Metadaten-Verwaltung in einem verteilten RAM-basierten Speicherdienst

Inaugural-Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Florian Klein
aus Düsseldorf

Monheim am Rhein, 25. September 2015

Aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent: Prof. Dr. Michael Schöttner
Korreferent: Prof. Dr. Martin Mauve

Tag der mündlichen Prüfung: 16.11.2015

Kurzfassung

Große interaktive Web-Anwendungen und Echtzeit-Graphverarbeitung erfordern schnelle Datenzugriffszeiten auf Milliarden kleiner Objekte und stoßen damit an die Grenzen klassischer festplatten-basierter Speichersysteme. Auf Grund der häufig unregelmäßigen Zugriffsmuster werden große Teile der Daten in RAM-basierten Caches gehalten, die manuell mit dem Hintergrundspeicher synchronisiert werden müssen. Auch das Neubefüllen solcher Caches, zum Beispiel bei einem Stromausfall, ist sehr zeitaufwendig und damit kostspielig. DXRAM begegnet diesen Problemen, indem alle Anwendungsdaten permanent im Hauptspeicher vieler vernetzter Knoten eines Rechenzentrums gehalten werden. Die Verwaltung Milliarden kleiner Datenobjekte (16-64 Byte) bei gleichzeitiger Gewährleistung von Persistenz und Fehlertoleranz sind dabei die Hauptziele. Persistenz und Fehlertoleranz werden mit Hilfe einer für SSD zugeschnittenen transparenten Hintergrundprotokollierung sichergestellt, die eine schnelle Datenwiederherstellung ausgefallener Knoten ermöglicht. Obwohl Daten- und Metadaten-Verwaltung ausführlich erforscht sind, stellen die große Anzahl sehr kleiner Objekte neue Herausforderungen dar. Der Einsatz von Supercomputern kann die Probleme zwar abmildern, ist jedoch auch sehr kostenintensiv und daher der Einsatz herkömmlicher PCs zu bevorzugen.

In dieser Arbeit wird ein neues integriertes Konzept von lokaler und globaler Metadaten-Verwaltung vorgestellt, das einen schnellen Datenzugriff und einen hohen Durchsatz erlaubt und gleichzeitig sehr speichereffizient ist. Die lokale Metadaten-Verwaltung umfasst ein effizientes paging-ähnliches Schema für die Übersetzung von globalen IDs zu virtuellen Speicheradressen und eine Speicherverwaltung optimiert für viele kleine Objekte. Dabei erlaubt es ein speziell entwickelter Speicherallokator den Speicherverbrauch für Metadaten auf zwei Byte pro Objekt zu senken, so dass über eine Milliarde Objekte pro Knoten (bei 32 GB RAM pro Knoten) gespeichert werden können. Um sich ändernde Allokationsmuster zu unterstützen, wird ein effizienter inkrementeller Defragmentierungsalgorithmus eingesetzt, der im Hintergrund oder bei Bedarf parallel zum Gesamtsystem arbeitet.

Unter Verwendung eines Super-Peer-Overlays wird eine auf Bereichen basierte globale Metadaten-Verwaltung vorgestellt, die eine schnelle Objektsuche ermöglicht und gleichzeitig sehr speichereffizient ist, indem Objekt-IDs zu ID-Bereichen zusammengefasst werden. Die Super-Peers verwalten diese ID-Bereiche zusammen mit Backupknoten-Informationen und ermöglichen damit eine parallele und schnelle Wiederherstellung von Metadaten und Daten eines

ausgefallenen Knotens. Zudem lässt sich das gleiche Konzept auch für ein Caching von Metadaten nutzen. Der gewählte Peer-to-Peer-Ansatz ermöglicht es, dass das Speichersystem als eigenständiger Backend-Speicher genutzt oder Anwendungscode direkt auf den Speicherknoten ausgeführt werden kann.

Sowohl die lokale als auch die globale Metadaten-Verwaltung wurden erfolgreich evaluiert und mit modernen Ansätzen und Systemen neuester Technik verglichen. Die präsentierten Messergebnisse zeigen einen hohen Durchsatz und eine sehr effiziente Speicherausnutzung, die besser als die herkömmlicher Systeme sind.

Abstract

Traditional disk-based storage solutions face problems with fast data access to billions of small data objects, as needed by large-scale interactive web applications and online graph processing. Because of the often irregular access patterns they must keep almost all data in RAM caches, which need to be manually synchronized with secondary storage and need a lot of time to be reloaded in case of power outages. DXRAM addresses this challenge by keeping all data always in RAM of potentially many nodes aggregated in a data center. The main aims of DXRAM are support of billions of small data objects (16-64 byte) and providing persistence by a novel SSD-aware logging approach allowing to recover failed nodes very fast. Although data and meta-data management are widely researched the sheer amount of very small objects rises new problems. The use of supercomputers can lessen the problems, but is very cost-intensiv. Therefore the use of traditional PCs is preferred.

This thesis presents a novel integrated approach of local and global meta-data management allowing a fast data access and high throughput while being very space-efficient. The local meta-data management includes an efficient paging-like translation scheme for global IDs to virtual memory addresses and a memory management optimized for many small data objects. A novel memory allocator allows to reduce the meta-data for allocations down to two bytes per object, allowing to store over one billion objects per node (with 32 GB of memory per node). Changing data granularities is supported by an efficient incremental defragmentation which can run in the background or as needed parallel to the overall system.

A super-peer-overlay is used for a range-based meta-data management allowing fast node lookups while being space-efficient by combining object IDs in ranges. The super-peers manage these ranges together with backup-node information to support parallel and fast recovery of meta data and data of failed peers. Furthermore, the same concept can also be used for caching. The choosen peer-to-peer approach allows to use the storage system as self-contained backend storage or to run application code on the storage nodes.

Both local and global meta-data management have been successfully evaluated and compared with state-of-the-art approaches and systems and the results show the high throughput and the very efficient memory usage, which are better than traditional systems.

Danksagung

An dieser Stelle möchte ich Herrn Prof. Dr. Michael Schöttner meinen herzlichsten Dank für die Ermöglichung dieser Dissertation aussprechen. Sein persönlicher Einsatz und die großzügige Unterstützung haben wesentlich zu dieser Arbeit beigetragen und verdienen ein besonderes Dankeschön.
Mein Dank geht auch an Herrn Prof. Dr. Martin Mauve für die Betreuung als Mentor und die Begutachtung dieser Arbeit.

Meinen Kollegen Kevin Beineke, Michael Braitmeier und Kim-Thomas Rehmann danke ich für ihren fachlichen und persönlichen Austausch und für die (teilweise kurzfristige) Hilfe bei Aufgabenstellungen und Problemen. Besonderen Dank geht an Angela Rennwanz, die in allen organisatorischen Angelegenheiten für mich unverzichtbar gewesen ist.
Ich danke auch Marc Ewert, Carsten Schroeder, Sebastian Klüppel, Christof Switala und Yunus Kaplan, die durch ihre Bachelor-, Projekt- und Masterarbeiten zu dieser Dissertation beigetragen haben.

Ganz besonderer Dank gilt meiner zukünftigen Frau Vanessa für ihre volle und liebevolle Unterstützung meiner Anstrengungen und ihren Rückhalt auch in schwierigeren Phasen.
Meiner Familie und meinen Freunden danke ich für ihre Unterstützung während meiner gesamten Promotionszeit. Ohne sie wäre das Ganze nicht möglich geworden. Besonderen Dank geht dabei an meine Eltern, Bettina und Manfred Klein, die immer wieder Korrektur gelesen haben.

Als Letztes danke ich allen, die Interesse an dieser Arbeit zeigten und mich unterstützten.

Florian Klein

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1
1.2 Klassifizierung von Speichersystemen . . . 4
1.3 Ziel der Arbeit . . . 9
1.4 Beiträge dieser Arbeit . . . 10
1.5 Struktur der Arbeit . . . 11

2 Architektur von DXRAM **12**

2.1 Dienste und Komponenten . . . 14
2.1.1 Erweiterte Dienste . . . 15
2.1.2 Kern-Dienste . . . 15
2.2 Super-Peer-Overlay . . . 18
2.2.1 Objektsuche . . . 18
2.2.2 Knotenbeitritt . . . 19
2.2.3 Fehlertoleranz . . . 20
2.3 RAM-Management . . . 23
2.3.1 Wiederverwendung von CIDs . . . 23
2.3.2 Übersetzung von CIDs zu virtuellen Speicheradressen . . . 24
2.3.3 RAM-Layout . . . 25
2.4 Backup-Management . . . 27
2.4.1 Backup-Prozess . . . 28
2.4.2 SSD-Charakteristiken . . . 29
2.4.3 Protokoll-Architektur . . . 29
2.4.4 Datenwiederherstellung . . . 32
2.4.5 Reorganisation . . . 33
2.4.6 Hot-and-Cold Zonen . . . 34
2.5 Verwandte Arbeiten . . . 34
2.6 Zusammenfassung . . . 45

3 Lokale Metadaten-Verwaltung **46**
3.1 Architektur 46
3.1.1 Komponenten der lokalen Metadaten-Verwaltung 47
3.1.2 Funktionen der lokalen Metadaten-Verwaltung 47
3.2 Adressübersetzung (CID -> VMA) 49
3.2.1 Hashtabellen 49
3.2.2 Indizes 51
3.2.3 CID-Tabellen 52
3.2.4 Freie CIDs 54
3.3 Speicherverwaltung für sehr viele sehr kleine Objekte 56
3.3.1 Traditionelle Speicherallokatoren 56
3.3.2 Initialisierung der Speicherverwaltung 57
3.3.3 Speicher allozieren (Malloc) 59
3.3.4 Speicher freigeben (Free) 61
3.3.5 Speichereffizienz 61
3.3.6 Inkrementelle Defragmentierung 63
3.3.7 Besonderheiten bei Java 67
3.4 Multi-Core-Optimierung 68
3.4.1 Synchronisierung 68
3.4.2 Synchronisierung der Adressübersetzung 80
3.4.3 Synchronisierung der Speicherverwaltung 86
3.4.4 Eingesetzte Threads und Sperren 90
3.5 Verwandte Arbeiten 93
3.6 Zusammenfassung 97

4 Globale Metadaten-Verwaltung **100**
4.1 Architektur 100
4.2 Super-Peer-Overlay 101
4.2.1 Objektsuche 102
4.2.2 Migrationen 103
4.2.3 Ausfall eines Peers 103
4.2.4 Ausfall eines Super-Peers 105
4.2.5 Totalausfall 106
4.3 CID-Bereiche 106
4.3.1 Datenstrukturen zur Verwaltung von CID-Bereichen 108
4.3.2 Integration der Backupknoten 112
4.3.3 Caching auf Clientseite 114
4.3.4 Paralleler Zugriff 118

4.4 Namensdienst . . . 120
4.4.1 Zuordnung eintragen . . . 120
4.4.2 CID ermitteln . . . 121
4.4.3 Modifizierte Hashtabelle . . . 121
4.5 Verwandte Arbeiten . . . 121
4.6 Zusammenfassung . . . 128

5 Messungen **131**
5.1 Testumgebungen . . . 131
5.2 Evaluation der lokalen Metadaten-Verwaltung . . . 132
5.2.1 Vergleich von Hash- und CID-Tabellen . . . 132
5.2.2 Speicherallokation . . . 136
5.2.3 Defragmentierung . . . 140
5.2.4 Adressübersetzung und Speicherverwaltung . . . 141
5.2.5 Vergleich mit RAMCloud . . . 143
5.3 Evaluation der globalen Metadaten-Verwaltung . . . 145
5.3.1 CID-Baum . . . 145
5.3.2 Caching von CID-Bereichen . . . 147
5.4 Evaluation des Gesamtsystems mit BG-Benchmark . . . 155
5.4.1 BG-Benchmark . . . 155
5.4.2 MongoDB . . . 157
5.4.3 Cassandra . . . 157
5.4.4 Performanz des Gesamtsystems . . . 158
5.5 Zusammenfassung . . . 160

6 Zusammenfassung **162**
6.1 Resultat . . . 162
6.2 Ausblick . . . 166

Abbildungsverzeichnis **169**

Tabellenverzeichnis **171**

Algorithmenverzeichnis **173**

Literaturverzeichnis **174**

A Messergebnisse **189**
A.1 Lokale Metadaten-Verwaltung . . . 189
A.1.1 Speicherverwaltung . . . 189

A.1.2 CID-Tabellen 190
A.1.3 CID-Tabellen und Speicherverwaltung 193
A.1.4 Vergleich mit RAMCloud 194
A.2 Globale Metadaten-Verwaltung 196
A.2.1 CID-Baum 196
A.2.2 Caching 198
A.3 Gesamtsystem 202
A.3.1 BG-Benchmark 202

Kapitel 1

Einleitung

Durch die fallenden Preise für Speichermedien ist in den letzten Jahren der Trend entstanden, dass Anwendungen immer mehr Daten speichern und verarbeiten. Diese Datenmengen, häufig als Big Data bezeichnet, sind meist so groß oder komplex, dass eine klassische Verarbeitung auf einem einzelnen Computer nicht mehr möglich ist [1]. Supercomputer können diese Probleme abmildern, sind jedoch sehr kostenintensiv im Gegensatz zu herkömmlichen PCs. Die Entwicklung von neuen verteilten und parallelen Systemen und Algorithmen zur Speicherung, Verwaltung und Auswertung dieser Daten ist daher unerlässlich.
Die gespeicherten Daten können aus vielen verschiedenen Quellen stammen und umfassen beispielsweise soziale Netzwerke, Mitarbeiter- und Kundendaten, elektronische Kommunikation, Gesundheitsdaten, Aufzeichnungen von Überwachungssystemen, Sensorauswertungen, Fotos und Videos und vieles mehr. Entsprechende Anwendungen sollen diese Daten auswerten und nutzbar machen. Mögliche Anwendungsfälle sind Marktbeobachtung, personalisierte Produktempfehlungen, Kündigerfrüherkennung, Umsichtige Steuerung, etc. [2].

1.1 Motivation

Große interaktive Web-Anwendungen, wie beispielsweise soziale Netzwerke, und Online Graphverarbeitung erlangen immer mehr Popularität und stellen besondere Anforderungen an die zugrunde liegende Speichersysteme. Diese Anwendungen verwalten nicht nur sehr große Datenmengen, sondern erfordern einen sehr schnellen Zugriff auf die Daten. Eine Untersuchung von Google und Bing (Microsoft) zeigt, dass Anfrage-Verzögerungen nicht nur eine Verschlechterung der Nutzerstatistiken nach sich zieht, sondern dass diese Verschlechterung dauerhaft ist und mit der Zeit zunimmt [3]. Geringe Latenzzeiten sind daher nicht nur ein Wunschkriterium, sondern beeinflussen erheblich Umsatz und Gewinn von Unternehmen. Die Zugriffsmuster dieser Anwendungen sind häufig unregelmäßig, lassen sich nur schwer voraussagen und werden von Lesezugriffen dominiert. Schreibzugriffe sind selten und abhängig vom Typ

der Daten spielen Aktualisierungs- und Löschoperationen nur eine untergeordnete Rolle [4, 5]. Festplatten-basierte Speichersysteme, wie klassische Datenbanken oder verteilte Dateisysteme, können diese Anforderungen nur schwer oder gar nicht gewährleisten und werden zunehmend durch andere Speicherverfahren ersetzt.

Ein erster Ansatz für die Verbesserung der Zugriffszeiten und eine bessere Performanz ist der Einsatz von SSD-basierten Speichersystemen, wie FlashStore [6] oder BloomStore [7]. Diese Systeme speichern die Daten auf SSD oder Flashspeicher und verwalten einen Index auf die Daten im RAM. Zusätzlich werden häufig nachgefragte Daten ebenfalls im RAM vorgehalten, um Zugriffe zu beschleunigen. Jedoch sind diese Systeme immer noch zu langsam und werden oftmals durch weitere Caching-Lösungen ergänzt. Da viele Anwendungen über keine Datenlokalität verfügen, müssen Caches sehr groß sein und sehr viele Daten beinhalten, weswegen dieser Ansatz nur zu einem gewissen Maße hilft und zusätzliche Synchronisierung von Cache und flash-basiertem Speichersystem erfordert.

RAM-basierte Cache-Lösungen erlauben einen schnellen Zugriff auf regelmäßig benötigte Daten und können den Bedarf dieser Anwendungen teilweise decken. So verwendete Facebook lange Zeit das weit verbreitete Memcached-System um 75% des Datenbestandes (mehr als 150 TB) in rund 1.000 Servern zu halten und so die Anforderungen der mehr als eine Milliarden interaktiver Nutzer zu gewährleisten [8, 9]. Inzwischen hat Facebook mit TAO ein eigenes Cachesystem entwickelt, das besser auf den zugrunde liegenden sozialen Graphen angepasst ist [5]. Suchmaschinen sind ein weiteres Beispiel solcher Anwendungen, bei denen Suchergebnisse für eine bessere Reaktionszeit im RAM gecacht werden [10, 11, 12]. Cache-Lösungen erfordern jedoch, dass Programmierer sich um die Synchronisierung von Cache und Hintergrundspeicher kümmern müssen. Darüber hinaus, müssen die Caches für irreguläre Zugriffsmuster (wie beispielsweise in sozialen Netzwerken) sehr groß sein und trotz dessen sind Cachefehler teuer. Letztendlich ist besonders das Neubefüllen eines Caches nach einem Ausfall sehr zeitaufwendig und somit kostspielig. So führte beispielsweise im September 2010 ein Softwarefehler bei Facebook zur Löschung von 28 TB an Cachedaten, der die Internetseite für 2,5 Stunden unbenutzbar machte, während die Caches von den wesentlich langsameren Datenbank-Servern neu befüllt wurden [13]. Die genannten Punkte erschweren es das gesamte Potenzial des RAM voll auszunutzen.
Der nächste sinnvolle Schritt ist daher zunehmend Anwendungsdaten komplett in den RAM zu verlagern [14] und SSDs und Festplatten nur noch für die Persistenz zu verwenden. Abbildung 1.1 zeigt die typischen Zugriffslatenzen auf unterschiedlichen Stufen der Speicherhierarchie. Dabei fällt besonders auf, dass der Zugriff auf den Hauptspeicher 10.000-mal schneller ist als auf eine Festplatte und 1.000-mal schneller als auf eine SSD. Der Zugriff auf entfernten Hauptspeicher ist langsamer als auf lokalen RAM, aber immer noch schneller als auf eine SSD

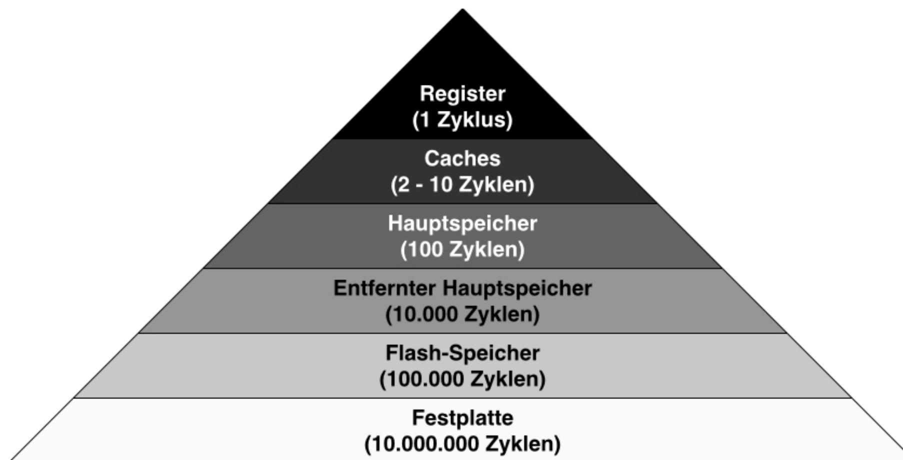


Abbildung 1.1: Speicherhierarchie. *Jede Stufe zeigt die typischen Zugriffslatenzen in CPU-Zyklen.*

und Festplatte.

