroffen ist der Workload M, bei welchem der Durchsatz zwischen den Konfigurationen mit 10 Mio. und 120 Mio. Dokumenten um maximal $\sim 1{,}51$ % sinkt.

4.1.5. Auto-Compaction

Dieses Kapitel untersucht die Auswirkungen verschiedener Konfigurationen bzgl. der automatischen Aktivierung der Compaction Funktionalität. Standardmäßig muss die Couchbase-Datenbank eine Fragmentierung von 30 % erreichen, bevor der Prozess der Compaction zur Defragmentierung automatisch gestartet wird. Innerhalb von Couchbase kann der Schwellwert, der die Defragmentierung auslöst, in einem Wertebereich zwischen 2 und 100 Prozent konfiguriert werden. Neben der Default-Konfiguration werden die Auswirkungen durch die Schwellwerte von 60 %, 30 %, 15 %, 10 % und einer deaktivierten Auto-Compaction Funktionalität untersucht.

Bedingt durch die Tatsache, dass die zur Ausführung der Auto-Compaction benötigte Fragmentierung lediglich durch Insert und Update-Operationen generiert werden kann, entfallen sämtliche Evaluierungen auf Basis des Read-Only Workloads.

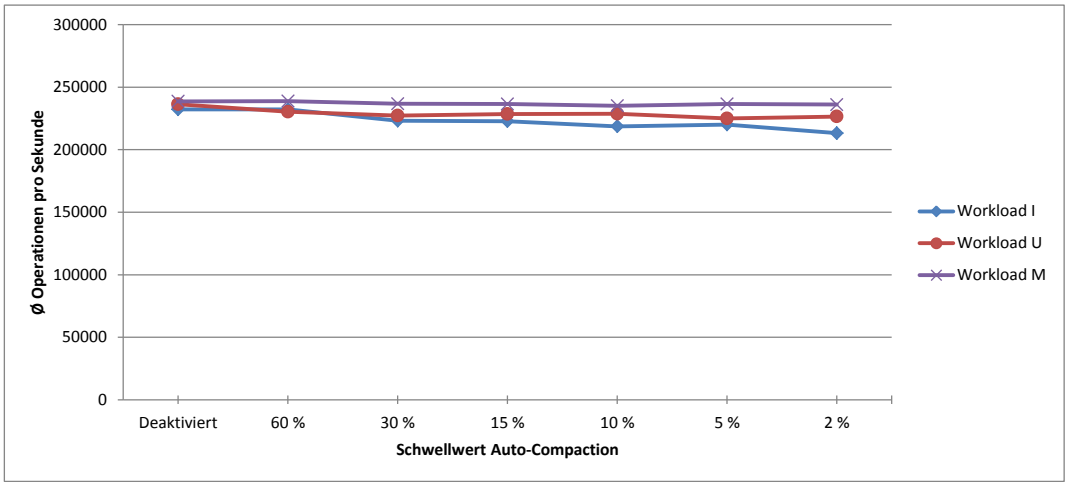


Abbildung 41: Couchbase: Compaction - Durchsatz

Wie sich anhand von Abbildung 41 erkennen lässt, sind die gemessenen Unterschiede zwischen den verschiedenen Konfigurationen insgesamt verhältnismäßig gering. Der größte Effekt kann in Bezug auf den Insert-Only Workload festgestellt werden. Im Vergleich zum standardmäßigen 30 % Schwellwert sinkt der durchschnittliche Durchsatz beim 2 % Schwellwert um $\sim$ 4,44 % und steigt dagegen bei deaktivierter Auto-Compaction um $\sim$ 4,06 %. Beim direkten Vergleich dieser maximal gegensätzlichen Konfigurationen ergibt sich ein maximaler Leistungsunterschied von $\sim$ 8,89 %.

Ein grundsätzlich ähnliches Verhalten ist bei Workload U zu beobachten, jedoch wirken sich insbesondere niedrigere Schwellwerte weniger stark aus. Entsprechend steigt der Durchsatz gegenüber der deaktivierten Auto-Compaction um $\sim$ 3,94 % und sinkt beim 2 % Schwellwert um $\sim$ 0,35 %.

Im Gegensatz dazu wirken sich die unterschiedlichen Konfigurationen weniger stark auf den gemischten Workload M aus. Während bei deaktiverter Auto-Compaction ein $\sim$ 0,79 % höherer Durchsatz erzielt wird als bei der Default-Konfiguration, sinkt der durchschnittliche Durchsatz dazu um $\sim$ 0,31 % beim 2 % Schwellwert. In Anbetracht der bei Workload I und U stärker zu beobachtenden Auswirkung könnte die Ursache hierfür auf den 50%igen Anteil an Read-Operationen zurückzuführen sein. Bedingt durch die vollständige Zwischenspeicherung der Dokumente im Cache und der folglich fehlenden Leseoperationen auf der Festplatte erfolgen somit insgesamt weniger Operationen, welche sich direkt auf die persistierten Dokumente auswirken.

Abschließend ist jedoch anzumerken, dass die verhältnismäßig geringe Laufzeit des Benchmarks mit 10 min (5 min Warmup und 5 min Messung), nicht zwingend eine Aussage über das Langzeitverhalten erlaubt. Die Untersuchung dieses Aspekts könnte somit Thema nachfolgender Arbeiten sein.

4.1.6. Couchbase Bucket vs. Memcached Bucket

Dieses Kapitel untersucht die Leistungsunterschiede zwischen den zwei, innerhalb von Couchbase konfigurierbaren, Bucket Varianten. Hierbei gilt zu beachten, dass es sich beim Memcached Bucket um ein reines In-Memory-System handelt, welches keine Dauerhaftigkeit in Form einer Persistierung oder Replikation unterstützt. Dem entgegen stellt der standardmäßige Couchbase Bucket jene Funktionalitäten zusätzlich bereit.

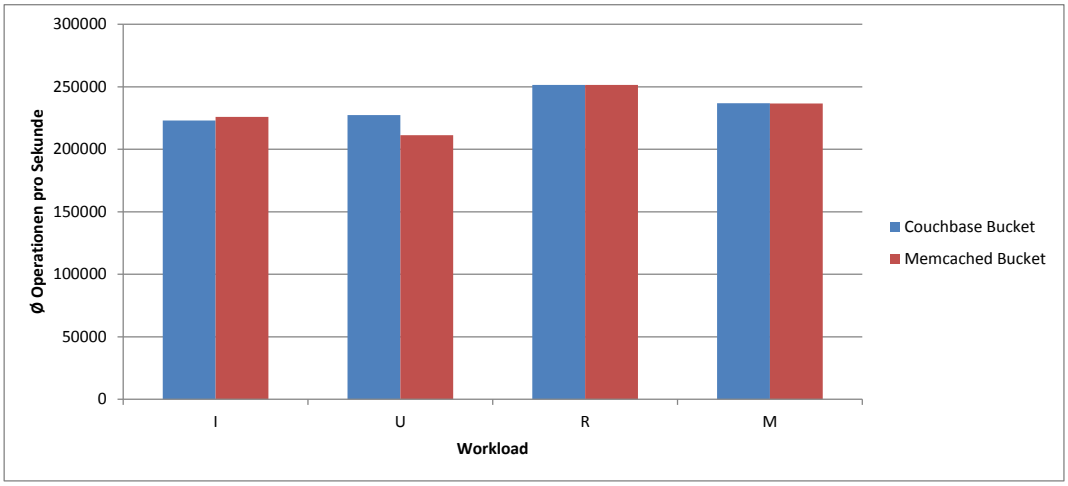


Abbildung 42: Couchbase: Couchbase Bucket vs. Memcached Bucket - Durchsatz

Anhand der Abbildung 42 lässt sich erkennen, dass beide Bucket Varianten zu relativ ähnlichen Ergebnissen gelangen. Mit einer Abweichung von weniger als 0,04 % erreichen beide Bucket Varianten bei den zwei Workloads R und M einen fast identischen Durchsatz. Gegenüber der Verwendung des standardmäßigen Couchbase Buckets steigert sich der Durchsatz beim Memcached Bucket in Hinblick auf Workload I nur geringfügig. Entsprechend steigt der durchschnittliche Durchsatz von $\sim$ 223.178 auf $\sim$ 225.895 Operationen pro Sekunde, was einer Steigerung von $\sim$ 1,22 Prozent entspricht. Ein deutlicher Unterschied besteht dagegen bei den erzielten Durchsatzwerten von Workload U. Hier erreicht der Couchbase Bucket einen Durchsatz von durchschnittlich $\sim$ 227.363 Operationen pro Sekunde, wogegen der Memcached Bucket einen Durchsatz von nur $\sim$ 211.378 Operationen pro Sekunde erreicht. Somit erzielt der Memcached Bucket einen $\sim$ 7 % geringeren Durchsatz. Bedingt durch die nahezu identischen Ergebnisse der beiden Bucket Typen bei Workload I, R und M wurden die Messungen mehrfach wiederholt, um einen Fehler bzgl. der bei Workload U festgestellten Ergebnisse auszuschließen. Als Resümee dieser Wiederholungen bleibt festzustellen, dass die gemessenen Resultate reproduzierbar sind.

4.1.7. Cache Eviction

Couchbase besitzt zwei unterschiedliche Strategien zum Entfernen von Daten aus dem internen Cache. Im Wesentlichen unterscheiden sich diese bzgl. des Umfangs an Daten, welche aus dem Cache entfernt werden. Während bei der Value Eviction lediglich der Wert bzw. das Dokument aus dem Cache entfernt wird, wird demgegenüber bei der Full Eviction der komplette Datensatz inklusive seiner Metadaten aus dem Cache entfernt. [Cou14, S. 104]

Bezüglich der Evaluierung dieser beiden Varianten existieren einige Probleme hinsichtlich auftretender Leistungseinbrüche und Serverabstürze. Um eine Beeinflussung der Messungen untereinander noch weiter als bisher ausschließen zu können, wurde das Benchmarking-Vorgehen für diese Untersuchung geändert. So wurde die Datenbank zusätzlich vor jeder Workload-Messung zurückgesetzt, anschließend gestoppt, ggf. genutztes Swapping zurückgesetzt, die Datenbank erneut gestartet und abschließend wieder mit neuen Dokumenten initialisiert. Rückwirkend betrachtet konnte durch diese Änderung jedoch kein wesentlich anderes Ergebnis erzielt werden.

Zur Evaluierung der unterschiedlichen Auswirkungen auf die Leistungsfähigkeit wurden die Eviction Varianten hinsichtlich verschieden großer initialer Dokumentenmengen getestet (siehe Tabelle 12). Als Ausgangspunkt wurde eine Messung auf Basis der 0,25 fachen Menge an max. empfohlenen[64] Dokumenten für das zugrundeliegende Cluster durchgeführt. Die hierbei verwendete geringe Menge an Dokumenten stellt sicher, dass die Ausführung der vier Workloads keine Auslagerung von Daten aus dem Cache erfordert. Sämtliche weiteren Dokumentenmengen wurden derart gewählt, dass bereits das initiale Befüllen der Datenbank ein Auslagern erfordert.

Max. empfohlene Dokumentenmenge	**Faktor 0,25**	**Faktor 1,25**	**Faktor 1,5**	**Faktor 1,75**
Anzahl Dokumente	65.310.000	326.550.000	391.880.000	457.170.000
Speicherbedarf	~ 25,11 GB	~ 125,53 GB	~ 150,64 GB	~ 175,74 GB

Tabelle 12: Couchbase: Ressourcenbedarf bei unterschiedlich großen initialen Dokumentenmengen (Cache Eviction)

Hinsichtlich der ausgewählten Konfigurationen ist anzumerken, dass größere initiale Dokumentenmengen nicht getestet werden konnten. Insbesondere bei der Value Eviction kam es ab der 1,75 fachen Dokumentenmenge bereits im Rahmen der initialen Befüllung zu vereinzelten Serverabstürzen[65], woraufhin deren Evaluierung abgebrochen wurde. Auffällig in diesem Zusammenhang ist, dass die entsprechenden Server am intensivsten vom Swapping Gebrauch gemacht haben (siehe Anhang ab Seite 133, Abbildung 81, 82, 83). Zudem bleibt anzumerken, dass der Absturz nicht immer während des Befüllens der Datenbank aufgetreten ist, sondern auch

[64] Gemäß der Couchbase Formel aus Kapitel 4.1.1.2 (Seite 60).

[65] Couchbase Datenbank war nicht mehr erreichbar. Zudem konnte keine Secure Shell (SSH) Verbindung mehr zum Node aufgebaut werde. Infolgedessen wurde jeweils ein Hard Reset durchgeführt.

teils bei der anschließend[66] explizit gestarteten Compaction. Eine konkrete Fehlerursache kann jedoch anhand der erhobenen Statistiken nicht identifiziert werden und bleibt somit als Untersuchungsaspekt zukünftigen Untersuchungen vorbehalten.

Value Eviction

Die nachfolgende Abbildung 43 zeigt die Durchsatzergebnisse für die drei zugrunde gelegten initialen Dokumentenmengen.

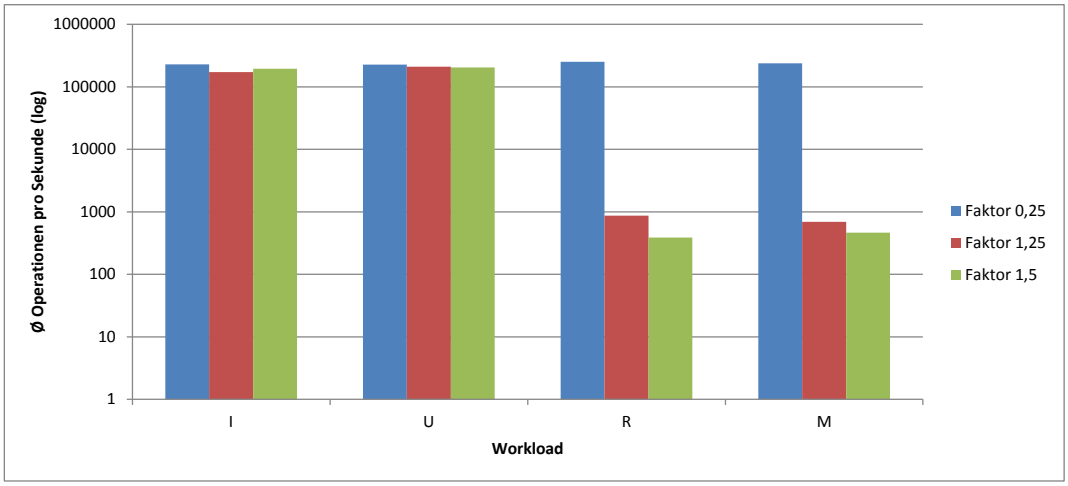


Abbildung 43: Couchbase: Value Eviction bei unterschiedlichen initial geladenen Dokumentenmengen - Durchsatz

Deutlich wird hierbei, dass insbesondere das Lesen von Datensätzen durch die Auslagerung betroffen ist. Gegenüber dem erzielten Durchsatz, welcher bei einer 0,25 fachen Dokumentenmenge ($\sim$ 252.911 Op/s) erreicht wurde, sinkt der Durchsatz bei der 1,25 fachen Dokumentenmenge um $\sim$ 99,7 % auf $\sim$ 867 Operationen pro Sekunde und bei der 1,5 fachen Dokumentenmenge um $\sim$ 99,8 % auf $\sim$ 388 Operationen pro Sekunde.

Bezüglich der schreibenden Operationen wirkt sich die zunehmende initiale Dokumentenmenge am stärksten beim Einfügen neuer Dokumente aus. So sinkt der Durchsatz von $\sim$ 229.664 Operationen pro Sekunde bei der 1,25 fachen Dokumentenmenge um $\sim$ 24,7 % (auf $\sim$ 172.941 Op/s) und bei der 1,5 fachen Dokumentenmenge um $\sim$ 15,3 % (auf $\sim$ 194.593 Op/s). Auffällig in diesem Zusammenhang ist, dass die Messungen für die 1,25 fache Dokumentenmenge einen niedrigeren Durchsatz ergeben, als die für die 1,5 fache Dokumentenmenge. Aufgrund dieser

[66] Sämtliche Dokumente wurden von den YCSB Clients zu diesem Zeitpunkt bereits übertragen und die Disk Write Queue von Couchbase ebenfalls abgearbeitet.

unerwarteten Gegebenheit wurden die Messungen mehrfach wiederholt. Schlussendlich bestätigen diese jedoch alle das dargestellte Ergebnis.

Der geringste Effekt ist hinsichtlich des Workloads U zu erkennen. Gegenüber der 0,25 fachen Dokumentenmenge ($\sim$ 226.108 Op/s) sinkt der Durchsatz bei der 1,25 fachen Dokumentenmenge um $\sim$ 6,9 % (auf $\sim$ 210.614 Op/s) und bei der 1,5 fachen Dokumentenmenge um $\sim$ 9,2 % (auf $\sim$ 205.362 Op/s).

Infolge des jeweiligen Leistungsabfalls bei den drei Operationsarten reinen Workloads ist der größte Durchsatzeinbruch beim gemischten Workload M zu beobachten. Hierbei sinkt der Durchsatz von $\sim$ 237.420 Operationen pro Sekunde bei der 1,25 fachen Dokumentenmenge um $\sim$ 99,66 % (auf $\sim$ 696 Op/s) und bei der 1,5 fachen Dokumentenmenge um $\sim$ 99,85 % (auf $\sim$ 466 Op/s).

Full Eviction

Aufgrund eines festgestellten Leistungseinbruches kann der Full Eviction Modus effektiv nur auf Basis der 0,25 fachen Dokumentenmenge evaluiert werden. Während das initiale Befüllen der Datenbank für die 1,25 fachen Dokumentenmenge beim Value Eviction Modus durchschnittlich 25 Minuten dauert, wurden beim Full Eviction Modus auch nach über 130 Minuten erst 71 % der Dokumente eingefügt. Daraufhin wurde der Vorgang abgebrochen. Die Abbildung 44 zeigt wie sich der Durchsatz über die Zeit bis zum Abbruch entwickelt.

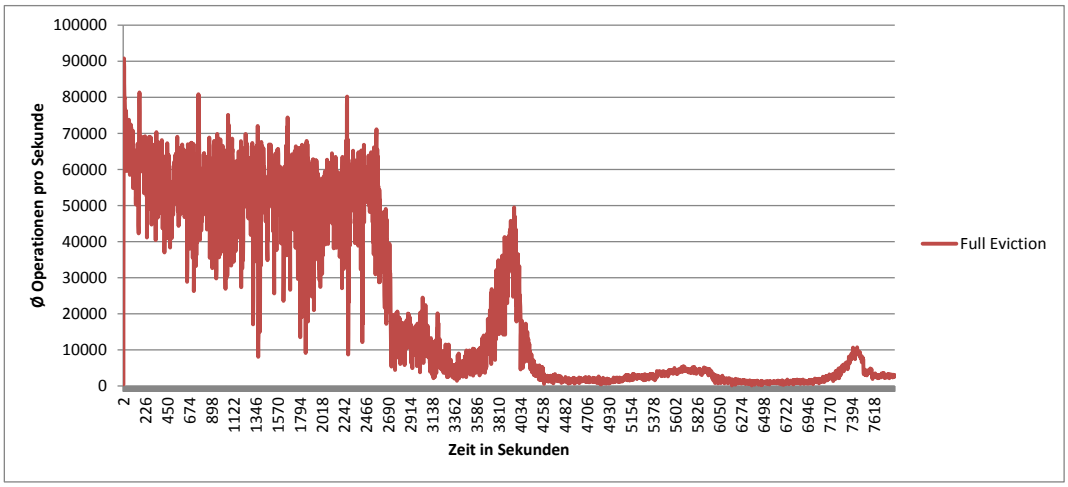


Abbildung 44: Couchbase: Durchsatzeinbruch beim Versuch, die 1,25 fache Menge an empfohlenen Dokumenten initial einzufügen

Im Austausch mit einem Couchbase Entwickler [Nit14] wurde die gegebenen Si-

tuation als abnormal eingestuft. Als Folge dessen wurde zudem ein Fehler-Ticket[67] für die zukünftige Behebung erstellt. Unter diesen Umständen und in Anbetracht der für diese Arbeit begrenzten Zeit wird auf eine Evaluation des Full Eviction Modus in Konfigurationen abseits der 0,25 fachen Dokumentenmenge verzichtet. Demnach erfolgt bzgl. des Full Eviction Modus keine Untersuchung von Szenarien, in welchen eine Auslagerung von Inhalten und Metadaten erforderlich ist.

Beim direkten Vergleich der beiden Eviction Varianten (siehe Abbildung 45) wird ersichtlich, dass unter Verwendung der 0,25 fachen Dokumentenmenge kein wesentlicher Unterschied bzgl. der Workloads U und R festgestellt werden kann.

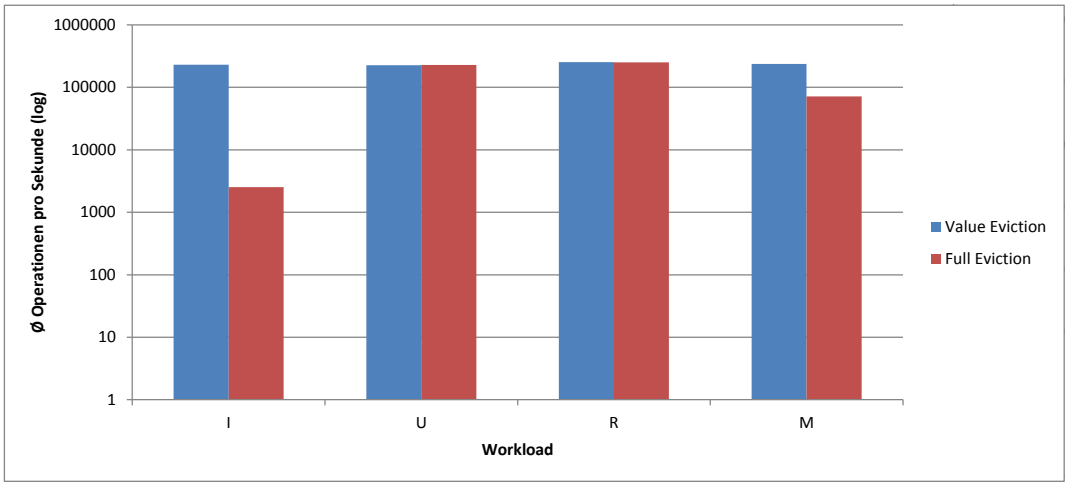


Abbildung 45: Couchbase: Eviction Strategien im Vergleich unter Verwendung der 1,25 fachen Menge an empfohlenen Dokumenten - Durchsatz

Einen deutlichen Leistungseinbruch verzeichnet dagegen der Full Eviction Modus beim Workload I. Gegenüber dem Value Eviction Modus fällt dieser von $\sim$ 229.664 Operationen pro Sekunde auf 2.524 Operationen pro Sekunde und somit um mehr als 98,9 %.

4.1.8. Asynchrone Replikation

Dieses Kapitel untersucht, inwiefern sich das Aktivieren der Couchbase Cluster internen Replikation auf dessen Leistungsfähigkeit auswirkt. Diesbezüglich erlaubt Couchbase separate Replikate auf bis zu drei weiteren Nodes zu erzeugen. Hierbei gilt zu beachten, dass die Replikation standardmäßig asynchron zur eigentlichen

[67] `http://www.couchbase.com/issues/browse/MB-12752`, zuletzt besucht am 14.01.2015. Es gilt zu beachten, dass dieses Ticket bereits während der Benchmark-Vorbereitung erstellt wurde. Folglich entsprechen dessen Beschreibungen nicht der letztendlich verwendeten Default-Konfiguration.

Client-Operation erfolgt. Ein weiterer Aspekt im Kontext dieser Untersuchung ist der durch die Replikation erhöhte Speicherbedarf und der dadurch ggf. erforderlichen Auslagerung von Dokumenten. Entsprechend erfordert bereits das initiale Befüllen der Datenbank bei der dreifachen Replikation eine Auslagerung von Dokumenten. Des Weiteren wurden während der Workload-Durchführung bei Workload I ab der einfachen Replikation und bei Workload M mit der dreifachen Replikation Dokumente ausgelagert.

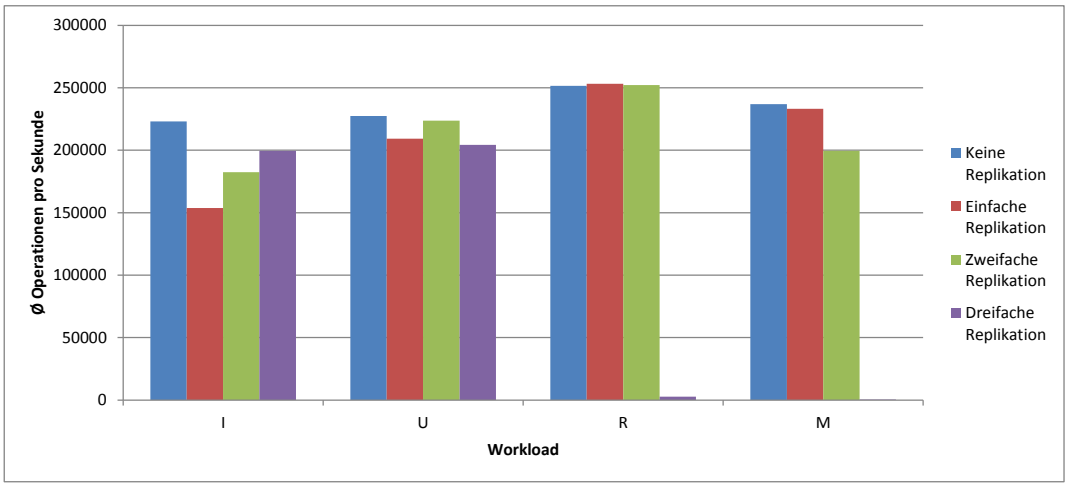
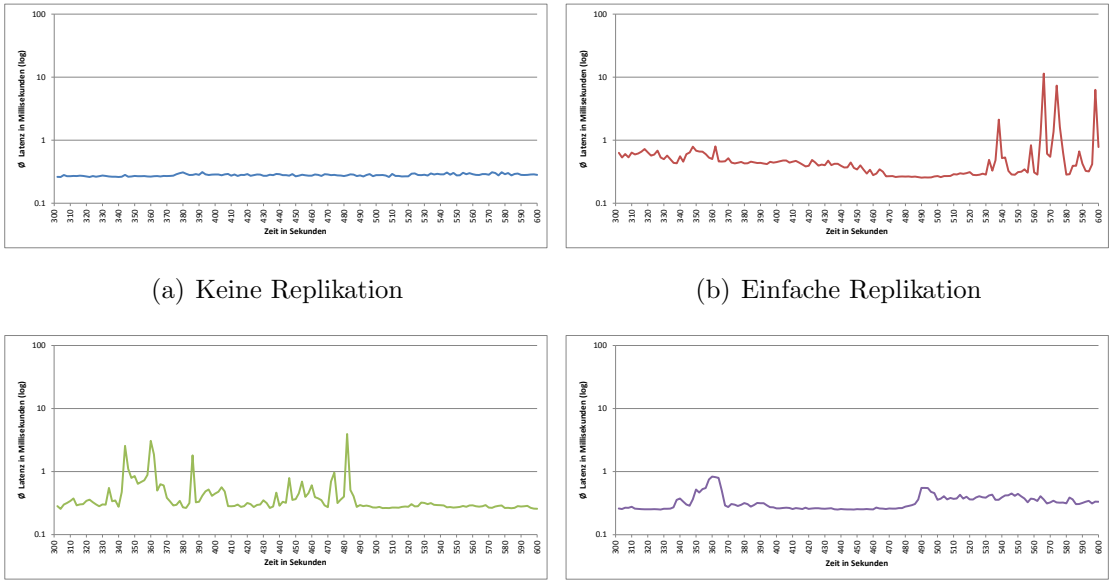


Abbildung 46: Couchbase: Asynchrone Replikation - Durchsatz

Mit Blick auf Abbildung 46 wird unmittelbar deutlich, dass bzgl. der Workloads I und U unerwartete Messergebnisse erzielt wurden. Insbesondere bei Workload I fällt auf, dass der Durchsatz gegenüber der Konfiguration mit einer deaktivierten Replikation umso geringer sinkt, je mehr Replikate generiert werden. Gegenüber der deaktivierten Replikation sinkt der durchschnittliche Durchsatz bei der einfachen Replikation um $\sim 31\,\%$, bei der zweifachen Replikation um $\sim 18\,\%$ und bei der dreifachen Replikation um nur noch $\sim 11\,\%$. Als Ursache für dieses Verhalten kann eine stark schwankende Latenz während der Benchmark-Ausführung[68] festgehalten werden (siehe Abbildung 47). Hierbei fällt auf, dass einerseits die Latenz bei der deaktivierten Replikation vergleichsweise konstant bleibt, während andererseits bei der einfachen Replikation die stärksten Ausschläge und bei der dreifachen Replikation die geringsten Ausschläge zu beobachten sind.

[68] Diskussion des Laufzeitverhaltens erfolgt auf Basis jener der drei durchgeführten Wiederholungen, die gegenüber deren Durchschnitt die geringste Abweichung aufweist.



(a) Keine Replikation

(b) Einfache Replikation

(c) Zweifache Replikation

(d) Dreifache Replikation

Abbildung 47: Couchbase: Asynchrone Replikation bei Insert-Operationen - Latenz zur Laufzeit

Eine direkte Gegenüberstellung des Latenzverhaltens kann dem Anhang auf Seite 135 (Abbildung 85) entnommen werden.

Wie eingangs bereits erwähnt, wurde bzgl. Workload U ein ebenso unerwartetes Ergebnis erzielt. Gegenüber der deaktivierten Replikation sinkt der durchschnittliche Durchsatz bei der einfachen Replikation um $\sim 8\,\%$, bei der zweifachen Replikation um lediglich $\sim 1{,}6\,\%$ und bei der dreifachen Replikation wieder um $\sim 10\,\%$. Mit Betrachtung der durchschnittlichen Latenz über die Dauer der Benchmark-Ausführung (siehe Abbildung 48) hinweg, wird deutlich, dass sämtliche Konfigurationen über den größten Zeitraum der Messung eine vergleichsweise identische Latenz vorweisen. Lediglich die Konfigurationen mit einfacher und dreifacher Replikation weisen zur Mitte der Messungen einen extremen Ausschlag hinsichtlich der Latenz auf.