Ein weiterer wichtiger Aspekt ist die Größe der gespeicherten Daten. Facebook beispielsweise verwaltet für seine über eine Milliarden Nutzer rund 200 TB an Daten [9]. Rund 70% dieser Daten sind kleiner als 64 Byte [4, 8], was dazu führt, dass jeder Server mehrere hundert Millionen Objekte verwalten muss. Eine ebenfalls große Anzahl an kleinen Objekten lässt sich auch in vielen Graphalgorithmen und -systemen finden, zum Beispiel [15].
Die große Menge an kleinen Objekten stellt besondere Anforderungen an die Speicher- und Metadaten-Verwaltung. Jedes Byte zusätzlicher Metadaten für ein Objekt potenziert sich so schnell auf mehrere Gigabyte zusätzlichen Speicherbedarf für das gesamte Speichersystem. Auch das traditionelle Alignement von Speicherallokatoren (beispielsweise an 32- oder 64-Byte Grenzen) und feste Allokationsklassen können zu einem erheblichen Speichermehraufwand führen. Da RAM vergleichsweise teuer ist, sind die Möglichkeiten den verfügbaren Speicher durch mehr Hauptspeicher oder weitere Knoten zu erweitern begrenzt und erfordert eine kompakte und speichereffiziente Verwaltung der Objekte.

RAMCloud war eines der ersten Projekte mit dem Ziel alle Anwendungsdaten permanent im verteilten RAM zu halten und somit einen sehr schnellen Datenzugriff zu ermöglichen [16]. Ein transparentes Hintergrundprotokoll und ein Fast-Recovery-Ansatz gewährleisten dabei Persistenz und Fehlertoleranz gegenüber Knotenausfällen. Objekte werden auf potentiell vielen Knoten verteilt und in Tabellen verwaltet, wobei ein zentraler Koordinator die Metadaten speichert und sich um die Objektsuche und die Koordinierung der Datenwiederherstellung kümmert. Die Objekte können zwischen ein paar Bytes und einem Megabyte groß sein. Der Metadaten-Aufwand für kleine Objekte ist jedoch sehr hoch und folglich eignet sich RAMCloud für viele der oben aufgeführten Anwendungen nicht.

DXRAM wird an der Heinrich-Heine-Universität Düsseldorf entwickelt und ist ebenfalls ein RAM-basiertes Speichersystem, das alle Anwendungsdaten permanent im RAM hält und von RAMCloud inspiriert wurde. Dabei werden zwar einige Ideen geteilt, aber an verschiedenen Stellen bestehen erhebliche Unterschiede. DXRAM ist für die Verwaltung von Milliarden sehr kleiner Datenobjekten (16-64 Byte) entwickelt worden, die in Schlüssel-Wert-Paaren verwaltet werden. Zielanwendungen sind dabei vor allem soziale Netzwerke und Graphanwendungen. Ziel ist es bis zu eine Milliarde Objekte pro Knoten (bei 32 GB Arbeitsspeicher pro Knoten) zu speichern. Ein eigener Speicherallokator und ein neuer Defragmentierungsalgorithmus sorgen dabei für die notwendige Speichereffizienz. Anstatt eines zentralen Koordinators wird ein Super-Peer-Overlay für die Verwaltung der Metadaten, die Überwachung der Knoten und die Koordinierung der Datenwiederherstellung eingesetzt. Für jede Objekt-ID muss ein Zuordnung zu einem Knoten verwaltet werden. Das bedeutet, dass für jeden Knoten bis zu eine Milliarde Zuordnungen existieren. Die Kombination von sequentieller ID-Erzeugung und globaler Metadaten-Verwaltung ermöglicht dabei die Zusammenfassung von mehrere Millionen Zuordnungen zu einer Einzigen. Aus Gründen der Persistenz und der Fehlertoleranz werden alle Daten durch eine asynchrone Hintergrund-Protokollierung auf mehrere Backupknoten auf SSD repliziert. Der verwendete Mechanismus weicht dabei an mehreren Punkten von der Protokollierung in RAMCloud ab. Erstens unterliegen Aktualisierungen einer strengen Ordnung, die es ermöglicht jederzeit die aktuellste Version eines Objektes zu ermitteln und dadurch die Datenwiederherstellung erheblich vereinfacht. Zweitens verwenden die Backupknoten mehrere Protokolle, in denen die Aktualisierungen mehrerer Knoten vorsortiert werden und bei einem Knotenausfall mit einer minimalen Anzahl an Zugriffen wieder eingelesen werden können. Und drittens wurde ein Protokoll-Layout entwickelt, das eine Maximierung des Lese- und Schreibdurchsatzes der SSDs ermöglicht. Vergleichbar mit RAMCloud wird ein Fast-Recovery-Ansatz für die Datenwiederherstellung verfolgt, der allerdings für die Behandlung von sehr vielen sehr kleinen Objekten optimiert ist und weniger darauf abzielt mit spezieller Hardware und Kernel-Optimierungen die Zeitgrenzen der Datenwiederherstellung zu verbessern.

1.2 Klassifizierung von Speichersystemen

Verteilte Speichersysteme können anhand mehrerer Kriterien klassifiziert werden. Nachfolgend wird eine einfache Klassifizierung verschiedener verteilter Speichersysteme vorgenommen. Die aufgeführten Klassen erheben keinen Anspruch auf Vollständigkeit, geben aber einen guten Überblick.

Verteilte Dateisysteme

Verteilte Dateisysteme existieren schon seit mehreren Jahrzehnten und erlauben den einfachen Zugriff auf Dateien über Rechnergrenzen hinweg. In der Regel besitzt jede Datei einen definierten Namen innerhalb eines hierarchischen Namensraumes und ist über den Pfad von der Wurzel des Namensraumes zum eigenen Eintrag eindeutig identifizierbar. Die meisten verteilten Dateisysteme lassen sich in das verwendete Betriebssystem einbinden und sind somit transparent für den Nutzer.

Viele verteilte Dateisysteme verfolgen einen Client-Server-Ansatz und speichern Daten auf lokalen Festplatten ab, beispielsweise Lustre [17], Coda [18], GlusterFS [19], etc. Einige Systeme verfolgen aber auch einen Peer-to-Peer-Ansatz, wie zum Beispiel IVY [20].

In den letzten Jahren haben vor allem verteilte Dateisysteme für große Datenmengen an Bedeutung gewonnen. So wird das Google File System (GFS) [21] in vielen Google-Produkten wie MapReduce, Google Maps, YouTube und Anderen eingesetzt und ist Grundlage für weitere Speichersysteme, wie beispielsweise BigTable [22]. Die Apache Foundation stellt mit dem Hadoop Distributed File System (HDFS) [23] und HBase [24] freie Implementierungen von GFS und BigTable zur Verfügung und nutzt diese im eigenen Hadoop System (MapReduce). Ein weiteres Beispiel ist BlobSeer [25], das es erlaubt bis zu ein Terabyte große Binärdaten zu speichern.

Häufig wird mit einem verteilten Dateisystem nicht ein komplettes System sondern nur ein Protokoll beschrieben, dass den Austausch von Dateien über Rechnergrenzen hinweg ermöglicht, zum Beispiel NFS [26] und CIFS [27].

Klassische Datenbanken

Datenbanken verwalten große Datenmengen und dienen zur dauerhaften Speicherung. Die Struktur der Daten hängt vom verwendeten Datenmodell ab und kann beispielsweise in Form von Tabellen, Graphen, Objekten, etc. erfolgen. Häufig werden die Daten in Dateien gespeichert und der Zugriff durch Indizes realisiert. Für den Zugriff auf die Daten wird in der Regel eine Abfragesprache (beispielsweise SQL) zur Verfügung gestellt, die es ermöglicht auch sehr komplexe Anfragen an die Datenbank zu stellen.

Datenbanken unterscheiden sich im Wesentlichen durch ihr Datenmodell voneinander. Relationale Datenbanken, wie beispielsweise DB2 [28], MySQL [29], H2 [30] und Oracle Database [31], organisieren Daten in Form von Tabellen, die über definierte Relationen miteinander verknüpft sind. Objektorientierte Datenbanken, zum Beispiel ObjectDB [32] oder Gemstone [33], verwalten Daten in Form von Objekten im Sinne der Objektorientierung. Ein Objekt kann dabei aus mehreren Datentypen zusammengesetzt sein und weitere Objekte beinhalten. Das Bindeglied zwischen relationalen und objektorientierten Datenbanken sind die sogenannten objektrelationalen Datenbanken, die Eigenschaften beider Datenbankmodelle verknüpfen, bei-

spielsweise PostgrSQL [34] oder Microsoft SQL Server [35]. Neo4j [36] und OrientDB [37] gehören zu den Graphdatenbanken, die Daten in Form von Knoten und Kanten abspeichern und spezielle Algorithmen beispielsweise für das Traversieren von Graphen bereitstellen.

NoSQL-Datenbanken

Der Begriff NoSQL-Datenbank ist etwas schwammig, bezeichnet aber zumeist Datenbanksysteme, die die Dominanz der relationalen Datenbanken brechen möchten und dafür auf viele Funktionalitäten klassischer Datenbanken verzichten und im Ausgleich dazu für spezielle Anwendungsfälle optimiert sind. So wird häufig die Abfragesprache SQL oder bestimmte Funktionsteile (beispielsweise Joins) nicht unterstützt, die verwendeten Abfragesprachen oder Schnittstellen haben aber meist Ähnlichkeit mit SQL. Die Daten werden beispielsweise in Schlüssel-Wert-Paaren (Redis [38], Dynamo [39], Scatter [40] und G-Store [41]), Dokumenten (MongoDB [42] und CouchDB [43]) oder zeilenorientierten Tabellen (Apache Cassandra [44]) organisiert.

Viele NoSQL-Datenbanken skalieren besser als klassische Datenbanken und bieten schnellere Zugriffszeiten, garantieren dafür aber beispielsweise keine ACID-Eigenschaften. Daneben existieren aber auch NoSQL-Datenbanken, die ACID-Eigenschaften und Transaktionen unterstützen, zum Beispiel Scalaris [45].

SSD-Speichersysteme

SSD-Speichersysteme sind für SSDs und Flashspeicher optimiert und ermöglichen so bessere Zugriffszeiten im Vergleich zu Speichersystemen, die Festplatten verwenden. SSDs haben nicht nur geringere Zugriffslatenzen, sondern erlauben auch eine höhere Datenrate und einen wahlfreien Zugriff. Trotz dessen bieten sequentielle Zugriffe mit einer vielfachen Größe einer Flash-Seite die beste Performanz. Im Vergleich zu Lesezugriffen sind Schreib- und Löschoperationen relativ teuer und erfordern jeweils das Neuschreiben oder Löschen ganzer Blöcke.

SSD-Speichersysteme verwenden SSDs entweder als Cache für langsamere Festplatten (FlashCache [46] und eNVy [47]), halten die Daten komplett auf SSDs (FlashStore [6], BloomStore [7] und Hyder [48]) oder kombinieren die Vorteile von SSDs und Festplatten (HybridStore [49]).

RAM-Caches

RAM-Caches halten häufig genutzte Daten oder Ergebnisse von früheren Anfragen im Hauptspeicher und vermeiden den Zugriff auf den langsameren Hintergrundspeicher (Datenbank, Dateisystem, etc.). Caches erfordern jedoch, dass Programmierer sich manuell um die Synchronisierung von Cache und Hintergrundspeicher kümmern. Da der Speicher im Regelfall

begrenzt ist und nur ein Teil der Daten im RAM vorgehalten werden kann, müssen beim Hinzufügen neuer Daten häufig bereits bestehende Daten entfernt werden. Memcached [50], Hazelcast [51] und Ehcache [52] sind Beispiele für RAM-Caches.

Memcached ist ein verteilter RAM-basierter Cache, der häufig von interaktiven Internetseiten eingesetzt wird. Memcached stellt für die Daten eine Hashtabelle zur Verfügung, bei der die Schlüssel bis zu 250 Bytes und die Werte bis zu 1 MB groß sein können [50]. Die Hashtabelle ist meist sehr groß, weswegen sie über mehrere Server in einem Rechenzentrum verteilt wird. Jeder Memcached-Client kennt alle Server und wendet sich immer an den Server, auf dem die notwendigen Daten liegen. Hierfür bestimmt der Client lokal einen Hashwert und kontaktiert den zugeordneten Server.

Hazelcast ist ein verteilter RAM-basierter Cache, der die Daten in Partitionen unterteilt und jedem Knoten mindestens eine Partition als Besitzer und eine Partition als Backup zuweist [53]. Damit wird die Last auf alle Knoten unterteilt und der Ausfall eines Knotens kann durch das Backup kompensiert werden. Ferner bietet Hazelcast Collections (ähnlich wie in Java), Sperren und Synchronisierungswerkzeuge, Trigger (Anwendungen werden bei bestimmten Ereignissen informiert) und die Möglichkeit Anwendungscode direkt auf den Daten auszuführen.

EhCache ist ein verteilter Cache, der sowohl RAM- als auch Festplattenspeicher unterstützt und sehr flexibel mit Hilfe von unterschiedlichen Modulen (Listeners, Cache Loader, Cache Extensions, etc.) erweitert werden kann [52]. Neben reinen Binärdaten erlaubt EhCache auch die Speicherung von nicht-serialisierbaren Objekten und bietet eine eigene Abfragesprache.

Neben diesen allgemeinen RAM-Caches existieren auch Caches für bestimmte Datenstrukturen oder Anwendungsfälle. So wurde zum Beispiel TAO [5] von Facebook entwickelt um gezielt Daten des sozialen Netzwerkgraphen zu cachen. Apache Spark [54] ist ein weiteres Beispiel, das große, vormals berechnete oder aggregierte Datensätze oder Teildatenbestände für schnelle Berechnungen im Hauptspeicher vorhält (nur Lesezugriffe).

Caching im RAM ist ein guter Ansatz und erlaubt in der Regel die langsamen Zugriffszeiten der Sekundärspeicher zu maskieren. Die in Abschnitt 1.1 beschriebenen Systeme besitzen allerdings kaum oder gar keine Lokalität und wahlfreie Zugriffe. Als Resultat müssen fast alle Daten im Cache vorgehalten werden. Und selbst dann sorgt schon eine 1%-ige Fehlschlagquote und die dadurch notwendigen langsameren Zugriffe auf den Sekundärspeicher für einen Performanzverlust um den Faktor 10 [16]. Aber auch wenn 100% der Daten im Cache gehalten werden, bieten Caches keine Persistenz und häufig können bei Knotenausfällen Daten verloren gehen, so dass immer noch manuell Cache und Sekundärspeicher synchronisiert werden müssen. Ferner erfordern besonders Schreibzugriffe, dass die Daten irgendwann vom Cache in den Sekundärspeicher gelangen, was sehr teuer ist. Daher erscheint der Ansatz alle Daten permanent im Hauptspeicher zu halten, und Persistenz über eine transparente Hintergrundprotokollierung zu erreichen, als zielführender.

RAM-basierte Datenbanken

RAM-basierte Datenbanken entsprechen in den meisten Aspekten und Funktionalitäten klassischen Datenbanken. Für schnellere Zugriffs- und Verarbeitungszeiten werden die Daten jedoch komplett oder zumindest überwiegend im Hauptspeicher gehalten. Eine transparente Hintergrundprotokollierung sorgt dabei für Persistenz und Beständigkeit (ACID-Eigenschaft). Viele RAM-basierte Datenbanken sind für die Echtzeit-Transaktionsverarbeitung (OLTP) oder Echtzeit-Analysen (OLAP) konzipiert, die im operativen Tagesgeschäft häufig in oder in der Kombination mit ERP- oder Data-Warehouse-Systemen eingesetzt werden. Zu diesen Datenbanken zählen beispielsweise SAP Hana [55], SanssouciDB [56], HyPer [57] oder H-Store [58]. RAM-basierte Datenbanken erlauben einen schnellen Zugriff auf die Daten, sind aber bedingt durch Transaktionen, ACID-Eigenschaften, Unterstützung von flexiblen Abfragesprachen, etc. in der Regel sehr komplex. Dies erschwert nicht nur den Einsatz, sondern verringert auch die Zugriffszeiten und den Durchsatz. Ferner ist die Verwendung von starren Datenmodellen (wie Tabellen) für viele Anwendungen zu unflexibel und daher ungeeignet.

RAM-basierte Speichersysteme

Als RAM-basierte Speichersysteme werden hier allgemein alle Speichersysteme bezeichnet, die Anwendungsdaten permanent im Hauptspeicher halten und nicht zu den Datenbanken oder Caches gehören. Die einzelnen Systeme unterscheiden sich zumeist sehr stark in ihren Anwendungsfällen und den damit verbundenen Daten. Die Größe einzelner Objekte kann von wenigen Byte (DXRAM [59] und Sedna [60]), über einige Kilobyte (RAMCloud [16]) bis zu mehreren Megabyte (FaRM [61]) reichen. Die Daten werden dabei in Form von Schlüssel-Wert-Paaren (DXRAM, Sedna, RAMCube [62] und Trinity [63]), Hashtabellen (FaRM) oder in Tabellen organisiert (RAMCloud). Die Anwendungsfälle sind teilweise sehr allgemein gehalten (RAMCloud) oder auf spezielle Szenarien zugeschnitten, wie beispielsweise Trinity für Graphalgorithmen und -analysen oder DXRAM für soziale Netzwerke und sehr große Graphen. Fast alle diese Systeme können den Speicher vieler Knoten aggregieren und sorgen durch eine transparente Hintergrundprotokollierung für Persistenz und Fehlertoleranz gegenüber Knotenausfällen.
RAMCloud ist eines der ersten Systeme, mit dem Ziel alle Anwendungsdaten jederzeit im Hauptspeicher zu halten, wozu der RAM hunderter oder tausender Knoten in einem Rechenzentrum aggregiert wird [16]. Besonderen Wert wird auf extrem geringe Latenzzeiten gelegt, die durch spezielle Hardware und Anpassungen von Netzwerkkarte und Kernel erreicht werden und Lesezugriffe in 5-10 Mikrosekunden ermöglichen [64]. Daten werden in Tabellen als Schlüssel-Wert-Paare gespeichert und je nach Tabellengröße auf einem oder mehreren Knoten gespeichert.
Trinity Graph Engine ist ein verteilter, RAM-basierter Schlüssel-Wert-Speicher für die Online-

Anfragebearbeitung und Offline-Analyse von großen Graphen [15]. Die Graphen können dabei aus Milliarden von Knoten bestehen, wie zum Beispiel in sozialen Netzwerken oder anderen Webgraphen [63]. Die Daten des Graphen werden in einer Speicher-Cloud im RAM vieler verteilter Knoten gespeichert. Dabei befinden sich nicht zwangsweise alle Daten im RAM, sondern vor allem die Topologie und häufig benutzte Daten werden im Speicher gehalten [65].
FaRM ist eine RAM-basierte verteilte Berechnungsplattform, die Remote Direct Memory Access (RDMA) für den rechnerübergreifenden Datenzugriff nutzt [61]. Gespeicherte Objekte sind von 64 Byte bis mehreren Megabyte groß und werden in einem Schlüssel-Wert-Speicher oder einem Graph-Speicher (ähnlich wie TAO) verwaltet. Der Datenzugriff erfolgt mit Transaktionen oder sperrfreien Lesezugriffen über RDMA, wofür nicht nur spezielle Hardware (Netzwerkkarten, Switches, etc.) zum Einsatz kommt, sondern auch das Betriebssystem und die Treiber der Netzwerkkarten modifiziert wurden.
RAMCube ist ein RAM-basierter Speicher, der auf die BCube [66] Netzwerk-Architektur aufsetzt und diese für Fehlerbehebung auf Netzwerkebene (Ausfall eines Knotens oder Switches) nutzt [62]. Daten werden in Tabellen, bestehend aus Schlüssel-Wert-Paaren gespeichert, wobei Schlüssel bis zu einem Kilobyte und Werte bis zu einem Megabyte groß sein können. RAMCube befasst sich überwiegend mit dem Problem der Fehlererkennung und -behebung in einem RAM-basierten Speicher. Dabei wird BCube verwendet, um Knoten über mehrere Netzwerkpfade zu erreichen und dadurch temporäre und permanente Knotenausfälle erkennen und unterscheiden zu können.
Sedna ist ein RAM-basierter Speicher für Echtzeit-Verarbeitung in der Cloud, der besonders auf Skalierbarkeit ausgerichtet ist [60]. Daten werden in Schlüssel-Wert-Paaren in einem hierarchischen Adressraum gespeichert. Trigger erlauben es geänderte Daten aktiv an Clients zu verteilen und benutzerdefinierte Aktionen auszuführen.

Für die in Abschnitt 1.1 betrachteten Anwendungen sind an erster Stelle die Klasse der RAM-basierten Speichersysteme relevant, da diese den notwendigen schnellen Zugriff bieten und unregelmäßige Zugriffsmuster durch die Eigenschaften des Hauptspeichers ermöglichen.

1.3 Ziel der Arbeit

Obwohl eine Vielzahl an verschiedenen Speichersystemen existiert, können besonders die Bedürfnisse großer interaktiver Web-Anwendungen nach schnellem Datenzugriff kaum gedeckt werden. Spezielle Lösungen bestehen zwar für unterschiedliche Anwendungsszenarien, allerdings ist die Verwaltung von sehr vielen sehr kleinen Objekten, wie sie beispielsweise in sozialen Netzwerken oder interaktiven Graphanwendungen auftreten, in RAM-basierten Systemen

kaum erforscht. DXRAM ist eines der ersten Systeme, das sich auf die kompakte und speichereffiziente Verwaltung von Milliarden sehr kleiner Objekte konzentriert.
Im Rahmen dieser Arbeit wird ein integriertes Konzept für die lokale und globale Metadaten-Verwaltung entwickelt, die explizit auf die Besonderheiten sehr vieler sehr kleiner Objekte zugeschnitten ist. Die dabei verwendete Speicherverwaltung soll einen eigenen Speicherallokator beinhalten, der den Aufwand für Metadaten minimiert und so die Unzulänglichkeiten bestehender Ansätze vermeidet. Zusammen mit einer effizienten lokalen Adressübersetzung von globalen IDs zu virtuellen Speicheradressen soll es ermöglicht werden, dass bis zu einer Milliarde Objekte pro Knoten (herkömmlicher Server) gespeichert werden können. Aufbauend auf den lokalen Strukturen soll eine globale Metadaten-Verwaltung in das Super-Peer-Overlay integriert werden, die es erlaubt die Metadaten für die Objektsuche und die Verwaltung der Backupknoten zu kombinieren und diese möglichst kompakt zu verwalten. Da die Super-Peers Metadaten für alle Peers bereithalten, ist es notwendig, dass jeder Super-Peer mehrere Milliarden Einträge verwalten kann. Die Metadaten sollen so organisiert werden, dass jederzeit der Ausfall eines Peers oder Super-Peers kompensiert werden kann und das Gesamtsystem weiterhin einsatzbereit bleibt.

1.4 Beiträge dieser Arbeit

Die in dieser Dissertation vorgestellten Konzepte wurden in Form von Beiträgen bei verschiedenen Konferenzen vorgestellt und publiziert.

1. Die Gesamtarchitektur von DXRAM und der neuartige Ansatz für sehr viele sehr kleine Objekte wurde auf der PDCAT13 publiziert [59]. Wesentlicher Inhalt waren die grundlegenden Architekturentscheidungen für die lokale und globale Metadaten-Verwaltung und die transparente Hintergrundprotokollierung.
2. Die lokale Metadaten-Verwaltung wurde detaillierter auf der IEEE Cluster14 publiziert [67]. Dabei ging es hauptsächlich um die Speicherverwaltung und Defragmentierung für sehr viele sehr kleine Objekte, die eine hohe Speichereffizienz ermöglicht, sowie um den schnellen Adressübersetzung mithilfe hierarchischer Tabellen.
3. Die globale Metadaten-Verwaltung wurde grundlegend auf den Informatiktagen14 [68] und detaillierter auf der BigDataCloud15 [69] publiziert. Schwerpunkt war der integrierte Ansatz von globaler und lokaler Metadaten-Verwaltung im Zusammenspiel mit der sequentiellen ID-Erzeugung und der Zusammenfassung von Metadaten zu ID-Bereichen. Hierbei ging es auch um die hohe Speichereffizienz und die schnelle Objektsuche.

1.5 Struktur der Arbeit

In Kapitel 2 wird mit DXRAM der Hintergrund für das in dieser Arbeit vorgestellte integrierte Konzept aus lokaler und globaler Metadaten-Verwaltung eingeführt. Neben einer allgemeinen Beschreibung der Komponenten werden substanzielle Eigenschaften des Speichers und mögliche Zielanwendungen erläutert.
Die lokale Metadaten-Verwaltung wird detailliert in Kapitel 3 dargestellt. Dabei wird zuerst die Adressübersetzung von globalen IDs zu virtuellen Speicheradressen geschildert und anschließend die Speicherverwaltung, inklusive eigenem Speicherallokator, ausgeführt. Abschließend werden mehrere Optimierungen vorgestellt, die eine Nutzung der vorgestellten Konzepte mit mehreren parallelen Threads ermöglichen.
Kapitel 4 beschäftigt sich mit der globalen Metadaten-Verwaltung, bei dem ein Super-Peer-Overlay für die Objektsuche und die Koordinierung der Datenwiederherstellung eingesetzt wird. Der Fokus liegt besonders auf der Verwaltung der Metadaten-Einträge in Form von ID-Bereichen. Zuletzt wird der Ansatz für den Namensdienst beschrieben, um Objekten benutzerdefinierte Schlüssel zuzuweisen.
Die vorgestellten Konzepte werden in Kapitel 5 ausführlich evaluiert und mit gebräuchlichen und modernen Ansätzen und Systemen verglichen.
Abschließend werden in Kapitel 6 alle vorgestellten Konzepte zusammengefasst, reflektiert und ein Ausblick auf mögliche weiterführende Arbeiten gegeben.

Kapitel 2

Architektur von DXRAM

RAM-basierte Speichersysteme sind in den letzten Jahren immer populärer geworden. Insbesondere große interaktive Webanwendungen nutzen RAM-Caches, wie das weit verbreitete Memcached System[1]. So hält zum Beispiel Facebook 75% seiner Datenbasis (ca. 150 TB) in rund 1.000 Memcached Servern, um den notwendigen schnellen Datenzugriff für die Anforderungen der interaktiven Nutzer zu erfüllen [9]. Rund 70% dieser Objekte sind kleiner als 64 Byte [4, 8], so dass insgesamt mehrere Milliarden Objekte gespeichert werden müssen. Ähnlich viele kleine Objekte existieren auch bei anderen sozialen Netzwerken wie Twitter [70] und Instagram [71]. Auf die Objekte wird überwiegend lesend zugegriffen und nur selten werden Objekte aktualisiert oder gelöscht [4, 5]. Die große Anzahl Objekte solcher Größe und lesedominante Zugriffsmuster finden sich auch in vielen Graphanwendungen, zum Beispiel [72].

Ein erster Ansatz zur Verbesserung der Performanz ist der Einsatz von SSD-basierten Speicherlösungen, wie z.B. FlashStore [6] oder BloomStore [7], die einen Index bzw. Hashtabelle im RAM halten um den Datenzugriff zu beschleunigen. Dennoch sind diese Systeme immer noch zu langsam für viele Anwendungen und werden daher häufig mit Middleware-Caching-Lösungen (Memcached, Hazelcast[2], Gemfire[3], etc.) kombiniert.

Die Nutzung des RAM lediglich als Cache bürdet den Programmierern die Verantwortung auf Caches und Sekundärspeicher zu synchronisieren. Bei unregelmäßigen oder gar zufälligen Zugriffsmustern, wie sie beispielsweise in sozialen Netzwerken auftreten, müssen die Caches sehr groß sein und trotzdem treten Cachefehler (Cache Misses) auf, die sehr teuer sind. Ferner ist das Neubefüllen der Caches nach einem Fehler sehr zeitaufwendig. So dauerte beispielsweise das Neubefüllen der Caches bei Facebook im September 2010 ca. 2.5 Stunden, weswegen die Seite unbenutzbar war [13]. Ein Softwarefehler sorgte dafür, dass die Caches geleert wurden

[1] `http://www.memcached.org`
[2] `http://hazelcast.com`
[3] `http://www.vmware.com/products/vfabric-gemfire/overview`

und von (wesentlich langsameren) Datenbankservern neu befüllt werden mussten. All diese Aspekte sorgen dafür, dass die volle Performanz des RAM nicht genutzt werden kann.
RAMCloud war eines der ersten Projekte, welches sich mit diesen Herausforderungen auseinandergesetzt hat, indem es alle Anwendungsdaten permanent im RAM hält [16]. Eine transparente Hintergrundprotokollierung auf Festplatte kombiniert mit einem Fast-Recovery-Ansatz sorgen für die notwendige Persistenz und schnelle Wiederherstellung der Daten im Fehlerfall.

DXRAM wurde von einigen Ideen RAMClouds inspiriert, zielt aber auf andere Anwendungsdomänen ab. Es ermöglicht die Speicherung von Milliarden kleiner Binärobjekte (16-64 Byte) in Form von Schlüssel-Wert-Paaren. Um alle Objekte permanent im Hauptspeicher zu halten, werden die Ressourcen von gegebenenfalls vielen Knoten aggregiert. Dabei wird davon ausgegangen, dass jeder Knoten mindestens 32 GB Hauptspeicher zur Verfügung hat, was zur aktuellen Zeit keine Einschränkung in einem Rechenzentrum darstellt. Das System ist sowohl für die Ausführung von Algorithmen auf den Speicherknoten als auch für den Einsatz als extrem schneller Backend-Speicher konzipiert.
Jeder Speicherknoten speichert bis zu einer Milliarde Objekte des Gesamtsystems. Die Metadaten der Objekte sind in einem Super-Peer-Overlay auf dedizierten Super-Peers gespeichert. Neben der Metadaten-Verwaltung übernehmen die Super-Peers auch die Koordinierung der Datenwiederherstellung und die Überwachung ihrer Peers. Um die Performanz der Objekte zu gewährleisten und einen Knotenausfall zu maskieren, wird jedes Objekt asynchron auf weitere Knoten auf Flash-Speicher repliziert. Standardmäßig werden die Daten eines Knoten auf drei weitere Knoten repliziert, die Anzahl ist jedoch konfigurierbar und kann bei Bedarf angepasst werden. Die Replikate eines Knotens sind dabei auf gegebenenfalls vielen Knoten verteilt. Dies ermöglicht eine schnelle Wiederherstellung der Daten, indem die Daten parallel auf allen Knoten wiederhergestellt werden.

Neben den bereits erwähnten Problemen müssen verteilte Speichersysteme auch das CAP Dilemma betrachten. Das CAP Dilemma besagt, dass von den drei Eigenschaften Konsistenz, Verfügbarkeit und Toleranz gegenüber Netzwerkpartitionen lediglich zwei gleichzeitig erfüllt sein können [73, 74]. Netzwerkpartitionierungen sind in Rechenzentren nur sehr selten, können aber auftreten. DXRAM nutzt den fehlertoleranten Koordinierungsdienst ZooKeeper[4] für den Systemstart (Bootstrapping) und die Integration neuer Knoten. Darüber hinaus wird ZooKeeper für die Erkennung einer Netzwerkpartitionierung genutzt. Wenn eine Partitionierung auftritt, kann nur noch ein Teil der Knoten ZooKeeper erreichen und daran die Partitionierung erkennen (siehe Abbildung 2.1).
DXRAM kann abhängig von den Anwendungsanforderungen so konfiguriert werden, dass die Konsistenz abgeschwächt oder die Verfügbarkeit einzelner Objekte, im Falle einer Netzwerk-

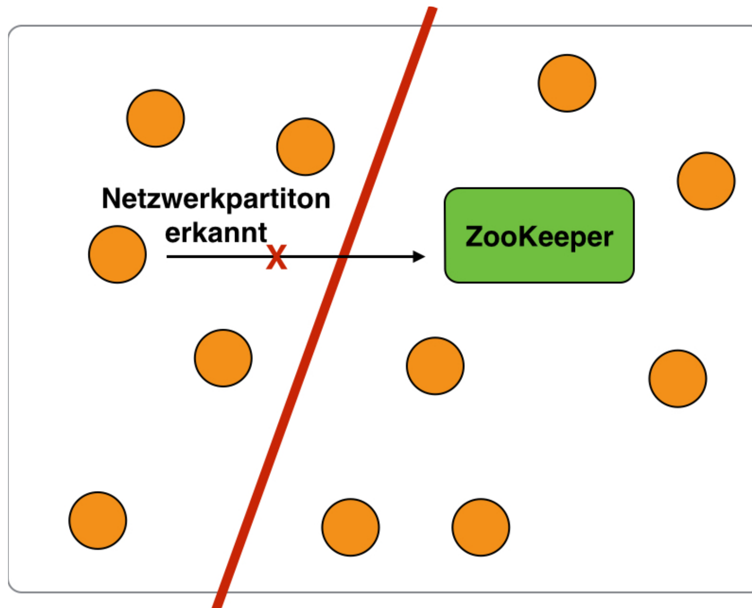


Abbildung 2.1: Erkennung einer Netzwerkpartitionierung. *Eine Netzwerkparitionierung wird daran erkannt, dass ein Teil der Knoten ZooKeeper nicht mehr erreichen können.*

partition, blockiert wird. Allerdings kann selbst bei einer Abschwächung der Konsistenz die Verfügbarkeit aller Objekte nicht sichergestellt werden, da es passieren kann, dass Objekte inklusive der drei Replikate in der anderen Partition liegen und somit nicht erreichbar sind.

In den nachfolgenden Abschnitten werden die Dienste und Komponenten von DXRAM kurz beschrieben. Die für DXRAM besonders relevanten Komponenten (Super-Peer-Overlay, RAM-Management und Backup-Management) werden anschließend noch tiefergehend betrachtet und wurden in [59] und [75] publiziert.

2.1 Dienste und Komponenten

Die Dienste und Komponenten von DXRAM lassen sich in die beiden Kategorien Erweiterte Dienste und Kern-Dienste unterteilen. Abbildung 2.2 zeigt die wichtigsten Dienste und Komponenten. Anwendungen, die DXRAM nutzen, können auf die Dienste und Komponenten sowohl der Kern-Dienste als auch der Erweiterten Dienste zugreifen.

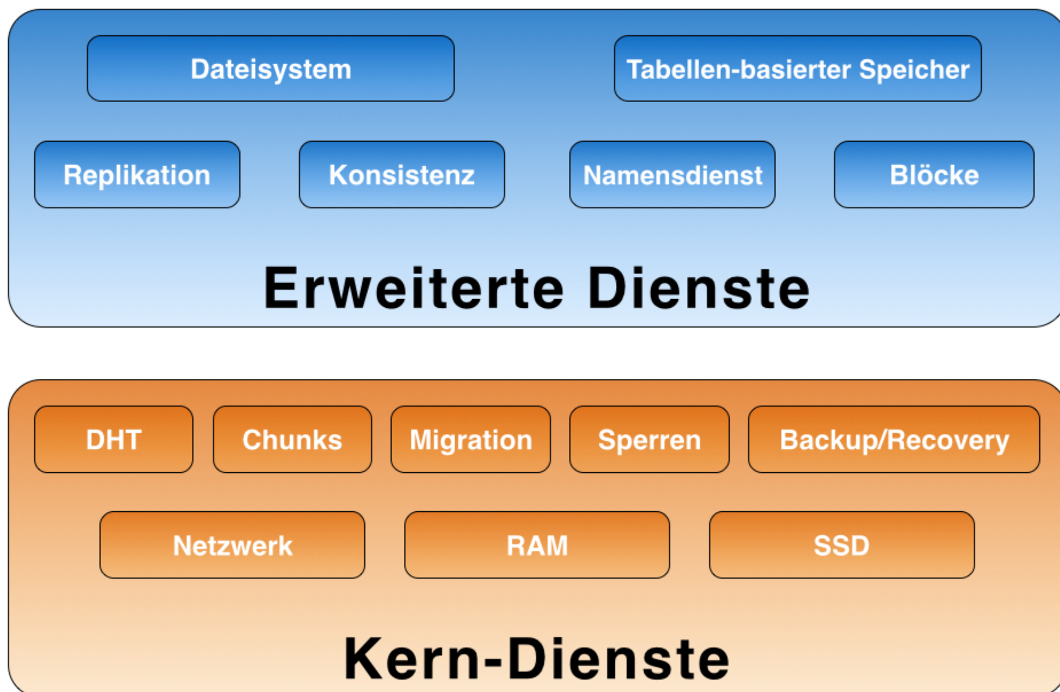


Abbildung 2.2: Dienste und Komponenten von DXRAM. *Die Erweiterten Dienste bauen auf Kern-Diensten auf, die die grundlegende Funktionalität zur Verfügung stellen.*

2.1.1 Erweiterte Dienste

Die Erweiterten Dienste umfassen generelle Dienste und erweiterte Datenmodelle, die auf den Kern-Diensten aufsetzen. Replikation (aus Performanzgründen), Konsistenzmodelle und Namensdienst-Implementierungen sind hier angesiedelt. Zu den hier angeordneten Datenmodellen gehören eine Dateisystemschnittstelle und ein tabellenbasierter Speicher, die das zugrunde liegende Datenmodell erweitern. Die Erweiterten Dienste benutzen *Blocks* zur Datenverwaltung, wohingegen die Kern-Dienste Schlüssel-Wert-Paare (*Chunks*) nutzen. Abhängig von der Konfiguration kann ein Block in einem einzelnen Chunk gespeichert oder auf mehrere Chunks aufgeteilt werden. Die Zuordnung zwischen Blocks und Chunks erfolgt in der jeweiligen Datenmodellimplementierung.