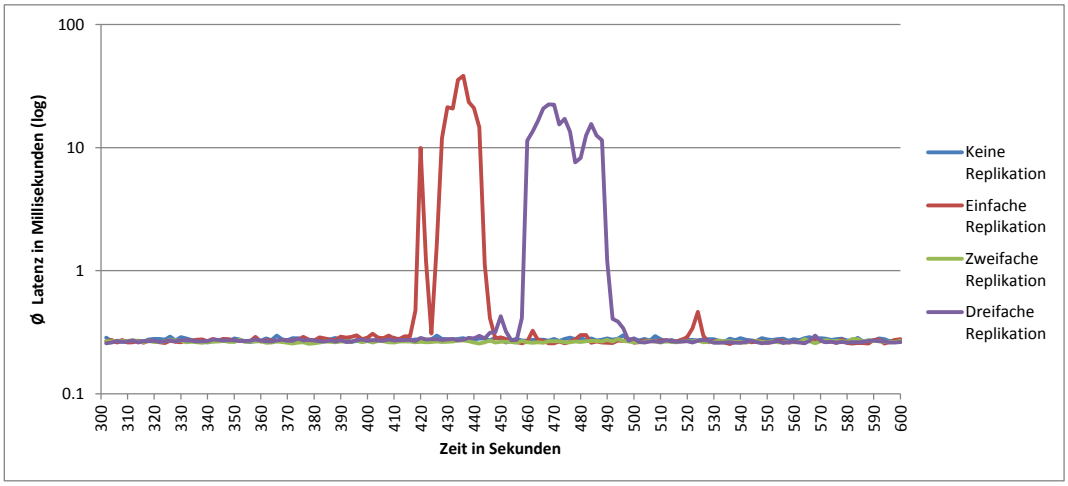


Abbildung 48: Couchbase: Asynchrone Replikation bei Update-Operationen - Durchsatz

Es bleibt anzumerken, dass anhand der durchgeführten Messungen keine Aussage darüber getroffen werden kann, inwiefern die beobachteten Ausschläge ggf. auch bei den anderen Konfigurationen bei längerer Laufzeit auftreten würden. Genauso wenig kann eine Aussage darüber getroffen in welcher Frequenz die beobachteten Ausschläge auftreten. Insgesamt bleibt zudem offen, ob es sich bei dem beobachteten Verhalten um ein fehlerhaftes oder um ein reguläres Verhalten handelt. Eine tiefer gehende Analyse dieses Sachverhalts obliegt somit zukünftigen, ggf. auf dieser Arbeit aufbauenden Untersuchungen. Abschließend bleibt bzgl. der Messergebnisse für Workload U jedoch festzuhalten, dass die Grundlatenz bei allen vier Konfigurationen verhältnismäßig einheitlich ist.

Anhand der für Workload R gewonnen Messergebnisse wird deutlich, dass sich die Replikation im Normalfall nicht wesentlich auf die zum Lesen erforderliche Latenz oder den folglich erzielbaren Durchsatz auswirkt. Entsprechend beträgt die Abweichung des Durchsatzes bei den Konfigurationen mit zweifacher oder geringerer Replikation weniger als 0,7 %. Der massive Leistungseinbruch ($\sim 99\%$), welcher in Verbindung mit der dreifachen Replikation einhergeht, ist auf die durch den erhöhten Speicherbedarf bedingte Auslagerung von Dokumenten zurückzuführen. Infolgedessen können viele der Datensätze nicht mehr aus dem Cache gelesen werden, sodass für diese ein lesender Zugriff auf den Sekundärspeicher erforderlich ist.

Beim Workload M sinkt der Durchsatz gegenüber der deaktivierten Replikation, bei der einfachen Replikation um $\sim 1{,}5$ %, bei der zweifachen Replikation um $\sim 15{,}7$ % und bei der dreifachen Replikation um mehr als 99,8 %. Der massive

Leistungseinbruch bei der dreifachen Replikation ist, wie bei Workload R, auf das Lesen bereits ausgelagerter Dokumente vom Sekundärspeicher zurückzuführen.

4.1.9. Bestätigung schreibender Operationen

Im Rahmen dieses Kapitels wird untersucht inwiefern sich die unterschiedlichen Bestätigungsmechanismen bei schreibenden Operationen auswirken. Couchbase erlaubt in diesem Zusammenhang auf Operationsebene eine differenzierte Anforderung in Bezug auf Persistierung und Replikation. Je nach erfolgter Konfiguration wartet der Client somit bis die Persistierung und/oder Replikation auf einer festgelegten Menge an Server Nodes erfolgt ist. Beide Varianten werden im Folgenden separat voneinander untersucht. Deren Kombination bleibt dagegen unbetrachtet.

4.1.9.1. Bestätigung nach Replikation

Bei der Ausführung schreibender Operationen erlaubt es Couchbase Client-seitig auf die Rückmeldung erfolgter Replikationen von bis zu drei weiteren Nodes abzuwarten. Die im Folgenden präsentierten Ergebnisse beziehen sich auf Konfigurationen in denen identisch viele Replikationsbestätigungen angefordert werden, wie insgesamt Replikate erzeugt werden.

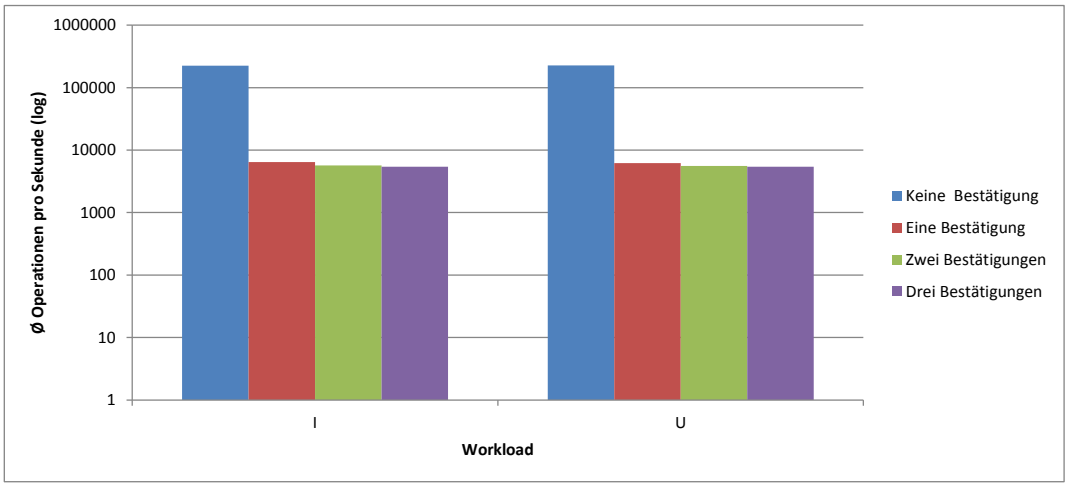


Abbildung 49: Couchbase: Bestätigung nach Replikation - Durchsatz

Die im Rahmen der Messungen gewonnenen Ergebnisse (siehe Abbildung 49) zeigen einen einheitlichen Effekt bzgl. der vier getesteten Konfigurationen auf die Workloads I und U. Besonders prägnant ist der Leistungseinbruch, welcher mit dem Wechsel von keiner angeforderten Replikationsbestätigung zu einer einzigen Bestätigung einhergeht. Hierbei sinkt der Durchsatz bei Workload I von $\sim$ 223 Tausend auf $\sim$ 6.448 Operationen pro Sekunde und gleichermaßen beim Workload U

von ∼ 227 Tausend auf ∼ 6.174 Operationen pro Sekunde. Folglich fällt der Durchsatz bei beiden Workloads um mehr als 97 Prozent.

Die nachfolgende Tabelle 13 stellt die Auswirkungen durch das Abwarten weiterer Bestätigungsmeldungen im direkten Vergleich dar. Dabei wird angegeben inwieweit der Durchsatz durch das Anfordern einer jeden weiteren Bestätigung prozentual sinkt.

	Workload I	**Workload U**
Keine vs. eine Bestätigung	∼ 97,11 %	∼ 97,28 %
Eine vs. zwei Bestätigungen	∼ 11,78 %	∼ 9,28 %
Zwei vs. drei Bestätigungen	∼ 5,01 %	∼ 2,93 %

Tabelle 13: Couchbase: Prozentual sinkender Durchsatz durch die Bestätigung erfolgter Replikationen

Mit Betrachtung der in Tabelle 13 dokumentierten Messergebnisse lässt sich feststellen, dass der Durchsatz bei der Anforderung von mehr als einer Bestätigung, sowohl für Workload I als auch für Workload U vergleichsweise geringfügig sinkt. Zudem verringert sich mit jeder weiteren angeforderten Bestätigung deren jeweilige zusätzliche Auswirkung.

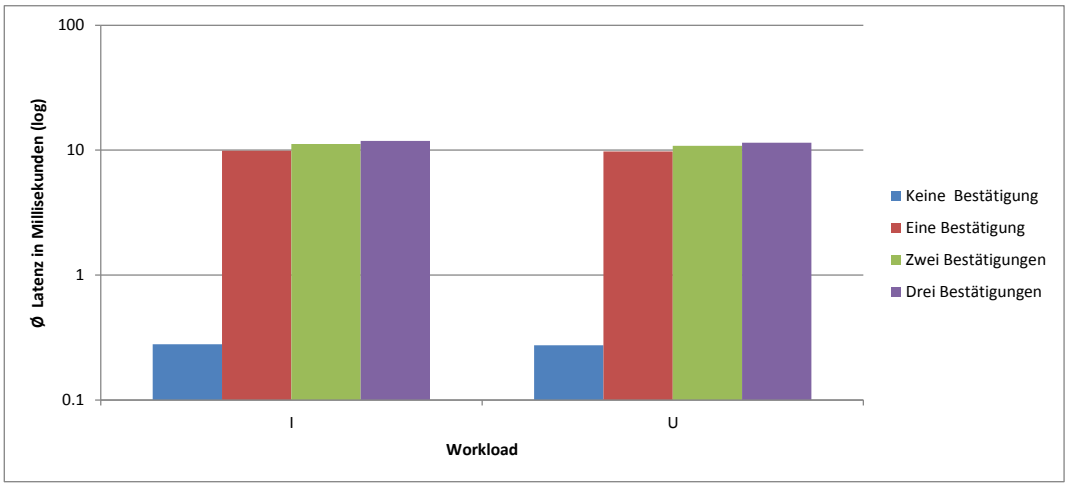


Abbildung 50: Couchbase: Bestätigung nach Replikation - Latenz

Bezüglich der Latenz zeigt sich ein entsprechend der vorangegangen Beschreibung vergleichbares Verhalten (siehe Abbildung 50). Insofern steigt die Latenz durch das Anfordern einer einzigen Replikationsbestätigung von ∼ 0,28 ms auf über 9,91 ms beim Workload I und beim Workload U von ∼ 0,27 ms auf über 9,77 ms. Die höchste Latenz wird beim Abwarten von drei Bestätigungen erreicht und beträgt beim Workload I ∼ 11,84 ms und ∼ 11,45 ms beim Workload U.

4.1.9.2. Bestätigung nach Persistierung

Couchbase erlaubt es bei der Ausführung schreibender Operationen auf die Persistierungsrückmeldung von bis zu vier unterschiedlichen Nodes zu warten. Sofern lediglich eine einzige Bestätigung benötigt wird, erfolgt diese sobald der für das Dokument zuständige Server Node den Datensatz persistiert hat. Werden dagegen von mehr als einem Server Node Bestätigungen erfordert, ist eine entsprechende Konfiguration der Replikation erforderlich. Im Rahmen der durchgeführten Messungen wurden für Benchmark-Konfigurationen mit x angeforderten Persistierungsbestätigungen $x - 1$ Replikate im Couchbase Cluster konfiguriert.

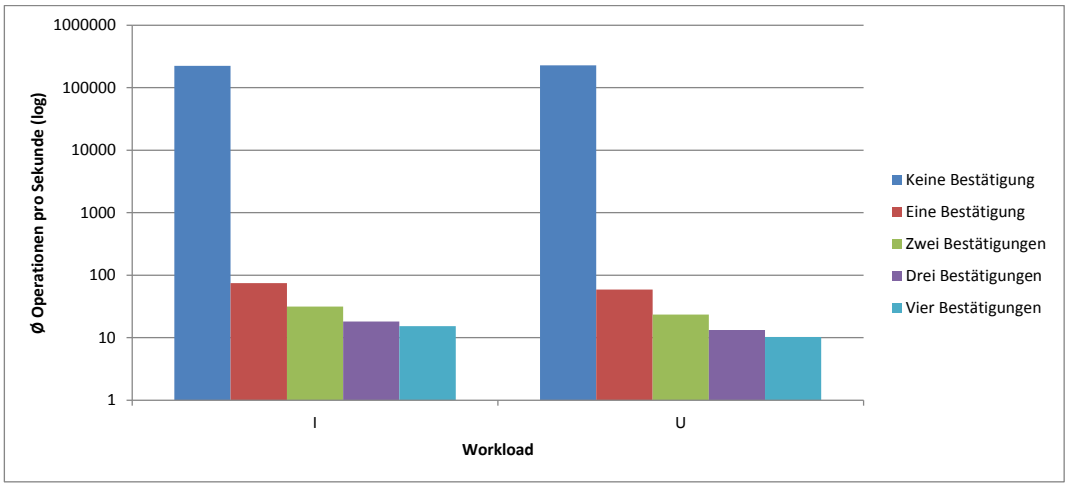


Abbildung 51: Couchbase: Bestätigung nach Persistierung - Durchsatz

Wie sich anhand der Abbildung 51 eindeutig erkennen lässt, sinkt der durchschnittliche Durchsatz mit zunehmender Anzahl abzuwartender Bestätigungen. Besonders prägnant erweist sich dabei der Unterschied zwischen keiner angeforderten Bestätigung und der ersten Persistenzbestätigung. Hierbei sinkt der Durchsatz bei Workload I von ~ 223 Tausend auf ~ 74 Operationen pro Sekunde und in ähnlicher Weise beim Workload U von ~ 227 Tausend auf ~ 59 Operationen pro Sekunde. Somit sinkt der Durchsatz bei beiden Workloads um mehr als 99,9 Prozent. Demgegenüber sinkt der Durchsatz mit jeder weiteren angeforderten Bestätigung vergleichsweise moderat.

Zur weiteren Analyse der jeweiligen Unterschiede gibt nachfolgende Tabelle 14 einen Überblick über den Verlauf der sinkenden Durchsatzleistung bei zunehmender Anzahl angeforderter Bestätigungen. Dargestellt wird hierbei die prozentuale Differenz zwischen der angegebenen Konfiguration und jener Konfiguration, welche eine Bestätigung weniger erfordert.

	Workload I	**Workload U**
Keine vs. eine Bestätigung	∼ 99,967 %	∼ 99,974 %
Eine vs. zwei Bestätigungen	∼ 57,32 %	∼ 60,36 %
Zwei vs. drei Bestätigungen	∼ 42,47 %	∼ 43,18 %
Drei vs. vier Bestätigungen	∼ 15,77 %	∼ 23,09 %

Tabelle 14: Couchbase: Prozentual sinkender Durchsatz durch die Bestätigung erfolgter Persistierungen

Mit Betrachtung der Tabelle 14 wird deutlich, dass die Erhöhung um eine weitere erforderte Bestätigung bei zunehmender Gesamtzahl zu einer vergleichsweise immer geringeren Reduzierung des Durchsatzes führt. Ein identisches Verhalten konnte bereits bei der Analyse der Auswirkungen durch das Abwarten von Replikationsbestätigungen in Kapitel 4.1.9.1 beobachtet werden. Bedingt durch die stets stärker gemessenen Effekte bei Workload U, lässt sich zudem schlussfolgern, dass die zunehmende Verwendung von Persistenzbestätigungen bei Update-Operationen zu einer geringfügig stärkeren Beeinflussung führt als bei Insert-Operationen.

Bezüglich der Latenz zeigt sich ein entsprechend der vorangegangen Beschreibung vergleichbares Verhalten (siehe Abbildung 52). Insofern steigt die Latenz durch das Anfordern einer einzigen Persistierungsbestätigung von ∼ 0,28 ms auf über 855 ms beim Workload I und beim Workload U von ∼ 0,27 ms auf über 1.072 ms. Die höchste Latenz wird beim Abwarten von drei Bestätigungen erreicht und beträgt beim Workload I ∼ 4.114 ms und ∼ 5.991 ms beim Workload U.

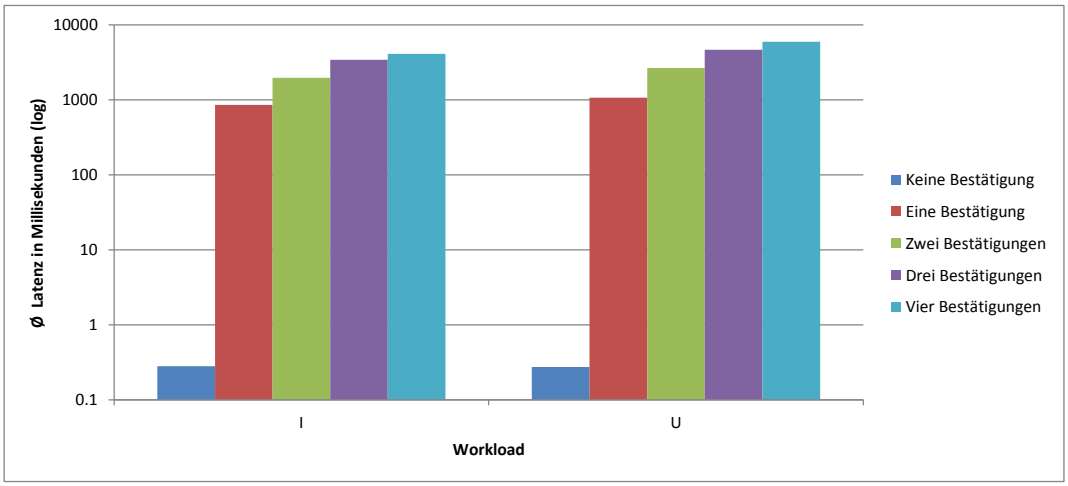


Abbildung 52: Couchbase: Bestätigung nach Persistierung - Latenz

4.1.10. Zusammenfassung der Messergebnisse

Von den drei Grundoperationen ist die Verarbeitung lesender Anfragen bei Couchbase am schnellsten. Darauf folgt die Verarbeitung von Updates und am Schluss die geringfügig langsameren Inserts.

Hinsichtlich der schreibenden Operationen wirkt sich die Anforderung von Replikations- und/oder Persistierungsbestätigungen am stärksten auf das Leistungsvermögen aus. Entsprechend fällt der Durchsatz bei den Workloads I und U um mehr als 97 %, sobald eine einzelne Replikationsbestätigung abgewartet wird. Jede zusätzliche Bestätigung führt zu einer weiteren Durchsatzminderung von jeweils ~ 3 % bis ~ 12 %. Mit einer Durchsatzeinbuße von über 99,9 % wirkt sich das Abwarten einer einzelnen Persistenzbestätigung nochmals stärker auf die Workloads I und U aus. Jede weitere Bestätigung führt hierbei zu einer zusätzliche Verminderung des Durchsatzes von jeweils ~ 16 % bis ~ 60 %.

Die größte Leistungsbeeinträchtigung bei lesenden Operationen tritt auf, sobald ein Teil der relevanten Daten nicht mehr im Cache gehalten werden kann. Infolge des somit erforderlichen Lesens vom Sekundärspeicher sinkt der Durchsatz um mehr als 99 %. Bei der Evaluierung der Eviction Varianten konnte zudem festgestellt werden, dass sich die Auslagerung bei der Value Eviction der Durchsatz bei Workload I um bis zu ca. 25 % und bei Workload U bis zu annähernd 9,2 % reduziert. Aufgrund eines Leistungseinbruches konnte die Full Eviction Variante nicht in einem Szenario getestet werden, bei welchem ein Teil der initialen Dokumentenmenge bereits ausgelagert ist. Dagegen zeigt der Test mit einer In-Memory Dokumentenmenge, dass beim Full Eviction Modus der Durchsatz von Insert-Operationen um mehr als 98,9 % geringerer ist, als im Value Eviction Modus.

Im Zusammenhang mit der asynchronen Replikation konnte festgestellt werden, dass hierbei vor allem schreibende Operationen negativ beeinträchtigt werden. Lesende Operationen werden dagegen nur beeinträchtigt, wenn durch die zusätzlichen Kopien Dokumente aus dem Cache ausgelagert werden. Hinsichtlich der Auswirkungen auf schreibende Operationen sind unerwartete Resultate aufgetreten. Beispielsweise erzielt eine einfache Replikation einen niedrigeren Durchsatz als eine dreifache Replikation. Als Ursache hierfür konnten stark schwankende Anfragelatenzen identifiziert werden.

Bei der Evaluierung verschiedener Dokumentengrößen wurde zudem festgestellt,

dass ab einer Dokumentengröße von $\sim 17{,}7$ KB jede Verzehnfachung der Dokumentengröße zu einer nahezu umgekehrt proportionalen Verminderung des Durchsatzes führt. Bei kleineren Dokumenten sinkt der Durchsatz in Folge einer Verzehnfachung der Dokumentengröße dagegen nur in geringerem Ausmaß.

Weitere Untersuchungen bzgl. verschieden großer initialer Dokumentenmengen, der Auto-Compaction und der beiden Bucket-Typen ergaben, dass deren unterschiedlichen Konfigurationen lediglich eine geringfügige Leistungsbeeinflussung (weniger als 10 %) bewirken. Bezüglich der Verwendung unterschiedlich überladener Operationen aus dem Couchbase Java Client SDK konnte zudem keine abweichende Performanz festgestellt werden.

4.2. Apache Cassandra

Im Rahmen dieses Kapitels wird beschrieben, inwiefern sich verschiedene Konfigurationen auf die Leistungsfähigkeit von Apache Cassandra 2.1.2 auswirken. Zunächst wird hierfür eine Default-Konfiguration festgelegt, welche den anschließenden Messungen zugrunde liegt. Daraufhin wird Cassandra hinsichtlich der folgenden acht Aspekte untersucht:

- Existenzvalidierung bei Insert-Operationen
- Prepared Statements
- Unterschiedliche Datensatzgrößen
- Unterschiedliche initiale Dokumentenmengen
- Compaction
- Komprimierung
- Write-Ahead Logging
- Replikation

Abschließend erfolgt eine kurze Zusammenfassung der wichtigsten Ergebnisse.

4.2.1. Festlegung einer geeigneten Default-Konfiguration

Bevor mit der Messung verschiedener Benchmark-Konfigurationen begonnen werden kann, ist es erforderlich eine Default-Konfiguration festzulegen. Diese Maßnahme ist notwendig, damit eine weitestgehende Vergleichbarkeit zwischen den für Cassandra beschriebenen Messungen gegeben ist. Sofern nicht Änderungen an den hier definierten Parametern im Speziellen evaluiert werden sollen, wird jede der im gesamten Kapitel 4.2 beschriebenen Messungen mit den hier festgelegten Parametern durchgeführt. Ausnahmen hiervon werden im Rahmen der Ergebnispräsentation dokumentiert.

Wie bereits in Kapitel 3.3 beschrieben, wird für die Default-Konfiguration standardmäßig eine Menge von neun Nodes verwendet. Diese teilt sich in vier YCSB Client Nodes, vier Cassandra Server Nodes und einen Cassandra Monitoring Node auf. Für die Persistierung von Cassandra selbst und der dazugehörigen Daten werden insgesamt drei Festplatten verwendet. Dabei wurde der Apachae Cassandra

Server 2.1.2 auf der gleichen Festplatte installiert wie das Betriebssystem. Cassandra selbst speichert seine (Nutzer-)Daten und Commit Logs jeweils getrennt voneinander auf eigenständigen Festplatten, welche entsprechend exklusiv verwendet werden.

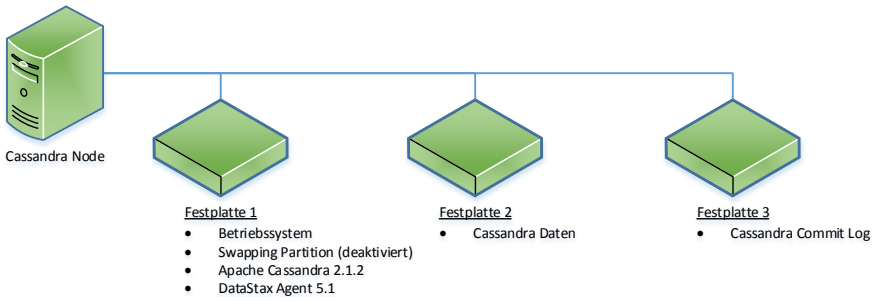


Abbildung 53: Cassandra: Physische Datenverteilung

Bedingt durch die Limitierung des Hauptspeichers auf 8 GB führt das Cassandra eigene Startskript den (Java) Cassandra-Instanz-Prozess mit einer maximalen Heap Größe von 2 GB aus. Der restliche Hauptspeicher verbleibt maßgeblich für die durch Cassandra außerhalb des Heaps gehaltenen Daten (bspw. den Bloom Filter [Dat15b, S. 116]) und den Linux-eigenen Page Cache. Gemäß der in den internen Konfigurationsdateien enthaltenen Empfehlungen [Apa15] wurde zum einen die Anzahl der gleichzeitig schreibenden Threads auf 64 erhöht. Zum anderen wurde der *endpoint_snitch* Parameter auf *GossipingPropertyFileSnitch* geändert und entspricht somit der für den Produktivbetrieb empfohlenen Konfiguration. Zur Beschleunigung der Restore Prozedur wurde zudem die auto_snapshot Funktionalität deaktiviert. Somit entfallen der andernfalls unnötig durchgeführte Flush und die anschließende Snapshot-Erstellung. Entsprechend der Empfehlung [Pop14] erfolgt die Anfragedurchführung unter der Verwendung von Prepared Statements. Abschließend bleibt anzumerken, dass innerhalb der Default-Konfiguration keine Replikation erfolgt, deren Auswirkung jedoch in Kapitel 4.2.9 explizit betrachtet wird.

4.2.1.1. Festlegung einer geeigneten initialen Dokumentenmenge

Zu Beginn ist es erforderlich eine Datenmenge zu definieren, welche standardmäßig in das Cassandra Cluster geladen wird und somit einer jeden Benchmark-Durchführung zugrunde liegt. Im Gegensatz zu Couchbase liegt der typische Einsatzbereich von Cassandra nicht in einem In-Memory Szenario, sondern in einem Anwendungsfall welcher die Persistierung von mehr Daten vorsieht, als im Hauptspeicher vorgehalten werden können. Deutlich wird dies unter anderem an der maximal

empfohlenen Datenmenge pro Node von drei bis fünf Terabyte [Dat15b, S. 25]. Auf der einen Seite ist es somit erforderlich die Datenmenge entsprechend groß zu wählen, sodass ein typisches Betriebsszenario vorherrscht. Andererseits sollte die Menge an Datensätzen möglichst gering sein, um den zeitlichen Aufwand für das Einfügen der Datensätze und der im Anschluss folgenden Compaction zu minimieren, wodurch erst die Durchführung einer Vielzahl von Messungen möglich wird. An dieser Stelle ist nochmals die in Kapitel 3.3 beschriebene Begrenzung des verfügbaren Hauptspeichers von 32 GB auf 8 GB hervorzuheben. Als Folge dieser realisierten Maßnahme ist eine bedeutend geringere Menge an Daten erforderlich, um ein In-Memory Betriebsszenario zu verlassen. Um ein unbeabsichtigtes In-Memory Szenario auszuschließen, wird eine initiale Datenmenge mit einem zweifachen Volumen des verfügbaren Hauptspeichers gewählt. Da hierbei lediglich die reinen Inhaltsdaten (Key- und Feldinhalte) zugrunde gelegt werden und effektiv noch Cassandra eigene Metadaten dazukommen, ist selbst unter Verwendung der internen Kompression ein Vorhalten sämtlicher Datensätze im Hauptspeicher nahezu ausgeschlossen. Zur Sicherstellung dieses Aspekts werden vor und nach jeder[69] Workload-Messung die internen Statistiken für den relevanten Keyspace abgerufen[70].

Zur Generierung der erforderlichen 16 GB Inhaltsdaten wird die Anzahl der Felder bei den standardmäßigen zehn belassen und die Feldgröße auf 60 Byte erhöht. Unter Berücksichtigung der durchschnittlichen Keygröße von 22,88 Byte[71] ergibt sich folglich eine Anzahl von 110.330.000 Datensätzen. Es gilt zu beachten, dass die Auswirkungen unterschiedlicher Datensatzgrößen in Kapitel 4.2.4 und die Auswirkungen unterschiedlich großer initialer Datensatzmengen in Kapitel 4.2.5 separat untersucht werden.

4.2.1.2. Festlegung einer geeigneten Thread-Anzahl pro Client Node

Neben der Definition einer initialen Dokumentenmenge ist es erforderlich, die Menge an parallel ausgeführten Threads pro YCSB-Client Node festzulegen. Dies ist notwendig, damit jeder Client Node in der Default-Konfiguration so viel Last wie möglich erzeugt, ohne dass Messergebnisse unbeabsichtigt negativ beeinträchtigt werden. Mit der hier ermittelten Anzahl an Threads pro Node werden alle folgen-

[69] Mit Ausnahme der im nachfolgenden Kapitel 4.2.1.2 beschriebenen Messungen. Hier erfolgte die Erhebung nur partiell. Aufgrund der bei sämtlichen Messungen identischen Anzahl von Datensätzen ist jedoch davon auszugehen, dass die erhoben Statistiken repräsentativ sind.

[70] Die Abfrage der Statistiken erfolgt auf jedem Server Node mittels "nodetool cfstats hwekeyspace".

[71] 4 Byte Konstante + 18,88 Byte FNV Hash

den Cassandra Benchmark-Konfigurationen getestet.

Zu diesem Zweck wurden elf unterschiedliche Messungen mit Mengen von einem bis 1024 Threads pro Client durchgeführt. Wie sich anhand der Abbildung 54 erkennen lässt, steigt zunächst sowohl bei Workload I als auch bei U der Durchsatz mit jeder Verdoppelung der eingesetzten Thread-Menge. Bzgl. des Workloads U fällt jedoch auf, dass der erreichbare Durchsatz ab einer Menge von 256 Threads pro Client stagniert und bei $\sim$ 192 Tausend Operationen pro Sekunde verharrt. Der Durchsatz bei Workload I steigt dagegen kontinuierlich auf bis zu $\sim$ 85.677 Operationen pro Sekunde. Hierbei ist jedoch anzumerken, dass ab einer Menge von 128 Threads während der Durchführung *WriteTimeoutExceptions*[72] aufgetreten sind. Mit einem Anteil von weniger als 0,005 Prozent der durchgeführten Operationen stellt deren Vorkommnis jedoch keine relevante Größe dar.

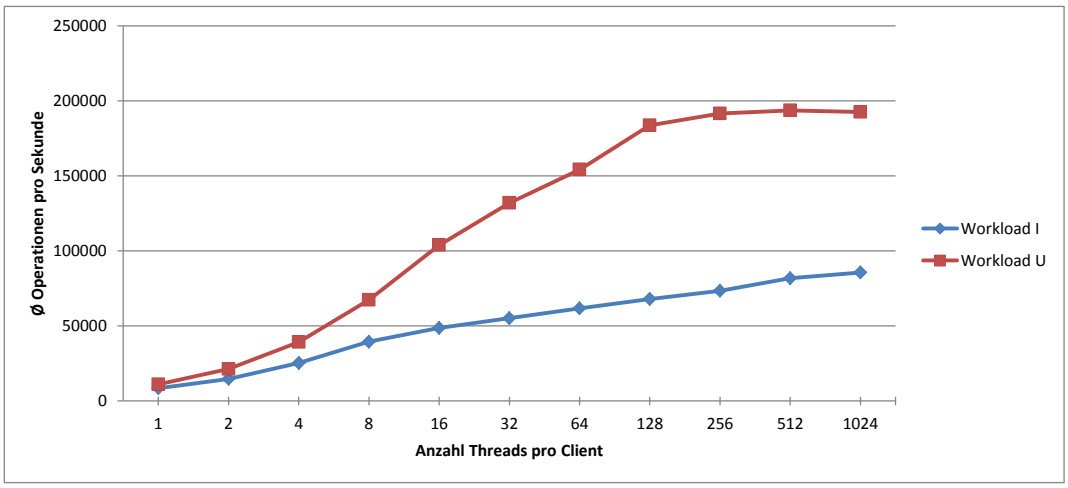


Abbildung 54: Cassandra: Threads pro Client - Durchsatz (Workload I & U)

Anhand der Abbildung 55 wird deutlich, dass der erzielbare durchschnittliche Durchsatz durch das Verdoppeln der Thread-Mengen früher bei den Workloads R und M stagniert, als noch beim zuvor betrachteten Workload U. Entsprechend ist erkennbar, dass der Durchsatz bei Workload R bereits ab einer Menge von 16 Threads pro Client in einem Bereich zwischen 640 und 750 Operationen pro Sekunden verharrt. Vergleichbar hierzu steigt der Durchsatz bei Workload M mit 8 Threads pro Client zunächst auf bis zu $\sim$ 1.400 Operationen pro Sekunde an. Die nachfolgenden Verdoppelungen der verwendeten Thread-Mengen bewirken dagegen zunächst keine weiteren Erhöhungen des erzielten Durchsatzes. Erst unter der Verwendung von 1024 Threads pro Client steigt der Durchsatz weiter auf $\sim$ 1.600

[72] Die *WriteTimeoutException* bezieht sich auf die Kommunikation zwischen den Cassandra Nodes und nicht auf die Kommunikation zwischen Client und Node.

Operationen pro Sekunde. Bedingt durch die ausbleibende Durchsatzsteigerung bei geringeren Thread-Mengen wurde die Messung mit 1024 Threads wiederholt, um eine Fehlmessung auszuschließen. Deren Resultat bestätigt jedoch das hier dargestellte Ergebnis.

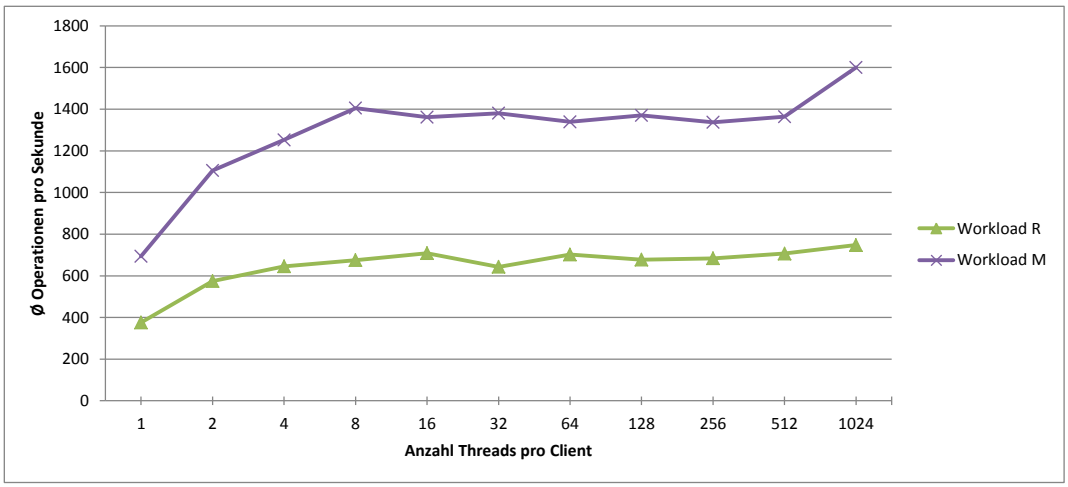


Abbildung 55: Cassandra: Threads pro Client - Durchsatz (Workload R & M)

Neben der Betrachtung des erzielbaren Durchsatzes ist es gleichermaßen erforderlich, die jeweiligen Latenzwerte bei den unterschiedlichen Thread-Mengen zu berücksichtigen. Diesbezüglich wird ersichtlich, dass die Latenzwerte mit jeder Verdoppelung der Thread-Menge kontinuierlich bei sämtlichen vier Workloads steigen (siehe Abbildung 56 und 57).

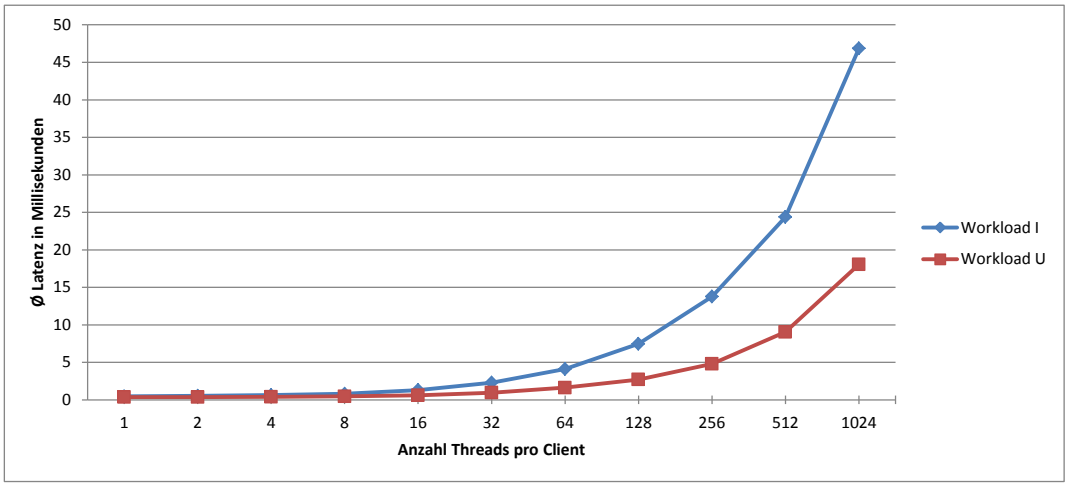


Abbildung 56: Cassandra: Threads pro Client - Latenz (Workload I & U)

Der geringste Latenzanstieg ist bzgl. des Workloads U festzustellen. Bei diesem steigt die durchschnittliche Latenz von $\sim 0{,}36$ ms auf $\sim 18{,}06$ ms (Faktor 50).

Demgegenüber nimmt die durchschnittliche Latenz beim Workload R am stärksten zu. Hier steigt die Latenz von $\sim$ 10,6 ms auf über $\sim$ 5.200 ms (Faktor 491).

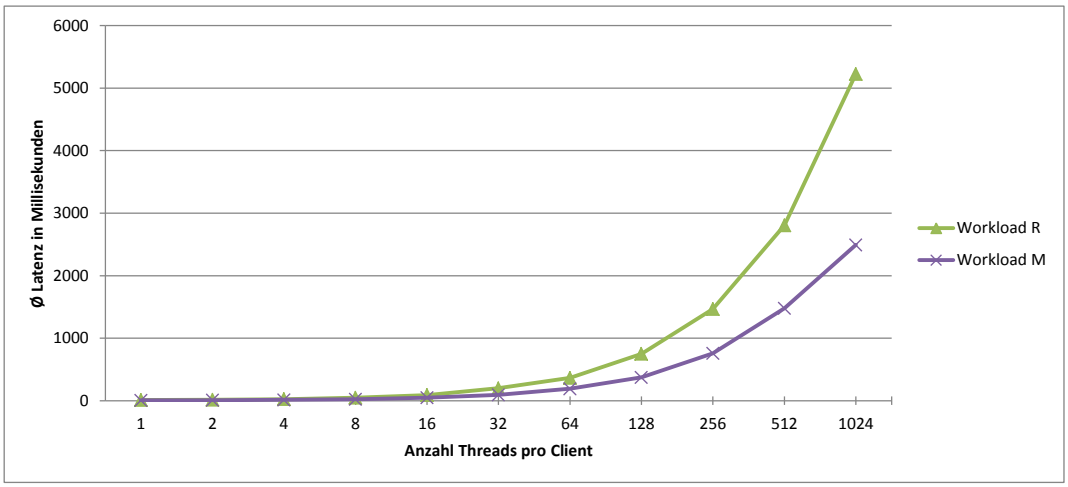


Abbildung 57: Cassandra: Threads pro Client - Latenz (Workload R & M)

Bei gleichzeitiger Betrachtung der gemessenen Durchsatz- und Latenzwerte wird deutlich, dass es nicht möglich ist, eine einzelne Thread-Menge zu identifizieren, welche sich für sämtliche Workloads gleichermaßen eignet. Wird die Thread-Menge sehr groß gewählt, unterstützt diese auf der einen Seite die Auslastung der Workloads I und U. Auf der anderen Seite erhöht sich jedoch die Latenz bei den Workloads R und M, ohne dass eine weitere Durchsatzsteigerung erzielt wird. Eine sehr kleine Thread-Menge vermeidet dagegen unnötig hohe Latenzwerte bei den Workloads R und M, unterbindet jedoch gleichzeitig eine optimale Auslastung der Workloads I und U. In Anbetracht dieser Problematik wäre eine Workload-spezifische Festlegung der zu verwendenden Thread-Menge einerseits optimal für die bestmögliche Auslastung des jeweiligen Workloads. Anderseits würde eine derart unterschiedliche Messgrundlage die Vergleichbarkeit zwischen den Workloads aufheben und somit eine Workload-übergreifende Betrachtung verhindern.

Als Kompromiss wird die größte Thread-Menge gewählt, welche noch für mindestens einen der Workloads eine höhere prozentuale Steigerung des Durchsatzes gegenüber der Latenz bewirkt. Mit Tabelle 15 sind sämtliche prozentualen Steigerungen infolge der Verdoppelung der eingesetzten Thread-Mengen dargestellt. Die grün hinterlegten Zellen kennzeichnen Messungen, in denen die Verdoppelung der Thread-Menge eine höhere Steigerung des Durchsatzes als der Latenz zufolge hat. Die rot hinterlegten Zellen dagegen kennzeichnen Messungen, in denen die Latenz stärker ansteigt. Somit wird deutlich, dass mit der Verdoppelung von 8 auf

16 Threads das letzte Mal der Durchsatz höher bei einem der Workloads steigt als die dabei erforderliche Latenz.

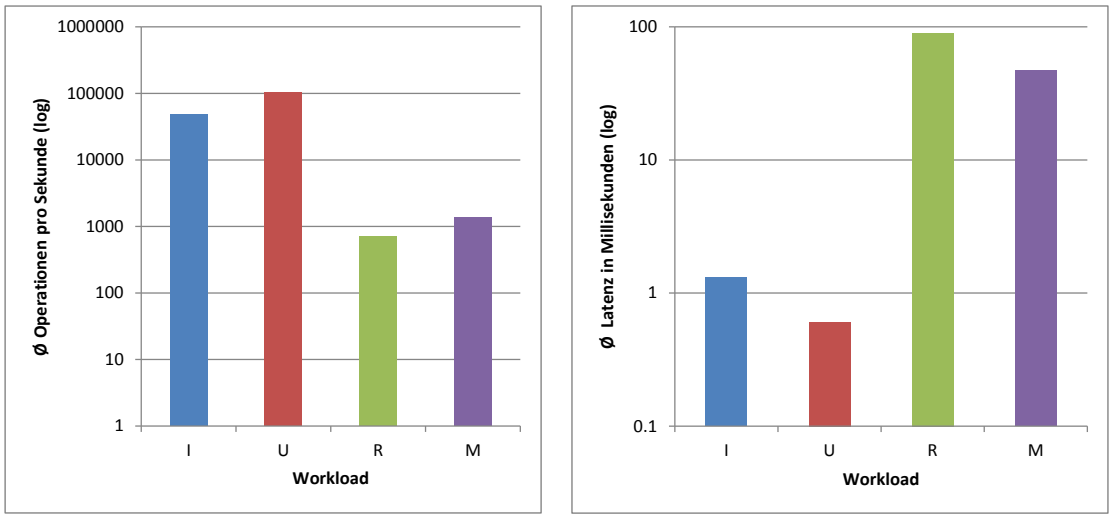
Threads pro Client	**Workload I**		**Workload U**		**Workload R**		**Workload M**	
	Durchsatz	**Latenz**	**Durchsatz**	**Latenz**	**Durchsatz**	**Latenz**	**Durchsatz**	**Latenz**
1 → 2	74,4	14,6	93,7	2,9	53,0	30,8	59,4	25,6
2 → 4	72,0	16,5	84,1	8,6	12,3	78,2	13,3	76,7
4 → 8	56,0	28,7	71,5	16,8	4,6	91,2	12,2	78,4
8 → 16	23,2	62,8	54,3	29,6	5,0	90,4	-3,1	106,3
16 → 32	13,5	76,7	27,0	58,2	-9,2	120,4	1,4	97,0
32 → 64	11,7	78,8	16,9	70,1	9,1	82,8	-3,0	106,1
64 → 128	10,2	81,4	19,1	66,3	-3,5	106,3	2,3	94,9
128 → 256	7,9	84,4	4,3	77,5	0,9	95,8	-2,4	104,1
256 → 512	11,5	77,2	1,1	88,2	3,3	91,1	2,1	94,9
512 → 1024	4,7	92,1	-0,6	99,7	5,8	86,4	17,3	68,5

Tabelle 15: Cassandra: Durch Verdoppelung verwendeter Thread-Mengen erzielte Durchsatz- und Latenzsteigerungen in Prozent

In der Konsequenz wird eine Menge von 16 Threads pro Client als geeignete Default-Konfiguration betrachtet. Es gilt jedoch zu beachten, dass es sich hierbei nicht zwangsläufig um die insgesamt bestmögliche Menge an Threads handelt. Mit zusätzlichen Benchmarks könnte untersucht werden, ob eine Thread-Menge zwischen 16 und 32 noch höhere Durchsätze ermöglicht, ohne dass die Latenz stärker ansteigt als der Durchsatz. Von einer derartigen Optimierung wird hinsichtlich des zeitlichen Aufwandes jedoch abgesehen.

4.2.1.3. Messergebnisse zur Default-Konfiguration

Dieses Kapitel beschreibt die gemessenen Leistungsunterschiede zwischen den Workloads in Bezug auf die Default-Konfiguration. Wie anhand von Abbildung 58(a) deutlich wird, erreicht Workload U mit $\sim$ 103.910 Operationen den höchsten Durchsatz pro Sekunde. Gegenüber Workload U (Modifikation einzelner Felder) erreicht Workload I (Neuanlage vollständiger Datensätze) mit $\sim$ 48.666 Operationen pro Sekunde einen um $\sim$ 53,16% geringeren Durchsatz. Mit $\sim$ 709 Operationen pro Sekunde erzielt das Lesen der Datensätze (Workload R) den insgesamt niedrigsten Durchsatz. Folglich ist das Einfügen neuer Datensätze um den Faktor 68,66 schneller als das Lesen bestehender Datensätze. Bedingt durch die Kombination der Operationsarten erreicht Workload M einen Durchsatz von $\sim$ 1.362 Operationen pro Sekunde.



(a) Durchschnittlicher Durchsatz (b) Durchschnittliche Latenz

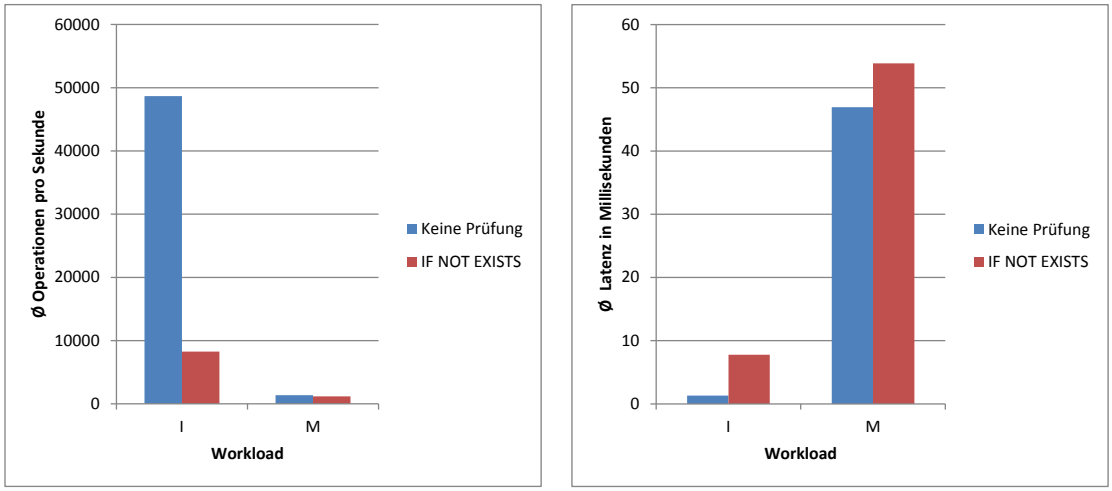
Abbildung 58: Cassandra: Messergebnisse zur Default-Konfiguration

In Hinblick auf die gemessenen Latenzwerte zeigt sich ein entsprechend vergleichbares Bild (siehe Abbildung 58(b)). Demnach benötigt Workload U die geringste durchschnittliche Latenz mit $\sim 0{,}6$ ms und Workload R mit durchschnittlich $\sim 90{,}2$ ms die höchste Latenz.

4.2.2. Existenzvalidierung bei Insert-Operationen

Bei der zugrundeliegenden Version 2.1.2 unterstützt Cassandra für Insert-Operationen die zusätzliche Prüfung auf das Vorhandensein eines bestehenden Datensatzes mit identischem Key. Durch die Erweiterung des entsprechenden Insert CQL Statements um den Anhang "IF NOT EXISTS" wird verhindert, dass ggf. bestehende Datensätze mit Inhalten des neuen Datensatzes überschrieben werden. Im Rahmen dieses Kapitels wird diese Funktionalität lediglich für Insert-Operationen getestet, da die entsprechende Update-Implementierung bis zur Version 2.1.2 fehlte und erst mit Version 2.1.3 nachgereicht wurde [Leb15].

Wie mittels Abbildung 59(a) ersichtlich wird, geht die Verwendung dieser Funktionalität mit einer erheblichen Durchsatzeinbuße einher. So sinkt der Durchsatz von ~ 48.667 auf ~ 8.269 Operationen pro Sekunde und folglich um mehr als 83 %. Umgekehrt wird ohne zusätzliche Validierung ein um den Faktor $\sim 5{,}9$ höherer Durchsatz erzielt. Bedingt durch den lediglich 25 %igen Anteil an Insert-Operationen beim Workload M wirkt sich die Leistungseinbuße insgesamt weniger stark aus. Entsprechend sinkt der Durchsatz lediglich um $\sim$12,8 % von ~ 1.362 auf ~ 1.187 Operationen pro Sekunde.



(a) Durchschnittlicher Durchsatz (b) Durchschnittliche Latenz

Abbildung 59: Cassandra: Existenzvalidierung bei Insert-Operationen - Durchsatz und Latenz

Bezüglich der gemessenen Latenz (siehe Abbildung 59(b)) ist festzuhalten, dass das zuvor beschriebene Verhalten des Durchsatzes in entsprechender Weise vergleichbar ist. Demnach erfolgt hierfür keine gesonderte Beschreibung.

4.2.3. Prepared Statements

Dieses Kapitel untersucht, inwiefern sich die Verwendung von Prepared Statements bzw. deren expliziter Verzicht auf die Leistungsfähigkeit bei den vier Workloads auswirkt. Mit der Betrachtung von Abbildung 60 wird deutlich, dass durch die Verwendung von Prepared Statements eine wesentliche Durchsatzsteigerung bei schreibenden Operationen erzielt werden kann. So steigt der Durchsatz bei Workload I von $\sim$ 32.248 auf $\sim$ 48.667 Operationen pro Sekunde ($\sim$ 50,9 %). Vergleichbar hierzu steigt der Durchsatz bei Workload U von $\sim$ 66.309 auf $\sim$ 103.910 Operationen pro Sekunde ($\sim$ 56,7 %). In Anbetracht der geringen Durchsatzsteigerung von $\sim$ 0,78 % bei Workload R ist mit größerer Wahrscheinlichkeit von einer Messungenauigkeit auszugehen, als von einer tatsächlichen Leistungsbeeinflussung lesender Operationen.

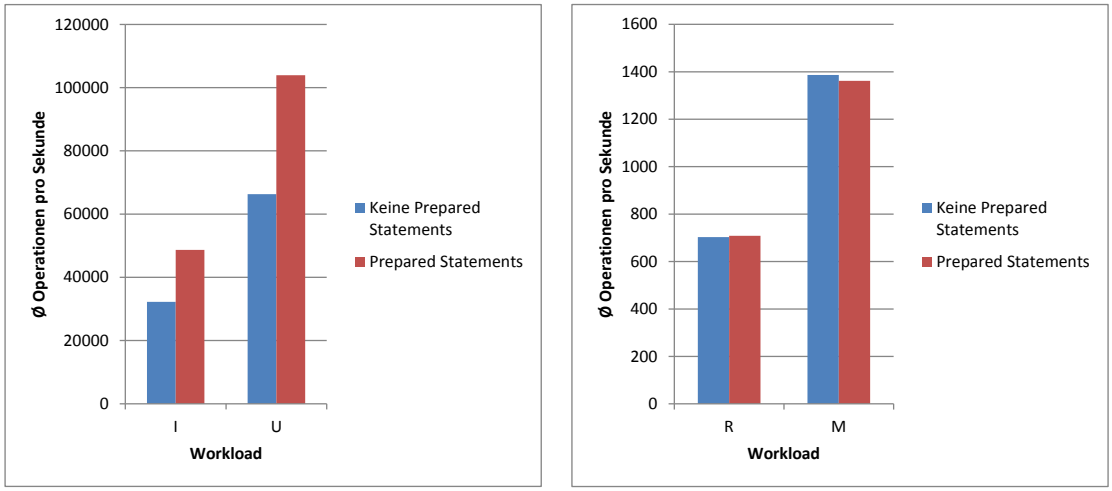


Abbildung 60: Cassandra: Prepared Statements - Durchsatz

Unerwarteter Weise wird bei Workload M unter der Verwendung von Prepared Statements sogar ein um $\sim 1{,}8\ \%$ geringerer Durchsatz erreicht als ohne deren Einsatz. Bedingt durch den hier ebenfalls nur geringfügig feststellbaren Leistungsunterschied ist auch dieses Ergebnis höchstwahrscheinlich auf eine Messschwankung zurückzuführen. Interessant ist jedoch der Aspekt, dass selbst die über 50%ige Durchsatzsteigerung bei den schreibenden Operationen zu keiner Erhöhung des Durchsatzes bei Workload M führen. In diesem Zusammenhang ist eine Betrachtung der gemessenen Latenzwerte für die einzelnen Operationsarten hilfreich (siehe Abbildung 61).

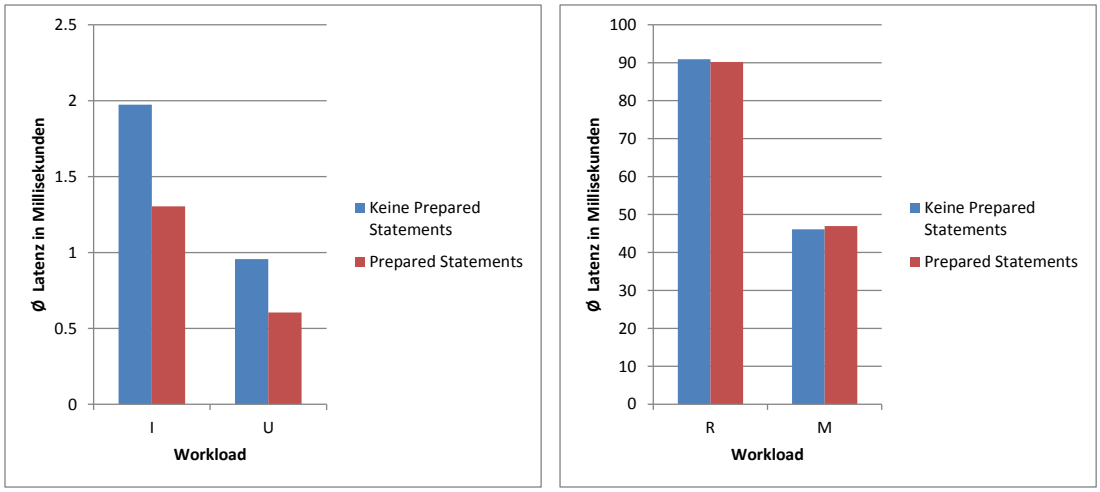


Abbildung 61: Cassandra: Prepared Statements - Latenz

Einerseits sinkt bei den Workloads I und U die Latenz merklich durch die Verwendung der Prepared Statements ($\sim 34\ \%$ und $\sim 36{,}8\ \%$). Andererseits bleibt

die Latenz bei Workload R nahezu konstant auf einem Wert von 90 ms. Bedingt durch die großen absoluten Unterschiede zwischen den erforderlichen Latenzen bei den Workloads I, U und R fallen die erzielten Leistungsgewinne der schreibenden Operationen beim Workload M jedoch nur geringfügig ins Gewicht.

4.2.4. Unterschiedliche Datensatzgrößen

Dieses Kapitel untersucht das Verhalten von Cassandra in Bezug auf die Verwendung unterschiedlicher Datensatzgrößen. Ausgehend von der standardmäßigen Größe wurden drei weitere Messungen durchgeführt, bei denen sich die Datensatzgröße jeweils verzehnfachte. Die Anzahl der initial geladenen Datensätze wurde derart gewählt, dass die Gesamtgröße der Daten (exklusive DB-spezifischem Overhead) nahezu konstant bleibt. Eine Übersicht der zugrundeliegenden Konfigurationen kann der Tabelle 16 entnommen werden.

Faktor	**1**	**10**	**100**	**1.000**
Feldgröße in Byte	60	600	6.000 ≈ 5,86 KB	60.000 ≈ 58,6 KB
Datensatzgröße in Byte	622,88	6.022,88 ≈ 5,88 KB	60.022,88 ≈ 58,6 KB	600.022,88 ≈ 585,96 KB
Anzahl Datensätze	110.330.000	11.410.000	1.144.900	114.532
Gesamte Datengröße in GB	∼ 16 GB	∼ 16 GB	∼ 16 GB	∼ 16 GB

Tabelle 16: Cassandra: Übersicht evaluierter Datensatzgrößen

Der Versuch, Konfigurationen mit noch größeren Datensätzen (ab Faktor 10.000) zu untersuchen, musste bereits beim initialen Befüllen der Tabelle das Vorgehen abgebrochen werden. Die Ursache liegt darin, dass einzelne Cassandra Nodes mit voranschreitender Laufzeit nicht mehr von den YCSB Clients erreicht[73] werden konnten.

[73] Der Versuch weitere Datensätze einzufügen schlägt mit einer UnavailableException (Not enough replica available for query at consistency ONE (1 required but only 0 alive)) fehl.

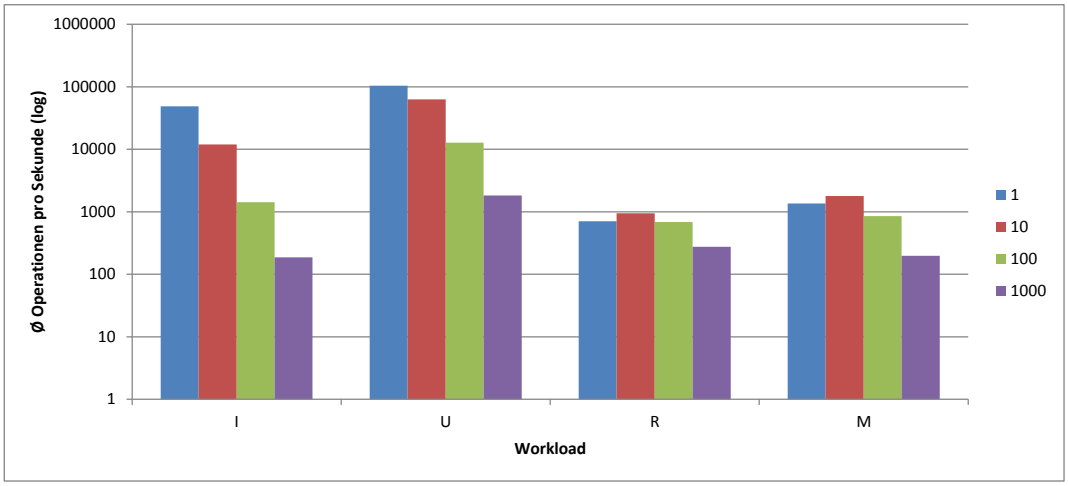


Abbildung 62: Cassandra: Unterschiedliche Datensatzgrößen - Durchsatz

Anhand von Abbildung 62 wird ersichtlich, dass der Durchsatz erwartungsgemäß bei den Workloads I und U mit zunehmender Datensatzgröße sinkt. Demgegenüber ist jedoch auch zu erkennen, dass bei den Workloads R und M der Durchsatz infolge einer Vergrößerung der Datensätze um den Faktor 10 zunächst sogar zunimmt - mit jeder weiteren Verzehnfachung jedoch wiederum erneut abnimmt. Diesbezüglich bietet die Tabelle 17 eine Übersicht, anhand derer entnommen werden kann, wie stark sich jede Verzehnfachung der Datensatzgröße prozentual auf den Durchsatz gegenüber der vorherigen Größe auswirkt.