2.1.2 Kern-Dienste

Die Kern-Dienste stellen die Funktionalität zur Verwaltung, Speicherung und Transfer von Chunks bereit. Eines der Hauptziele ist es die Schnittstelle nach außen so kompakt wie möglich zu halten, um die Nutzung von DXRAM für die Programmierer zu vereinfachen. Die minimale Schnittstelle für Chunks beinhaltet die folgenden Funktionen:

Create: Erzeugt einen neuen Chunk der übergebenen Größe. Bereits bei der Erzeugung wird dem Chunk eine freie global eindeutige ID zugeordnet.

Remove: Löscht einen Chunk und gibt gegebenenfalls seine globale ID für eine erneute Benutzung frei.

Get: Holt einen Chunk an Hand seiner globalen ID, wobei sich der Chunk im lokalen RAM oder im Hauptspeicher eines anderen Knotens befinden kann.

Put: Aktualisiert die in DXRAM gespeicherten Chunkdaten. Liegt der Chunk nicht im lokalen Hauptspeicher, wird die Put-Operation auf dem Knoten der den Chunk speichert ausgeführt.

Recover: Stellt die Daten eines Knotens mithilfe der Knoten-ID wieder her. Dazu werden die Daten von der lokalen SSD eingelesen und im Hauptspeicher wieder hergestellt. Die Operation ist für den Einsatz nach einem Neustart oder Absturz des Systems durch einen Stromausfall gedacht.

Lock: Sperrt einen Chunk für den exklusiven Zugriff. Die Put- und Get-Operationen werden unabhängig von einer Sperre ausgeführt, so dass eine korrekte Synchronisierung immer voraussetzt, dass alle Zugriffe die Sperre verwenden. Neben der exklusiven Sperre wird auch eine Lese-Schreib-Sperre unterstützt.
Beim Aufruf der Lock-Operation wird implizit eine Get-Operation durchgeführt und der gesperrte Chunk zurückgegeben.

Unlock: Entsperrt einen Chunk wieder. Statt des Unlock-Aufrufs kann auch direkt bei einer Put-Operation der entsprechende Chunk entsperrt werden.

Die Funktionalitäten umfassen unter anderem die Verwaltung von RAM und Flash-Speicher, die Netzwerkkommunikation, die Daten- und Metadaten-Verwaltung und die Backup- und Datenwiederherstellungs-Koordinierung. Diese Dienste werden in den folgenden Abschnitten detaillierter beschrieben. Ferner existieren Dienste für Sperren und Migration von Chunks. Der erstere Dienst dient zur Kontrolle von Wettlaufsituationen und kann zur Implementierung von starken Konsistenzmodellen verwendet werden. Der zweite Dienst wird zur Lastverteilung eingesetzt. Da die Daten auf jedem Knoten im Hauptspeicher liegen und dadurch der Datenzugriff überall schnell ist, dienen Migrationen lediglich zur Auflösung von Hotspot-Häufungen auf einzelnen Knoten. Es kann zum Beispiel in einem sozialen Netzwerk passieren, dass mehrere berühmte Pop Stars mit tausenden Freunden (manche Künstler haben bis zu 40 Millionen Freunde / Follower [63]), auf einem einzelnen DXRAM Knoten gespeichert sind. Insgesamt existieren aber nicht Millionen von beliebten Pop Stars.
Wie bereits erwähnt, verwalten die Kern-Dienste Binärdaten in Form von Chunks. Jeder Chunk

besteht aus einer global eindeutigen ID, der *ChunkID* oder *CID*, und den eigentlichen Binärdaten. Die CID ist ein 64-Bit Wert, der aus der Knoten ID des erzeugenden Knotens (kurz NID_C) und einer lokal eindeutigen ID (kurz *LID*) besteht. Die NID_C ist 16 Bit lang und identifiziert global eindeutig den Erzeuger des Chunks. Ein Chunk ist nicht zwangsweise auf seinem Erzeuger gespeichert, sondern kann auch auf einem anderen Knoten liegen. Die LID ist ein lokal eindeutiger 48-Bit Wert, der bei jeder Chunkerzeugung intern um 1 inkrementiert wird. Die Bitgrößen von NID_C und LID erlauben DXRAM 65.536 Knoten zu adressieren, die jeweils bis zu 2^{48} ($\sim$ 280 Billionen) Chunks erzeugen können. Die Verwendung eines 64-Bit Wertes für die CID ermöglicht die einfache Verwaltung durch einen Long Datentyp. Grundsätzlich ist es aber möglich die CIDs zu erweitern, falls zukünftig mehr Knoten oder Chunks unterstützt werden sollen. Die Entscheidung für diesen Ansatz, anstatt benutzerdefinierter Schlüssel, bietet einige Vorteile. Zum einen erlaubt uns der Ansatz den ID-Raum kompakt zu halten und unterstützt lokalitäts-basierte Zugriffsmuster. Zum anderen verhindert er die Anpassung von Hashfunktionen im Fall von Knotenbeitritten und -austritten. Abgesehen davon ist die sequentielle ID-Erzeugung keine Einschränkung für Anwendungen. Die meisten Anwendungen, die Datenbanken nutzen, greifen auf die Daten mit Hilfe von auto-inkrementierten Tabellenschlüsseln zu (ähnlich zu den LIDs in DXRAM). Ferner verfügt DXRAM über einen Namensdienst, der es ermöglicht auf eine CID über einen benutzerdefinierten Schlüssel zuzugreifen. Die Intention ist, dass nicht jedes Objekt einen benutzerdefinierten Schlüssel benötigt, sondern lediglich eine Teilmenge, beispielsweise die Benutzer-Datensätze eines sozialen Netzwerkes.

Apache ZooKeeper wird für die Erzeugung und Zuordnung von NIDs verwendet. ZooKeeper bietet eine fehlertolerante Verwaltung von hierarchischen Schlüssel-Wert-Paaren mit starker Konsistenz. Wenn ein bestehender DXRAM Knoten abstürzt, kann die zugehörige NID sicher einem anderen beitretenden Knoten oder dem alten Knoten (nach einem Neustart und Wiederbeitritt) zugewiesen werden. Da DXRAM für Rechenzentren ausgelegt ist, werden allerdings Knotenbeitritte und -austritte nicht so häufig auftreten, so dass ZooKeeper keinen Flaschenhals darstellt.

Chunks haben variable Größen, die bei der Erzeugung festgelegt werden, und werden immer am Stück gespeichert. Auf Grund der begrenzten Kapazität des RAM werden Chunks aus Fehlertoleranzgründen nur auf Flash-Speicher von mehreren *Backupknoten* repliziert. Um den Netzwerktransfer von Backupreplikaten und Chunks in angemessener Zeit durchzuführen, ist die maximale Größe von Chunks auf 16 MB beschränkt. Größere Daten können mit Hilfe der Datenmodelle der Erweiterten Dienste realisiert werden. Dies erlaubt beispielsweise die Bündelung mehrerer Chunks zu größeren logischen Blöcken. Der Hauptfokus von DXRAM liegt allerdings bei der Verwaltung von sehr vielen sehr kleinen Objekten.

DXRAM verwendet ein Super-Peer-Overlay für die Metadaten-Verwaltung der Chunks. Die Super-Peers implementieren eine DHT, die eine schnelle Objektsuche für eine gegebene CID

ermöglicht. Die Super-Peers können beim Start festgelegt oder dynamisch, auf Basis der Topologie des Rechenzentrums, ausgewählt werden. Eine Anordnung der Super-Peers mit Berücksichtigung der Rackzuordnung verhindert den Ausfall mehrerer Super-Peers bei einem Totalausfall eines Racks. Zur Entlastung der Super-Peers werden die Ergebnisse von Objektsuchen auf den Peers gecacht. Darüber hinaus kümmern sich die Super-Peers um die Überwachung ihrer Peers und um die Koordinierung der Datenwiederherstellung. Die Wiederherstellung ausgefallener Knoten muss koordiniert werden, da die Backupreplikate über viele Backupknoten verteilt sind. Die Verteilung auf viele Backupknoten ermöglicht das parallele Wiederherstellen von sehr vielen Daten innerhalb von Sekunden. Die Verteilung der Backup- und Wiederherstellungslast auf die Backupknoten erfolgt mit Hilfe von so genannten *Backupzonen*, zum Beispiel 64-256 MB, für die jeweils drei Backupknoten zuständig sind. Der genutzte Hauptspeicher jedes Peers wird in entsprechende Backupzonen unterteilt, so dass beispielsweise 32 GB RAM in 128 Backupzonen unterteilt werden, die auf bis zu 384 Backupknoten verteilt sind. Schlussendlich wird bei der Wiederherstellung jede Backupzone von einem der zugehörigen Backupknoten wiederhergestellt.

2.2 Super-Peer-Overlay

Rund 5-10% der zur Verfügung stehenden DXRAM Knoten werden als dedizierte Super-Peers verwendet, die eine DHT ähnlich zu Chord realisieren [76]. Wenn beispielsweise 1.024 Knoten mit jeweils 32 GB RAM zur Verfügung stehen, würden 64 Knoten als Super-Peers eingesetzt. Demnach stehen 2 TB für die Metadaten-Verwaltung und 28,8 TB für Anwendungsdaten zur Verfügung. Wie im vorherigen Abschnitt beschrieben, werden die Super-Peers mit Rücksicht auf die Racks bestimmt. Die NIDs, die den Super-Peers beim Systemstart zugewiesen werden, sind in ZooKeeper hinterlegt und lassen sich über eine Konfigurationsdatei einstellen. Durch diese Konfigurationsdatei kennen sich die Super-Peers gegenseitig und können bei Bedarf eine Verbindung zueinander aufbauen. Im Gegensatz zu DHTs im Internet kann somit eine Objektsuche in $\mathcal{O}(1)$ durchgeführt werden.

2.2.1 Objektsuche

Die Super-Peers sind aufsteigend nach NID geordnet und jeder Super-Peer SP_i ist zuständig für alle Knoten und deren Chunks im Bereich $SP_{i-1} \leq CID < SP_i$, siehe Abbildung 2.3. Wie bereits erwähnt, sind Super-Peers für die schnelle Objektsuche zuständig und speichern dafür Zuordnungen von CIDs zu NIDs. Für die Objektsuche wird eine *NID$_C$-Tabelle* verwendet, die für jede mögliche NID einen Eintrag besitzt. Bei 2^{16} Knoten und 64-Bit großen Einträgen

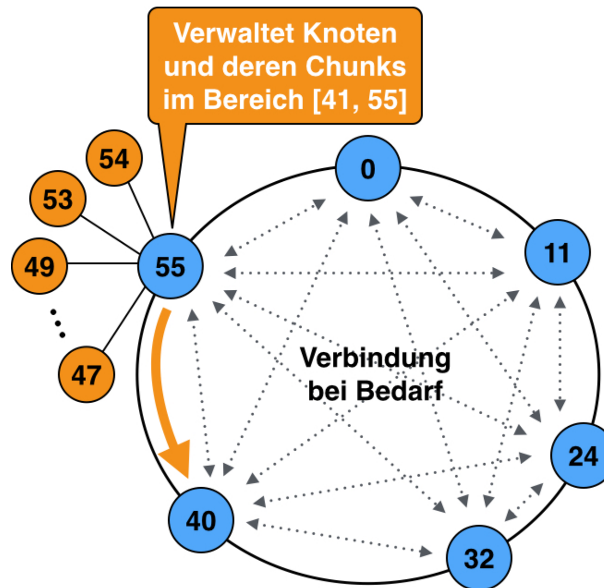


Abbildung 2.3: Super-Peer-Overlay. *Jeder Super-Peer SP_i ist zuständig für alle Knoten und deren Chunks im Bereich $SP_{i-1} \leq CID < SP_i$. Die Super-Peers kennen sich gegenseitig und können bei Bedarf eine Verbindung zueinander aufbauen.*

benötigt die Tabelle 512 KB RAM. Jeder belegte Eintrag verweist auf einen *CID-Baum*, der alle CIDs speichert, die vom Knoten mit der entsprechenden NID_C erzeugt wurden (das C steht für Creator) und existiert lediglich für die Knoten, für die der Super-Peer zuständig ist. Bei dem Beispiel mit 1.024 Knoten und 64 Super-Peers ist jeder Super-Peer für durchschnittlich 15 Peers zuständig und speichert somit 15 CID-Bäume. Bei einer Objektsuche wird die CID der Anfrage in NID_C und LID aufgeteilt. Die NID_C wird verwendet, um in der NID_C-Tabelle den zugehörigen CID-Baum auszuwählen. Die LID wird schließlich verwendet, um den passenden Eintrag im CID-Baum zu ermitteln. Die Struktur und Funktionalität des CID-Baumes wird im Kapitel 4 ausführlich beschrieben.

2.2.2 Knotenbeitritt

Ein neuer Knoten tritt dem System grundsätzlich als Peer bei, wie in Abbildung 2.4 gezeigt. Dazu kontaktiert der neue Knoten als erstes ZooKeeper und fordert eine NID an. ZooKeeper kennt alle freien NIDs und weist dem Knoten eine freie NID zu. Gleichzeitig wird dem Knoten durch ZooKeeper auch der zuständige Super-Peer für die NID mitgeteilt. Anschließend kontaktiert der Knoten den zuständigen Super-Peer und tritt als sein Peer dem System bei.

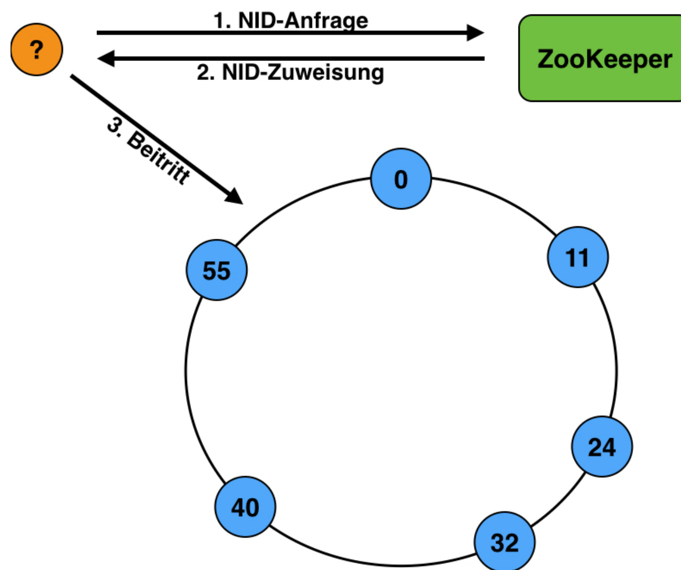


Abbildung 2.4: Knotenbeitritt. *Der Beitritt eines Knotens erfolgt in drei Schritten: 1. Der Knoten sendet eine NID-Anfrage an ZooKeeper. 2. ZooKeeper weist dem Knoten eine freie NID zu. 3. Beitritt über den zuständigen Super-Peer.*

2.2.3 Fehlertoleranz

In einer kontrollierten Umgebung, wie einem Rechenzentrum, ist keine hohe fehlerbedingte Knotenfluktuation zu erwarten. Natürlich kann ein Stromausfall zum gleichzeitigen Ausfall aller Knoten führen, was die Wiederherstellung aller Daten aus allen Protokollen erfordert und eine gewisse Zeit benötigt. Solch ein Fall kann aber als sehr selten angenommen werden. Abgesehen davon wird lediglich ein kontrolliertes Erhöhen und Vermindern der Knotenanzahl erwartet. Wenn dabei ein definierter Grenzwert über- bzw. unterschritten wird, werden neue Super-Peers bestimmt oder abgemeldet.

Ausfall eines Peers

Der Ausfall eines Peers erkennt der zugehörige Super-Peer durch ein regelmäßiges Heart-Beat-Protokoll. Alternativ kann auch ein anfragender Knoten den Ausfall bemerken. In diesem Fall wird der zuständige Super-Peer durch den anfragenden Knoten informiert und die freigewordene NID in ZooKeeper hinterlegt (siehe Abbildung 2.5). Der Super-Peer stößt anschließend die Wiederherstellung des ausgefallenen Peers an. Dazu kontaktiert er die relevanten Backupknoten und koordiniert die weitere Datenwiederherstellung. Nach der Wiederherstellung wird die (vom ausgefallenen Peer) zuletzt erzeugte CID ermittelt und in ZooKeeper hinterlegt. Die

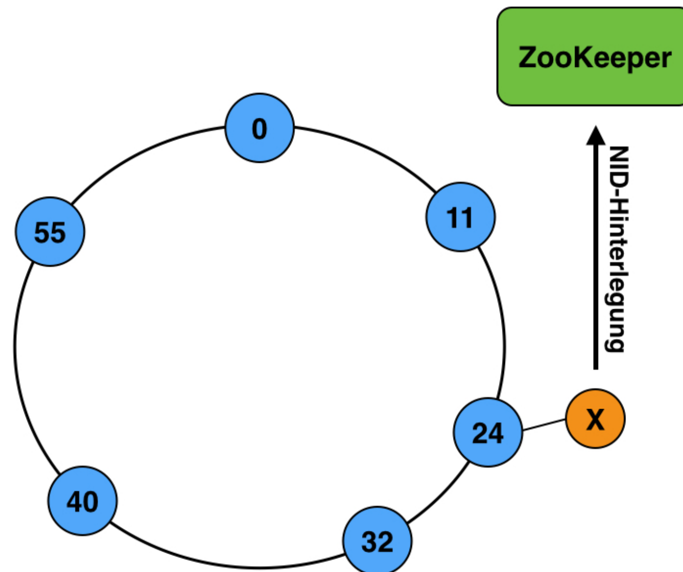


Abbildung 2.5: Ausfall eines Peers. *Beim Ausfall eines Peers wird die freigewordene NID in ZooKeeper hinterlegt, so dass diese erneut verwendet werden kann.*

letzte erzeugte CID wird benötigt, um CID Konflikte bei der Neuzuweisung der NID des ausgefallenen Peers zu vermeiden. Für die Datenwiederherstellung existieren drei grundlegende Ansätze:

1. Wiederherstellung der Daten auf einem neuen Knoten: Diese Strategie ist sehr langsam, da alle Daten nach dem Wiederherstellen zu einem einzelnen Knoten transferiert werden müssen. Bei 32 GB Daten und 10-Gigabit-Ethernet dauert alleine schon der Transfer der Daten ca. 30 Sekunden.

2. Wiederherstellung der Daten verteilt und parallel auf den Backupknoten: Diese Strategie ist am schnellsten, da alle Daten verteilt wiederhergestellt werden und direkt zur Verfügung stehen. Dabei stellen die Backupknoten parallel genau die Daten wieder her, die jeweils auf der lokalen SSD repliziert sind. Die Wiederherstellung der Daten erfolgt zwar schnell, kann aber dazu führen, dass die gegebenenfalls vorhanden Lokalität der Daten verloren geht. Die Daten können nur parallel wiederhergestellt werden, wenn die Backupknoten genügend freien Speicherplatz im RAM zur Verfügung haben.

3. Wiederherstellung der Daten verteilt und parallel auf den Backupknoten und anschließende Migration der Daten auf einen neuen Knoten: Diese Strategie ist eine Kombination der ersten beiden Strategien. Die Daten werden verteilt und parallel auf den Backupknoten wiederhergestellt und stehen direkt zu Verfügung. Im weiteren Verlauf werden die Daten asynchron zu einem neuen Knoten transferiert, um die Lokalität der Daten zu rekonstruieren. Hierfür kann

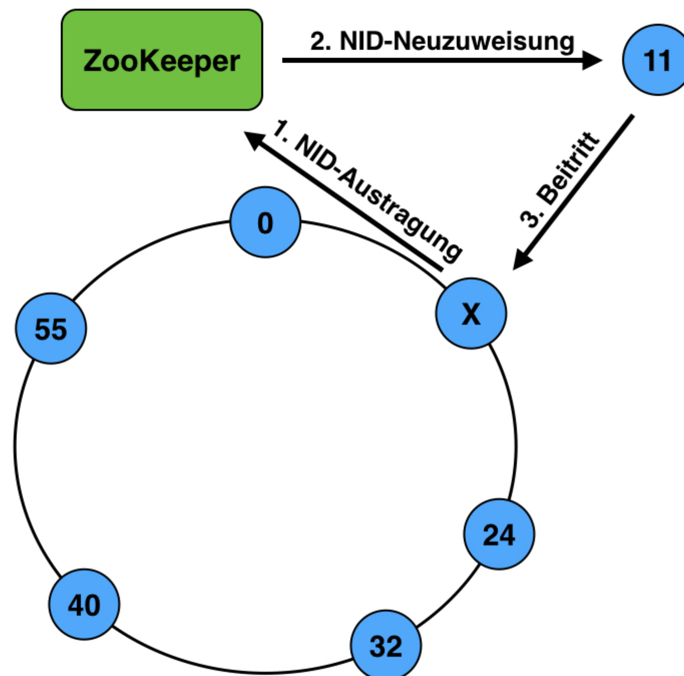


Abbildung 2.6: Ausfall eines Super-Peers. *Beim Ausfall eines Super-Peers wird die freigewordene NID in ZooKeeper hinterlegt. Die NID wird anschließend einem neuen Knoten zugewiesen, der den Platz des Super-Peers einnimmt.*

auch ein neuer Knoten verwendet werden, der in diesem Fall die NID des ausgefallenen Knotens erhält.

DXRAM unterstützt die zweite und dritte Strategie und überlässt der Anwendung die Passendste auszuwählen [75].
Abgesehen von den eigentlichen Anwendungsdaten müssen auch die Metadaten auf dem zugehörigen Super-Peer rekonstruiert werden. Dazu werden die Einträge im entsprechenden CID-Baum angepasst. Eine genaue Beschreibung des Prozesses findet sich in Kapitel 4.

Ausfall eines Super-Peers

Um den Ausfall eines Super-Peers zu maskieren, werden die gespeicherten Metadaten auf den nachfolgenden Super-Peers repliziert. Auch für die Wiederherstellung der Metadaten existieren zwei grundlegende Ansätze:

1. Wiederherstellung der Metadaten auf einem neuen Knoten: Bei dieser Strategie wird die NID des ausgefallenen Super-Peers durch ZooKeeper einem neuen Knoten zugewiesen und dieser zum Super-Peer befördert (siehe Abbildung 2.6). Anschließend bezieht der neue Super-Peer die Metadaten von seinem Nachfolger.

2. Wiederherstellung der Metadaten auf einem existierenden Peer: Diese Strategie wählt einen Peer des ausgefallenen Super-Peers aus und befördert diesen zum neuen Super-Peer. Da die Super-Peers keine Anwendungsdaten speichern, müssen die Daten des Peers vor der Beförderung auf andere Knoten migriert werden. Anschließend werden die Metadaten von einem der Nachfolger bezogen.

Offensichtlich ist die erste Strategie schneller und nicht so komplex wie die Zweite und beispielsweise in einer Cloudumgebung mit sehr vielen Ressourcen auch immer möglich. Bei beiden Strategien können während der Datenwiederherstellung Objektsuchen durch die nachfolgenden Super-Peers, mit Hilfe der replizierten Metadaten, durchgeführt werden.
Auch beim gleichzeitigen Ausfall mehrerer Super-Peers können die Metadaten noch rekonstruiert werden. In diesem unwahrscheinlichen Fall kontaktiert der neue Super-Peer seine Peers und fordert die gespeicherten gebündelt CIDs an. Dieser Prozess ist jedoch sehr aufwendig und sollte nach Möglichkeit vermieden werden.

2.3 RAM-Management

In diesem Kapitel werden verschiedene Strategien für die lokale Speicherverwaltung der im RAM gespeicherten Chunks vorgestellt. Die größte Herausforderung ist die effiziente Speicherung der Zuordnung von CIDs zu lokalen virtuellen Speicheradressen.

2.3.1 Wiederverwendung von CIDs

Entscheidend für die effiziente Speicherung ist die Frage, ob CIDs nach dem Löschen der entsprechenden Chunks wiederverwendet werden sollen oder nicht. Jeder Knoten kann lokal maximal 2^{48} ($\sim$ 280 Billionen) CIDs erzeugen. Wenn ein Knoten jede Sekunde 10^6 CIDs erzeugen würde, könnte er dies 8,9 Jahre lang, bevor keine CIDs mehr zur Verfügung stehen. Des Weiteren können so viele Chunks nicht gleichzeitig im Hauptspeicher gehalten werden. Zumindest aus theoretischer Sicht ist daher eine Wiederverwendung nicht notwendig.
Im Folgenden werden drei mögliche Ansätze beschrieben bei denen CIDs (teilweise) wiederverwendet beziehungsweise nicht wiederverwendet werden:

1. Wiederverwendung von CIDs: Durch die Wiederverwendung von CIDs bleibt der CID-Raum sehr kompakt und ermöglicht eine effiziente Verwaltung mit Hilfe von hierarchischen Tabellen. Die hierarchischen Tabellen ermöglichen gleichzeitig das schnelle Auffinden von freigegebenen CIDs. Besondere Aufmerksamkeit muss bei diesem Ansatz migrierten Chunks gewidmet werden. Wenn beispielsweise die CIDs [5, 10] von Knoten K_1 zu Knoten K_2 migriert wurden

und der Chunk mit der CID 7 gelöscht wird, muss K_1 über das Löschen informiert werden, damit die CID neu vergeben werden kann. Darüber hinaus hat dies auch Auswirkungen auf die Metadaten. In Abschnitt 4.3 wird erläutert, dass sich dadurch die Anzahl der Metadaten-Einträge erhöht. Offensichtlich ist die Wiederverwendung von CIDs, insbesondere von migrierten Chunks, sehr komplex, erfordert zusätzlichen Netzwerkverkehr und vergrößert die Metadaten.

2. Keine Wiederverwendung von CIDs: Wenn CIDs nicht wiederverwendet werden, wächst der CID-Raum mit der Zeit kontinuierlich an und die CIDs sind immer weiter verstreut. Eine effiziente Speicherung in einer hierarchischen Tabellenstruktur ist damit kaum möglich. Hashtabellen wären eine Alternative, benötigen aber für die Kollisionsauflösung zusätzlichen Speicherplatz. Es existieren weitere Datenstrukturen, zum Beispiel Array, Liste, Baum, Trie, etc., welche jedoch keinen Zugriff in $\mathcal{O}(1)$ erlauben und somit den schnellen Datenzugriff ausbremsen. Aus den genannten Gründen ist dieser Ansatz nicht zu empfehlen.

3. Teilweise Wiederverwendung von CIDs: Dieser Ansatz basiert auf dem Ersten, erweitert um einige Modifizierungen. Sowohl der Knoten, der einen Chunk erzeugt hat, als auch der Knoten, der den Chunk vorhält, können einen Chunk löschen. Die CID wird aber nur wiederverwendet, wenn der erzeugende Knoten den Chunk löscht. Wenn der Besitzer den Chunk löscht, wird das Löschen lediglich protokolliert, so dass der gelöschte Chunk bei der Datenwiederherstellung erkannt werden kann. Dieser Ansatz ist sehr interessant, da alle Nachteile des zweiten Ansatzes vermieden und gleichzeitig die Vorteile des ersten Ansatzes erreicht werden. Da nicht viele Migrationen erwartet werden, können dadurch die meisten CIDs wiederverwendet werden. Aus diesen Gründen implementiert DXRAM diese Strategie und verwendet für die Verwaltung der Zuordnungen hierarchische Tabellen.

2.3.2 Übersetzung von CIDs zu virtuellen Speicheradressen

Die Adressübersetzung der CIDs hat Ähnlichkeit mit der Adressübersetzung beim Paging bei PC-Betriebssystemen. Allerdings bleiben die Tabellen, durch die Wiederverwendung von CIDs, immer kompakt und erlauben damit eine schnelle Übersetzung. Die Adressübersetzung wird in Abbildung 2.7 dargestellt und in Abschnitt 3.2 ausführlich erläutert. An dieser Stelle soll lediglich das grundlegende Konzept erklärt werden.
Wie auch in der Metadaten-Verwaltung im Super-Peer-Overlay wird eine CID in NID- und LID-Teil zerlegt. Die NID wird als Index in eine NID_C-Tabelle verwendet. Im Gegensatz zu der NID_C-Tabelle eines Super-Peers verweisen die Einträge hier auf ein *CID-Verzeichnis*, dessen Einträge wiederum auf *CID-Tabellen* verweisen, usw. Als Index für das CID-Verzeichnis und die CID-Tabellen wird die LID verwendet. Dazu wird die LID in mehrere Teile gleicher

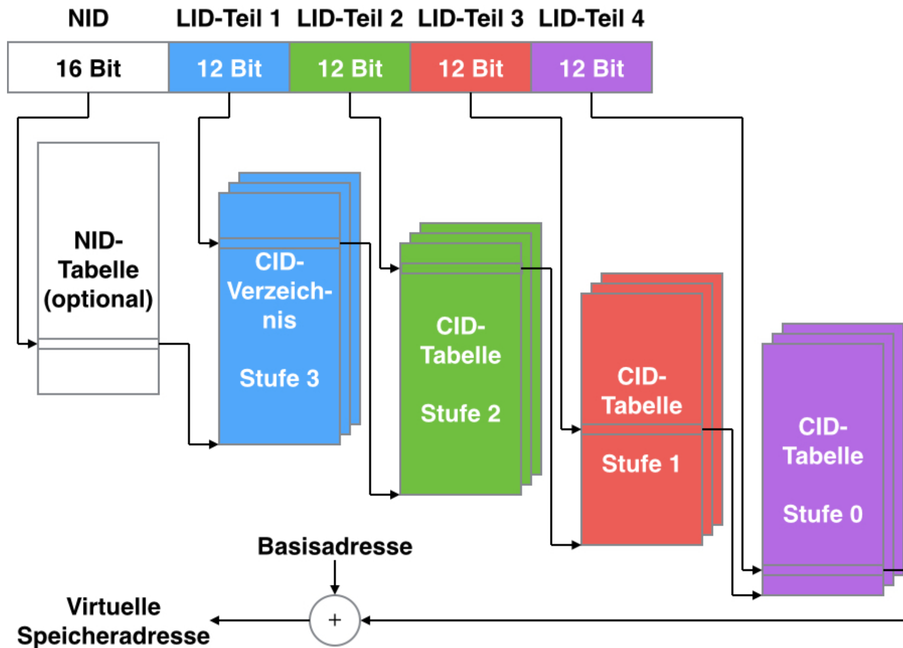


Abbildung 2.7: Adressübersetzung von CIDs. *Bei der Adressübersetzung kommen hierarchische Tabellen (ähnlich den Tabellen beim Paging) zum Einsatz.*

Größe unterteilt (standardmäßig vier Teile zu je 12 Bit). Der erste Teil wird als Index in das CID-Verzeichnis, die weiteren Teile als Indizes für die nachfolgenden CID-Tabellen verwendet. Wenn der Knoten nur seine eigenen Chunks speichert, kann die NID_C-Tabelle weggelassen werden, da nur ein CID-Verzeichnis benötigt wird. Sobald Chunks eines weiteren Knotens gespeichert werden, wird die NID_C-Tabelle erzeugt. Äquivalent zur NID_C-Tabelle werden alle CID-Tabellen nur bei Bedarf erzeugt und gelöscht, sobald sie nicht mehr benötigt werden. Alle Tabellen befinden sich permanent im Hauptspeicher und werden zu keinem Zeitpunkt ausgelagert.

2.3.3 RAM-Layout

Beim Start von DXRAM wird einmalig ein großer zusammenhängender virtueller Speicherblock (Virtual Memory Block, kurz *VMB*) angelegt. Die Größe des VMB orientiert sich an der Größe des physikalischen Hauptspeichers. Der VMB sollte nur so groß sein, dass der Hauptspeicher den VMB komplett aufnehmen kann und zusätzlich mindestens eine GB für das Betriebssystem und ein weiteres GB für die restlichen Datenstrukturen von DXRAM zur Verfügung stehen. Wenn der VMB größer ist, als der physikalische Hauptspeicher, müssen Teile des VMB durch die Speicherverwaltung des Betriebssystems ausgelagert und bei Bedarf wieder

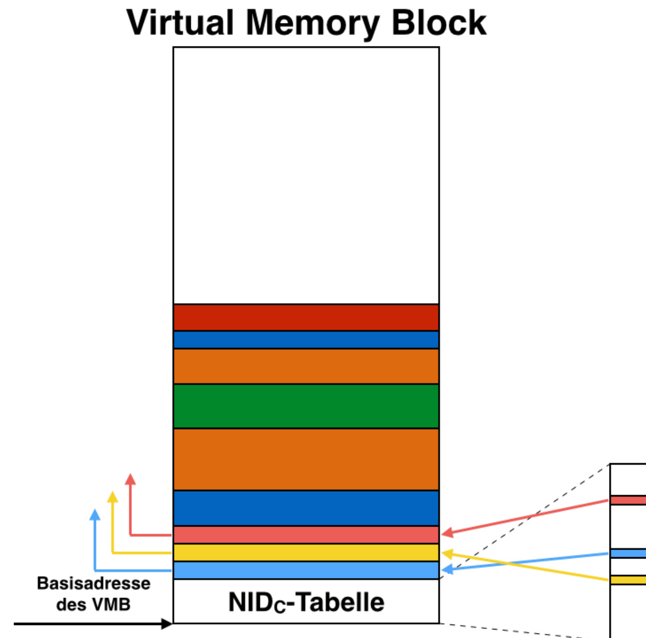


Abbildung 2.8: Struktur des VMB. *Beim Offset 0 ist die NID_C-Tabelle gespeichert. Von dort sind das CID-Verzeichnis und die CID-Tabellen verknüpft. Die eigentlichen Chunkdaten sind im gesamten VMB verteilt.*

eingelagert werden, was sehr teuer ist und die Grundidee, alle Anwendungsdaten permanent im Hauptspeicher zu halten, zunichtemacht.

Die Erzeugung des VMBs hat zwei große Vorteile. Erstens wird der komplette Speicher für die Anwendungsdaten bereits beim Systemstart alloziert und muss nicht erst bei Bedarf angefordert werden. Zweitens können so Offset-Adressen anstatt vollständige 64-Bit virtuelle Adressen eingesetzt werden. Die Offset-Adressen sind standardmäßig 39-Bit groß, können aber bei Bedarf auch vergrößert werden. Mit 39-Bit lassen sich jedoch bereits 512 GB adressieren. Die Offset-Adressen erlauben eine Reduzierung des Speicherbedarfs für Zeiger innerhalb des VMB. Die vollständige virtuelle Adresse für eine Offset-Adresse erhält man hierbei, indem die Offset-Adresse und die Basisadresse des VMB addiert werden.

Wie bereits erwähnt, werden alle Chunks permanent im Hauptspeicher gehalten. Als Speicherort wird der erzeugte VMB verwendet. Abgesehen von den Chunks, werden auch die NID_C-Tabelle, die CID-Verzeichnisse und die CID-Tabellen im VMB abgelegt. Die NID_C-Tabelle liegt dabei an einem fest definierten Offset. Das Anlegen einer neuen CID-Tabelle im Speicher läuft dabei genauso ab, wie das Anlegen eines neuen Chunks.

Abbildung 2.8 zeigt das hier beschriebene Layout des VMBs. Eine detaillierte Beschreibung der Speicherverwaltung, inklusive Allokation und Freigabe von Speicher und die Verwaltung von freien und belegten Blöcken, wird in Abschnitt 3.3 gegeben.

2.4 Backup-Management

Um Ausfälle von Knoten, Betriebsstörungen (z.B. Stromausfälle) zu maskieren und die Programmierer nicht mit der Persistenzverwaltung zu belasten, speichert DXRAM für jeden Chunk Backupreplikate auf SSD von mehreren Backupknoten ab. Die Backupknoten werden nicht für jeden einzelnen Chunk bestimmt, sondern sind immer für eine ganze Reihe hintereinander liegender Chunks zuständig (*Backupzone*). Die Größe einer Backupzone ist entweder durch eine maximale Speichergröße oder durch eine Maximalanzahl an Chunks definiert. In der Regel umfasst eine Backupzone 2,5 Millionen Chunks, was bei Objekten zwischen 16 und 64 Byte einer Größe von ungefähr 100 MB entspricht. Die Backupreplikate werden in einem für SSDs optimierten Protokoll abgelegt. Während einer fehlerfreien Ausführung wird auf das Protokoll nur schreibend zugegriffen. Im Fehlerfall wird das Protokoll einmalig komplett eingelesen.
In der Standardkonfiguration werden ein lokales und drei entfernte Replikate erzeugt. Aktualisierungen der Backupreplikate erfolgen im Regelfall asynchron, ausgenommen die Anwendung führt explizit einen blockierenden Put-Befehl aus (beispielsweise bei einer kritischen Operation). Die Backupreplikate eines Knotens müssen auf viele Backupknoten verteilt werden, da beispielsweise das Einlesen von 32 GB von einer schnellen SSD (2 GB/s Bandbreite) mehr als 16 Sekunden dauern würde. Alle verteilten Replikate auf einem neuen Knoten wiederherzustellen ist aber auch keine Alternative, da der reine Transfer von 32 GB über ein 10 GBit/s Netzwerk ungefähr 25 Sekunden in Anspruch nehmen würde. Die Datenwiederherstellung auf vielen Knoten kann die Lokalität der Daten beschädigen und erfordert das Aktualisieren der Metadaten auf dem zugehörigen Super-Peer, ermöglicht jedoch das Wiederherstellen von 32-64 GB in wenigen Sekunden [9]. Offenkundig gibt es keine optimale Lösung, daher erlaubt DXRAM der Anwendung die Anzahl der Backupknoten pro Peer, sowie die Anzahl der Knoten, auf denen die wiederhergestellten Daten gespeichert werden, zu konfigurieren und somit an die Anforderungen der Anwendung anzupassen.
Das lokale Backupreplikat ermöglicht das geordnete Herunterfahren von DXRAM, beispielsweise für Wartungszwecke, und spätere Wiederaufnahme des Betriebes. Der gleiche Mechanismus kann auch verwendet werden, um DXRAM nach einem Stromausfall des Rechenzentrums neu zu starten. Im einfachsten Fall wird dafür der VMB auf SSD kopiert und beim Neustart von SSD wiederhergestellt. Nach dem Neustart kann sich der VMB an einer anderen virtuellen Speicheradresse befinden, daher sorgt die konsequente Verwendung von Offset-Zeigern dafür, dass der VMB weiterhin nutzbar ist. Dieses Verfahren ist nur deswegen möglich, da Chunks keine Zeiger auf andere Chunks besitzen, wodurch die Speicheradressen der Chunks problemlos geändert werden können.