Vergrößerung	**Workload I**	**Workload U**	**Workload R**	**Workload M**
1 → 10	−75,4 %	−39,5 %	+34,0 %	+31,7 %
10 → 100	−88,0 %	−79,7 %	−27,7 %	−52,5 %
100 → 1.000	−86,9 %	−85,7 %	−60,0 %	−76,8 %

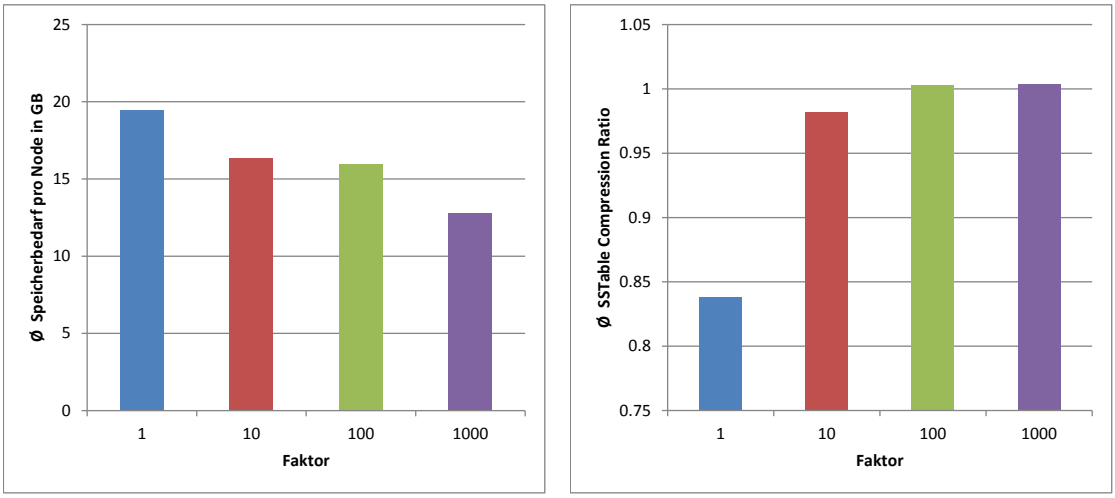
Tabelle 17: Cassandra: Unterschiedliche Datensatzgrößen - Prozentuale Durchsatzentwicklung

Infolge der fehlenden Berücksichtigung[74] des Cassandra-spezifischen Overheads entspricht der letztendliche physische Speicherbedarf nicht den zugrunde gelegten 16 GB an reinem Inhalt[75]. Von den vier gemessenen Konfigurationen besitzt die Ausgangskonfiguration (Faktor 1) mit durchschnittlich ∼ 19,45 GB den höchsten Speicherbedarf pro Node (siehe Abbildung 63(a)). Im Gegensatz dazu benötigt die Faktor 1.000 Konfiguration mit einem Speicherbedarf von durchschnittlich ∼ 12,78 GB pro Node den geringsten Speicher. Dass der abnehmende Speicher-

[74] Der verfügbaren Systemdokumentation [Dat15b] konnte keine zuverlässige Berechnungsgrundlage entnommen werden.

[75] (Keygröße + (10 Felder x Feldgröße)) x Anzahl Datensätze

bedarf nicht auf eine effizientere LZ4 Komprimierung bei zunehmender Datensatzgröße zurückzuführen ist, wird mittels Abbildung 63(b) deutlich. Während die Standardkonfiguration noch ein Kompressionsverhältnis[76] von $\sim 0{,}84$ erzielt, verschlechtert sich diese jedoch mit jeder weiteren Verzehnfachung der Datensatzgröße. Bei der Faktor 100 und der Faktor 1.000 Konfiguration führt dieses Verhalten sogar zu einem Kompressionsverhältnis > 1, sodass die Kompression letztendlich zu einer geringfügigen Erhöhung des Speicherbedarfs führt.



(a) Physischer Speicherbedarf (b) SSTable Compression Ratio

Abbildung 63: Cassandra: Unterschiedliche Datensatzgrößen - Speicherbedarf

Infolgedessen ist davon auszugehen, dass der insgesamt verringerte Speicherbedarf auf einen geringeren Anteil des Cassandra-spezifischen Overheads zurückzuführen ist. Schließlich wurde bei der Vergrößerung der Datensatzgröße auch gleichzeitig die Anzahl der Datensätze reduziert, um die Inhaltsgröße von 16 GB konstant zu halten. Theoretisch könnte dieser Umstand auch die beobachtete Steigerung des Durchsatzes bei der Erhöhung der Datensatzgröße von Faktor 1 auf 10 erklären. In diesem Fall wäre jedoch zu erwarten gewesen, dass der Durchsatz auch mit dem weiter sinkenden Speicherbedarf bei der Faktor 100 und Faktor 1.000 Konfiguration entweder steigt oder zu mindestens in abgeschwächter Intensität sinkt. Folglich bleibt die Ursache für das unerwartete partielle Steigen des Durchsatzes an dieser Stelle ungeklärt.

Abschließend ist anzumerken, dass mit zunehmender Datensatzgröße auch der Anteil fehlgeschlagener Operationen[77] ansteigt (siehe Tabelle 18). Der höchste

[76] Komprimierte Datengröße dividiert durch die unkomprimierte Datengröße.
[77] Server-seitiger Abbruch mittels WriteTimeoutException bzw. ReadTimeoutException.

Anteil fehlgeschlagener Operationen wird bei Workload M mit der Faktor 1.000 Konfiguration erreicht. Insgesamt schlugen hier durchschnittlich $\sim$ 10,06 % der durchgeführten Operationen fehl. Im Einzelnen scheiterten dabei durchschnittlich $\sim$ 20,52 % der Inserts, $\sim$ 19,87 % der Updates und weniger als 0,02 % der Reads. Entsprechend kann somit festgehalten werden, dass mit zunehmender Datensatzgröße insbesondere schreibende Operationen fehlschlagen und dass deren Ausmaß im gemischten Workload stärker zum Tragen kommt, als bei der Ausführung reiner Workloads.

Faktor	**Workload I**	**Workload U**	**Workload R**	**Workload M**
1	0 %	0 %	0 %	0 %
10	$< 0,01$ %	0 %	0 %	0 %
100	$\sim 0,67$ %	$< 0,01$ %	0 %	$\sim$ 1,80 %
1.000	$\sim 3,76$ %	$\sim 0,15$ %	0 %	$\sim$ 10,06 %

Tabelle 18: Cassandra: Unterschiedliche Datensatzgrößen - Durchschnittlicher Anteil fehlgeschlagener Operationen

4.2.5. Unterschiedliche initiale Datensatzmengen

Im Rahmen diese Kapitels wird untersucht, inwiefern die vier zugrundeliegenden Workloads durch verschieden große initiale Dokumentenmengen beeinflusst werden. Ausgehend von der Default-Menge mit 11,3 Mio. Datensätzen werden sowohl zwei kleinere als auch zwei größere Datenmengen untersucht. Der Tabelle 19 kann eine Übersicht der betrachteten Datenmengen entnommen werden.

	27 Mio.	**55 Mio.**	**110 Mio.**	**220 Mio.**	**441 Mio.**
Anzahl Datensätze	27.582.500	55.165.000	110.330.000	220.660.000	441.320.000
Ø Speicherbedarf Inhalt pro Node	$\sim$ 4 GB	$\sim$ 8 GB	$\sim$ 16 GB	$\sim$ 32 GB	$\sim$ 64 GB
Ø Speicherbedarf tatsächlich pro Node	$\sim$ 4,86 GB	$\sim$ 9,72 GB	$\sim$ 19,45 GB	$\sim$ 38,86 GB	$\sim$ 77,79 GB

Tabelle 19: Cassandra: Ressourcenbedarf bei unterschiedlich großen initialen Datensatzmengen

Hierbei gilt es den tatsächlichen Speicherbedarf im Vergleich zum berechneten Bedarf für den ausschließlichen Inhalt[78] zu beachten. Diesbezüglich ist zu beobachten,

[78] (Keygöße + (10 Felder x Feldgröße)) x Anzahl Datensätze

dass zur Speicherung der Datensätze, unabhängig von ihrer Anzahl, ein ~ 21 %iger Overhead[79] erforderlich ist.

Anhand von Abbildung 64 wird deutlich, dass die unterschiedlich großen, initial ins Datenbank-Cluster geladenen Datenmengen keine wesentlichen Auswirkungen auf die schreibenden Operationen haben. Entsprechend beträgt der maximale Unterschied bzgl. der gemessenen Durchsatzraten weniger als 2 % bei Workload I und weniger als 1,4 % bei Workload U.

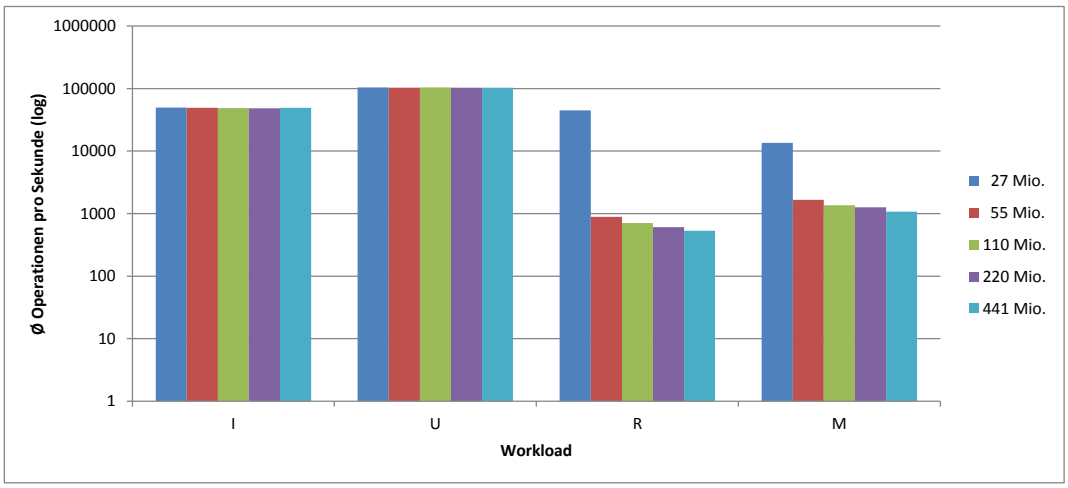


Abbildung 64: Cassandra: Unterschiedliche Datensatzmengen - Durchsatz

Demgegenüber wirken sich die unterschiedlichen Datensatzmengen deutlich auf den Durchsatz lesender Operationen aus. Insgesamt zeigt sich bei den Workloads R und M ein vergleichbares Bild. Während die Konfiguration mit ~ 27 Mio. Datensätzen den höchsten Durchsatz erreicht, sinkt dieser bereits drastisch bei einer Menge von ~ 55 Mio. Datensätzen ab. Die nachfolgenden Verdoppelungen der Datensatzmengen führen dagegen nur noch zu vergleichsweise moderaten Verminderungen des Durchsatzes. Eine entsprechende Übersicht bzgl. der ermittelten Durchsatzentwicklung ist in der nachfolgenden Tabelle 20 gegeben.

Vergrößerung	**Workload I**	**Workload U**	**Workload R**	**Workload M**
27 Mio. → 55 Mio.	$-0,68$ %	$-0,83$ %	$-98,03$ %	$-87,73$ %
55 Mio. → 110 Mio.	$-1,02$ %	$+0,44$ %	$-19,97$ %	$-18,31$ %
110 Mio. → 220 Mio.	$-0,29$ %	$-0,65$ %	$-14,29$ %	$-7,52$ %
220 Mio. → 441 Mio.	$+2,02$ %	$-0,28$ %	$-12,50$ %	$-15,12$ %

Tabelle 20: Cassandra: Unterschiedliche Datensatzmengen - Prozentuale Durchsatzentwicklung

[79] Sämtliche Konfigurationen verwendeten Datensätze mit 10 Feldern und jeweils 60 Byte Zufallsdaten. Abweichende Konfigurationsparameter könnten zu anderen Ergebnissen führen.

In Anbetracht des geringen Speicherbedarfs von nur Ø 4,86 GB pro Node für die Menge von insgesamt $\sim$ 27 Mio. Datensätze könnte das vollständige Vorhalten der Daten im Page Cache den außerordentlich hohen Durchsatz erklären. Bedingt durch den ansteigenden Speicherbedarf bei höheren Mengen von Datensätzen sinkt gleichzeitig der prozentuale Anteil im Page Cache vorgehaltener Daten. Durch den somit immer häufiger erforderlichen Zugriff auf den Sekundärspeicher erklärt sich auch der sinkende Durchsatz von lesenden Operationen bei zunehmenden Datenmengen.

4.2.6. Compaction

Im Rahmen dieses Kapitels wird der Leistungsunterschied zwischen der standardmäßigen Size-tiered Compaction und der Leveled Compaction untersucht. Diesbezüglich ist hervorzuheben, dass beide Compaction Strategien den nahezu identischen Speicherbedarf für die initiale Datenmenge erforderten[80]. Demnach kann eine gravierende Beeinflussung der Messergebnisse infolge eines unbeabsichtigten Caching-Vorteils[81] weitestgehend ausgeschlossen werden.

In Hinblick auf die Ausführung schreibender Operationen ist kein wesentlicher Unterschied zwischen den beiden Compaction Strategien festzustellen (siehe Abbildung 65).

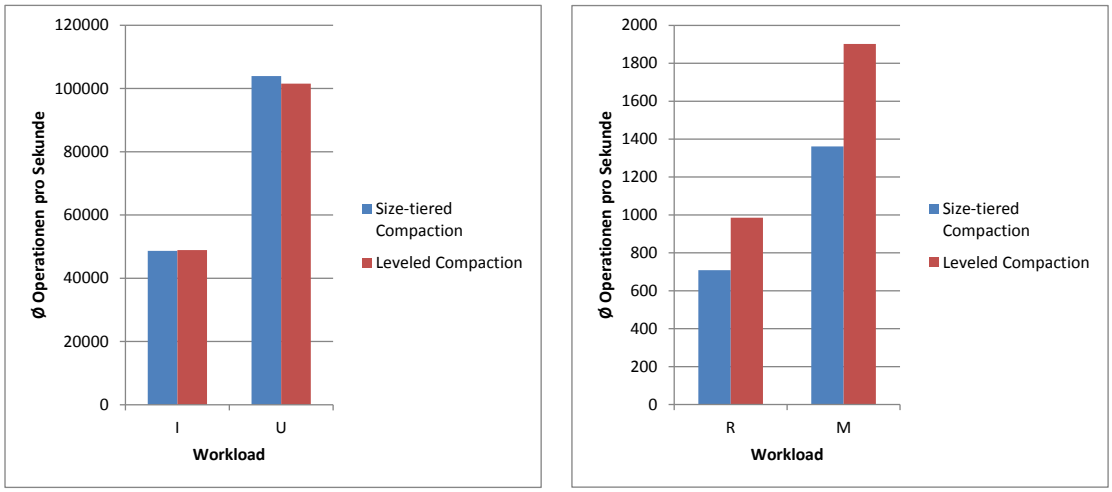


Abbildung 65: Cassandra: Compaction - Durchsatz

Entsprechend beträgt bei Workload I die Durchsatzsteigerung der Leveled Com-

[80] Die Abweichung betrug dabei ca. 0,183 %.

[81] Je kleiner der Speicherbedarf für eine Datenmenge ist, desto größer kann bspw. der Anteil im Page Cache gehaltener Daten sein.

paction lediglich $\sim 0{,}46$ % und bei Workload U die Durchsatzverringerung bei der Leveled Compaction $\sim 2{,}28$ %. Demgegenüber führt die Verwendung der Leveled Compaction sowohl beim Workload R als auch beim Workload M zu einer ~ 39 %igen Durchsatzsteigerung.

Bezüglich der gewonnenen Erkenntnisse ist jedoch anzumerken, dass die verhältnismäßig geringe Laufzeit des Benchmarks von 10 min (5 min Warmup und 5 min Messung), nicht zwingend eine Aussage über das Langzeitverhalten erlaubt. Die Untersuchung dieses Aspekts könnte somit Gegenstand anschließender Arbeiten sein.

4.2.7. Komprimierung

Dieses Kapitel untersucht die Auswirkungen, welche durch den Einsatz der mit Cassandra bereitgestellten Kompressionsalgorithmen entstehen. Neben der standardmäßigen LZ4 Kompression werden hierbei die Snappy und die Deflate Kompression betrachtet. Des Weiteren wird untersucht, wie sich dagegen die Deaktivierung der Kompressionsfunktionalität auswirkt.

Für die von YCSB generierten Testdaten erzielten die Kompressionsalgorithmen im Rahmen dieser Untersuchung bedeutende Kompressionsraten[82]. Entsprechend erreichte LZ4 eine Kompressionsrate von $\sim 0{,}839$, Snappy eine von $\sim 0{,}82$ und Deflate eine von $\sim 0{,}547$. Folglich reduziert die Deflate Komprimierung den Speicherbedarf auf nahezu die Hälfte der ursprünglichen Größe.

Hinsichtlich der schreibenden Operationsarten (siehe Abbildung 66) existieren keine markanten Leistungsunterschiede zwischen den Kompressionsarten. Einerseits wird durch das Deaktivieren der Kompression die höchste Schreibleistung erzielt. Andererseits beträgt die Differenz gegenüber der Verwendung von Kompressionsalgorithmen im Maximum[83] $\sim 2{,}67$ %. Wesentlich deutlicher wirkt sich dagegen die Verwendung auf Workload R und M aus. Bezüglich Workload R sinkt der Durchsatz im Vergleich zur LZ4 Komprimierung bei der Snappy Komprimierung um $\sim 6{,}3$ % und bei deaktivierter Komprimierung um $\sim 14{,}5$ %. Demgegenüber steigt der Durchsatz bei der Deflate Komprimierung um $\sim 17{,}6$ %. Hinsichtlich Workload M wurde lediglich durch die Deaktivierung der Kompression ein um $\sim 15{,}9$ % geringerer Durchsatz erreicht als bei der Verwendung von LZ4. Dagegen

[82] Komprimierte Datengröße dividiert durch die unkomprimierte Datengröße.
[83] Bezüglich der Deflate Kompression bei Workload U.

erzielte die Snappy Kompression einen $\sim$ 2,3 % und die Deflate Kompression einen $\sim$ 17 % höheren Durchsatz. Demnach ist festzuhalten, dass entgegen der Cassandra Dokumentation[Dat15a, S. 59] dennoch Szenarien existieren, in welchen der Deflate Algorithmus eine entsprechend hohe Kompression erreicht, sodass dessen langsamere Ausführung mehr als ausgeglichen wird.

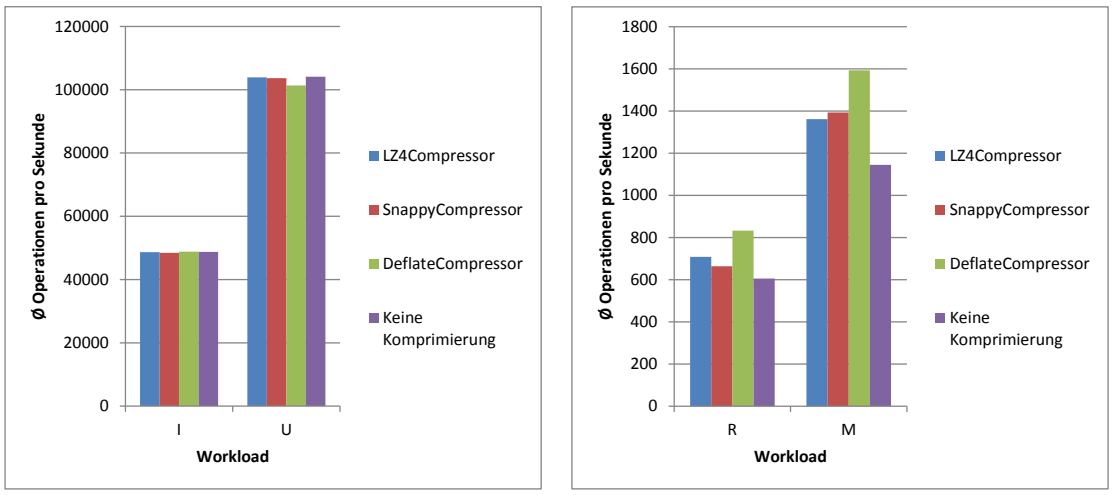


Abbildung 66: Cassandra: Komprimierung - Durchsatz

Insgesamt wird deutlich, dass die Verwendung der von Cassandra bereitgestellten Kompressionsalgorithmen lediglich einen geringen negativen Einfluss auf schreibenden Operationen ausüben. Deutlich positiver wirkt sich dagegen der erzielbare Leistungsgewinn bei den lesenden Operationen aus.

4.2.8. Write-Ahead Logging

Gegenstand dieses Kapitels ist die Evaluierung der Leistungsunterschiede infolge verschiedener Write-Ahead Logging (WAL) Konfigurationen[84]. Zu Beginn wird untersucht, inwiefern sich die Periodic und Batch Konfigurationen mit ihren Default-Werten im Vergleich zur deaktivierten WAL Funktionalität unterscheiden. Nach dieser Betrachtung werden die Periodic und Batch Modi zusätzlich mit unterschiedlichen Parametrisierungen separat evaluiert.

Mit Blick auf Abbildung 67 wird deutlich, dass sich die Wahl des verwendeten WAL Betriebsmodus entscheidend auf den erzielbaren Durchsatz auswirkt. Gegenüber dem bei Cassandra standardmäßigen Periodic Modus sinkt der Durchsatz bei

[84]Eine Beschreibung der unterschiedlichen WAL Konfigurationen kann dem Kapitel 2.2.5 entnommen werden.

Workload I von ∼ 48.667 auf ∼ 1.064 Operationen pro Sekunde (-97,8 %) und bei Workload U von ∼ 103.910 auf ∼ 1.050 Operationen pro Sekunde (-98,9 %). Anders ausgedrückt, führt der Periodic Modus gegenüber dem Batch Modus zu einen um den Faktor ∼ 45,8 höheren Durchsatz bei Workload I und einen ∼ 98,9 fach höheren Durchsatz bei Workload U. Gegenüber des Periodic Modus führt die Deaktivierung der WAL Funktionalität bei Workload I zu einer ∼ 61 %igen Steigerung des Durchsatzes auf ∼ 78.343 Operationen pro Sekunde und bei Workload U zu einer ∼ 16,3 %igen Steigerung des Durchsatzes auf ∼ 120.861 Operationen pro Sekunde.

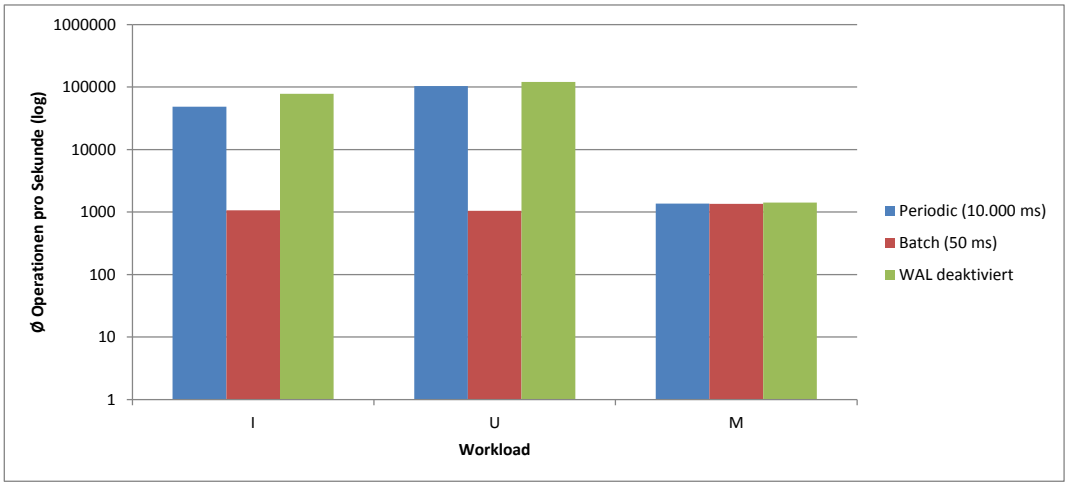


Abbildung 67: Cassandra: WAL - Durchsatz

Vergleichbar zu der in Kapitel 4.2.3 (Prepared Statements) gemachten Beobachtung, wirken sich auch hier die beachtlichen Auswirkungen auf die schreibenden Zugriffe letztendlich nicht in adäquater Form bei Workload M aus. Gegenüber der Periodic Konfiguration sinkt der Durchsatz im Batch Modus lediglich um ∼ 0,49 % und steigt dagegen bei der deaktivierten WAL Funktion um ∼ 3,99 %.

4.2.8.1. Separate Betrachtung von Periodic Konfigurationen

Bezüglich des Periodic Modus untersucht dieser Abschnitt inwieweit ein häufigeres (alle 5.000 ms) bzw. selteneres (alle 20.000 ms) Persistieren des Commit Logs eine Wirkung erzielt.

Insgesamt betrachtet (siehe Abbildung 68) bewirkt die Halbierung bzw. Verdoppelung des Persistierungsintervalls lediglich eine verhältnismäßig geringfügige Beeinflussung des Durchsatzes. Gegenüber der standardmäßigen 10.000 ms Parametrisierung steigt der Durchsatz durch die Verdoppelung des Intervalls bei Workload I

um ∼ 2,3 %, bei Workload U um ∼ 0,6 % und bei Workload M um ∼ 1,8 %. Infolge des halbierten Intervalls steigt einerseits bei Workload I der Durchsatz um ∼ 0,6 % und bei Workload U um ∼ 5,1 %. Andererseits sinkt der Durchsatz bei Workload M um ∼ 0,5 %.

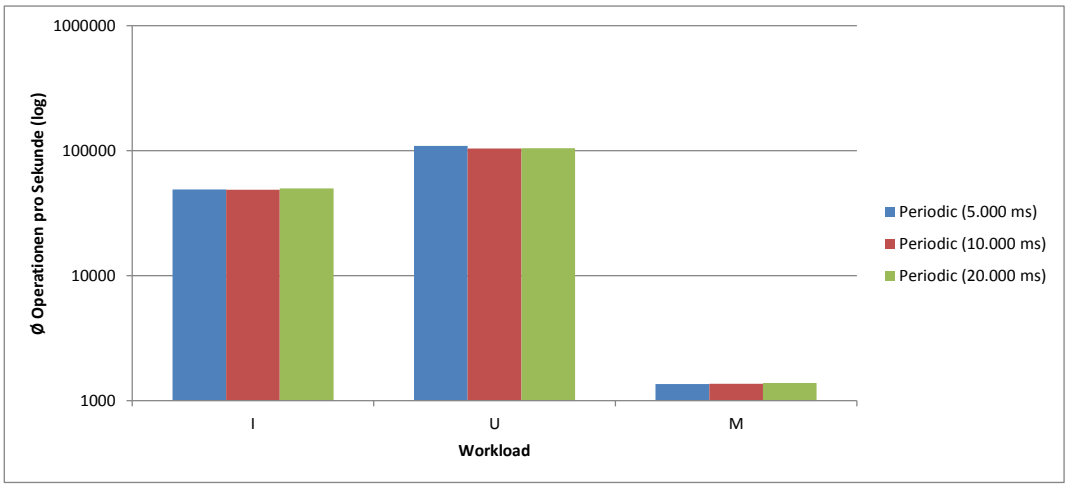


Abbildung 68: Cassandra: WAL Periodic - Durchsatz

Letztlich kann auf Basis der gemessenen Differenzen weder eine eindeutig durchsatzfördernde noch -senkende Wirkung festgestellt werden. Eine mögliche Ursache für die derart geringfügige Auswirkung könnte in der asynchronen Persistierung[85] des Commit Logs begründet sein.

4.2.8.2. Separate Betrachtung von Batch Konfigurationen

Dieser Abschnitt untersucht, inwieweit sich ein halb (alle 25 ms) bzw. doppelt (alle 100 ms) so häufiges Persistieren des Commit Logs im Batch Modus auswirkt. Infolge der synchronen Persistierung sollte sich zu mindestens theoretisch der Durchsatz durch die Halbierung des Persistierungsintervalls steigern lassen. Die Verdoppelung des Intervalls sollte dagegen einen niedrigeren Durchsatz zur Folge haben.

Mit der Betrachtung von Abbildung 69 wird jedoch ersichtlich, dass die standardmäßige Parametrisierung von 50 ms den insgesamt höchsten Durchsatz bei den drei betrachteten Workloads erzielt. Demgegenüber fällt der Durchsatz durch die Verdoppelung des Intervalls bei Workload I und U um ∼ 32 % und bei Workload M um ∼ 15,2 %. Dagegen sinkt bei der Halbierung des Intervalls der Durchsatz deutlich geringfügiger. Entsprechend fällt der Durchsatz bei Workload I um ∼ 5,8 %,

[85] Siehe Kapitel 2.2.5.

bei Workload U um $\sim 4{,}2\ \%$ und bei Workload M um $\sim 2{,}7\ \%$.

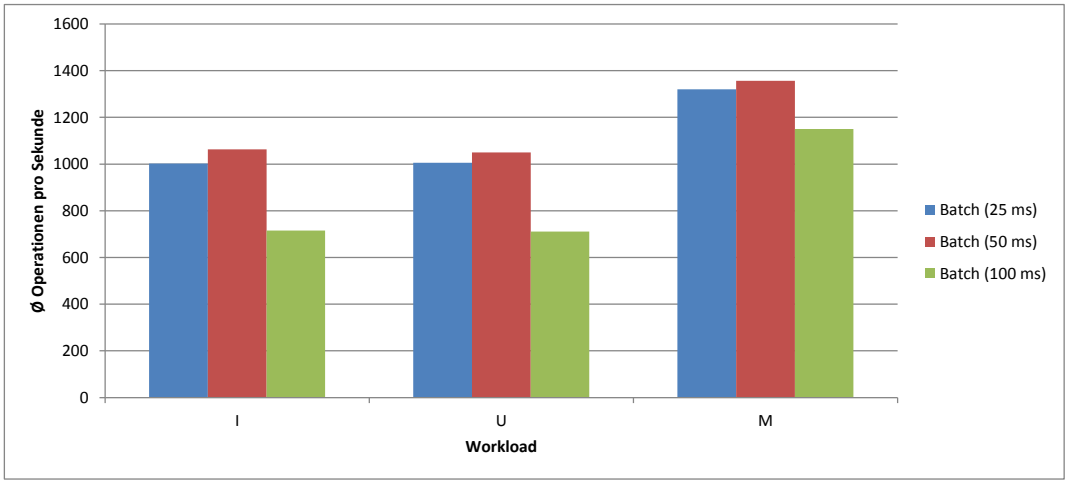


Abbildung 69: Cassandra: WAL Batch - Durchsatz

Da die beobachtete Durchsatzminderung infolge der Halbierung des Persistierungsintervalls zu mindestens konzeptionell widersprüchlich ist, wird als Ursache ein zu geringes Leistungsvermögen seitens des Sekundärspeichers vermutet. Bedingt durch die bisher fehlende Erhebung entsprechend unterstützender Metriken, bleibt die Bestätigung bzw. Widerlegung dieser These nachfolgenden Arbeiten vorbehalten.