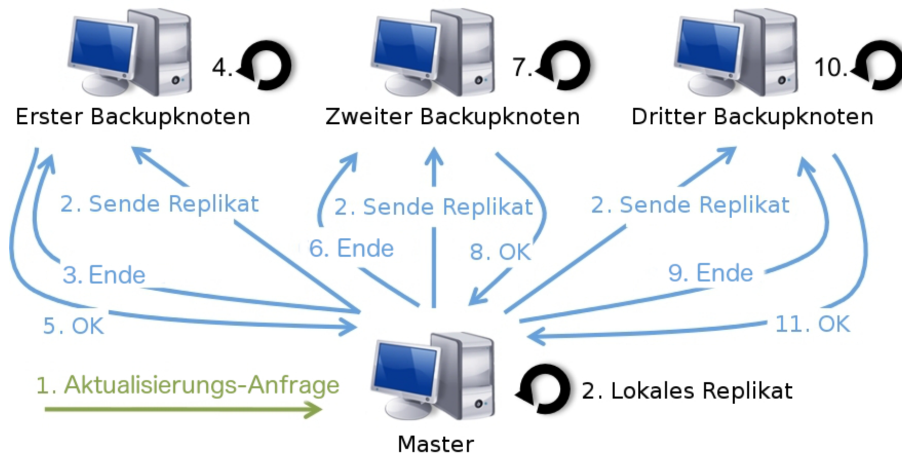


Abbildung 2.9: Backup-Prozess. *Eine Chunkaktualisierung wird sequentiell erst auf dem Master und anschließend auf dem ersten, zweiten und dritten Backupknoten durchgeführt.*

2.4.1 Backup-Prozess

Abbildung 2.9 zeigt die wichtigsten Backupschritte für das Anlegen / die Aktualisierung eines Chunks. Im weiteren Verlauf werden der Besitzer des Chunks *Master* und die Backupknoten geordnet und *Erster Backupknoten*, *Zweiter Backupknoten* und *Dritter Backupknoten* genannt. Welcher Chunk auf welchem Backupknoten gespeichert ist, ist in den Metadaten auf den Super-Peers hinterlegt (siehe Kapitel 4.2).

Die folgenden Schritte werden bei einem Backupvorgang ausgeführt:

1. Der Master empfängt eine Anfrage zum Anlegen / Aktualisieren eines Chunks.
2. Die lokalen Chunkdaten im RAM werden aktualisiert.
3. Das lokale Backupreplikat im Flash-Speicher wird aktualisiert und gleichzeitig eine Backupanfrage an alle Backupknoten geschickt.
4. Sobald der lokale Schreibvorgang erfolgreich war, schickt der Master eine Finish-Nachricht an den ersten Backupknoten.
5. Der erste Backupknoten aktualisiert sein Backupreplikat und sendet nach erfolgreichem Abschluss eine OK-Nachricht an den Master zurück.
6. Wiederhole die Schritte 5) und 6) für den zweiten und dritten Backup Knoten.

Die Backupreihenfolge garantiert, dass zu jeder Zeit bekannt ist, welcher Backupknoten die aktuellste Replikatversion hat. Bei einer fehlerfreien Ausführung ist dies immer der erste Backupknoten.
Im Regelfall wird der Backup-Prozess asynchron zum Anwendungscode ausgeführt. Ein Put-Befehl kehrt sofort nach der Aktualisierung des Chunks im RAM zur Anwendung zurück. Alternativ wird ein synchroner bzw. blockierender Put-Befehl zur Verfügung gestellt, der erst zur Anwendung zurückkehrt, nachdem alle Replikate auf den Backupknoten aktualisiert wurden.

2.4.2 SSD-Charakteristiken

Im Gegensatz zu herkömmlichen Festplatten werden bei SSDs Lese- und Schreibzugriffe zu Seiten gebündelt. Jede Seite ist 4 KB groß und liegt auf einem separaten NAND Flash-Speicher. Das bedeutet, dass grundsätzlich immer 4 KB geschrieben werden, selbst wenn nur 1 Byte geschrieben werden soll. Da DXRAM für viele kleine Objekte (16-64 Byte) ausgelegt ist, müssen Schreibzugriffe offensichtlich gebündelt werden. Durch die interne Parallelisierung einer SSD ist es sogar noch besser, wenn mehrere Seiten auf einmal geschrieben werden [77]. Darüber hinaus kann eine Seite nicht einfach überschrieben werden, sondern es muss zuerst der Block (64 bis 128 Seiten) gelöscht werden. Der Schreibzugriff wird anschließend auf einer von der SSD ausgewählten Seite durchgeführt. Die SSD verwendet eine spezifische Speicherbereinigung und eine Übersetzungsschicht, um diese Charakteristiken vor dem System zu verstecken. Die Folge ist ein wesentlich höherer Durchsatz wenn sequentiell anstatt wahlfrei geschrieben wird [77]. Schlussendlich müssen noch zwei weitere Aspekte berücksichtigt werden. Zum einen sollten Löschoperationen, auf Grund der großen Granularität (256 - 2.048 KB), möglichst vermieden werden. Zum andern sollten Lese- und Schreibzugriffe nicht gemischt werden, da sie sich gegenseitig beeinflussen können [78]. Aus diesem Grund wird im fehlerfreien Fall auf das Protokoll auch nur schreibend und im Fehlerfall nur lesend zugegriffen.

2.4.3 Protokoll-Architektur

Eine der größten Herausforderungen ist eine effiziente Protokoll-Architektur für viele kleine Chunks, die eine schnelle Datenwiederherstellung erlaubt. DXRAM begegnet dieser Herausforderung mit einem differenzierten Protokollierungsansatz [75]. Abbildung 2.10 zeigt die Komponenten des Protokolls und ihre Zusammenhänge.

Schreibpuffer: Dieser Puffer wird genutzt um Netzwerkanfragen vom langsameren I/O-Durchsatz des Protokolls zu entkoppeln. Darüber hinaus erlaubt der Puffer mehrere Schreibzugriffe

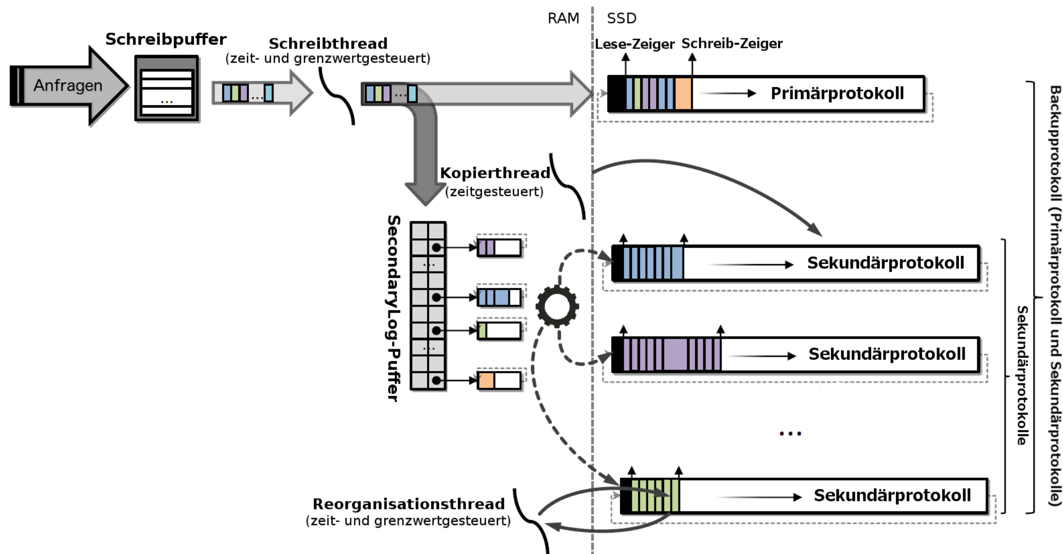


Abbildung 2.10: Protokoll-Architektur. *Die Protokoll-Architektur besteht aus mehreren Komponenten im RAM und auf SSD. Der Schreibpuffer und die Sekundärprotokoll-Puffer bündeln Schreibzugriffe. Das Primärprotokoll dient hauptsächlich zur schnellen Herstellung der Persistenz. Die Sekundärprotokoll enthalten jeweils nur Daten eines einzelnen Knotens und sind für die schnelle Datenwiederherstellung optimiert.*

zu bündeln, um die Schreibperformanz der SSD zu maximieren. Die Netzwerkthreads fungieren dabei als Erzeuger und schreiben ihre Daten in den Puffer. Ein einzelner *Schreibthread* liest den Puffer aus (Verbraucher) und schreibt die Daten auf SSD, sobald das Vielfache einer Seitengröße vorliegt. Bei geringer Schreiblast sorgt ein Zeitgeber für ein regelmäßiges Wegschreiben des Puffers. Wenn der Puffer dauerhaft überlastet ist, weil die SSD zu langsam beim Wegschreiben der Netzwerkanfragen ist, muss der Super-Peer die Backuplast für den Backupknoten anpassen.

Primärprotokoll: Abbildung 2.11 zeigt das Zusammenwirken von Schreibpuffer und Primärprotokoll etwas detaillierter. Das Primärprotokoll wird verwendet, um die Backupdaten so schnell wie möglich persistent zu machen. Es ist als Ringpuffer organisiert, ähnlich wie bei SpriteFS [79]. Während einer fehlerfreien Ausführung wird das Protokoll nie gelesen, daher ist es nicht nötig entsprechende Metadaten im RAM zu halten. Allerdings ist das Protokoll selbstbeschreibend. Jeder Eintrag besitzt einen Kopf mit einem Längenfeld, der CID des Objektes und optional eine Checksumme. Das Primärprotokoll wird nur bei einer vollständigen Knotenwiederherstellung komplett eingelesen.

Sekundärprotokolle: Jeder Knoten verteilt seine Backupreplikate über viele Backupknoten, zum Beispiel werden die 32 GB Hauptspeicher eines Knoten über 100 oder mehr Backup-

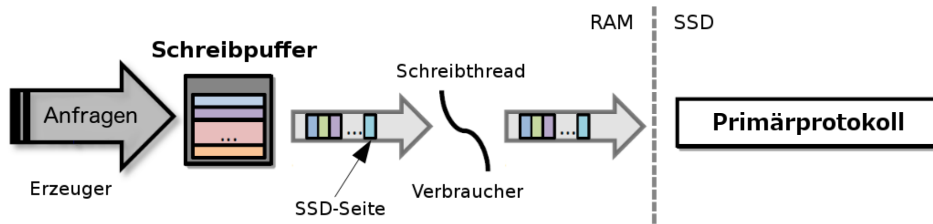


Abbildung 2.11: Schreibpuffer und Primärprotokoll. *Der Schreibpuffer bündelt die Schreibzugriffe der Netzwerkthreads (Erzeuger) zu einem Vielfachen der SSD-Seitengröße. Ein einzelner Schreibthread (Verbraucher) schreibt die Daten periodisch oder beim Erreichen eines bestimmten Grenzwertes auf SSD in das Primärprotokoll.*

knoten verteilt. Jeder Speicherknoten agiert daher gleichzeitig auch als Backupknoten für viele andere Speicherknoten. Um die Datenwiederherstellung zu beschleunigen, existiert für jeden solcher Knoten ein Sekundärprotokoll, in das nur die Backupreplikate dieses Knotens einsortiert werden. Dadurch muss bei der Datenwiederherstellung nur das Sekundärprotokoll des ausgefallenen Knotens eingelesen und nicht das komplette Primärprotokoll analysiert werden. Die Sekundärprotokolle sind ebenfalls als Ringpuffer organisiert und selbstbeschreibend. Der Header jedes Eintrags lässt sich im Gegensatz zum Primärprotokoll weiter reduzieren. So ist es nicht mehr notwendig die komplette CID zu speichern, sondern nur noch die LID. Dies ist möglich, da ein Sekundärprotokoll nur Backupreplikate eines einzigen Knoten beinhaltet. Der Kopf kann noch weiter reduziert werden, wenn Backupreplikate mehrere aufeinanderfolgende LIDs besitzen. In diesem Fall reicht es die volle LID für das erste Backupreplikat und ein Flag für alle weiteren Backupreplikate zu speichern. Da CIDs sequentiell erzeugt werden (siehe Abschnitt 2.1), kann erwartet werden, dass dieser Fall relativ häufig auftritt.
Die Größe der Sekundärprotokolle richtet sich nach der Anzahl der Backupknoten. Wenn die Speicherknoten über 32 GB RAM verfügen und ihre Daten auf 384 Backupknoten (128 Backupzonen mit je drei Backupknoten) repliziert werden, muss jedes Sekundärprotokoll mindestens 256 MB umfassen. Da die Protokolle gegebenenfalls mehrere Versionen der Replikate beinhalten, sollten die Protokolle jedoch größer sein, beispielsweise 512 MB. SSDs werden zunehmend größer und billiger, daher kann davon ausgegangen werden, dass zumindest 512 GB zur Verfügung stehen. Das erlaubt die Speicherung von 1.024 Sekundärprotokollen, womit die Größe der Sekundärprotokolle kein Problem darstellt.

Sekundärprotokoll-Puffer: Nachdem die Backupreplikate ins Primärprotokoll geschrieben wurden, werden die Daten zusätzlich in die Sekundärprotokoll-Puffer im RAM kopiert. Für jedes Sekundärprotokoll existiert so ein Puffer. Die Sekundärprotokoll-Puffer dienen zur Bündelung von Schreibzugriffen auf die Sekundärprotokolle, ähnlich wie der Schreibpuffer für

Schreibzugriffe auf das Primärprotokoll. Dabei ist zu bemerken, dass die Verwendung der Sekundärprotokoll-Puffer nicht zu Datenverlust führen kann, da alle Daten bereits im Primärprotokoll auf SSD stehen. Ähnlich wie der Schreibpuffer werden die Sekundärprotokoll-Puffer periodisch (aber seltener als der Schreibpuffer) in die Sekundärprotokolle geschrieben. Der Einsatz der Puffer hat drei maßgebliche Vorteile. Erstens können alle Daten, die in die entsprechenden Sekundärprotokolle geschrieben wurden, im Primärprotokoll, durch das Anpassen des Lese-Zeigers, gelöscht werden. Zweitens lässt sich die Reorganisation effizienter durchführen, wenn die Daten im Hauptspeicher liegen. Drittens kann die Datenwiederherstellung schneller durchgeführt werden, wenn das Primärprotokoll nicht gelesen werden muss.

2.4.4 Datenwiederherstellung

Wie bereits erwähnt, wird die Datenwiederherstellung durch den Super-Peer des ausgefallenen Knotens koordiniert (siehe Abschnitt 2.2). Relevante Backupknoten werden durch den Super-Peer kontaktiert und stellen gespeicherte Chunks des ausgefallenen Knoten aus dem lokalen Protokoll wieder her.
Der Super-Peer beauftragt alle ersten Backupknoten mit der Wiederherstellung (siehe Abbildung 2.12). Die entsprechenden Backupknoten spülen ihren Schreibpuffer in dem sich potentiell die neuesten Backupreplikate des ausgefallenen Knotens befinden. Anschließend wird das komplette Sekundärprotokoll in den Hauptspeicher eingelesen und zusammen mit dem Sekundärprotokoll-Puffer analysiert, um veraltete und gelöschte Chunks zu entfernen. Im Anschluss werden die Metadaten der wiederhergestellten Chunks gebündelt an den koordinierenden Super-Peer geschickt, der daraufhin seine Metadaten entsprechend aktualisiert. Alle folgenden Chunkanfragen landen bei den neuen Besitzern der Chunks und das System läuft regulär weiter. Während der Datenwiederherstellung puffert der Super-Peer entsprechende Anfragen und verzögert sie bis zum Abschluss der Wiederherstellung. Durch dieses Verhalten verläuft die Wiederherstellung vollkommen transparent für die Anwendung.
Nach der Wiederherstellung sind die ersten Backupknoten die neuen Chunkbesitzer, die zweiten Backupknoten werden zu den ersten Backupknoten und die dritten Backupknoten werden zu den zweiten Backupknoten. Neue dritte Backupknoten werden durch den Super-Peer bestimmt und von den beiden verbleibenden Backupknoten mit Backupreplikaten versorgt.
Durch die im Backup-Prozess beschriebenen Schritte ist garantiert, dass der erste Backupknoten die aktuellste Replikatversion besitzt. Im besten Fall haben der zweite und dritte Backupknoten auch die aktuellste Version. Es kann aber vorkommen, dass ein Knoten und der zugehörige erste Backupknoten gleichzeitig ausfallen und der zweite Backupknoten nicht die aktuellste Version besitzt. Offensichtlich handelt es sich dabei um ein sehr selten auftretendes Ereignis. In diesem Fall wird die ältere Version wiederhergestellt und die Anwendung diesbe-

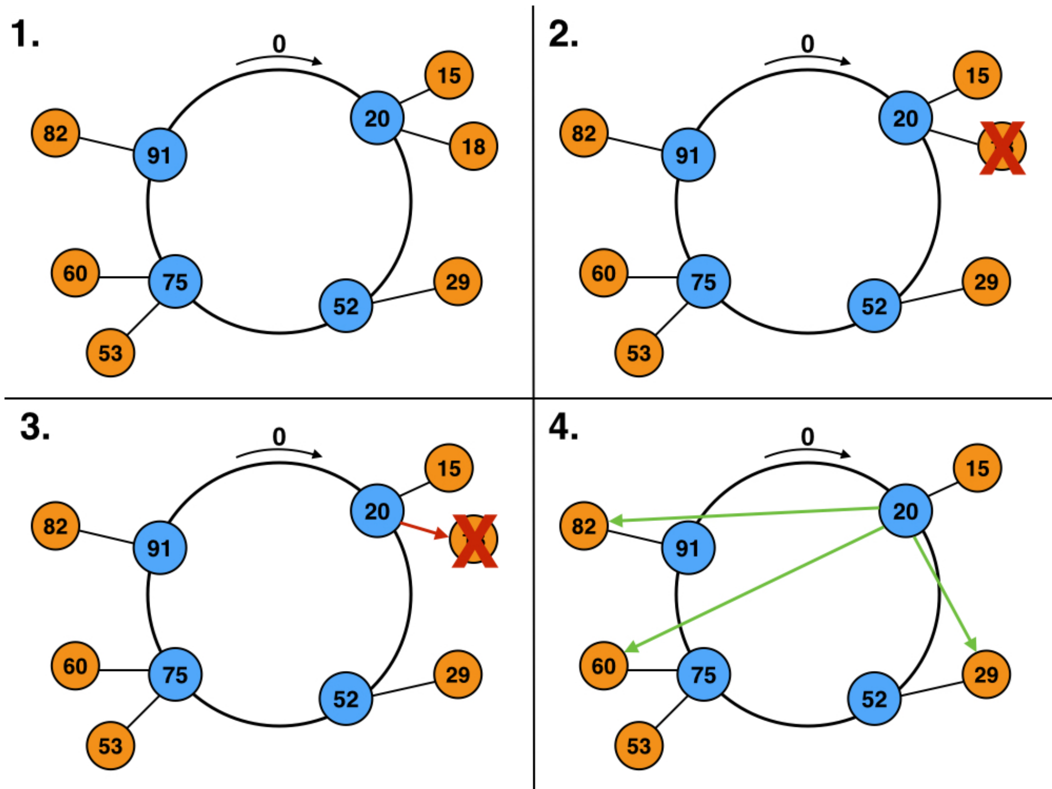


Abbildung 2.12: Datenwiederherstellung. *1. Super-Peer-Overlay mit vier Super-Peers (20, 52, 75, 104) und sechs Peers (15, 29, 53, 65, 100. 110). 2. Ausfall von Peer 110. 3. Ausfall wird erkannt. 4. Verteilung der Wiederherstellungs-Nachrichten.*

züglich informiert. An dieser Stelle ist noch einmal darauf hinzuweisen, dass ein synchroner Put-Befehl existiert, der die Aktualisierung aller Backupreplikate garantiert. Die Performanz des Gesamtsystems leidet allerdings darunter, wenn der synchrone Put-Befehl grundsätzlich eingesetzt wird.