4.2.9. Replikation

Im Rahmen dieses Kapitels wird untersucht, inwiefern sich die Verwendung der Replikation auf das Leistungsvermögen von Cassandra auswirkt. In diesem Zusammenhang wird vor allem das Verhalten mit zunehmender Anzahl an Replikationen betrachtet.

Bedingt durch den starken Einfluss der Größe der zugrundeliegenden Datenmenge (siehe Kapitel 4.2.5), wurde die Anzahl der vom System gehaltenen Datensätze für jede Konfiguration nahezu identisch gehalten. Hierzu sinkt die Anzahl der dem System zugefügten Daten im selben Verhältnis zum Anstieg des Replikationsfaktors (siehe Tabelle 21). Diese Maßnahme ist erforderlich, um die Vergleichbarkeit zwischen den Konfigurationen zu gewährleisten.

	Keine Replikation	**Einfache Replikation**	**Zweifache Replikation**	**Dreifache Replikation**
Cassandra Replikationsfaktor	1	2	3	4
Anzahl Datensätze ohne Replikation	110.330.000	55.165.000	36.776.668	27.582.500
Anzahl Datensätze mit Replikation	110.330.000	110.330.000	110.330.004	110.330.000

Tabelle 21: Cassandra: Replikation - Verwendete Datenmengen

Gegenüber der Default-Konfiguration aus Kapitel 4.2.1.3 betrug die Abweichung des erforderlichen Speicherbedarfs bei den nachfolgend diskutierten Replikationskonfigurationen jeweils weniger als 0,1%. Demnach kann eine wesentliche Beeinflussung infolge eines unterschiedlichen Speicherbedarfs ausgeschlossen werden.

Bzgl. der nachfolgend beschriebenen Messergebnisse ist anzumerken, dass die zugrundeliegende WAL Konfiguration der Standardeinstellung (Periodic) entsprach. Entsprechend erfolgte die Persistierung des Commit Logs asynchron zur Bestätigung schreibender Operationen.

4.2.9.1. Antwort von allen Replika-Nodes abwarten

In diesem Abschnitt wird das Systemverhalten untersucht, wenn auf eine Verarbeitungsbestätigung aller Replika-Nodes[86] gewartet wird. In diesem Fall wartet der jeweilige Coordinator Node[87] bei der Operationsausführung stets auf die Antwort sämtlicher Replika-Nodes, bevor er das Resultat an den Client zurücksendet. Bei einer einfachen Replikation sind es zwei Nodes, die auf eine Operationsanfrage antworten müssen. Bei einer zweifachen Replikation sind es drei und bei der dreifachen Replikation sind es alle vier Nodes. Sinnvoll erscheint diese Konfiguration für Szenarien, bei denen ein konsistentes Lesen von Daten erforderlich ist. Im Vergleich zu anderen Konfigurationen wird dies auch gewährleistet, wenn Datensätze augenblicklich geschrieben werden und erst ein Teil der Replika-Nodes den Schreibvorgang abgeschlossen hat.

In Bezug auf die gemessenen Replikationskonfigurationen stellt sich bei allen vier Workloads ein ähnliches Verhalten dar (siehe Abbildung 70). So wird stets der

[86] Cassandra Konsistenz-Level = All
[87] Siehe Kapitel 2.2.

höchste Durchsatz erzielt, sofern keine Replikation erfolgt und somit nur eine Antwort abgewartet wird. Der niedrigste Durchsatz wird dagegen unter Verwendung der dreifachen Replikation erreicht.

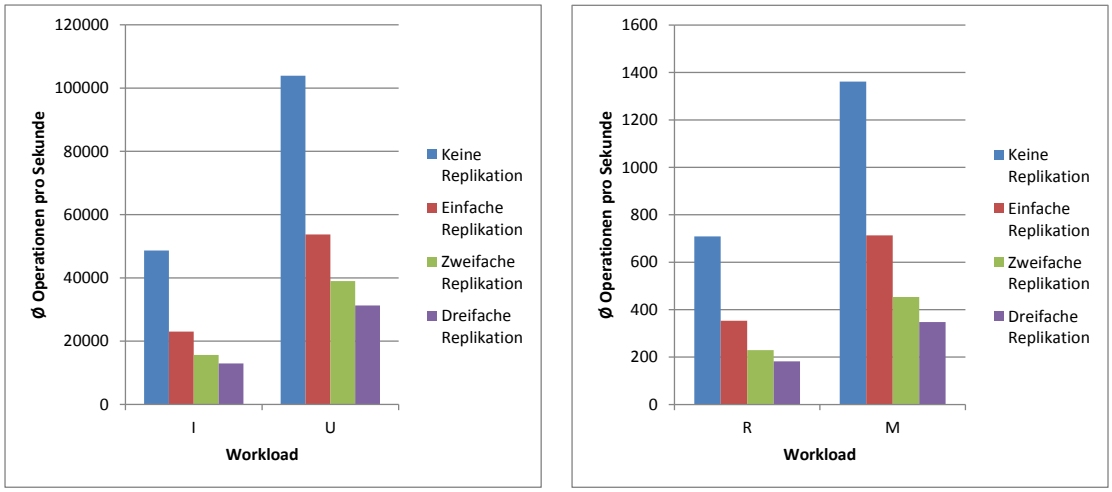


Abbildung 70: Cassandra: Replikation (alle Antworten abwarten) - Durchsatz

Die nachfolgende Tabelle 22 gibt einen Überblick über den prozentualen Durchsatzverlust, der bei den unterschiedlichen Replikationskonfigurationen im Vergleich zu der deaktivierten Replikation auftritt. Hierbei wird deutlich, dass das Abwarten der Antwort von sämtlich betroffenen Nodes zu Leistungseinbußen von mindestens $\sim$ 47 % und bis zu $\sim$ 75 % führt. Zudem ist ersichtlich, dass mit steigender Anzahl an Replikaten, der Durchsatz bei allen vier Workloads verhältnismäßig gleichartig sinkt. In diesem Zusammenhang beträgt die Abweichung zwischen den Workloads im Maximum $\sim$ 5 %.

	Einfache Replikation	**Zweifache Replikation**	**Dreifache Replikation**
Workload I	-52,7 %	-67,9 %	-73,3 %
Workload U	-48,3 %	-62,5 %	-69,9 %
Workload R	-50,2 %	-67,6 %	-74,3 %
Workload M	-47,6 %	-66,7 %	-74,5 %

Tabelle 22: Cassandra: Replikation - Prozentual sinkender Durchsatz gegenüber der deaktivierten Replikation, wenn alle Antworten abgewartet werden

4.2.9.2. Antwort von der Mehrheit aller Replika-Nodes abwarten

Innerhalb dieses Abschnittes wird untersucht, inwiefern sich die Leistungsfähigkeit bei steigender Anzahl an Replikaten entwickelt, sofern der Coordinator Node auf

die Antwort von der Mehrheit der Replika-Nodes[88] abwartet, bevor er deren Resultat an den Client zurücksendet. Die Verwendung dieser Konfiguration ist bspw. sinnvoll, sofern ein konsistentes Lesen erfolgreich geschriebener Daten erforderlich ist.

Ebenso wie beim Abwarten der Antwort von sämtlichen Replika-Nodes, sinkt der Durchsatz auch beim Warten auf die Antworten von der Mehrheit aller Replika-Nodes bei steigender Anzahl von Replikaten (siehe Abbildung 71).

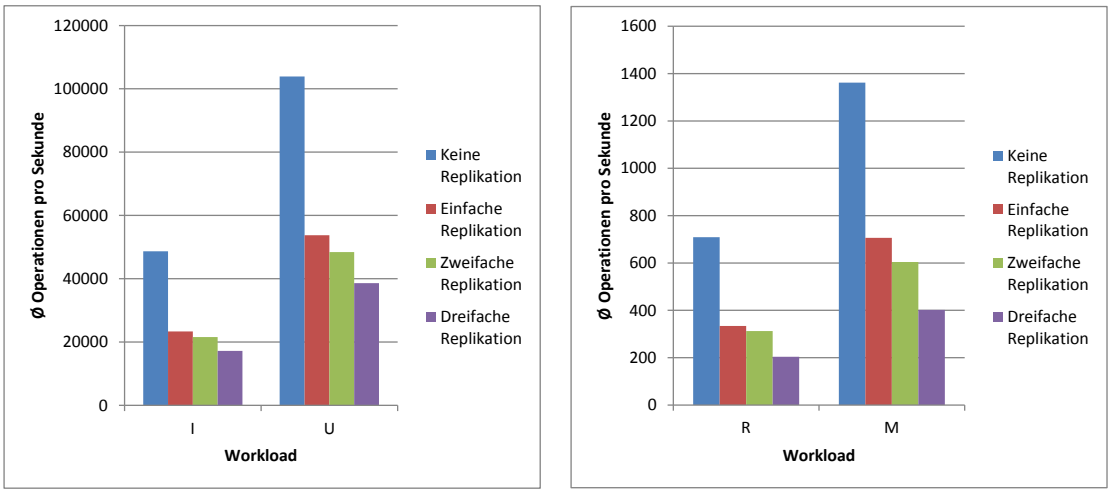


Abbildung 71: Cassandra: Replikation (Mehrheit aller Antworten abwarten) - Durchsatz

Gegenüber der deaktivierten Replikation zeigen sich Durchsatzeinbußen von mindestens ~ 48 % und bis zu ~ 71 % (siehe Tabelle 23). Bei der einfachen und zweifachen Replikation wirkt sich die Beeinträchtigung vergleichsweise ähnlich[89] auf die unterschiedlichen Workloads aus. Demgegenüber ist festzuhalten, dass infolge einer dreifachen Replikation schreibende Zugriffe weniger stark beeinträchtigt werden als lesende.

[88] Cassandra Konsistenz-Level = Quorum
[89] Abweichung zwischen den Workloads beträgt $\sim 2{,}5$ % bis $\sim 4{,}7$ %.

	Einfache Replikation	**Zweifache Replikation**	**Dreifache Replikation**
Workload I	-52,06 %	-55,65 %	-64,62 %
Workload U	-48,26 %	-53,41 %	-62,85 %
Workload R	-52,86 %	-55,87 %	-71,22 %
Workload M	-48,13 %	-55,62 %	-70,48 %

Tabelle 23: Cassandra: Replikation - Prozentual sinkender Durchsatz gegenüber der deaktivierten Replikation, wenn die Mehrheit aller Antworten abgewartet wird

Im Vergleich zur maximalen Durchsatzminderung beim Abwarten der Antworten von sämtlichen Replika-Nodes ($\sim$ 75 %) zeigt sich diese Konfiguration im Ganzen weniger stark beeinträchtigt. Die Ausnahme hiervon bildet die einfache Replikation, da hierbei lediglich zwei Instanzen desselben Datensatzes insgesamt im Cluster gehalten werden und beide zur Erfüllung der Mehrheitsbedingung erfolgreich gelesen bzw. geschrieben werden müssen. Folglich gleicht sich das letztendliche Verhalten der beiden Konfigurationen.

4.2.9.3. Antwort von einem Replika-Node abwarten

Im Folgenden wird untersucht, zu welchen Leistungsunterschieden es führt, wenn der Coordinator Node fortwährend nur die Antwort von einem der Replika-Nodes[90] abwartet, bevor er dessen Ergebnis an den Client zurücksendet. Sinnvoll kann diese Konfiguration für Anwendungsszenarien sein, in denen keine gesonderte Anforderung hinsichtlich der Konsistenz besteht.

Identisch zu den zuvor betrachteten Replikationskonfigurationen sinkt auch in dieser Konfiguration der Durchsatz mit steigender Anzahl an Replikaten, obwohl dabei stets nur eine einzelne Antwort abgewartet wird (siehe Abbildung 72). Im Gegensatz zu den vorangegangenen Konfigurationen wirkt sich die Erhöhung der Replikation hier jedoch am geringsten aus.

[90] Cassandra Konsistenz-Level = One

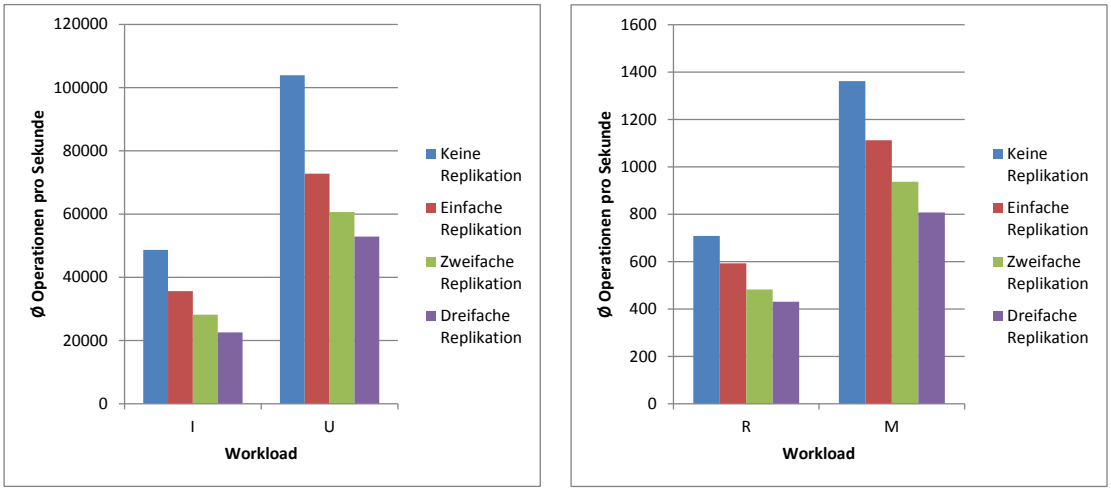


Abbildung 72: Cassandra: Replikation (eine Antwort abwarten) - Durchsatz

Im direkten Vergleich zur deaktivierten Replikation zeigen sich Durchsatzeinbußen von mindestens $\sim 16\,\%$ und bis zu $\sim 54\,\%$ (siehe Tabelle 24). Dabei ist ersichtlich, dass mit steigender Anzahl an Replikaten der Durchsatz bei schreibenden Operationen stärker sinkt als beim lesenden Zugriff. Die Differenz zwischen den Workloads beträgt hierbei je nach Konfiguration maximal $\sim 11\,\%$ bis $\sim 14\,\%$.

	Einfache Replikation	**Zweifache Replikation**	**Dreifache Replikation**
Workload I	-26,78 %	-42,12 %	-53,58 %
Workload U	-29,95 %	-41,62 %	-49,11 %
Workload R	-16,32 %	-31,91 %	-39,22 %
Workload M	-18,30 %	-31,18 %	-40,67 %

Tabelle 24: Cassandra: Replikation - Prozentual sinkender Durchsatz gegenüber der deaktivierten Replikation, wenn eine Antwort abgewartet wird

In Anbetracht des Aspekts, dass lesende Anfragen in dieser Konfiguration im Regelfall lediglich an ein Node gestellt werden, überrascht der sinkende Durchsatz bei Workload R mit zunehmender Replikation. Eine mögliche Ursache hierfür könnte der im Hintergrund stattfindende Read-Repair Prozess sein. Dieser wird standardmäßig mit Wahrscheinlichkeit von 10 % bei der Verarbeitung einer lesenden Anfrage vom Coordinator Node ausgelöst wird.

4.2.9.4. Operationsarten spezifisches Abwarten von Antworten

In Anbetracht der Tatsache, dass in einem typischen Anwendungsfall nicht nur Datensätze ausschließlich gelesen oder geschrieben werden, bietet sich je nach Bedarf

auch eine spezifische Konfiguration für die einzelnen Operationsarten an. Besonderes Interesse gilt hierbei vor allem Konfigurationen die sowohl ein konsistentes Lesen von erfolgreich geschrieben Daten, als auch einen größtmöglichen Durchsatz erzielen.

Für eine beliebige Menge von Replikationen erscheinen hierfür vor allem zwei unterschiedliche Kombinationen sinnvoll. Bei Variante 1 wird von sämtlichen Replika-Nodes gelesen, wogegen beim Schreiben lediglich auf eine einzige Bestätigung gewartet wird. Folglich ist garantiert, dass die stets aktuellste Version eines Datensatzes an den Coordinator Node und somit auch an den anfragenden Client zurückgesendet wird. Demgegenüber wird bei Variante 2 ausschließlich von einem Replika-Node gelesen[91], jedoch beim Schreiben die Bestätigung von sämtlichen Replika-Nodes abgewartet. Hierbei verfügt jeder Replika-Node stets über die aktuellste Version eines Datensatzes, sofern der Schreibvorgang vom Coordinator Node als erfolgreich an den Client gemeldet wird. Demnach kann das Lesen eines Datensatzes auch von einem einzigem beliebigem Replika-Node erfolgen.

Basierend auf den Erkenntnissen aus den vorangegangenen Abschnitten zeigt sich Variante 2 als vermeintlich bessere Wahl in Hinblick auf einen höchstmöglichen Durchsatz. Schließlich konnte in Kapitel 4.2.9.3 festgestellt werden, dass mit zunehmender Anzahl an Replikationen der Durchsatz bei schreibenden Operationen stärker sinkt als beim Lesen von Datensätzen, sofern jeweils nur eine Antwort abgewartet wird. Unter dem Aspekt, dass lesende Operationen eine vielfach höhere Latenz erfordern als schreibende Operationen, ist deren verminderte Beeinträchtigung besonders relevant. In Kapitel 4.2.9.1 konnte dagegen keine eindeutig bessere oder schlechtere Auswirkung bei steigender Anzahl von Replikaten zwischen den lesenden und schreibenden Zugriffen festgestellt werden, sofern auf die Antwort sämtlicher Replika-Nodes gewartet wird. Folglich kann diesbezüglich keine wesentlich bessere oder schlechtere Wahl getroffen werden.

Die nachfolgende Abbildung 73 zeigt wie sich diese Variante (Write All, Read One) im Vergleich zu den bisher betrachteten Konfigurationen bei Workload M verhält. Insgesamt wird deutlich, dass durch die gewählte Kombination bei sämtlichen Replikationsstufen ein nahezu identischer Durchsatz wie beim Abwarten von nur einer Antwort(Write One, Read One) erzielt werden kann. Bemerkenswert ist dieses Ergebnis vor allem, da im Vergleich zum Abwarten von der Mehrheit aller

[91] Ohne Berücksichtigung ggf. erfolgender Read Repair Anfragen.

Antworten (Write Quorum, Read Quorum) ein bis zu doppelt so großer Durchsatz bei vergleichbarem Konsistenzverhalten erreicht wird.

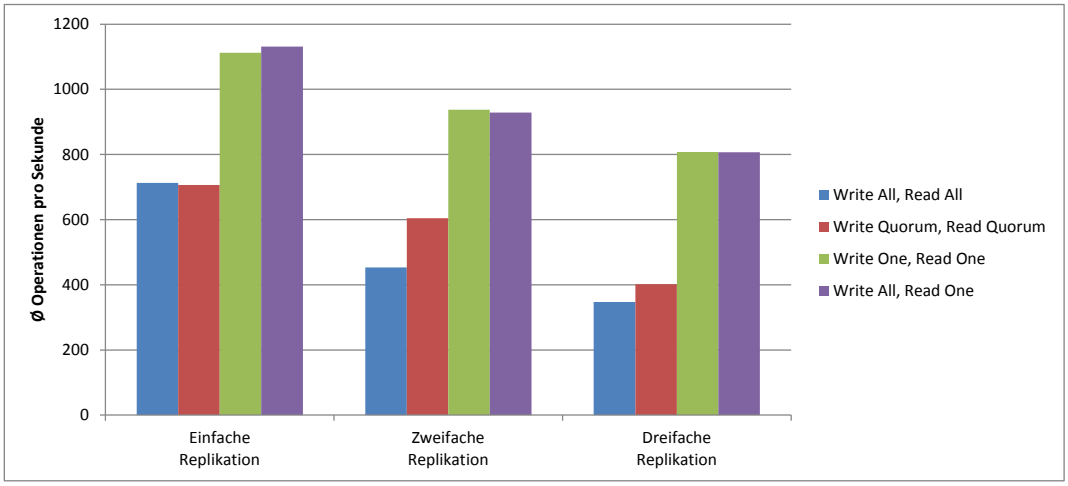


Abbildung 73: Cassandra: Konsistenz-Level im Vergleich (Workload M) - Durchsatz

Eine detaillierte Auflistung des erzielten Durchsatzes im Vergleich mit der Write All, Read One Konfiguration kann der Tabelle 25 entnommen werden.

Konsistenz-Level	**Einfache Replikation**	**Zweifache Replikation**	**Dreifache Replikation**
Write All, Read All	$-36,96\ \%$	$-51,17\ \%$	$-56,93\ \%$
Write Quorum, Read Quorum	$-37,54\ \%$	$-34,92\ \%$	$-50,19\ \%$
Write One, Read One	$-1,63\ \%$	$+0,92\ \%$	$+0,11\ \%$

Tabelle 25: Cassandra: Konsistenz-Level im Durchsatzvergleich zur Write All, Read One Konfiguration bei Workload M

4.2.10. Zusammenfassung der Messergebnisse

In Bezug auf die drei Grundoperationen erfolgt bei Cassandra die Verarbeitung schreibender Anfragen am schnellsten. Erwartungsgemäß erfolgt hierbei die Aktualisierung eines einzelnen Feldes schneller als die Neuanlage eines neuen, vollständigen Datensatzes. Demgegenüber nimmt die Verarbeitung lesender Anfragen die längste Zeit in Anspruch.

Den größten Einfluss auf die Leistungsfähigkeit schreibender Operationen üben die unterschiedlichen Konfigurationen des Write-Ahead Loggings (WAL) aus. Einerseits sinkt durch den Wechsel vom Periodic zum Batch Modus der Durchsatz bei den Workloads I und U um $\sim 97{,}8$ % bzw. $\sim 98{,}9$ %. Andererseits führt die Deaktivierung der WAL Funktionalität gegenüber der Verwendung im Periodic Modus zu einer Durchsatzsteigerung von ~ 61 % bei Workload I und ~ 16 % bei Workload U. Trotz dieser deutlichen Auswirkungen auf schreibende Operationen, bewirken die unterschiedlichen WAL Konfigurationen lediglich geringfügige Unterschiede beim gemischten Workload M.

Weitere deutliche Auswirkungen auf das Leistungsvermögen schreibender Operationen konnten zudem bzgl. der Existienzvalidierung bei Insert-Operationen und der Verwendung von Prepared Statements festgestellt werden. Die optionale Prüfung auf das Vorhandensein eines Keys bei Insert-Operationen führt zu einer Verringerung des Durchsatzes um mehr als 83 %. Dagegen steigert die Verwendung von Prepared Statements den Durchsatz bei Workload I um ~ 51 % und bei Workload U um ~ 57 %. Relevante Auswirkungen auf die Workloads R und M konnten dabei nicht festgestellt werden.

Hinsichtlich der lesenden Operationen wirkt sich die Größe der zugrundeliegenden Datenmenge maßgeblich aus. Durch die Folgen der mehrfachen Verdoppelung der Datenmenge konnte gezeigt werden, dass mit zunehmender Größe der Durchsatz kontinuierlich sinkt. Jede Verdoppelung der Datenmenge führte hierbei zu einer Durchsatzminderung von jeweils $\sim 12{,}5$ bis zu ~ 98 Prozent. Bedeutsame Auswirkungen auf schreibende Operationen konnten in diesem Zusammenhang jedoch nicht festgestellt werden.

Bei der Verwendung zunehmend größerer Datensätze wurde ein kontinuierliches Sinken des Durchsatzes beobachtet. Interessant in diesem Zusammenhang ist vor allem, dass einerseits die Effektivität der standardmäßigen LZ4 Komprimierung

mit ansteigender Datensatzgröße sinkt. Letztendlich führt die LZ4 Implementierung zu einem höheren Speicherbedarf, als ohne Komprimierung erforderlich wäre. Weiterhin steigt in diesem Zusammenhang auch die Anzahl fehlgeschlagener Insert- und Update-Operationen deutlich.

Bei der Evaluierung verschiedener Kompressionsalgorithmen konnte insbesondere festgestellt werden, dass der Deflate Algorithmus die gegebene Datenmenge auf $\sim 54{,}7\,\%$ der ursprünglichen Größe reduziert. Im Vergleich zu den Algorithmen LZ4 ($\sim 83{,}9\,\%$) und Snappy ($\sim 82\,\%$) stellt dieses Ergebnis eine deutliche Verbesserung dar. Gegenüber der standardmäßigen LZ4 Komprimierung erreicht die Deflate Kompression somit letztendlich einen 17 bis 18 Prozent höheren Durchsatz bei den Workloads R und M. Diesbezüglich bleibt jedoch anzumerken, dass die Datensätze ausschließlich mit Zufallsdaten gefüllt wurden, was abgesehen von verschlüsselten Daten eher praxisfernen Datenmustern entspricht.

Beim Vergleich der standardmäßigen Size-Tired Compaction und der Leveled Compaction konnte kein relevanter Unterschied bzgl. der schreibenden Operationen festgestellt werden. Dagegen erreicht die Leveled Compaction bei den Workloads R und M eine ~ 39 %ige Steigerung des Durchsatzes gegenüber der Size-Tired Compaction.

Bezüglich der untersuchten Replikationskonfigurationen bleibt festzuhalten, dass der Durchsatz unabhängig vom Konsistenz-Level zunehmend sinkt, je mehr Replikate existieren. Erwartungsgemäß konnte die stärkste Beeinträchtigung infolge des Abwartens einer Antwort von allen Replika-Nodes festgestellt werden. Die geringste dagegen beim Abwarten einer Antwort von nur einem Node. Durch die Kombination unterschiedlicher Konsistenz-Level für die verschiedenen Operationsarten konnte zudem gezeigt werden, dass für den Workload M einerseits ein konsistentes Lesen garantiert und andererseits ein ebenso hoher Durchsatz, wie beim Abwarten einer Antwort von nur einem Client, erreicht werden kann.

5. Fazit

In diesem Kapitel werden die Ergebnisse der Masterarbeit zusammengefasst. Bedingt durch die bereits in Kapitel 4.1.10 und 4.2.10 erfolgte detaillierte Zusammenfassung der Messergebnisse von Couchbase und Cassandra, beschränkt sich deren Zusammenfassung in diesem Kapitel auf die prägnanten Erkenntnisse. Deutlich umfangreicher werden an dieser Stelle die wichtigsten Erkenntnisse infolge der Verwendung und Modifikation von Thumbtacks YCSB Variante zusammengefasst. Abschließend wird ein Ausblick über potentiell aufbauende Untersuchungsaspekte gegeben.

5.1. Erreichte Ergebnisse

Im Rahmen der Benchmark-Vorbereitung wurden einige sinnvolle Erweiterungen an Thumbtacks YCSB Variante durchgeführt. Hierzu zählen unter anderem neue Datenbankadapter, welche durch die Verwendung aktuellerer SDK Bibliotheken eine Kompatibilität mit den aktuellen Datenbankversionen sicherstellen. Zudem wurde die von Thumbtack implementierte Warmup-Funktionalität überarbeitet, sodass fortan sämtliche Operationsarten unterstützt werden. Des Weiteren wurde die Multi-Client Unterstützung dahingehend erweitert, dass auch bei gemischten Workloads ebenfalls Insert-Operationen verwendet werden können. Die Realisierung einer Benchmark-Automatisierung stellt, für die im Rahmen dieser Arbeit vielzählig durchgeführten Messungen, eine weitere elementare Erweiterung dar. Aufgrund dieser konnte ein Großteil der durchgeführten Messungen vollautomatisiert erfolgen.

Neben der Erweiterung von Thumbtacks YCSB Variante konnten zudem einige grundlegende Fehler identifiziert und behoben werden. Hierzu zählen insbesondere die fehlerbehaftete Protokollierung der Messergebnisse während der Benchmark-Durchführung und die teilweise verfälschte Aggregation von Latenzwerten bei der nachträglichen Durchschnittsbildung. Darüber hinaus erwies sich ebenfalls die Implementierung der Benchmark-Terminierung als problematisch. Trotz einer maximal definierten Laufzeit wurde bei einzelnen Konfigurationen eine variierende Messdauer zwischen den eingesetzten Client-Threads festgestellt. Zur Vermeidung einer diesbezüglich bedingten Ergebnisverfälschung wurde die ursprüngliche Implementierung ersetzt, sodass keine Messwerte mehr nach Ablauf der maximal definierten Laufzeit protokolliert werden. Neben Thumbtacks YCSB Variante ist auch die originale YCSB Version von einigen der identifizierten Fehler betroffen.

Zusätzlich zu den durchgeführten Modifikationen konnten einige Aspekte identifiziert werden, die bei der Verwendung von YCSB berücksichtigt werden sollten. Hierzu zählt insbesondere, dass bei der Berechnung der durchschnittlichen Latenz- und Durchsatzwerte nicht unterschieden wird, ob eine Anfrage erfolgreich oder fehlerhaft abgeschlossen wurde. Infolge der zumeist verkürzten Latenz bei fehlgeschlagenen Operationen kann dieses Verhalten zu einer deutlichen Verfälschung der Messergebnisse führen. Ein weiterer wichtiger Aspekt betrifft das Benchmarking von Datenbanksystemen, welche kein partielles Update von bestehenden Daten unterstützen. Standardmäßig wird bei der Update-Ausführung versucht, lediglich den Wert eines einzelnen Feldes/Attributes zu ersetzten. Wird eine derartige partielle Aktualisierung nicht unterstützt, kann es je nach Implementierung des Datenbankadapters zu ungewollten Ersetzungen des vollständigen Datensatzes kommen. In der Konsequenz bewirkt dieses Verhalten mit voranschreitender Laufzeit einen sinkenden Speicherbedarf, wodurch ggf. ein größerer Anteil der Daten in Cache-Strukturen gehalten werden kann. Insbesondere beim direkten Vergleich unterschiedlicher (NoSQL-)Datenbanksysteme könnte dieses Verhalten zu unzuverlässigen Aussagen führen.

Als übergreifendes Resümee für die durchgeführte Evaluierung von Couchbase kann festgehalten werden, dass insbesondere die vorherrschende In-Memory Fokussierung und der Verzicht auf eine garantierte Dauerhaftigkeit bei schreibender Operationen, ein außerordentliches Leistungsverhalten ermöglicht. Sobald jedoch, bzgl. dieser beiden Aspekte, ein Anwendungsszenario ein abweichendes Verhalten erfordert, bricht der erreichbare Durchsatz an Operationen pro Sekunde essentiell ein.