Die Lokalität der Daten ist durch den hier vorgestellten Prozess nicht vollständig garantiert, da die Daten nach der Wiederherstellung potentiell über viele Knoten verteilt sind. Um die komplette Lokalität zu rekonstruieren, können die Daten asynchron zu einem neuen Knoten transferiert werden.

2.4.5 Reorganisation

Der Schreibpuffer und die Sekundärprotokoll-Puffer werden regelmäßig geleert, weshalb eine Reorganisation nicht notwendig ist. Die Sekundärprotokolle hingegen werden zunehmend voller, durch am Ende der Protokolle eingefügte Einträge. Für jede Chunkerzeugung, -aktualisierung

und -löschung wird ein Protokolleintrag erzeugt.
Sofern häufig aktualisiert wird, besteht offenkundig ein hoher Reorganisationsbedarf, um veraltete und gelöschte Protokolleinträge zu entfernen und Speicher freizugeben. In DXRAM wird die Reorganisation angestoßen, sobald ein Sekundärprotokoll zu 75% gefüllt ist. Bei der Reorganisation wird ein komplettes Sekundärprotokoll in den RAM geladen und eine Reinigung mit mehreren Threads durchgeführt. Die Reinigungsthreads überprüfen das komplette Protokoll auf gültige, veraltete und gelöschte Einträge. Durch die sequentielle Ordnung beim Schreiben stehen neuere Versionen hinter älteren Versionen im Protokoll.
Auch wenn dieser hier vorgestellte Ansatz gut funktioniert, erscheint ein inkrementeller Ansatz vielversprechender. In [79] wird beispielsweise ein Protokoll in mehrere Segmente unterteilt und die zu reorganisierenden Segmente anhand einer Kosten-Nutzen-Strategie ausgewählt.

2.4.6 Hot-and-Cold Zonen

Das Konzept der Hot-and-Cold Zonen mindert die Problematik des häufigen Umkopierens selten geänderter Daten in den Sekundärprotokollen. Dabei wird ein Sekundärprotokoll in zwei Zonen unterteilt. In der *Hot-Zone* befinden sich die Backupreplikate, die sich häufig ändern, wohingegen in der *Cold-Zone* sich die Backupreplikate befinden, die selten geändert werden. In beiden Zonen wird die oben vorgestellte Reorganisation durchgeführt, wobei die Hot-Zone häufiger reorganisiert wird als die Cold-Zone. Grundsätzlich ist es auch möglich mehr als zwei Zonen zu verwenden, allerdings führt jede weitere Zone zu zusätzlicher Komplexität bei der Protokollierung und der Datenwiederherstellung.

2.5 Verwandte Arbeiten

Speichersysteme waren und werden immer ein wichtiges Forschungsgebiet sein und es existieren sehr viele Publikationen, deren Inhalte in irgendeiner Weise mit den hier vorgestellten Konzepten verwandt sind. Im Folgenden werden die relevantesten verwandten Arbeiten betrachtet. In Kapitel 3.5 und 4.5 werden darüber hinaus weitere verwandte Arbeiten betrachtet, die mit den dort vorgestellten Konzepten in Bezug stehen.

RAMCloud
RAMCloud ist eines der ersten Systeme, mit dem Ziel alle Anwendungsdaten immer im Hauptspeicher zu halten, wozu der RAM von hunderten oder tausenden Knoten in einem Rechenzentrum aggregiert wird [16]. Besonderen Wert wird auf extrem geringe Latenzzeiten gelegt,

die durch spezielle Hardware und Anpassungen von Netzwerkkarte und Kernel erreicht werden und Lesezugriffe in 5-10 Mikrosekunden ermöglichen [64]. Daten werden in Tabellen als Schlüssel-Wert-Paare gespeichert und je nach Tabellengröße auf einem oder mehreren Knoten gespeichert. Der Schlüsselraum der Tabelle wird dafür in so genannte Tablets (zusammenhängende Schlüsselbereiche) unterteilt und den Knoten zugewiesen. Schlüssel sind variabel lang und werden gehasht, um das zuständige Tablet und damit den zuständigen Knoten zu ermitteln. Neben dem Schlüssel besitzt jedes Objekt eine Versionsnummer, die bei einem Schreibzugriff inkrementiert wird und bei Zugriffen als Bedingung verwendet werden kann (beispielsweise ändere Objekt X nur, wenn die aktuelle Version des Objektes 5 ist). Die eigentlichen Daten liegen in Form eines Byte-Arrays vor, das bis zu 1 MB groß sein kann. Der Hauptspeicher ist wie ein Protokoll aufgebaut, dass aus Segmenten zu je 8 MB besteht, wobei Schreibzugriffe nur am Kopf des Protokolls möglich sind [80]. Eine Hashtabelle enthält die Referenzen auf die aktuelle Version der Objekte. Um Datenverlust bei einem Knotenausfall zu vermeiden, werden die Segmente des Protokolls auf jeweils 2-3 Backupknoten verteilt, die die Segmente in einem gleich aufgebauten Protokoll auf Festplatte speichern. Insgesamt wird das Protokoll somit auf alle Knoten des Systems verteilt und bei der Datenwiederherstellung können alle Segmente parallel eingelesen werden. Dieser Fast-Recovery-Ansatz erlaubt es 35 GB Daten in 1,6 Sekunden auf 1.000 Backupknoten parallel wiederherzustellen [9]. Globale Metadaten werden durch einen zentralen Koordinator verwaltet (verwendet ZooKeeper), der auch die Datenwiederherstellung anstößt und koordiniert. Dazu wird bei einem Knotenausfall mit einer Broadcast-Nachricht ermittelt, welcher Knoten welche Segmente des ausgefallenen Knotens besitzt.

DXRAM ist von einigen Ideen RAMClouds inspiriert worden, unterscheidet sich aber in mehreren wichtigen Aspekten. Anstatt die Daten in Tabellen zu organisieren, verwaltet DXRAM Daten in Schlüssel-Wert-Paaren. Ferner ist RAMCloud für wesentlich größere Daten als DXRAM konzipiert [81], so dass DXRAM eine natürlichere Modellierung von Graphen erlaubt. Auf Grund des unterschiedlichen Datenmodells mussten verschiedene Komponenten in DXRAM andersartig konzipiert werden, beispielsweise die Metadaten-Verwaltung. Im Gegensatz zu dem zentralen Koordinator in RAMCloud, kommt ein dezentrales Super-Peer-Overlay zum Einsatz. Die Protokoll-Architektur und die Datenwiederherstellung unterscheidet sich auch in mehreren Punkten. DXRAM vermeidet das Senden einer Broadcast-Nachricht (um eine globale Sicht zu erlangen) und setzt statt dessen auf eine strenge Ordnung der Backupreplikate auf den Backupknoten. Welche Backupknoten bei einer Wiederherstellung in Betracht kommen, speichert der zuständige Super-Peer in seinen Metadaten. Eine Hashtabelle zum Nachverfolgen von Replikatversionen, wie sie in RAMCloud eingesetzt wird, ist für DXRAM nicht praktikabel. Da DXRAM für sehr viele sehr kleine Objekte ausgelegt ist, würde eine Hashtabelle zu viel Speicher beanspruchen. Aus diesem Grund wird ein selbstbeschreibendes Protokoll

verwendet, in dem zu jedem Eintrag ein Header mit den notwendigen Metadaten gespeichert wird.
In Abschnitt 3.5 und 4.5 wird noch einmal detaillierter auf die Unterschiede bei der lokalen und globalen Metadaten-Verwaltung eingegangen.

Trinity Graph Engine

Trinity Graph Engine ist ein verteilter, RAM-basierter Schlüssel-Wert-Speicher für die Online-Anfragebearbeitung und Offline-Analyse von großen Graphen [15]. Die Graphen können dabei aus Milliarden von Knoten bestehen, wie zum Beispiel in sozialen Netzwerken oder anderen Webgraphen [63]. Der Benutzer kann mit Hilfe der Trinity Specification Language (TLS) das Graphschema, das Kommunikationsprotokoll und die Knoten- und Kantenstruktur definieren und auf seine Anforderungen anpassen. Die Daten des Graphen werden in einer Speicher-Cloud im RAM vieler verteilter Knoten gespeichert, wobei sich nicht zwangsweise alle Daten im RAM befinden, sondern vor allem die Topologie und häufig benutzte Daten im Speicher gehalten werden [65]. Andere Daten befinden sich gegebenenfalls auf Festplatte oder in einem Datenbanksystem [63].
Daten-Objekte in der Speicher-Cloud werden in Schlüssel-Wert-Paaren verwaltet, wobei der Schlüssel 64 Bit lang ist und die Werte von ein paar Bytes bis zu mehreren Kilobytes umfassen können. Der Speicher ist in Memory Trunks zu je 2 GB aufgeteilt, auf mehreren Knoten verteilt und zusätzlich im Trinity File System (TFS) repliziert (ähnlich zu HDFS).
Fehlertoleranz wird in Trinity auf zwei Weisen erreicht. Erstens werden periodische Schnappschüsse erzeugt, für die das System kurzzeitig angehalten wird und zweitens wird das bereits erwähnte TFS eingesetzt. Wenn ein Knoten ausfällt, werden die gespeicherten Memory Trunks aus dem TFS auf noch laufenden Knoten wiederhergestellt [63].
Für Trinity existieren zwei Erweiterungen, die es ermöglichen Wissensgraphen (Knowledge Graphs) und RDF-Daten zu verwalten [82, 83].
Im Gegensatz zu Trinity werden in DXRAM alle Daten permanent im RAM gehalten und nicht nur häufig benötigte Daten. Dadurch wird zu jeder Zeit ein wahlfreier und schneller Datenzugriff gewährleistet. DXRAM ist speziell für sehr viele sehr kleine Objekte konzipiert, wohingegen die Objekte in Trinity bis zu mehreren Kilobyte groß sind. Auch die Speicherverwaltung unterscheidet sich erheblich. So sind in Trinity die 2 GB großen Memory Chunks die Einheit für die Speicherverwaltung, für die Datenwiederherstellung und für Migrationen, wohingegen DXRAM Migrationen auf Basis einzelner Objekte (16-64 Byte) erlaubt und für die Datenwiederherstellung Backupzonen mit ungefähr 100 MB verwendet werden. Bei der Datenwiederherstellung setzt DXRAM auf einen Fast-Recovery-Ansatz, der es ermöglicht Backupzonen auf potentiell sehr vielen Knoten parallel wiederherzustellen und Daten schon während der Wiederherstellung zur Verfügung stehen. Im Unterschied dazu setzt Trinity mit TFS ein

verteiltes Dateisystem ein, um komplette Memory Trunks auf einem anderen Knoten wiederherzustellen.
Auch bei Trinity wird in Abschnitt 3.5 und 4.5 noch einmal detaillierter auf die Unterschiede bei der lokalen und globalen Metadaten-Verwaltung eingegangen.

FaRM

FaRM ist eine RAM-basierte verteilte Berechnungsplattform, die RDMA für den rechnerübergreifenden Datenzugriff nutzt [61]. Gespeicherte Objekte sind von 64 Byte bis mehreren Megabyte groß und werden in einem Schlüssel-Wert-Speicher oder einem Graph-Speicher (ähnlich wie TAO) verwaltet. Der Datenzugriff erfolgt mit Transaktionen oder sperrfreien Lesezugriffen über RDMA, wofür nicht nur spezielle Hardware (Netzwerkkarten, Switches, etc.) zum Einsatz kommt, sondern auch das Betriebssystem und die Treiber der Netzwerkkarten modifiziert wurden. Der Aufwand für die verteilten Transaktionen lässt sich in bestimmten Szenarien reduzieren, indem die Funktionen auf dem entfernten Rechner in einer lokalen Transaktion durchgeführt werden. Für die Maskierung von Knotenausfällen wird ein zu RAMCloud vergleichbarer Protokollierungs- und Wiederherstellungsansatz verfolgt.
Der verteilte Speicher ist in 2 GB große, gemeinsam genutzte Speicherregionen aufgeteilt, die die Basis für die Adressierung, die Datenwiederherstellung und die RDMA-Registrierung bilden. Jede globale ID besteht aus einer 32-Bit großen Regions-ID und einem 32-Bit-Offset. Zum Auffinden einer Region wird Consistent-Hashing mit mehreren virtuellen Ringen eingesetzt und die Metadaten der Region über RDMA bezogen und lokal gecacht. Für die Objektsuche kann neben der globalen ID auch ein Namensdienst verwendet werden.
Im Gegensatz zu DXRAM benötigt FaRM spezielle Hardware, um die RDMA Zugriffe zu ermöglichen. FaRM erlaubt zwar die Speicherung von kleinen Objekten, allerdings erscheint der Einsatz von RDMA für sehr viele sehr kleine Objekte (wie in DXRAM) nicht sinnvoll, da der RDMA-Zugriff für kleine Objekte in der Regel inperformant ist [84].
Die lokale und globale Metadaten-Verwaltung von FaRM werden in Abschnitt 3.5 und 4.5 noch einmal detaillierter betrachtet und mit den vorgestellten Konzepten verglichen.

RAMCube

RAMCube ist ein RAM-basierter Speicher, der auf die BCube [66] Netzwerk-Architektur aufsetzt und diese für Fehlerbehebung auf Netzwerkebene (Ausfall eines Knotens oder Switches) nutzt [62]. Daten werden in Tabellen, bestehend aus Schlüssel-Wert-Paaren, gespeichert, wobei die Schlüssel bis zu einem Kilobyte und die Werte bis zu einem Megabyte groß sein können. Daten eines Servers (Primary-Server) werden auf Festplatte mehrerer Backup-Server repliziert und bei einem Knotenausfall parallel wiederhergestellt (ähnlich wie in RAMCloud). Neben den Backup-Servern existieren für jeden Server mehrere Recovery-Server, auf den die Daten

bei einem Knotenausfall wiederhergestellt werden. Für jeden Server wird in RAMCube eine Baumstruktur erstellt, in dem der Server die Wurzel, die Recovery-Server die inneren Knoten und die Backup-Server die Blätter darstellen. Ein Heart-Beat-Protokoll zwischen Server und Recovery-Server wird für die Ausfallerkennung genutzt, wobei die Recovery-Server die Daten von den Backup-Servern wiederherstellen, sobald der Primary-Server nicht mehr erreichbar ist. Die Bäume werden auf das Netzwerk-Overlay in Form von mehreren logischen Ringen abgebildet, wobei der erste Ring alle Primary-Server umfasst. Für jeden Primary-Server existiert ein weiterer Ring, der seine Recovery-Server beinhaltet, die wiederum jeweils einen weiteren Ring besitzen, der aus den entsprechenden Backup-Servern besteht.
RAMCube befasst sich überwiegend mit dem Problem der Fehlererkennung und -behebung in einem RAM-basierten Speicher. Dabei wird BCube verwendet, um Knoten über mehrere Netzwerkpfade zu erreichen und dadurch temporäre und permanente Knotenausfälle erkennen und unterscheiden zu können. DXRAM hingegen konzentriert sich auf die speichereffiziente Verwaltung von Daten und Metadaten und schnelle Zugriffszeiten und setzt für die Fehlererkennung und -behebung auf bewährte Techniken bekannter P2P-Systeme und einen Fast-Recovery-Ansatz. Bei einem Knotenausfall werden in DXRAM die Daten des ausgefallenen Knotens direkt auf den Backupknoten wiederhergestellt, wohingegen RAMCube alle Daten über das Netzwerk zu definierten Recovery-Server schickt und dort erst zur Verfügung stellt.

Sedna

Sedna ist ein RAM-basierter Speicher für Echtzeit-Verarbeitung in der Cloud, der besonders auf Skalierbarkeit ausgerichtet ist [60]. Daten werden in Schlüssel-Wert-Paaren in einem hierarchischen Adressraum gespeichert. Trigger erlauben es geänderte Daten aktiv an Clients zu verteilen und benutzerdefinierte Aktionen auszuführen. Mehrere ZooKeeper-Server sorgen für eine konsistente Verwaltung von Status-Informationen und Metadaten für das gesamte System. Consistent Hashing (mit virtuellen Ringen und Knoten) wird verwendet, um Datenobjekte über die Speicherknoten zu verteilen, wobei jedes Objekt auf einem Knoten gespeichert und auf zwei weiteren Knoten repliziert wird. Knotenausfälle werden durch das Heart-Beat-Protokoll von ZooKeeper erkannt, allerdings die Wiederherstellung erst beim nächsten Lese- oder Schreibzugriff auf die Daten durchgeführt.
Sedna verwaltet Metadaten in mehreren ZooKeeper-Instanzen und verwendet Consistent Hashing für die Objektsuche. In DXRAM hingegen werden die Metadaten in Form von ID-Bereichen in einem Super-Peer-Overlay verwaltet und ein modifizierter B-Baum für die Objektsuche eingesetzt. Replikate eines Objektes werden auf SSD mehrerer Backupknoten gespeichert und ein Fast-Recovery-Ansatz sorgt für die Datenwiederherstellung bei einem Knotenausfall, wohingegen Sedna jedes Objekt im Hauptspeicher zweier weiterer Knoten repliziert und dadurch die Datenwiederherstellung erst verzögert durchführen muss. Dieses Vorgehen erfordert jedoch

einen wesentlich höheren Speicherverbrauch durch die beiden zusätzlichen Replikate. Darüber hinaus ist Sedna für Echtzeit-Verarbeitung ausgelegt und unterstützt mit den Triggern ein anderes Programmierparadigma als DXRAM.