Bezüglich der erfolgten Evaluierung von Cassandra kann ein ausgeprägtes Leistungsverhalten bei der Verarbeitung schreibender Anfragen festgehalten werden. Erreicht wird dies unter anderem durch die asynchrone Persistierung des Commit Logs und einer standardmäßig unterlassenen Existenzprüfung der betroffenen Keys bei schreibenden Operationen. Beachtliche Leistungseinbußen erfolgen dagegen durch die Anpassung der WAL Konfiguration, sodass die Dauerhaftigkeit erfolgter Datenänderungen garantiert wird. Ebenso deutlich beeinträchtigt die Replikation mit zunehmender Konsistenzanforderung das Leistungsvermögen von Cassandra.

Insgesamt lässt sich aus dieser Arbeit ein positives Fazit ziehen. Entsprechend

der Zielsetzung konnten die unterschiedlichen Auswirkungen infolge verschiedener Konfigurationen für Couchbase und Cassandra erfolgreich herausgearbeitet werden. Mit der Vielzahl an durchgeführten Modifikationen an Thumbtacks YCSB Variante wurden zudem sinnvolle Erweiterungen am Framework vorgenommen und bestehende Schwächen sowohl aufgezeigt als auch behoben.

5.2. Ausblick

Einerseits zeigt die Vielzahl der erfolgten Modifikationen an Thumbtacks YCSB Variante, dass das YCSB Framework noch weitreichendes Optimierungspotential zur Verbesserung und/oder Vereinfachung des Benchmarkings von (NoSQL-)Datenbanksystemen besitzt. Beispielsweise wäre es sinnvoll, die Benchmark-Automatisierung um Funktionalitäten der Fehlerdetektion zu erweitern. Anhand der festgestellten Fehler wird andererseits jedoch auch deutlich, dass das bisherige Benchmarking mit (Thumbtacks) YCSB je nach Parametrisierung zur Verfälschung der Messergebnisse führen kann. Aus diesem Grund sollte zunächst eine separate Prüfung deren konkrete Ausmaße evaluieren und den zugrundeliegenden YCSB Code auf weitere Schwächen untersuchen.

Neben der Evaluierung zusätzlicher NoSQL-Datenbanksysteme könnten weiterführende Arbeiten die hier gewonnen Erkenntnisse vertiefen. Diese liegen z.B. in der weiteren Untersuchung der aufgetretenen Fragestellungen. Hierzu zählt unter anderem, inwiefern sich die gewonnen Ergebnisse bzgl. der Compaction bei einer Langzeituntersuchung beider Systeme verändern. Des Weiteren bleibt bspw. für Cassandra zu untersuchen, welchen Einfluss der verwendete Sekundärspeicher auf die WAL Batch Konfiguration ausübt. In diesem Zusammenhang wären die Auswirkungen durch den Einsatz von SSD Speichermedien auf schreibende Anfragen ein interessanter Aspekt.

Weiterhin könnte untersucht werden, inwieweit sich verschiedene Konfigurationskombinationen beeinflussen. Eine interessante Fragestellung wäre beispielsweise, ob bei Cassandra die Deflate Komprimierung genauso effektiv bleibt, wenn statt der standardmäßigen Size-tiered Compaction die Leveled Compaction Strategie verwendet wird. Bedingt durch die Vielzahl an möglichen Konfigurationsparametern aus den Bereichen Hardware, Betriebssystem, Datenbanksystem und YCSB bieten sich hierbei potentiell beliebig viele Kombinationen.

A. Anhang

A.1. Modifikationen an Thumbtacks YCSB Variante

A.1.1. Protokollauszug zu einem Fehler beim Erzeugen eines YCSB-Client-Jobs

Der nachfolgende Protokollauszug zeigt beispielhaft eine Fehlermeldung entsprechend der in Kapitel 3.2.2.5 beschriebenen Fehlersymptomatik.

```
1  ...
2  [2014-12-08 23:58:22] fab -f fabfile/ ycsb_load:db=couchbase2
3  [redfort.fbi.h-da.de] Executing task 'ycsb_load'
4  [1;32m2014-12-09 00:00:00+01:00[0m
5  [redfort.fbi.h-da.de] run: echo "/opt/ycsb/bin/ycsb load
       couchbase2 -s -p couchbase.user= -p timeseries.granularity=100
       -p readretrycount=1000 -p couchbase.checkOperationStatus=true -
       p couchbase.views= -p fieldlength=10 -p reconnectiontime=1000 -
       p couchbase.ddocs= -p warmupexecutiontime=300000 -p couchbase.
       hosts=gulltown.fbi.h-da.de,runestone.fbi.h-da.de,dreadfort.fbi.
       h-da.de,karhold.fbi.h-da.de -p operationcount=2147483647 -p
       reconnectionthroughput=10 -p recordcount=10000000 -p couchbase.
       opTimeout=86400000 -p fieldcount=10 -p threadcount=32 -p
       maxexecutiontime=5999940 -p couchbase.password= -p
       measurementtype=timeseries -p exportmeasurementsinterval=30000
       -p workload=com.yahoo.ycsb.workloads.CoreWorkload -p
       updateretrycount=1000 -p couchbase.bucket=default -p
       ignoreinserterrors=false -p insertretrycount=1000 -p
       fieldnameprefix=f -p insertstart=0 -p insertcount=10000000 > /
       run/shm/2014-12-09_00-00_couchbase_load.out 2> /run/shm
       /2014-12-09_00-00_couchbase_load.err" | at 00:00 today
6  [redfort.fbi.h-da.de] out: at: refusing to create job destined in
       the past
7
8  Disconnecting from redfort.fbi.h-da.de... done.
9  [2014-12-08 23:58:22] Auf YCSB-Clients warten
10 ...
```

Listing 8: Benchmark-Vorbereitung: Protokollauszug bzgl. fehlschlagender YCSB Client Ausführungen

A.1.2. Probleme mit Couchbase Java SDK 2.0.1

Im Rahmen verschiedener Tests konnte nach der Erhöhung des Client-seitigen Timeouts, konnte unter Verwendung des Couchbase Java SDKs 2.0.1 ein zunehmend steigender Speicherbedarf beobachtet werden. Schlussendlich führt dieser zu einer OutOfMemory-Exception und somit zum Abbruch der gesamten Applikation.

Bedingt durch die hohen Ressourcenkapazitäten der Cluster Nodes tritt der Fehler erst mit fortgeschrittener Laufzeit auf. Zur schnelleren Reproduktion des Fehlers wurden die nachfolgenden Untersuchungen, außerhalb des Clusters, isoliert auf einem Notebook, durchgeführt. Hierfür wurde ein separates Java Projekt ohne YCSB Bezug angelegt, wodurch die Fehlerquelle eindeutig auf das Couchbase Client SDK reduziert werden konnte.

Anhand der Abbildung 74 lässt sich deutlich der erhöhte Speicherverbrauch bei zunehmender Anzahl durchgeführter Operationen erkennen. Der Vergleich erfolgt hierbei bzgl. des standardmäßigen Timeouts von 2,5 Sekunden[92] und einem beispielhaft gewähltem Timeout von 600 Sekunden.

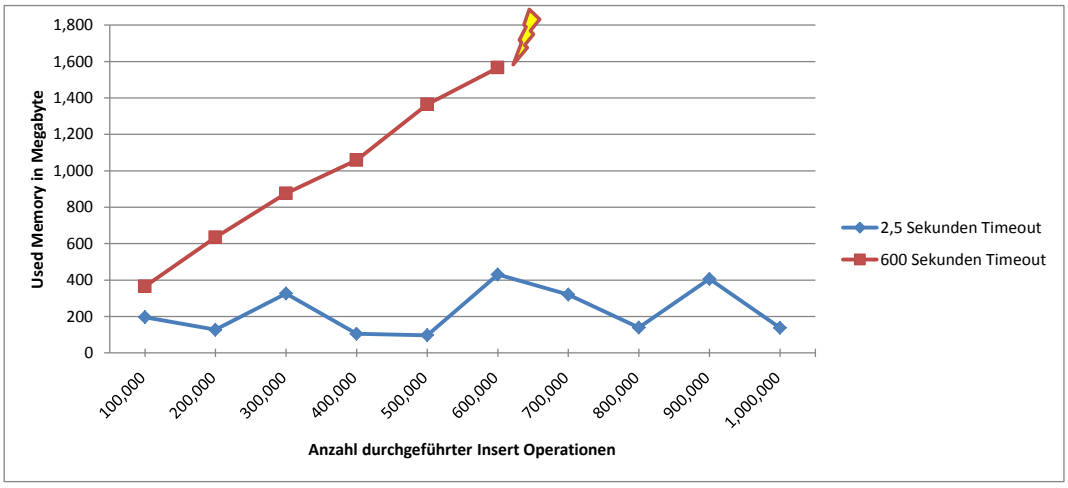


Abbildung 74: Couchbase: Speicherbedarf bei unterschiedlichen Timeout-Werten

Während die Konfiguration mit dem 2,5 Sekunden Timeout einen verhältnismäßig gleichbleibenden Speicherbedarf besitzt, steigt der Speicherbedarf unter Verwendung des 600 Sekunden Timeouts fast linear bis zum Auftreten der OutOfMemory-Exception. Diesbezüglich gilt zu beachten, dass die Operationen von einem einzelnen Thread synchron ausgeführt wurden, sodass vom Prinzip her kein Anwachsen

[92] Gilt für Couchbase Java SDK $\geq$ 2.0. Dagegen besitzt die SDK Version 1.5.4 einen standardmäßigen Timeout von 5 Sekunden.

des Speichers erwartet werden würde. Zudem bleibt anzumerken, dass der Timeout von 600 Sekunden niemals erreicht wurde.

Für dieses Problem wurde ein Ticket[93] bei Couchbase eröffnet, welches vom Couchbase eigenen SDK Entwickler Michael Nitschinger als Fehler bestätigt wurde[94]. Aufgrund der vorherrschenden Kompatibilität zwischen der derzeitig aktuellsten Java Client SDK Version des letzten Major-Releases (1.4.5) und dem verwendetem Couchbase Server 3.0.1 [Ing14], wird auf diese zurückgegriffen werden.

A.2. Benchmark-Ablauf

A.2.1. Überprüfung der Betriebsbereitschaft von Couchbase

```
1  @roles('all_server')
2  @parallel
3  def couchbase_waitForReadyState():
4          """Server alle 30 Sekunden kontrollieren, ob bereit fuer
               naechsten Workload"""
5          script = """while [ 1 ]; do /opt/couchbase/bin/cbstats
               127.0.0.1:11210 all | awk '
6                    /ep_flusher_todo/ { sum=sum+$2 }        #
                         DiskWriteQueue
7                    /ep_queue_size/ { sum=sum+$2 }          #
                         DiskWriteQueue
8                    /ep_pending_compactions/ { sum=sum+$2 } #Compaction
9                    /vb_replica_queue_size/ { sum=sum+$2 }  #Replication
10                   END {if (sum == 0) { exit 0} else {exit 1}}
11                   '; if [ $? -eq 0 ]; then break; fi; sleep 30; done"""
12         run(script)
```

Listing 9: Benchmark-Ablauf: Skript zur Überprüfung der Betriebsbereitschaft von Couchbase

[93] `https://www.couchbase.com/issues/browse/JCBC-639`, zuletzt besucht am 02.12.2014.

[94] Für weitere Informationen siehe `https://github.com/ReactiveX/RxJava/issues/1919`, zuletzt besucht am 02.12.2014.

A.2.2. Überprüfung der Betriebsbereitschaft von Cassandra

```
1  @roles('all_server')
2  @parallel
3  def cassandra_waitForReadyState():
4          script = """while [ 1 ]; do
5                  cassandra_ready=0
6                  nodetool status > /dev/null; if [ $? -eq 0 ]; then cassandra_ready
                       =$((cassandra_ready+1)); fi;
7                  nodetool status | awk ' { if($1 == "UN"){counter=counter+1;}} END
                       {if (counter == %s) { exit 0} else {exit 1}}' ; if [ $? -eq 0
                       ]; then cassandra_ready=$((cassandra_ready+1)); fi;
8                  nodetool compactionstats | awk -F':' '
9                          /Active compaction remaining time / { sum=sum+1 }
10                         END {if (sum == 0) { exit 0} else {exit 1}}
11                         '; if [ $? -eq 0 ]; then cassandra_ready=$((
                               cassandra_ready+1)); fi;
12                 nodetool tpstats | awk -F':' '
13                         /CounterMutationStage/    { sum=sum+$2+$3+$5 }
14                         /ReadStage/               { sum=sum+$2+$3+$5 }
15                         /RequestResponseStage/    { sum=sum+$2+$3+$5 }
16                         /MutationStage/           { sum=sum+$2+$3+$5 }
17                         /ReadRepairStage/         { sum=sum+$2+$3+$5 }
18                         /GossipStage/             { sum=sum+$2+$3+$5 }
19                         /MigrationStage/          { sum=sum+$2+$3+$5 }
20                         /ValidationExecutor/      { sum=sum+$2+$3+$5 }
21                         /MemtableReclaimMemory/   { sum=sum+$2+$3+$5 }
22                         /InternalResponseStage/   { sum=sum+$2+$3+$5 }
23                         /AntiEntropyStage/        { sum=sum+$2+$3+$5 }
24                         /MiscStage/               { sum=sum+$2+$3+$5 }
25                         /CommitLogArchiver/       { sum=sum+$2+$3+$5 }
26                         /MemtableFlushWriter/     { sum=sum+$2+$3+$5 }
27                         /PendingRangeCalculator/ { sum=sum+$2+$3+$5 }
28                         /MemtablePostFlush/       { sum=sum+$2+$3+$5 }
29                         /CompactionExecutor/      { sum=sum+$2+$3+$5 }
30                         /AntiEntropySessions/     { sum=sum+$2+$3+$5 }
31                         /HintedHandoff/           { sum=sum+$2+$3+$5 }
32                         END {if (sum == 0) { exit 0} else {exit 1}}
33                         '; if [ $? -eq 0 ]; then cassandra_ready=$((
                               cassandra_ready+1)); fi;
34                 nodetool netstats | awk '
35                         /Commands/ { sum=sum+$3 }
36                         /Responses/ { sum=sum+$3 }
37                         END {if (sum == 0) { exit 0} else {exit 1}}
38                         '; if [ $? -eq 0 ]; then cassandra_ready=$((
                               cassandra_ready+1)); fi;
39                 if [ $cassandra_ready -eq 5 ]; then echo "ende";break; fi;
40                 sleep 60; done""" % len(env.roledefs['server'])
41         run(script)
```

Listing 10: Benchmark-Ablauf: Skript zur Überprüfung der Betriebsbereitschaft von Cassandra

Vornehmlich nach dem Neustart der Cassandra Instanzen verursacht dieses Skript in seltenen Fällen ein ordnungsgemäßes Blockieren des weiteren Benchmark-Ablaufs. Ursache hierfür ist stets die ausbleibende Verarbeitung von drei ausstehenden Netzwerkbefehlen. Zur Behebung dieser Blockade kann der auf dem Node laufende DataStax Agent neu gestartet werden, woraufhin die Verarbeitung augenscheinlich fortgesetzt wird. Inwiefern es sich beim DataStax Agent auch um den Auslöser der Symptomatik handelt, konnte jedoch nicht zweifelsfrei ermittelt werden.

A.3. Benchmark-Durchführung

Diese Kapitel enthält eine Übersicht der gemessenen Werte. Für eine bessere Lesbarkeit wurden die nachfolgenden Durchsatzwerte auf ganze Zahlen und die Latenzwerte auf drei Nachkommastellen gerundet. Die originalen Messwerte können den Protokollen auf der beigefügten DVD entnommen werden.

Ebenfalls enthalten sind in diesem Kapitel, die noch ausstehenden Latenz-Diagramme.

A.3.1. Messwerte zu Couchbase

A.3.1.1. Festlegung einer geeigneten Thread-Anzahl pro Client Node

Threads	Workload I	Workload U	Workload R	Workload M
1	4.601	4.576	5.289	4.823
2	9.035	8.982	10.666	9.387
4	16.420	16.227	18.365	16.710
8	29.713	29.258	35.439	31.134
16	63.260	63.148	63.882	61.555
32	90.519	92.149	93.045	88.179
64	98.422	97.545	99.890	95.652
128	96.801	96.386	96.990	93.669

Tabelle 26: Couchbase: Threads pro Client - Durchsatz in Operationen pro Sekunde

Threads	Workload I	Workload U	Workload R	Workload M
1	0,211	0,212	0,185	0,201
2	0,214	0,215	0,183	0,207
4	0,235	0,238	0,213	0,232
8	0,261	0,265	0,221	0,250
16	0,246	0,247	0,246	0,254
32	0,348	0,342	0,341	0,358
64	0,645	0,651	0,637	0,664
128	1,316	1,322	1,316	1,361

Tabelle 27: Couchbase: Threads pro Client - Latenz in Millisekunden

Threads	Workload I	Workload U	Workload R	Workload M
16 (1x16)	63.260	63.148	63.882	61.555
16 (2x8)	58.761	57.511	66.496	60.028
16 (4x4)	63.984	64.065	71.784	66.539
32 (1x32)	90.519	92.149	93.045	88.179
32 (2x16)	124.014	123.425	126.755	121.460
32 (4x8)	114.507	113.247	128.104	117.862
64 (1x64)	98.422	97.545	99.890	95.652
64 (2x32)	172.569	178.230	182.505	173.737
64 (4x16)	224.098	239.087	253.629	239.879

Tabelle 28: Couchbase: Client x Thread Kombinationen - Durchsatz in Operationen pro Sekunde

Threads	Workload I	Workload U	Workload R	Workload M
16 (1x16)	0,246	0,247	0,246	0,254
16 (2x8)	0,264	0,270	0,236	0,259
16 (4x4)	0,242	0,241	0,218	0,233
32 (1x32)	0,348	0,342	0,341	0,358
32 (2x16)	0,251	0,252	0,248	0,258
32 (4x8)	0,271	0,274	0,245	0,264
64 (1x64)	0,645	0,651	0,637	0,664
64 (2x32)	0,365	0,354	0,347	0,364
64 (4x16)	0,279	0,261	0,248	0,261

Tabelle 29: Couchbase: Client x Thread Kombinationen - Latenz in Millisekunden

A.3.1.2. Default-Konfiguration

Workload I	Workload U	Workload R	Workload M
223.178	227.363	251.547	236.823

Tabelle 30: Couchbase: Default-Konfiguration - Durchsatz in Operationen pro Sekunde

Workload I	Workload U	Workload R	Workload M
0,280	0,275	0,250	0,264

Tabelle 31: Couchbase: Default-Konfiguration - Latenz in Millisekunden

A.3.1.3. Couchbase Java Client SDK

Methodenaufruf	**Workload I**	**Workload U**	**Workload M**
operation()	223.178	227.363	236.823
operation(replicateTo=Zero)	224.522	227.526	236.889
operation(persistTo=Zero)	223.816	229.616	236.917
operation(persistTo=Zero, replicateTo=Zero)	221.877	227.303	237.136

Tabelle 32: Couchbase: Java Client SDK 1.4.5 - Durchsatz in Operationen pro Sekunde

Methodenaufruf	**Workload I**	**Workload U**	**Workload M**
operation()	0,280	0,275	0,264
operation(replicateTo=Zero)	0,278	0,274	0,264
operation(persistTo=Zero)	0,279	0,272	0,264
operation(persistTo=Zero, replicateTo=Zero)	0,281	0,275	0,264

Tabelle 33: Couchbase: Java Client SDK 1.4.5 - Latenz in Millisekunden

A.3.1.4. Unterschiedliche Dokumentengrößen

Faktor	Workload I	Workload U	Workload R	Workload M
1	223.178	227.363	251.547	236.823
10	129.120	131.809	180.364	157.702
100	23.798	23.708	24.762	35.715
1.000	2.379	2.399	2.490	3.971
10.000	241	241	247	384
100.000	24	24	24	37

Tabelle 34: Couchbase: Unterschiedliche Dokumentengrößen - Durchsatz in Operationen pro Sekunde

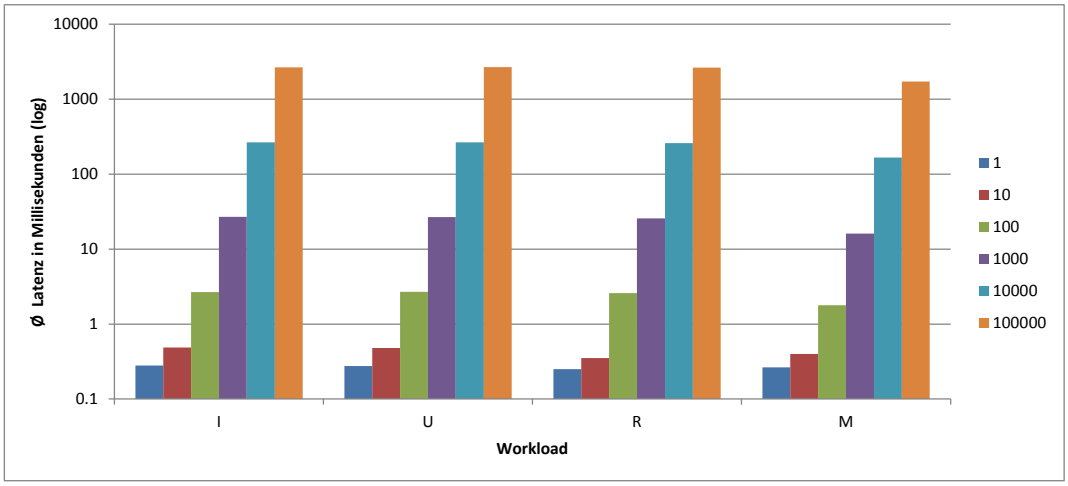


Abbildung 75: Couchbase: Unterschiedliche Dokumentengrößen - Latenz

Faktor	Workload I	Workload U	Workload R	Workload M
1	0,280	0,275	0,250	0,264
10	0,487	0,477	0,351	0,400
100	2,680	2,691	2,578	1,784
1.000	26,879	26,659	25,693	16,103
10.000	265,010	265,512	259,238	166,344
100.000	2657,787	2689,107	2641,652	1722,335

Tabelle 35: Couchbase: Unterschiedliche Dokumentengrößen - Latenz in Millisekunden

A.3.1.5. Unterschiedliche initiale Dokumentenmengen

Dokumenten-menge	Workload I	Workload U	Workload R	Workload M
10 Mio.	224.098	239.087	253.629	239.879
30 Mio.	217.246	233.392	252.875	236.449
60 Mio.	223.178	227.363	251.547	236.823
90 Mio.	210.608	219.339	252.799	237.309
120 Mio.	206.748	221.746	253.024	236.253

Tabelle 36: Couchbase: Unterschiedliche Dokumentenmengen - Durchsatz in Operationen pro Sekunde

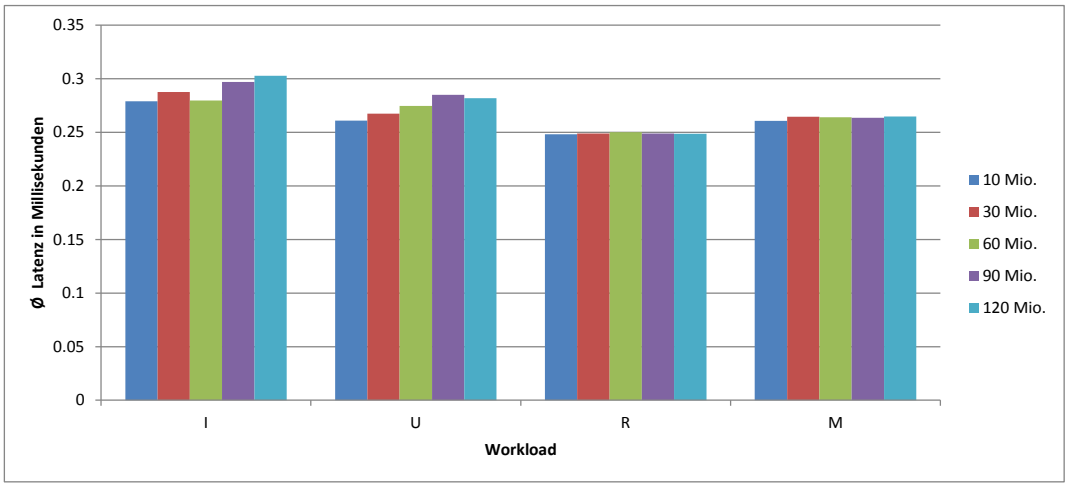


Abbildung 76: Couchbase: Unterschiedliche Dokumentenmengen - Latenz

Dokumenten-menge	Workload I	Workload U	Workload R	Workload M
10 Mio.	0,279	0,261	0,248	0,261
30 Mio.	0,288	0,268	0,249	0,265
60 Mio.	0,280	0,275	0,250	0,264
90 Mio.	0,297	0,285	0,249	0,264
120 Mio.	0,303	0,282	0,249	0,265

Tabelle 37: Couchbase: Unterschiedliche Dokumentenmengen - Latenz in Millisekunden

A.3.1.6. Auto-Compaction

Methoden-aufruf	Workload I	Workload U	Workload M
Deaktiviert	232.246	236.324	238.712
60 %	232.220	230.354	238.851
30 %	223.178	227.363	236.823
15 %	222.798	228.468	236.562
10 %	218.724	228.767	235.169
5 %	220.146	225.035	236.520
2 %	213.266	226.560	236.078

Tabelle 38: Couchbase: Compaction - Durchsatz in Operationen pro Sekunde

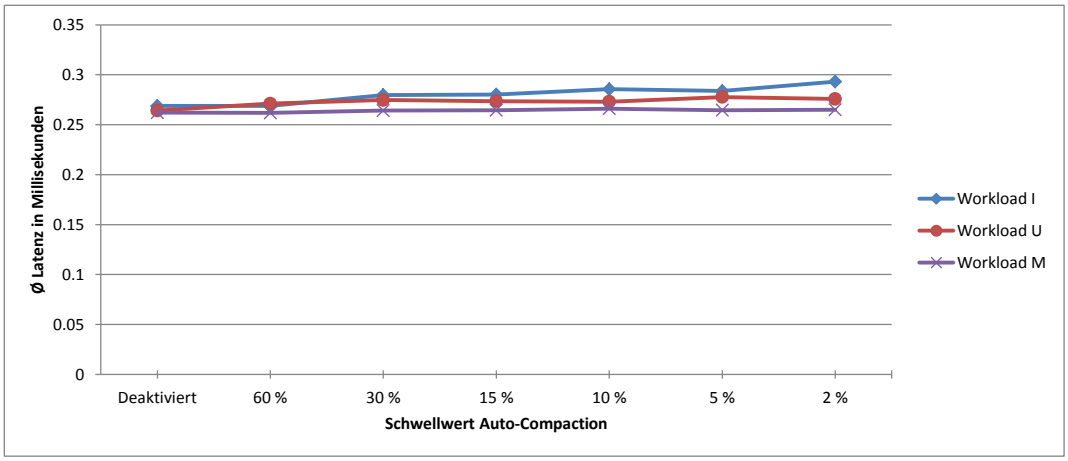


Abbildung 77: Couchbase: Compaction - Latenz

Methoden-aufruf	Workload I	Workload U	Workload M
Deaktiviert	0,269	0,264	0,262
60 %	0,269	0,271	0,262
30 %	0,280	0,275	0,264
15 %	0,280	0,273	0,265
10 %	0,286	0,273	0,266
5 %	0,284	0,278	0,265
2 %	0,293	0,276	0,265

Tabelle 39: Couchbase: Compaction - Latenz in Millisekunden

A.3.1.7. Couchbase Bucket vs. Memcached Bucket

Buckettyp	Workload I	Workload U	Workload R	Workload M
Couchbase Bucket	223.178	227.363	251.547	236.823
Memcached Bucket	225.895	211.378	251.630	236.730

Tabelle 40: Couchbase: Couchbase Bucket vs. Memcached Bucket - Durchsatz in Operationen pro Sekunde

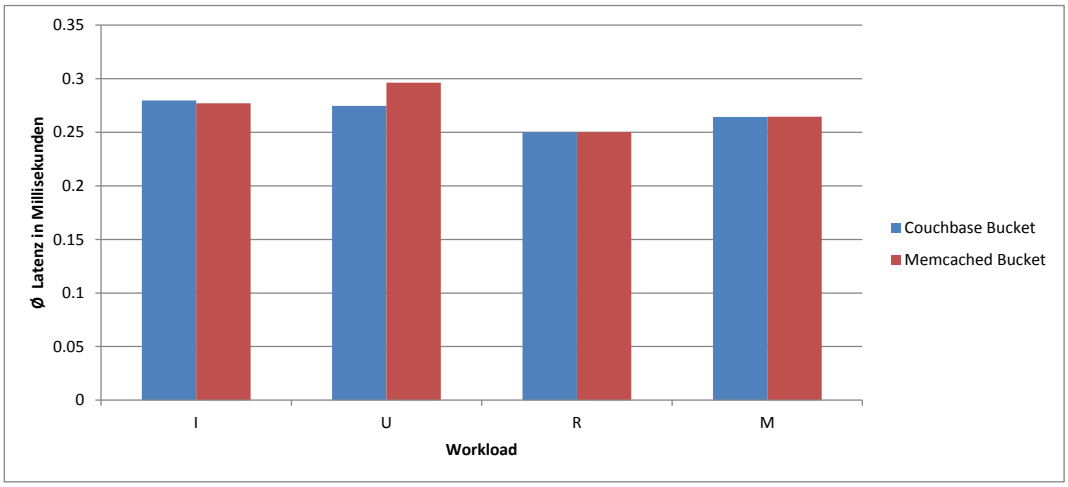


Abbildung 78: Couchbase: Couchbase Bucket vs. Memcached Bucket - Latenz

Buckettyp	Workload I	Workload U	Workload R	Workload M
Couchbase Bucket	0,280	0,275	0,250	0,264
Memcached Bucket	0,277	0,296	0,250	0,264

Tabelle 41: Couchbase: Couchbase Bucket vs. Memcached Bucket - Latenz in Millisekunden

A.3.1.8. Cache Eviction

Dokumenten-menge	**Workload I**	**Workload U**	**Workload R**	**Workload M**
Faktor 0,25 (Value Eject.)	229.664	226.108	252.912	237.420
Faktor 1,25 (Value Eject.)	172.941	210.614	867	696
Faktor 1,5 (Value Eject.)	194.593	205.362	388	466
Faktor 0,25 (Full Eject.)	2.524	228.658	252.200	71.533

Tabelle 42: Couchbase: Eviction - Durchsatz in Operationen pro Sekunde

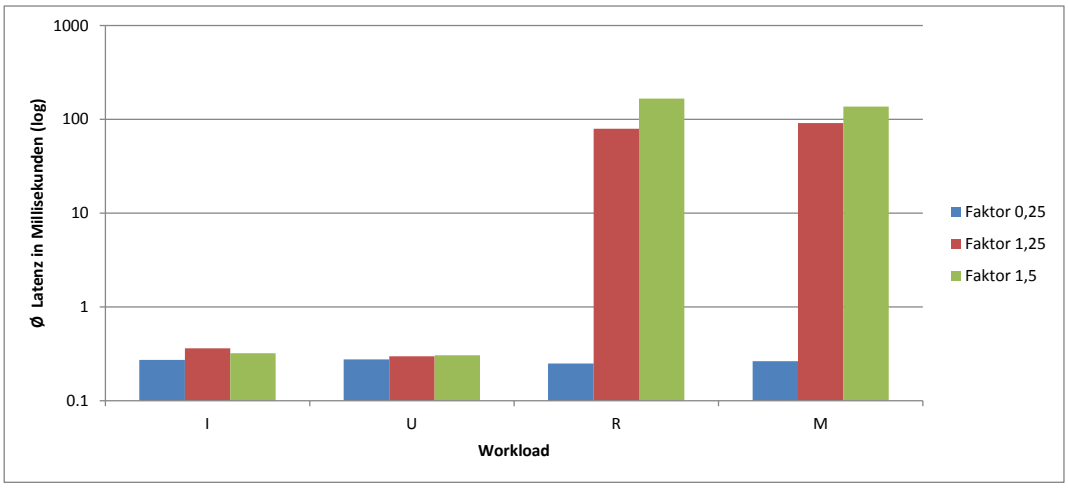


Abbildung 79: Couchbase: Value Eviction bei unterschiedlichen initial geladenen Dokumentenmengen - Latenz

Dokumenten-menge	**Workload I**	**Workload U**	**Workload R**	**Workload M**
Faktor 0,25 (Value Eject.)	0,272	0,276	0,249	0,264
Faktor 1,25 (Value Eject.)	0,363	0,297	79,478	91,636
Faktor 1,5 (Value Eject.)	0,321	0,304	166,958	136,644
Faktor 0,25 (Full Eject.)	35,330	0,273	0,250	0,887

Tabelle 43: Couchbase: Eviction - Latenz in Millisekunden

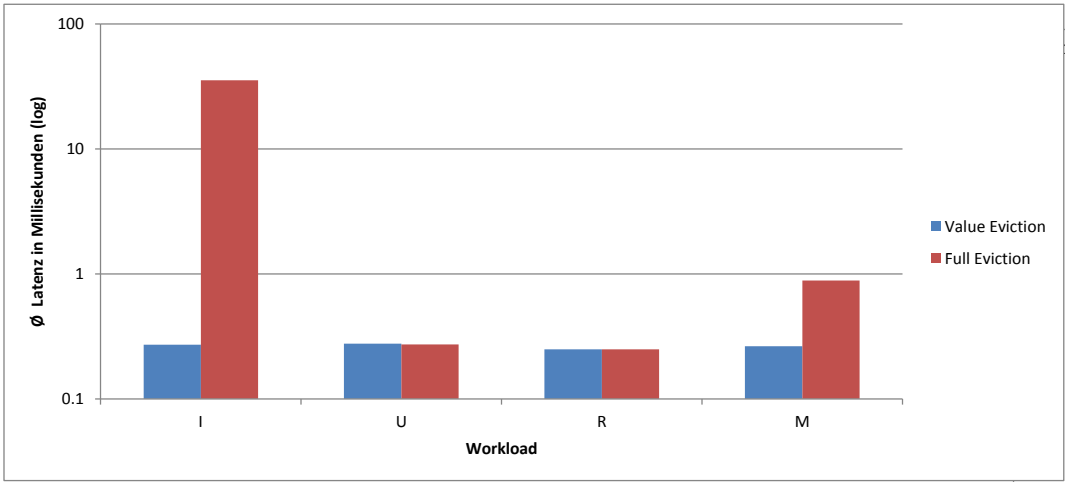


Abbildung 80: Couchbase: Eviction Strategien im Vergleich bei einer initialen Dokumentenmenge für $\sim$ 1 Node - Latenz

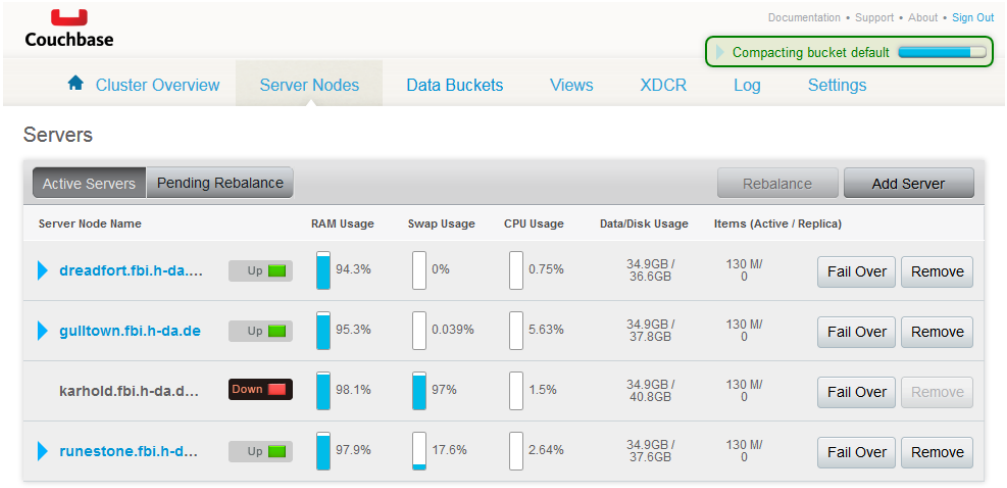


Abbildung 81: Couchbase: Serverabsturz I

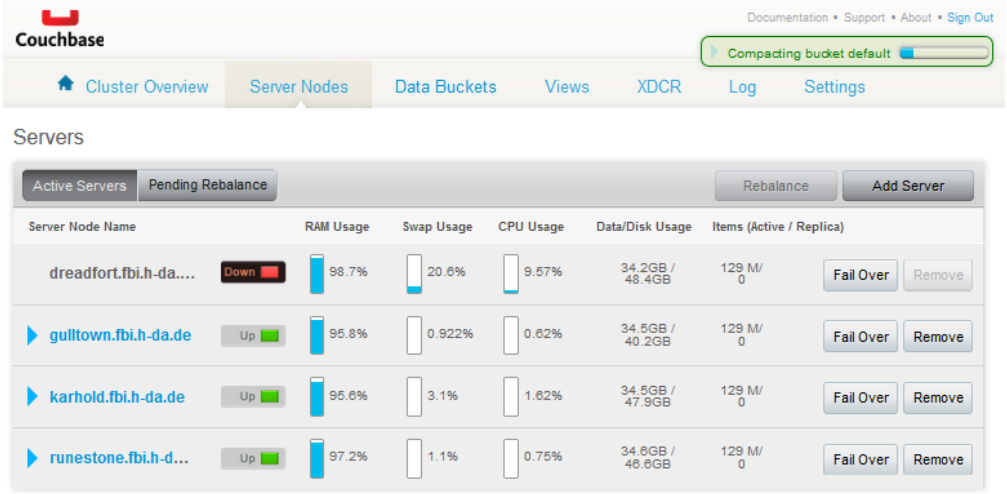


Abbildung 82: Couchbase: Serverabsturz II

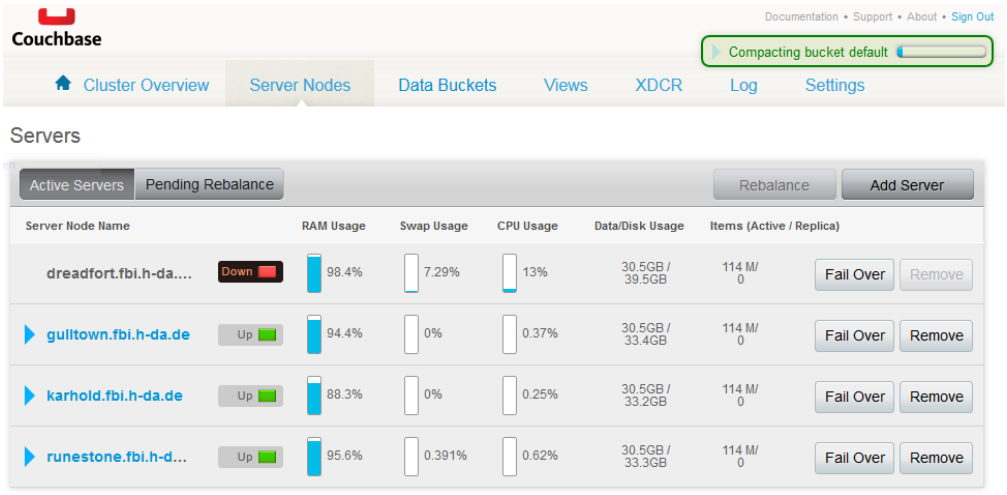


Abbildung 83: Couchbase: Serverabsturz III

A.3.1.9. Asynchrone Replikation

	Workload I	Workload U	Workload R	Workload M
Keine Replikation	223.178	227.363	251.547	236.823
Einfache Replikation	153.843	209.200	253.285	233.256
Zweifache Replikation	182.492	223.756	252.256	199.637
Dreifache Replikation	199.509	204.228	2.676	392

Tabelle 44: Couchbase: Asynchrone Replikation - Durchsatz in Operationen pro Sekunde

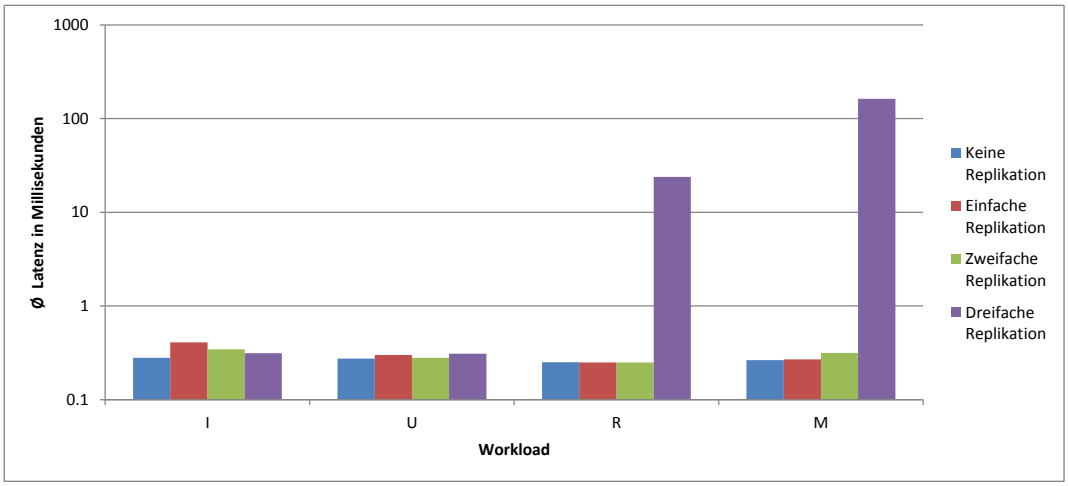


Abbildung 84: Couchbase: Asynchrone Replikation - Latenz

	Workload I	Workload U	Workload R	Workload M
Keine Replikation	0,280	0,275	0,250	0,264
Einfache Replikation	0,409	0,299	0,248	0,268
Zweifache Replikation	0,344	0,280	0,249	0,315
Dreifache Replikation	0,314	0,308	23,892	162,819

Tabelle 45: Couchbase: Asynchrone Replikation - Latenz in Millisekunden

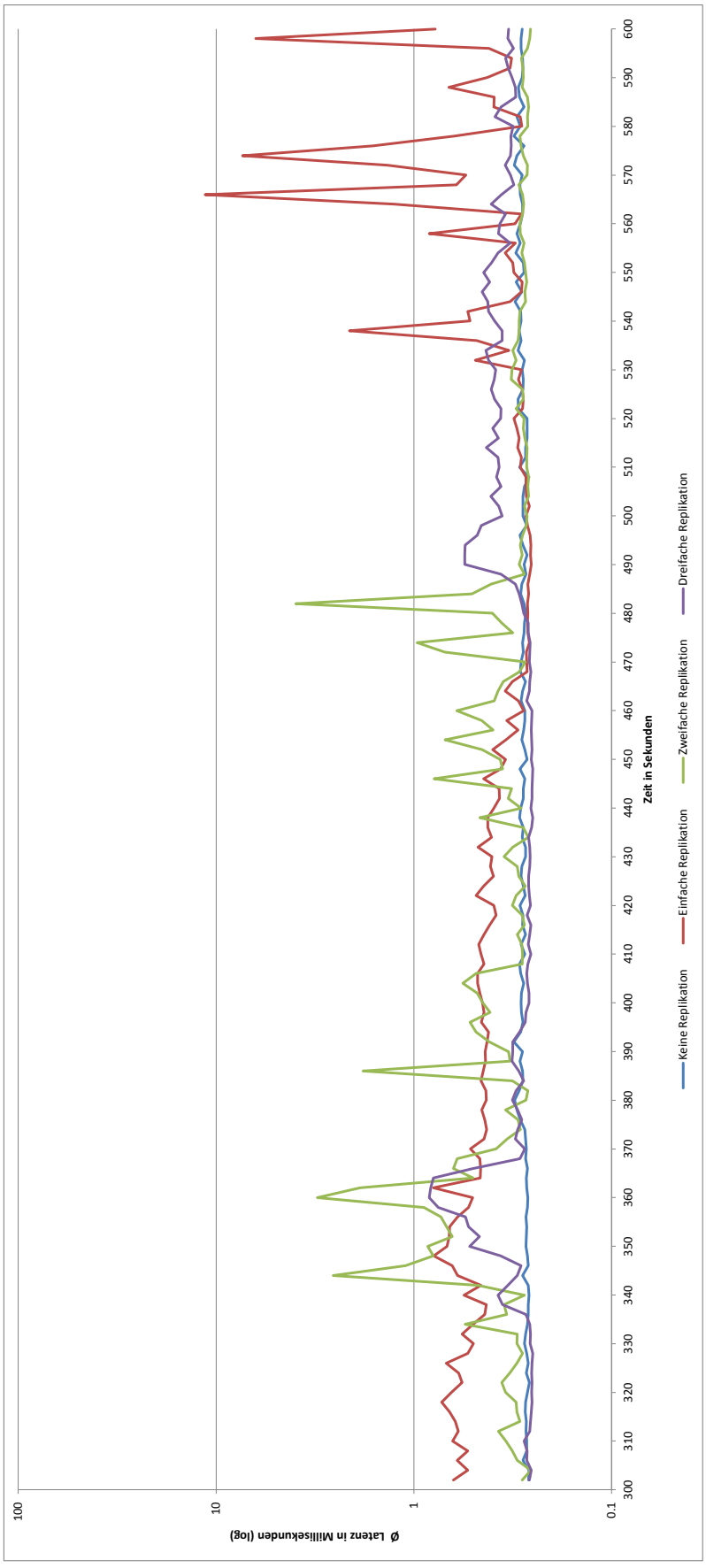


Abbildung 85: Couchbase: Asynchrone Replikation - Latenzverlauf (Inserts)

A.3.1.10. Bestätigung schreibender Operationen

	Workload I	**Workload U**
Keine Bestätigung	223.178	227.363
Eine Bestätigung	6.449	6.174
Zwei Bestätigungen	5.689	5.601
Drei Bestätigungen	5.404	5.437

Tabelle 46: Couchbase: Bestätigung nach Replikation - Durchsatz in Operationen pro Sekunde

	Workload I	**Workload U**
Keine Bestätigung	0,280	0,275
Eine Bestätigung	9,912	9,773
Zwei Bestätigungen	11,207	10,856
Drei Bestätigungen	11,835	11,448

Tabelle 47: Couchbase: Bestätigung nach Replikation - Latenz in Millisekunden

	Workload I	**Workload U**
Keine Bestätigung	223.178	227.363
Eine Bestätigung	74	59
Zwei Bestätigungen	32	23
Drei Bestätigungen	18	13
Vier Bestätigungen	15	10

Tabelle 48: Couchbase: Bestätigung nach Persistierung - Durchsatz in Operationen pro Sekunde

	Workload I	**Workload U**
Keine Bestätigung	0,280	0,275
Eine Bestätigung	855,931	1,072,855
Zwei Bestätigungen	1.970,152	2.671,510
Drei Bestätigungen	3.443,731	4.668,482
Vier Bestätigungen	4.113,983	5.990,800

Tabelle 49: Couchbase: Bestätigung nach Persistierung - Latenz in Millisekunden

A.3.2. Messwerte zu Cassandra

A.3.2.1. Festlegung einer geeigneten Thread-Anzahl pro Client Node

Threads	Workload I	Workload U	Workload R	Workload M
1	8.441	11.015	376	694
2	14.722	21.334	575	1.106
4	25.319	39.275	645	1.253
8	39.500	67.350	675	1.405
16	48.667	103.910	709	1.362
32	55.250	131.961	643	1.381
64	61.713	154.262	702	1.339
128	68.000	183.734	677	1.370
256	73.399	191.624	684	1.337
512	81.815	193.758	707	1.364
1024	85.677	192.630	747	1.600

Tabelle 50: Cassandra: Threads pro Client - Durchsatz in Operationen pro Sekunde

Threads	Workload I	Workload U	Workload R	Workload M
1	0,466	0,357	10,632	5,748
2	0,534	0,368	13,902	7,220
4	0,622	0,400	24,775	12,754
8	0,801	0,467	47,376	22,753
16	1,304	0,605	90,206	46,941
32	2,303	0,957	198,806	92,482
64	4,117	1,627	363,428	190,641
128	7,468	2,707	749,701	371,626
256	13,771	4,804	1467,720	758,518
512	24,398	9,043	2804,545	1478,702
1024	46,870	18,059	5227,886	2491,931

Tabelle 51: Cassandra: Threads pro Client - Latenz in Millisekunden

A.3.2.2. Default-Konfiguration

Workload I	**Workload U**	**Workload R**	**Workload M**
48.667	103.910	709	1.362

Tabelle 52: Cassandra: Default-Konfiguration - Durchsatz in Operationen pro Sekunde

Workload I	**Workload U**	**Workload R**	**Workload M**
1,304	0,605	90,206	46,941

Tabelle 53: Cassandra: Default-Konfiguration - Latenz in Millisekunden

A.3.2.3. Existenzvalidierung bei Insert-Operationen

Existenzvalidierung	**Workload I**	**Workload M**
Keine Prüfung	48.667	1.362
IF NOT EXISTS	8.269	1.187

Tabelle 54: Cassandra: Existenzvalidierung bei Insert-Operationen - Durchsatz in Operationen pro Sekunde

Existenzvalidierung	**Workload I**	**Workload M**
Keine Prüfung	1,304	46,941
IF NOT EXISTS	7,780	53,872

Tabelle 55: Cassandra: Existenzvalidierung bei Insert-Operationen - Latenz in Millisekunden

A.3.2.4. Prepared Statements

	Workload I	**Workload U**	**Workload R**	**Workload M**
Prepared Statements	48.667	103.910	709	1.362
Keine Prepared Statements	32.248	66.309	703	1.387

Tabelle 56: Cassandra: Prepared Statements - Durchsatz in Operationen pro Sekunde

	Workload I	**Workload U**	**Workload R**	**Workload M**
Prepared Statements	1,304	0,605	90,206	46,941
Keine Prepared Statements	1,974	0,957	90,914	46,109

Tabelle 57: Cassandra: Prepared Statements - Latenz in Millisekunden

A.3.2.5. Unterschiedliche Datensatzgrößen

Faktor	**Workload I**	**Workload U**	**Workload R**	**Workload M**
1	48.667	103.910	709	1.362
10	11.953	62.878	950	1.794
100	1.431	12.763	687	852
1.000	187	1.824	275	198

Tabelle 58: Cassandra: Unterschiedliche Datensatzgrößen - Durchsatz in Operationen pro Sekunde

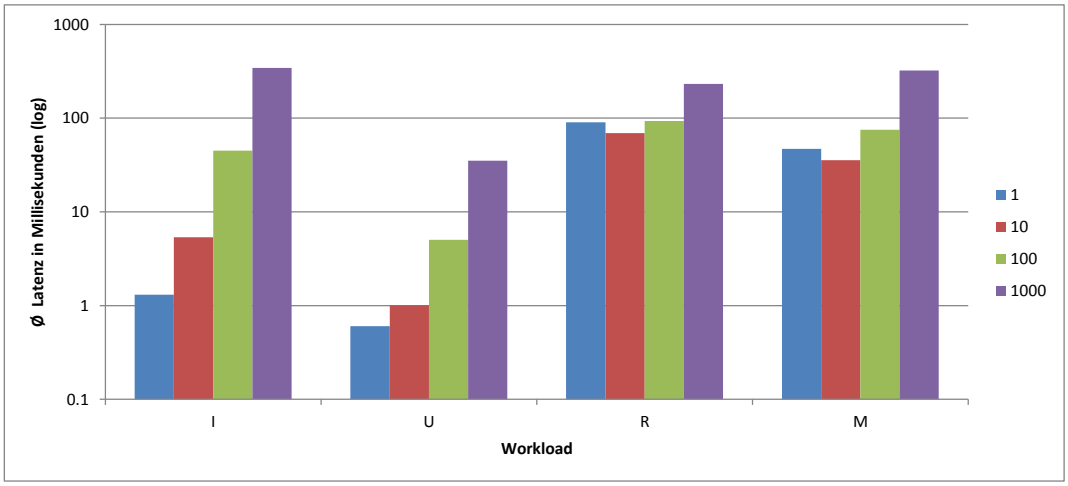


Abbildung 86: Cassandra: Unterschiedliche Datensatzgrößen - Latenz

Faktor	Workload I	Workload U	Workload R	Workload M
1	1,304	0,605	90,206	46,941
10	5,351	1,006	69,388	35,644
100	44,996	5,022	93,073	75,119
1.000	342,648	35,118	232,363	322,149

Tabelle 59: Cassandra: Unterschiedliche Datensatzgrößen - Latenz in Millisekunden

Faktor	Speicherbedarf gesamt in GB	Speicherbedarf pro Node in GB	SSTable Compression Ratio
1	77,793	19,448	0,839
10	65,470	16,367	0,982
100	63,901	15,975	1,003
1.000	51,103	12,776	1,004

Tabelle 60: Cassandra: Unterschiedliche Datensatzgrößen - Speicherbedarf

A.3.2.6. Unterschiedliche initiale Datensatzmengen

Faktor	Workload I	Workload U	Workload R	Workload M
27 Mio.	49.506	104.321	44.907	13.583
55 Mio.	49.167	103.450	886	1.667
110 Mio.	48.667	103.910	709	1.362
220 Mio.	48.524	103.233	607	1.259
441 Mio.	49.503	102.946	532	1.069

Tabelle 61: Cassandra: Unterschiedliche Datensatzmengen - Durchsatz in Operationen pro Sekunde

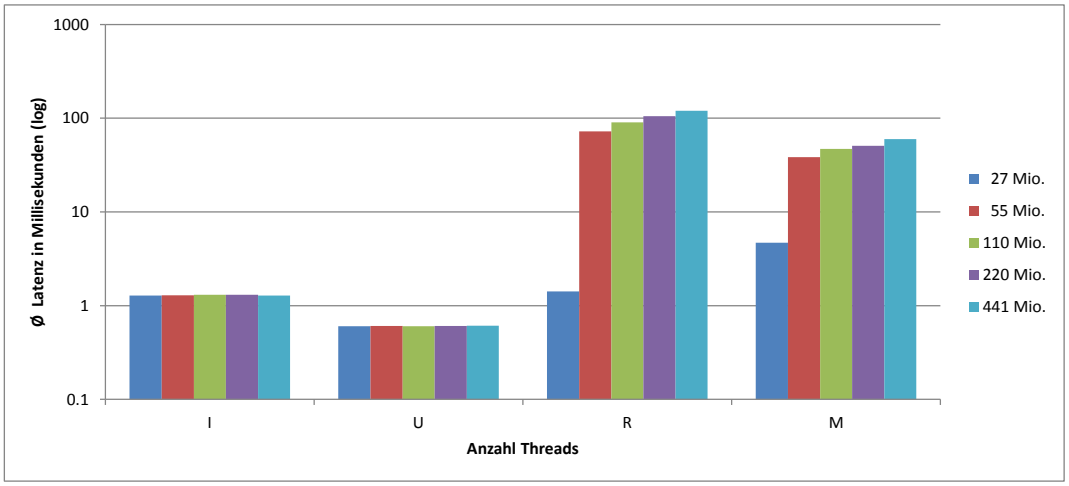


Abbildung 87: Cassandra: Unterschiedliche Datensatzmengen - Latenz

Faktor	Workload I	Workload U	Workload R	Workload M
27 Mio.	1,281	0,602	1,419	4,699
55 Mio.	1,292	0,607	72,186	38,338
110 Mio.	1,304	0,605	90,206	46,941
220 Mio.	1,307	0,609	105,267	50,773
441 Mio.	1,281	0,611	120,343	59,820

Tabelle 62: Cassandra: Unterschiedliche Datensatzmengen - Latenz in Millisekunden

A.3.2.7. Compaction

	Workload I	Workload U	Workload R	Workload M
Size-tiered Compaction	48.667	103.910	709	1.362
Leveled Compaction	48.889	101.539	986	1.902

Tabelle 63: Cassandra: Compaction - Durchsatz in Operationen pro Sekunde

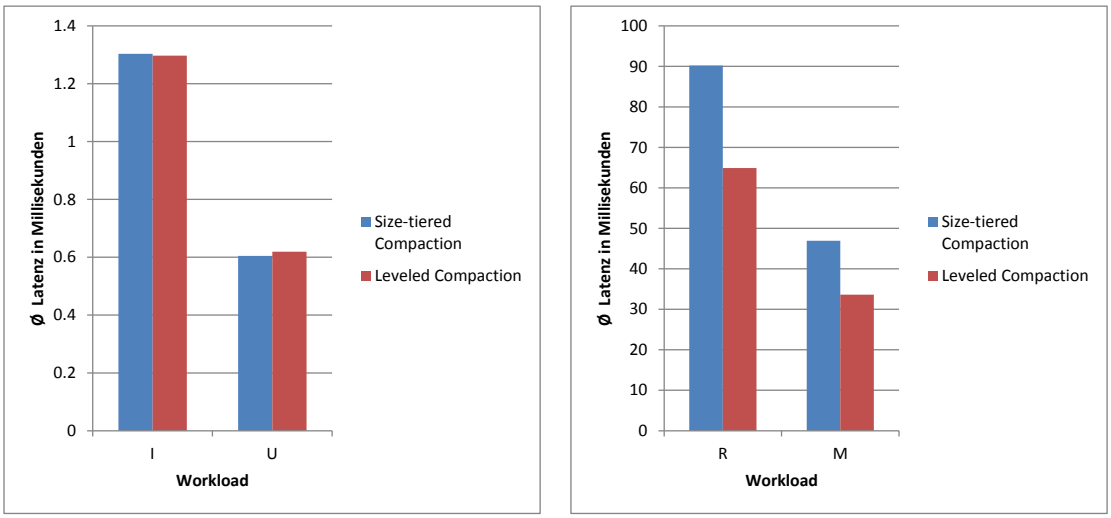


Abbildung 88: Cassandra: Compaction - Latenz

	Workload I	Workload U	Workload R	Workload M
Size-tiered Compaction	1,304	0,605	90,206	46,941
Leveled Compaction	1,297	0,619	64,897	33,614

Tabelle 64: Cassandra: Compaction - Latenz in Millisekunden

A.3.2.8. Komprimierung

	Workload I	Workload U	Workload R	Workload M
LZ4-Compressor	48.667	103.910	709	1.362
Snappy-Compressor	48.442	103.645	664	1.393
Deflate-Compressor	48.831	101.334	833	1.593
Keine Komprimierung	48.736	104.111	606	1.145

Tabelle 65: Cassandra: Komprimierung - Durchsatz in Operationen pro Sekunde

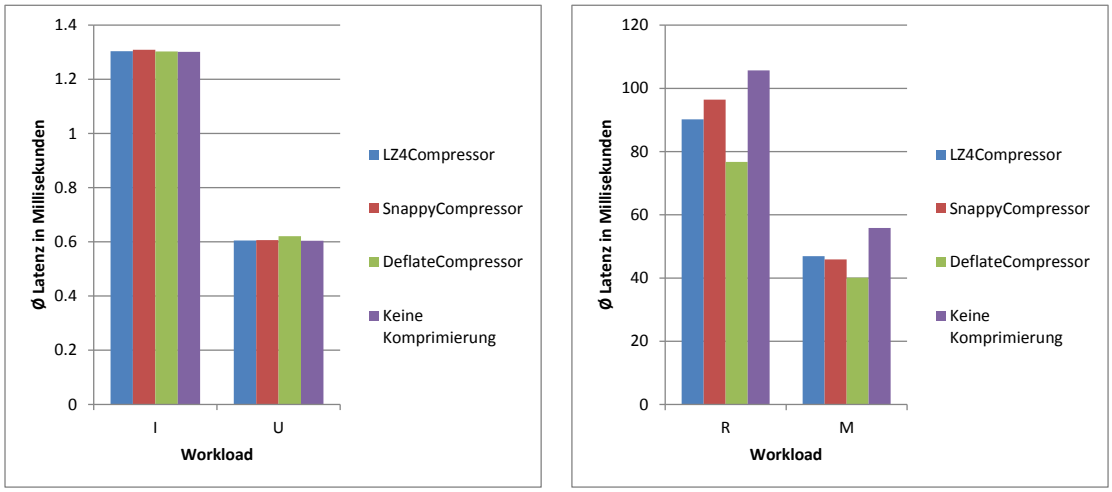


Abbildung 89: Cassandra: Komprimierung - Latenz

	Workload I	Workload U	Workload R	Workload M
LZ4-Compressor	1,304	0,605	90,206	46,941
Snappy-Compressor	1,309	0,606	96,436	45,876
Deflate-Compressor	1,302	0,621	76,751	40,124
Keine Komprimierung	1,301	0,604	105,674	55,837

Tabelle 66: Cassandra: Komprimierung - Latenz in Millisekunden

A.3.2.9. Write-Ahead Logging

	Workload I	Workload U	Workload M
Periodic (10.000 ms)	48.667	103.910	1.362
Batch (50 ms)	1.064	1.050	1.356
WAL deaktiviert	78.343	120.861	1.416

Tabelle 67: Cassandra: WAL - Durchsatz in Operationen pro Sekunde

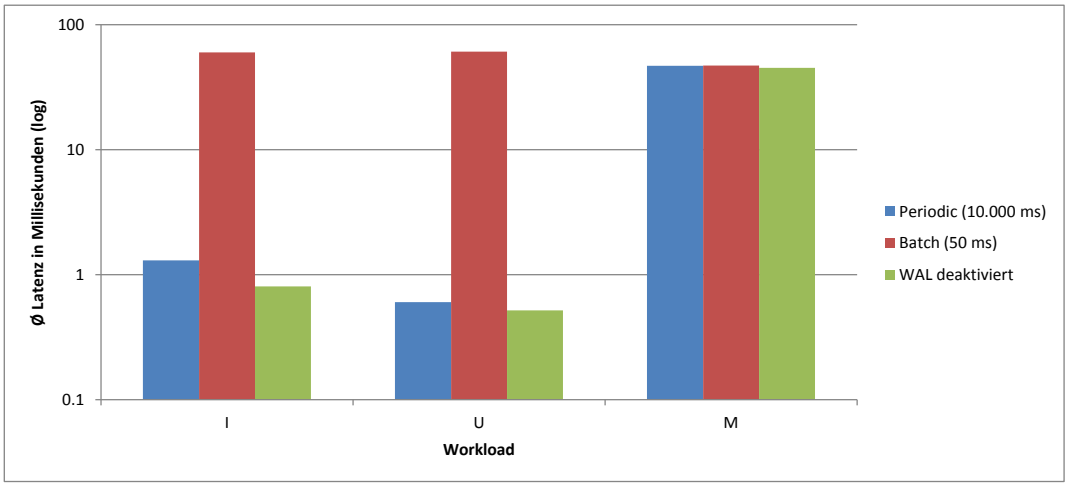


Abbildung 90: Cassandra: WAL - Latenz

	Workload I	Workload U	Workload M
Periodic (10.000 ms)	1,304	0,605	46,941
Batch (50 ms)	60,156	60,921	47,170
WAL deaktiviert	0,806	0,518	45,141

Tabelle 68: Cassandra: WAL - Latenz in Millisekunden

WAL - Separate Betrachtung von Periodic Konfigurationen

Intervall	Workload I	Workload U	Workload M
Periodic (5.000 ms)	48.975	109.177	1.355
Periodic (10.000 ms)	48.667	103.910	1.362
Periodic (20.000 ms)	49.785	104.501	1.386

Tabelle 69: Cassandra: WAL Periodic - Durchsatz in Operationen pro Sekunde

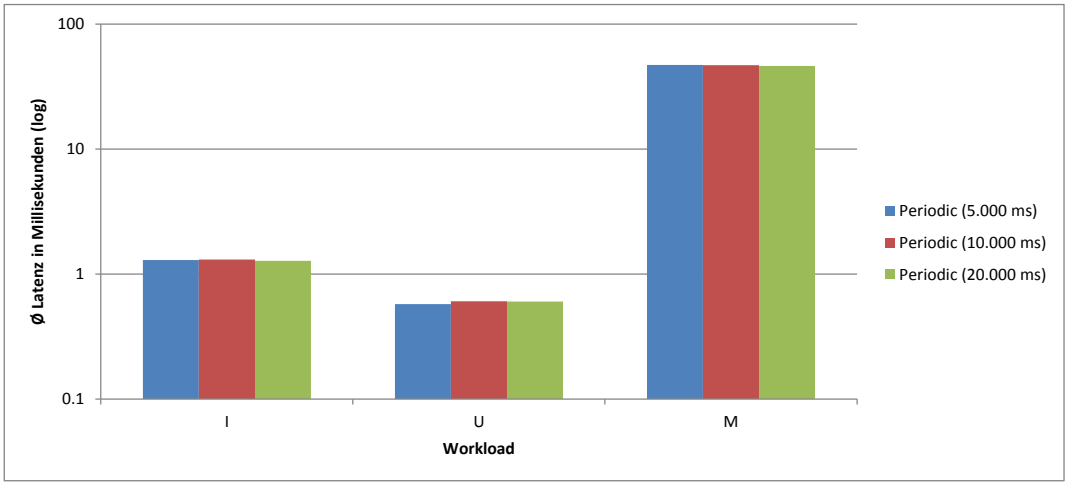


Abbildung 91: Cassandra: WAL Periodic - Latenz

Intervall	Workload I	Workload U	Workload M
Periodic (5.000 ms)	1,295	0,575	47,166
Periodic (10.000 ms)	1,304	0,605	46,941
Periodic (20.000 ms)	1,275	0,601	46,127

Tabelle 70: Cassandra: WAL Periodic - Latenz in Millisekunden

WAL - Separate Betrachtung von Batch Konfigurationen

Intervall	Workload I	Workload U	Workload M
Batch (25 ms)	1.002	1.006	1.320
Batch (50 ms)	1.064	1.050	1.356
Batch (100 ms)	715	711	1.150

Tabelle 71: Cassandra: WAL Batch - Durchsatz in Operationen pro Sekunde

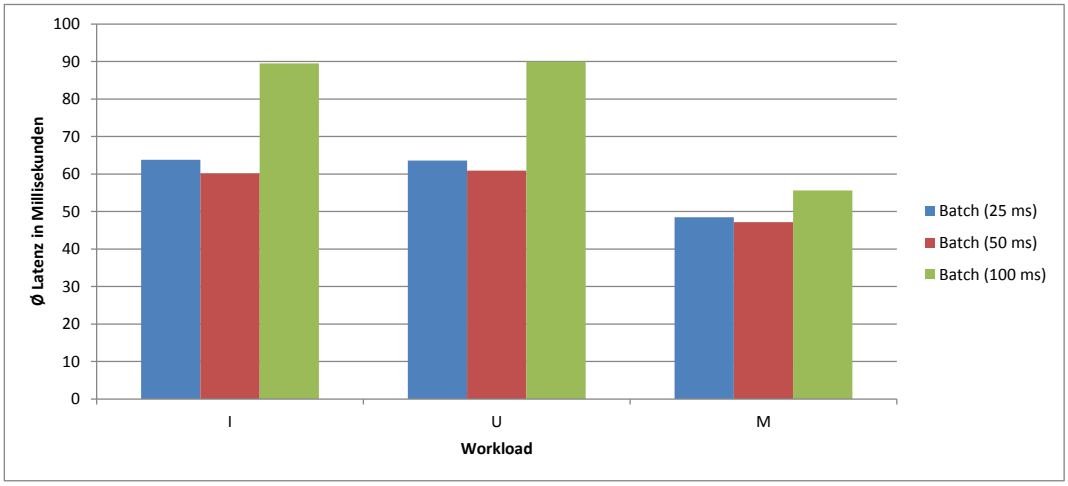


Abbildung 92: Cassandra: WAL Batch - Latenz

Intervall	Workload I	Workload U	Workload M
Batch (25 ms)	63,823	63,584	48,464
Batch (50 ms)	60,156	60,921	47,170
Batch (100 ms)	89,491	89,947	55,619

Tabelle 72: Cassandra: WAL Batch - Latenz in Millisekunden

A.3.2.10. Replikation

Antwort von allen Replika-Nodes abwarten

	Workload I	Workload U	Workload R	Workload M
Keine Replikation	48.667	103.910	709	1.362
Einfache Replikation	23.029	53.728	353	713
Zweifache Replikation	15.646	39.003	229	453
Dreifache Replikation	12.974	31.279	182	348

Tabelle 73: Cassandra: Replikation (alle Antworten abwarten) - Durchsatz in Operationen pro Sekunde

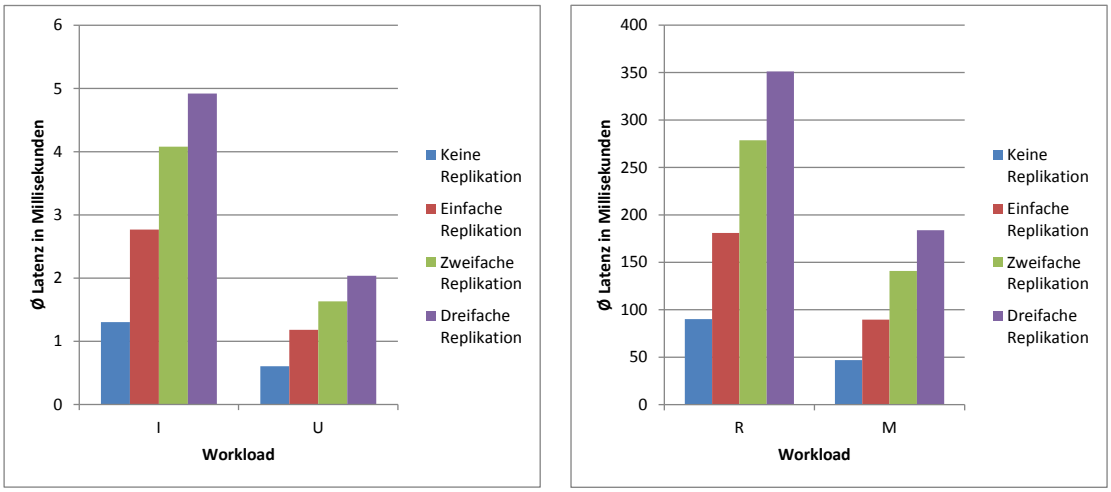


Abbildung 93: Cassandra: Replikation (alle Antworten abwarten) - Latenz

	Workload I	Workload U	Workload R	Workload M
Keine Replikation	1,304	0,605	90,206	46,941
Einfache Replikation	2,769	1,182	181,012	89,643
Zweifache Replikation	4,079	1,632	278,680	140,941
Dreifache Replikation	4,919	2,036	351,113	183,826

Tabelle 74: Cassandra: Replikation (alle Antworten abwarten) - Latenz in Millisekunden

Antwort von der Mehrheit aller Replika-Nodes abwarten

	Workload I	Workload U	Workload R	Workload M
Keine Replikation	48.667	103.910	709	1.362
Einfache Replikation	23.330	53.761	334	706
Zweifache Replikation	21.582	48.416	313	604
Dreifache Replikation	17.219	38.598	204	402

Tabelle 75: Cassandra: Replikation (Mehrheit aller Antworten abwarten) - Durchsatz in Operationen pro Sekunde

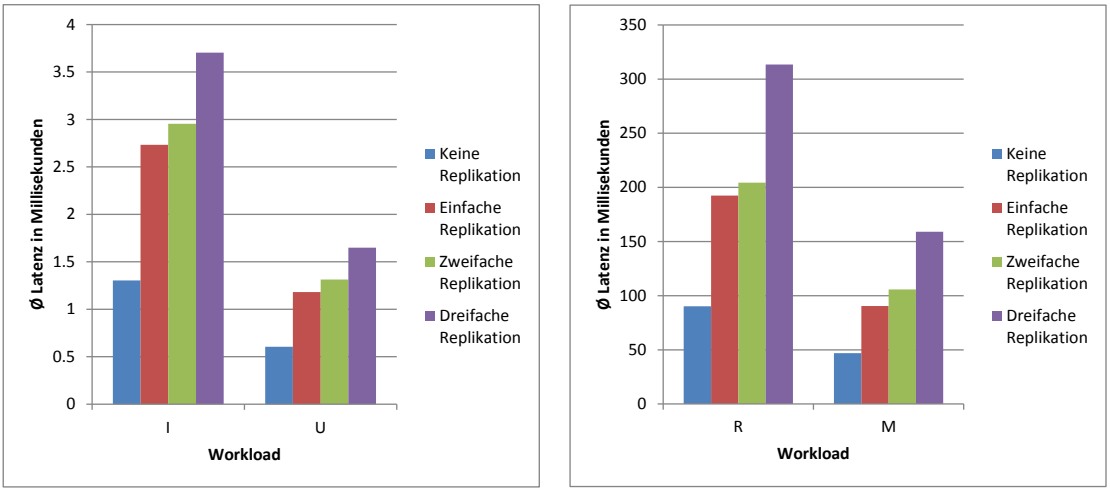


Abbildung 94: Cassandra: Replikation (Mehrheit aller Antworten abwarten) - Latenz

	Workload I	Workload U	Workload R	Workload M
Keine Replikation	1,304	0,605	90,206	46,941
Einfache Replikation	2,732	1,182	192,471	90,489
Zweifache Replikation	2,954	1,313	204,403	105,785
Dreifache Replikation	3,704	1,649	313,466	158,986

Tabelle 76: Cassandra: Replikation (Mehrheit aller Antworten abwarten) - Latenz in Millisekunden

Antwort von einem Replika-Node abwarten

	Workload I	Workload U	Workload R	Workload M
Keine Replikation	48.667	103.910	709	1.362
Einfache Replikation	35.632	72.789	593	1.113
Zweifache Replikation	28.168	60.662	483	937
Dreifache Replikation	22.592	52.879	431	808

Tabelle 77: Cassandra: Replikation (eine Antwort abwarten) - Durchsatz in Operationen pro Sekunde

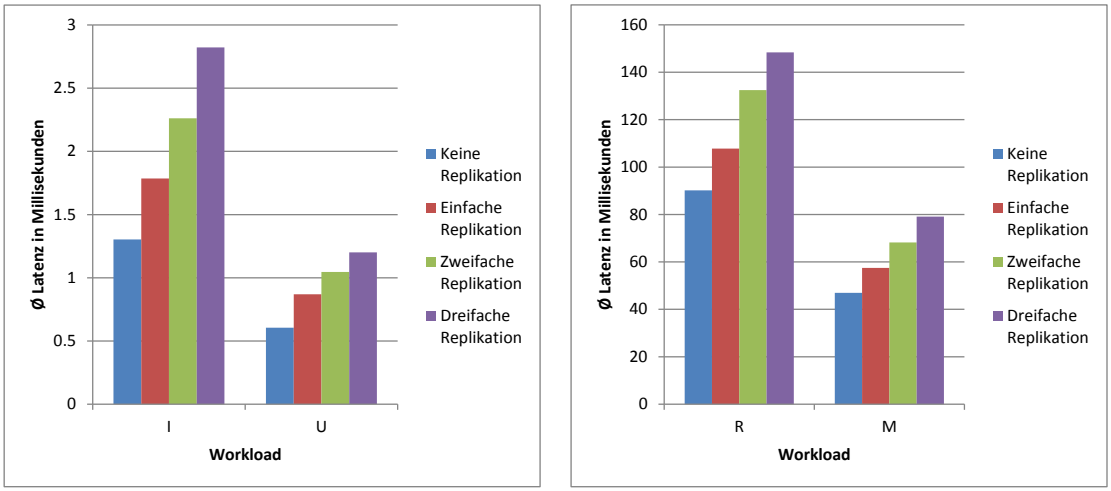


Abbildung 95: Cassandra: Replikation (eine Antwort abwarten) - Latenz

	Workload I	Workload U	Workload R	Workload M
Keine Replikation	1,304	0,605	90,206	46,941
Einfache Replikation	1,785	0,870	107,823	57,472
Zweifache Replikation	2,261	1,046	132,487	68,228
Dreifache Replikation	2,822	1,202	148,401	79,124

Tabelle 78: Cassandra: Replikation (eine Antwort abwarten) - Latenz in Millisekunden

Operationsarten spezifisches Abwarten von Antworten

	Write All, Read One	**Write All, Read All**	**Write One, Read One**	**Write Quorum, Read Quorum**
Einfache Replikation	1.131	713	1.113	706
Zweifache Replikation	929	453	937	604
Dreifache Replikation	807	348	808	402

Tabelle 79: Cassandra: Konsistenz-Level im Vergleich (Workload M) - Durchsatz in Operationen pro Sekunde

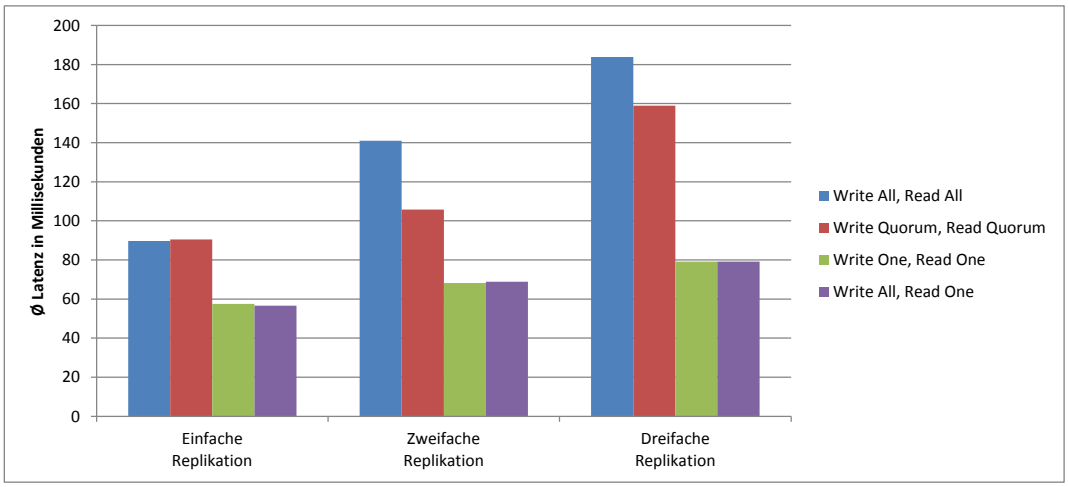


Abbildung 96: Cassandra: Konsistenz-Level im Vergleich (Workload M) - Latenz

	Write All, Read One	**Write All, Read All**	**Write One, Read One**	**Write Quorum, Read Quorum**
Einfache Replikation	56,542	89,643	57,472	90,489
Zweifache Replikation	68,849	140,941	68,228	105,785
Dreifache Replikation	79,200	183,826	79,124	158,986

Tabelle 80: Cassandra: Konsistenz-Level im Vergleich (Workload M) - Latenz in Millisekunden

Abkürzungsverzeichnis

AG	Aktiengesellschaft
API	Application Programming Interface
CQL	Cassandra Query Language
DB	Datenbank
DOAG	Deutsche ORACLE-Anwendergruppe
EMF	Eclipse Modeling Framework
JDK	Java Development Kit
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
LTS	Long Term Support
MDE	Model Driven Engineering
NoSQL	Not only SQL
NSA	National Security Agency
NTP	Network Time Protocol
NTPd	Network Time Protocol daemon
SDK	Software Development Kit
SLA	Service Level Agreement
SoAR	Social Action Rating
SQL	Structured Query Language
SSD	Solid State Drive
SSH	Secure Shell
SSTable	Sorted String Table
WAL	Write-Ahead Logging
XMI	XML Metadata Interchange
XML	Extensible Markup Language
YCSB	Yahoo! Cloud Serving Benchmark

Literatur- und Quellenverzeichnis

[Alt14] Altoros: *The NoSQL Technical Comparison Report: Cassandra (DataStax), MongoDB, and Couchbase Server*, 09.2014. http://www.altoros.com/nosql-tech-comparison-cassandra-mongodb-couchbase.html. – zuletzt besucht am 02.03.2015.

[Apa15] Apache Cassandra: *Cassandra storage config YAML*, 05.03.2015. https://raw.githubusercontent.com/apache/cassandra/cassandra-2.1/conf/cassandra.yaml. – zuletzt besucht am 06.03.2015.

[Ava15] Avalon Consulting, LLC: *Comparing Couchbase Server 3.0.2 with MongoDB 3.0: Benchmark Results and Analysis*, 19.03.2015. http://info.couchbase.com/rs/northscale/images/couchbase_benchmark.pdf. – zuletzt besucht am 27.03.2015.

[BG13] Barahmand, Sumita ; Ghandeharizadeh, Shahram: *BG: A Benchmark to Evaluate Interactive Social Networking Actions*, 2013. http://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper93.pdf. – zuletzt besucht am 29.03.2015.

[Cha12] Chakravarthi, Srigurunath: *GridMix3*, 08.05.2012. http://clds.sdsc.edu/sites/clds.sdsc.edu/files/wbdb2012/presentations/WBDB2012Presentation44Chakravarthi.pdf. – zuletzt besucht am 02.03.2015.

[Coo10] Cooper, Brian F.: *Core Workloads*, 14.09.2010. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads. – zuletzt besucht am 31.12.2014.

[Cou13] Couchbase: *Compaction magic in Couchbase Server 2.0*, 18.02.2013. http://blog.couchbase.com/compaction-magic-couchbase-server-20. – zuletzt besucht am 21.03.2015.

[Cou14] Couchbase: *Couchbase Server 3.0 Documentation*, 16.10.2014. http://docs.couchbase.com/prebuilt/pdfs/couchbase-3.0-2014-10-16.pdf. – zuletzt besucht am 17.12.2014.

[CST+10] Cooper, Brian F. ; Silberstein, Adam ; Tam, Erwin ; Ramakrishnan, Raghu ; Sears, Russell: Benchmarking Cloud Serving Systems with YCSB. In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. New York, NY, USA : ACM, 2010 (SoCC '10), S. 143–154.

[Dat13] DataStax Corporation: *Benchmarking Top NoSQL Databases: A Performance Comparison for Architects and IT Managers*, 02.2013. `https://www.datastax.com/wp-content/uploads/2013/02/WP-Benchmarking-Top-NoSQL-Databases.pdf`. – zuletzt besucht am 02.03.2015.

[Dat14] DataStax Corporation: *Apache Cassandra™ 2.0: The read path*, 07.08.2014. `http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_about_read_path_c.html`. – zuletzt besucht am 15.03.2015.

[Dat15a] DataStax Corporation: *CQL for Cassandra 2.x Documentation: March 13, 2015*, 13.03.2015. `http://www.datastax.com/documentation/cql/3.1/pdf/cql31.pdf`. – zuletzt besucht am 15.03.2015.

[Dat15b] DataStax Corporation: *Apache Cassandra™ 2.1 Documentation: January 16, 2015*, 16.01.2015. `http://www.datastax.com/documentation/cassandra/2.1/pdf/cassandra21.pdf`. – zuletzt besucht am 19.01.2015.

[DFNR14] Dey, Akon ; Fekete, Alan ; Nambiar, Raghunath ; Röhm, Uwe: YCSB+T: Benchmarking web-scale transactional databases. In: *Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on*, 2014, S. 223–230.

[DG13] Diomin, Alexey ; Grigorchuk, Kirill: *Benchmarking Couchbase Server for Interactive Applications*, 2013. `http://www.couchbase.com/sites/default/files/uploads/all/Benchmarking-Couchbase-Server-for-Interactive-Applications.pdf`. – zuletzt besucht am 03.03.2015.

[Ell11] Ellis, Jonathan: *DataStax Developer Blog: Leveled Compaction in Apache Cassandra*, 10.10.2011. `http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra`. – zuletzt besucht am 16.03.2015.

[FAC+14] Ferrarons, Jaume ; Adhana, Mulu ; Colmenares, Carlos ; Pietrowska, Sandra ; Bentayeb, Fadila ; Darmont, Jérôme: PRIMEBALL: A Parallel Processing Framework Benchmark for Big Data Applications in the Cloud. In: *Performance Characterization and Benchmarking* Bd. 8391. Springer International Publishing, 2014, S. 109–124.

[GGK+14] Gandini, Andrea ; Gribaudo, Marco ; Knottenbelt, WilliamJ. ; Osman, Rasha ; Piazzolla, Pietro: Performance Evaluation of NoSQL Databases. In: *Computer Performance Engineering* Bd. 8721. Springer International Publishing, 2014, S. 16–29.

[GP14] Gusev, Andrew ; Pozdnyakov, Sergey: *NoSQL Performance when Scaling by RAM: Benchmarking Cassandra, Couchbase, and MongoDB for RAM-heavy Workloads*, 2014. http://info.couchbase.com/rs/northscale/images/NoSQL_Performance_Scaling_by_RAM.pdf. – zuletzt besucht am 02.03.2015.

[HHD+10] Huang, Shengsheng ; Huang, Jie ; Dai, Jinquan ; Xie, Tao ; Huang, Bo: The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In: *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, 03.2010, S. 41–51.

[Ing14] Ingenthron, Matt: *Couchbase java sdk 1.4.5 compatible with couchbase 3.0.1*, 14.11.2014. https://forums.couchbase.com/t/couchbase-java-sdk-1-4-5-compatible-with-couchbase-3-0-1/2123. – zuletzt besucht am 12.01.2015.

[Kir14] Kirkconnell, Kirk: *Often Overlooked Linux OS Tweaks*, 06.03.2014. http://blog.couchbase.com/often-overlooked-linux-os-tweaks. – zuletzt besucht am 06.01.2015.

[Kup12] Kuppuswamy, Hariprasad: *Mavenisation of the YCSB with a (tar ball) distribution packager*, 16.02.2012. https://github.com/thumbtack-technology/ycsb/commit/3532283de4d1f2baae6502e7c58dfbcbb8924a24#diff-5f93d862ed7f6d1183093e445394e19a. – zuletzt besucht am 30.12.2014.

[Leb15] Lebresne, Sylvain: *Allow IF EXISTS for UPDATE statements*, 04.02.2015. https://issues.apache.org/jira/browse/CASSANDRA-8610. – zuletzt besucht am 10.03.2015.

[Lin15] Linux Kernel Organization: *Kernel Parameters*, 23.02.2015. https://www.kernel.org/doc/Documentation/kernel-parameters.txt. – zuletzt besucht am 06.03.2015.

[LM10] Lakshman, Avinash ; Malik, Prashant: Cassandra: A Decentralized Structured Storage System. In: *SIGOPS Oper. Syst. Rev.* 44 (2010), April, Nr. 2, S. 35–40.

[Mic14] Michaels, Justin: *Best Practices: Managing a Healthy Couchbase Server Deployment: Couchbase Connect 2014*, 20.10.2014. http://de.slideshare.net/Couchbase/best-practicesmanagingahealthydeployment-justinmichaels. – zuletzt besucht am 17.12.2012.

[MK13] McCreary, Dan ; Kelly, Ann: *Making Sense of NoSQL: A Guide for Managers and the Rest of Us*. Shelter Island, NY: Manning Publications Co., 2013.

[MR15] Morreale, Peter W. ; Riel, Rik van: *Documentation for /proc/sys/vm/**, 15.02.2015. https://www.kernel.org/doc/Documentation/sysctl/vm.txt. – zuletzt besucht am 05.03.2015.

[NE13a] Nelubin, Denis ; Engber, Ben: *Ultra-High Performance NoSQL Benchmarking: Analyzing Durability and Performance Tradeoffs*, 11.01.2013. http://www.aerospike.com/wp-content/uploads/2013/01/Ultra-High-Performance-NoSQL-Benchmarking.pdf. – zuletzt besucht am 13.01.2015.

[NE13b] Nelubin, Denis ; Engber, Ben: *NoSQL Failover Characteristics: Aerospike, Cassandra, Couchbase, MongoDB*, 2013. http://www.thumbtack.net/whitepapers/nosql-failover-characteristics.html. – zuletzt besucht am 02.03.2015.

[Nel13] Nelubin, Denis: *Modified processing of persistTo and replicateTo parameters*, 09.01.2013. https://github.com/thumbtack-technology/ycsb/commit/735f2ecd082767c469f3659017821bfb8ced90e4. – zuletzt besucht am 23.12.2014.

[Nit14] Nitschinger, Michael: *Couchbase 3.0.1 performance issue in full metadata ejection mode*, 14.11.2014. https://forums.couchbase.com/t/couchbase-3-0-1-performance-issue-in-full-metadata-ejection-mode/2124/3. – zuletzt besucht am 14.01.2015.

[ORD15a] ORDIX AG: *Bewährte Technologien - Leistungen auf hohem Niveau*, 2015. http://www.ordix.de/dienstleistung/technologien.html. – zuletzt besucht am 27.03.2015.

[ORD15b] ORDIX AG: *ORDIX Dienstleistungen - einfach. gut. beraten.*, 2015. http://www.ordix.de/dienstleistung/ordix-dienstleistungen.html. – zuletzt besucht am 27.03.2015.

[ORD15c] ORDIX AG: *ORDIX® – Qualität seit 25 Jahren*, 2015. http://www.ordix.de/unternehmen/wir-ueber-uns.html. – zuletzt besucht am 27.03.2015.

[Pop14] Popescu, Alex: *4 simple rules when using the DataStax drivers for Cassandra*, 05.06.2014. http://planetcassandra.org/blog/4-simple-rules-when-using-the-datastax-drivers-for-cassandra/. – zuletzt besucht am 19.03.2015.

[PPR+11] Patil, Swapnil ; Polte, Milo ; Ren, Kai ; Tantisiriroj, Wittawat ; Xiao, Lin ; López, Julio ; Gibson, Garth ; Fuchs, Adam ; Rinaldi, Billie: YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. New York, NY, USA : ACM, 2011 (SOCC '11), S. 9:1–9:14.

[RGVS+12] Rabl, Tilmann ; Gómez-Villamor, Sergio ; Sadoghi, Mohammad ; Muntés-Mulero, Victor ; Jacobsen, Hans-Arno ; Mankovskii, Serge: Solving Big Data Challenges for Enterprise Application Performance Management. In: *Proc. VLDB Endow.* 5 (2012), August, Nr. 12, S. 1724–1735.

[Roo13] Roos, Alexis: *Calculating average document size of documents stored in Couchbase*, 12.07.2013. `http://blog.couchbase.com/calculating-average-document-size-documents-stored-couchbase`. – zuletzt besucht am 22.12.2014.

[SF12] Sadalage, Pramod J. ; Fowler, Martin: *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence.* München: Addison-Wesley, 2012

[SWK+14] Shah, SeyyedM. ; Wei, Ran ; Kolovos, DimitriosS. ; Rose, LouisM. ; Paige, RichardF. ; Barmpis, Konstantinos: A Framework to Benchmark NoSQL Data Stores for Large-Scale Model Persistence. In: *Model-Driven Engineering Languages and Systems* Bd. 8767. Springer International Publishing, 2014, S. 586–601.

[Til14] Tillman, Luke: *Introduction to Apache Cassandra*, 23.10.2014. `http://de.slideshare.net/LukeTillman/introduction-to-cassandra-40657854`. – zuletzt besucht am 17.03.2015.

[Tra15] Transaction Processing Performance Council: *TPC EXPRESS BENCHMARK^{TM}HS (TPCx-HS): Standard Specification Version 1.3.0*, 19.02.2015. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpcx-hs_specification_1.3.0.pdf`. – zuletzt besucht am 02.03.2015.

[WZL+14] Wang, Lei ; Zhan, Jianfeng ; Luo, Chunjie ; Zhu, Yuqing ; Yang, Qiang ; He, Yongqiang ; Gao, Wanling ; Jia, Zhen ; Shi, Yingjie ; Zhang, Shujie ; Zheng, Chen ; Lu, Gang ; Zhan, Kent ; Li, Xiaona ; Qiu, Bizhu: *BigDataBench: a Big Data Benchmark Suite from Internet Services*, 2014. `http://arxiv.org/pdf/1401.1406v2.pdf`. – zuletzt besucht am 02.03.2015.