GemFire
VMware® vFabric™ GemFire® ist eine verteilte, RAM-basierte Datenverwaltungsplattform, die für hohe Performanz und lineare Skalierbarkeit von daten-intensiven Anwendungen entwickelt wurde [85]. GemFire besteht grundsätzlich aus einem Peer-to-Peer-System, dass allerdings auch in Form einer Client-Server-Architektur oder einer Multi-Site-Topologie eingesetzt werden kann. Anwendungen können dabei sowohl auf den GemFire-Knoten als auch auf separaten Clients ausgeführt werden. Daten werden zu Regionen zusammen gefasst, die jeweils unabhängig voneinander konfiguriert werden können und einem oder mehreren von vier Typen entsprechen: Partitioned, Replicated (not distributed), Distributed (not replicated), Local (not distributed). Abhängig vom Typ werden die Daten der Region über mehrere Server verteilt und repliziert oder auch nicht. Um die Latenzen zu minimieren, verwendet GemFire mehrere Ansätze: 1. Anwendungen können im Speicherbereich arbeiten, in dem die Daten liegen, 2. Cache Listener ermöglichen die Ausführung von Programmlogik, sobald Daten eintreffen, 3. Anwendungslogik kann über definierte Methoden direkt auf den Daten ausgeführt werden. Bei Schreibzugriffen auf eine Region werden diese nach Möglichkeit nur im RAM durchgeführt, da Festplatten- oder Datenbankzugriffe zu langsam sind. Steht allerdings nicht genügend Speicher zur Verfügung, wird eine Garbage-Collection angestoßen, die alte Daten aus dem Speicher entfernt, um Platz für neue Daten zu erhalten.
Im Vergleich zu DXRAM hält GemFire Daten nicht permanent im RAM, sondern ist als verteilter RAM-basierter Cache ausgelegt [86]. Fehlertoleranz wird durch Replikation oder sekundäre Kopien erreicht, wohingegen in DXRAM eine transparente Hintergrundprotokollierung und ein Fast-Recovery-Ansatz verwendet wird. Für jedes Eintrag in einer Region existieren zwischen 87 und 243 Byte Metadaten [87], was einen Einsatz für sehr viele sehr klein Objekte, wie in DXRAM, ausschließt. In DXRAM existieren keine Konsistenzgarantien, sondern die Anwendungen müssen über bereitgestellte Sperren die gewünschte Konsistenz umsetzten. In GemFire hingegen wird besonderen Wert auf Konsistenz gelegt und verteilte Transaktionen auf mehreren Daten innerhalb einer Region mit ACID-Eigenschaften ermöglicht. Da GemFire nicht frei zugänglich ist, ist ein tiefergehender Einblick in die Konzepte und die Datenverwaltung leider nicht möglich.

Apache Spark
Apache Spark ist eine Erweiterung von Hadoop und ermöglicht es regelmäßig benötigte Sammlungen von Datensätzen für MapReduce-Anwendungen im RAM mehrerer Knoten vorzuhal-

ten [54]. Die so genannten Resilient Distributed Datasets (RDDs) werden auf mehreren Knoten verteilt, können nur lesend zugegriffen werden und ermöglichen eine Wiederherstellung der Daten falls eine Partition durch einen Knotenausfall verloren geht. Ein RDD wird in Spark durch ein Scala Objekt repräsentiert und kann nur durch vier verschiedene Arten erzeugt werden: 1. Durch das Einlesen einer Datei aus einem gemeinsamen Dateisystem (z.B. HDFS), 2. durch die Aufteilung einer Scala Collection (z.B. ein Array) in eine Anzahl von Stücken, 3. durch die Transformation eines bestehenden RDDs mit Hilfe gegebener Funktionen oder 4. durch die Änderung der Persistenzeigenschaften eines bestehenden RDDs. Normalerweise werden RDDs erst bei der Benutzung erzeugt und nach der Benutzung wieder freigegeben. Durch die Änderung der Persistenzeigenschaften kann der Programmierer Spark anweisen, ein RDD nach der Benutzung weiterhin im Speicher zu halten (Cache Action) oder in das gemeinsame Dateisystem zu schreiben (Save Action). Falls allerdings nicht genügend Speicher zur Verfügung steht, kann Spark von den Anweisungen abweichen und RDDs in das gemeinsame Dateisystem auslagern oder komplett verwerfen und bei der nächsten Benutzung neu erzeugen. Da Daten bei MapReduce üblicherweise sehr groß sind, sind auch RDDs meist sehr groß und belegen viel Speicher.
Im Gegensatz zu Spark, ist DXRAM für sehr viele sehr kleine Objekte konzipiert, die permanent im RAM gehalten werden. Beispielsweise speichert die Erweiterung für Graphverarbeitung GraphX[5] einen kompletten Graphen in einer RDD. Daten können in DXRAM grundsätzlich gelesen und verändert werden und Knotenausfälle werden durch eine transparente Hintergundprotokollierung und ein Fast-Recovery-Ansatz maskiert. In Spark hingegen kann auf RDDs nur lesend zugegriffen werden und Fehlertoleranz gegenüber Knotenausfällen wird durch ein gemeinsames Dateisystem gewährleistet.

TAO

TAO ist ein RAM-basierter Graph-Cache von Facebook, der besonders für Lesezugriffe optimiert ist [5] und die zuvor genutzten Memcached-Instanzen ablösen soll. Wie bereits die Memcached-Instanzen, wird TAO als Cache benutzt und befindet sich zwischen der Anwendung und vielen MySQL-Servern. TAO cacht Objekte, Verknüfungslisten und Verknüfungszähler und wird nach Bedarf gefüllt. Dabei werden gegebenenfalls bereits gecachte Daten nach einer LRU-Strategie entfernt. Jedes Objekt ist genau einem Shard zugewiesen, von denen mehrere auf einem Server gespeichert sind. Die Objekt-ID enthält eine eindeutige Shard-ID, um den zugeordneten Shard zu identifizieren.
Im Gegensatz zu TAO hält DXRAM permanent alle Anwendungsdaten im RAM und cacht nicht nur Abfrageergebnisse. Darüber hinaus ist TAO nur für den Einsatz bei Facebook entwickelt worden und erlaubt nur einen eingeschränkten Satz Befehle und Graphabfragen. DXRAM

[5] `http://spark.apache.org/graphx/`

hingegen verwendet eine schlanke Schnittstelle, die sehr viele Einsatzmöglichkeiten bietet.

Memcached

Memcached ist ein verteilter RAM-basierter Cache, der häufig von interaktiven Internetseiten eingesetzt wird. Die eigentlichen Daten kommen dabei meist aus externen Quellen (Datenbanken, APIs, etc.) und werden in Memcached abgelegt, um den nächsten Zugriff auf die Daten zu beschleunigen [88]. Memcached stellt für die Daten eine Hashtabelle zur Verfügung, bei der die Schlüssel bis zu 250 Bytes und die Werte bis zu 1 MB groß sein können [50]. Die Hashtabelle ist meist sehr groß, weswegen sie über mehrere Server in einem Rechenzentrum verteilt wird.

Jeder Memcached-Client kennt alle Server und wendet sich immer an den Server, auf dem die notwendigen Daten liegen. Hierfür bestimmt der Client lokal einen Hashwert und kontaktiert den zugeordneten Server [50]. Wenn alle Clients die selbe Hashfunktion verwenden (nicht zwingend erforderlich), können die Clients gegenseitig auf ihre Daten zugreifen. Bei einer Anfrage erzeugt ein Server einen zweiten Hashwert und ermittelt damit den Speicherort der Daten. Die Daten werden dabei immer auf dem kontaktierten Server gelesen oder abgelegt, eine Kommunikation zwischen den einzelnen Servern findet nicht statt. Hat ein Memcached-Server keinen Speicher mehr zur Verfügung um neue Daten aufzunehmen, werden die am längsten nicht mehr benutzten Daten verworfen. Daher kann nicht davon ausgegangen werden, dass einmal hinzugefügte Daten auch wieder abgerufen werden können [50, 88].

Memcached hält immer nur einen Teil der Anwendungsdaten im RAM vor und eine manuelle Synchronisierung von Cache und Sekundärspeicher ist notwendig. Im Unterschied dazu hält DXRAM zu jeder Zeit alle Anwendungsdaten komplett im Hauptspeicher, so dass eine Synchronisierung verschiedener Speicher überflüssig ist. Für die Objektsuche schicken Clients Anfragen an das Super-Peer-Overlay, cachen die Antworten und kommunizieren anschließend direkt mit dem Speicherknoten. Dieser Ansatz erlaubt es zur Laufzeit weitere Speicherknoten hinzuzufügen oder bestehende zu entfernen, wohingegen bei Memcached für das Hashing auf den Clients die Anzahl der Memcached-Server bekannt sein muss. Darüber hinaus bietet Memcached, keinerlei Persistenz, sondern überlässt die Persistenz der externen Datenquelle. In DXRAM sorgt eine transparente Hintergrundprotokollierung dafür, dass alle Daten persistent sind.

Cassandra

Cassandra ist ein verteiltes NoSQL-Speichersystem mit vielen Servern für eine sehr hohe Anzahl an strukturierten Daten [44]. Es wurde ursprünglich von Facebook entwickelt [89] und legt besonderen Wert auf eine hohe Verfügbarkeit des Systems, das keinen Single-Point-of-Failure

besitzt. Datenobjekte werden in Tabellen organisiert und durch eine beliebig lange Zeichenkette (Schlüssel) identifiziert. Spalten können zu Spaltenfamilien zusammengefasst und Werte nach Namen oder Zeit sortiert abgelegt werden. Aus Persistenzgründen werden alle Schreibzugriffe erst auf Festplatte ausgeführt, bevor gegebenenfalls Daten im RAM geändert werden. Consistent-Hashing wird eingesetzt, um die Daten über die einzelnen Speicherknoten zu verteilen. Der Adressraum wird in Form eines Ringes abgebildet, wobei jeder Knoten eine Position auf dem Ring hat und für einen Teil des Rings zuständig ist. Zur Lastverteilung ist es in Cassandra möglich, dass die Knoten ihre Ringposition anpassen und somit ihren zuständigen Bereich verkleinern oder vergrößern. Jedes Datenobjekt wird auf mehreren Knoten repliziert, wobei die Auswahl und die Anzahl der Knoten durch verschiedene Strategien (Rack Unaware, Rack Aware, Datacenter Aware) beeinflusst werden kann.
Cassandra ist ein festplatten-basiertes Speichersystem für strukturierte Daten, wohingegen DXRAM alle Daten permanent im RAM hält und unstrukturierte Binärdaten verwaltet. Statt Consistent-Hashing wird in DXRAM ein Super-Peer-Overlay für die Objektsuche eingesetzt. Replikate dienen in Cassandra dazu die Last zu verteilen und einen Knotenausfall zu maskieren. Im Gegensatz dazu werden Objekte in DXRAM, auf Grund des Speicherbedarfs, nur auf SSD mehrere Knoten repliziert und ein Fast-Recovery-Ansatz sorgt für die Datenwiederherstellung bei einem Knotenausfall.

MongoDB

MongoDB ist eine NoSQL-Datenbank, die ihre Daten auf Festplatte / SSD speichert und nach Möglichkeit zusätzlich im RAM vorhält [42]. Daten sind in Dokumenten (ähnlich zu JSON-Objekten) organisiert, von denen mehrere zu Collections zusammengefasst werden können. MongoDB unterstützt weder Verbunde (Joins) noch Transaktionen, dafür werden aber Sekundäre Indizes, eine ausdrucksstarke Abfragesprache, atomare Schreibzugriffe auf Dokumenten und (voll) konsistente Lesezugriffe bereitgestellt.
DXRAM speichert sehr viele sehr kleine Objekte die aus unstrukturierten Binärdaten bestehen und permanent im RAM gehalten werden, wohingegen MongoDB in Form von Dokumenten strukturierte Daten auf Festplatte speichert und nur nach Möglichkeit einen Teil der Daten im RAM cacht. Statt einer Abfragesprache verfügt DXRAM über eine schmale Schnittstelle, die sehr flexible Einsatzmöglichkeiten bietet.

SAP HANA

SAP HANA ist der Kern von SAP's neuer Datenverwaltungsplattform und ist für analytische und transaktionale Zugriffe in einer hoch skalierbaren Umgebung entworfen [55]. Für beide Zugriffsarten wird das gleiche Datenmodell verwendet, das spaltenorientierte Tabellen mit

Hilfe eines Merging-Algorithmus verwendet. Neben SQL können auch ein paar weitere Abfragesprachen verwendet werden. Ein Anfrage wird zuerst in eine optimierte interne Repräsentation überführt, bevor ein Berechnungsgraph erzeugt wird. Um die Daten schnell zu adressieren kommen Indexstrukturen zum Einsatz.
SAP HANA ist eine RAM-basierte Datenbank, verwaltet Daten in Tabellenform und ist für Echtzeit-Transaktionsverarbeitung (OLTP) oder Echtzeit-Analysen (OLAP) ausgelegt, wofür eine umfangreiche Unterstützung von Abfragesprachen gegeben ist. Im Unterschied dazu ist DXRAM für soziale Netzwerke und Graphverarbeitung konzipiert und besteht aus Schlüssel-Wert-Paaren mit unstrukturierten Binärdaten. Eine schmale Schnittstelle ermöglicht einen vielfältigen Einsatz und flexiblen Zugriff nach Anwendungsanforderungen.

Sinfonia

Sinfonia ist eine Plattform für die einfache Implementierung von skalierbaren, fehlertoleranten verteilten Systemen [90]. Zu diesem Zweck werden ein verteilter RAM-Speicher, Mini-Transaktionen und Fehlertoleranzmechanismen zur Verfügung gestellt, so dass sich Programmierer nur noch um das Design der Datenstrukturen und die Manipulation der Daten kümmern müssen. Jeder Speicherknoten stellt einen eigenen linearen Adressraum zur Verfügung und speichert Daten entweder im RAM oder auf Festplatte beziehungsweise SSD. Alle Objekte besitzen eine globale ID, die aus der ID des zugehörigen Speicherknotens und einer Adresse in dessen Adressraum besteht. Der Zugriff auf die Objekte erfolgt mit Hilfe von Mini-Transaktionen, die die Manipulation von Daten in zwei Round-Trip-Zeiten ermöglichen. Da Sinfonia weder Lastverteilung noch Metadaten für eine Objektsuche zur Verfügung stellt, müssen sich jedoch entsprechende verteilte Anwendungen selbst um die Verwaltung von Speicher und globalen IDs kümmern.
Sinfonia stellt lediglich einen großen verteilten Speicher zur Verfügung, kümmert sich jedoch nicht um die Verwaltung des Speichers oder die Metadaten. DXRAM hingegen bietet ein Gesamtkonzept für die effiziente Verwaltung von sehr vielen sehr kleinen Objekten und kümmert sich dabei auch um die Metadaten für die Objektsuche und den schnellen Zugriff.

LSM-trie

LSM-trie ist ein festplatten-basierter Schlüssel-Wert-Speicher mit begrenztem RAM-Verbrauch für sehr viele sehr kleine Objekte und erweitert die Konzepte von LevelDB[6], um den zusätzlichen Schreibaufwand für die Datenverwaltung zu minimieren und Zugriffe insgesamt zu beschleunigen [91]. Daten werden mit Hilfe eines modifizierten LSM-Baumes (präfix-basierte Baum-Struktur), Bloomfiltern und Hashing verwaltet. Durch dieses Verfahren können Indizes

[6] `http://github.com/google/leveldb`

minimiert und der Datenzugriff kann in maximal zwei Lese- beziehungsweise Schreiboperationen erfolgen. Beim Datenzugriff werden Schlüssel gehasht und der resultierende Hashcode in einem Präfix-Baum (SSTable-trie) gesucht. Jedes Level des Präfix-Baumes entspricht einem Level in LevelDB und umfasst den Schlüssel-Bereich des Vaterknotens. Der Einsatz des Präfix-Baumes verhindert bei der Kompaktifizierung (Schlüssel-Wert-Paare werden vom Vaterknoten in die Schlüssel-Wert-Paare der Kinder einsortiert) eine Überlappung der Schlüssel-Bereiche und verringert dadurch die Anzahl der Zugriffe bei der Suche nach einem Schlüssel-Wert-Paar. Im LSM-trie werden größere Bloomfilter eingesetzt, um einzelne Level bei der Suche auszuschließen und die Anzahl und Größe der Indizes zu minimieren.Durch die größeren Bloomfilter lässt sich die Rate der False Positives selbst bei sehr vielen Leveln stark reduzieren. Da die Bloomfilter auf Grund ihrer Größe und der Anzahl der Knoten im SSTable-trie nicht alle im Hauptspeicher gecacht werden können, werden mehrere Bloomfilter zu einem Cluster zusammen gefasst und auf Festplatte gespeichert. Die Bloomfilter werden dabei so ausgewählt, dass alle für einen Datenzugriff notwendigen Bloomfilter mit einem einzelnen Lesezugriff (4 KB) eingelesen werden können. Anstatt der ursprünglichen index-basierten SSTable in LevelDB setzt LSM-trie auf die hash-basierte HTable, bei der jeder Datenblock als Bucket aufgefasst wird und Schlüssel-Wert-Paare auf einen der Buckets gehasht werden. Lediglich bei sehr großen Datenobjekten (selten), die in keinen Bucket passen, wird innerhalb der HTable ein Index eingesetzt und im RAM für einen schnellen Zugriff repliziert.
Bei DXRAM werden auch sehr viele sehr kleine Objekte verwaltet, allerdings werden diese permanent im RAM gehalten und nicht, wie beim LSM-trie, auf Festplatte gespeichert. Durch den wahlfreien Zugriff auf den RAM, ist es bei DXRAM nicht notwendig Schlüssel-Wert-Paare in sortierter Reihenfolge zu speichern und Lesezugriffe auf definierte Blockgrößen anzupassen.

Schlüssel-Wert-Speicher

Schlüssel-Wert-Speicher, wie zum Beispiel Amazons Dynamo [39] und Scatter [40], verwalten gleichfalls viele Schlüssel-Wert-Paare und beschäftigen sich mit Fehlertoleranz. Allerdings zielen diese Systeme nicht darauf ab alle Anwendungsdaten permanent im RAM zu halten.

Flash-Caches

Flash-Caches bieten einen persistenten Schlüssel-Wert-Speicher, der Flash-Speicher direkt oder als nicht-flüchtigen Cache zwischen RAM und Festplatte nutzt. Beispiele für solche System sind FlashStore [6] und BloomStore [7]. Diese Systeme verfolgen den Ansatz die Menge der Metadaten im RAM zu minimieren. In der Regel werden dazu Metadaten in den Flash-Speicher ausgelagert und nur bei Bedarf geladen, was zu langsameren Datenzugriff führt.

2.6 Zusammenfassung

RAM-basierte Speichersysteme sind, auf Grund ihres schnellen Zugriffs, zunehmend gefragter und besonders große interaktive Anwendungen profitieren vom schnellen Zugriff auf ihre Daten. Systeme, die nur einen Teil der Anwendungsdaten im Hauptspeicher cachen, reichen auf Dauer nicht mehr aus um den Bedarf dieser Anwendungen zu decken. Ferner müssen RAM-basierte Caches aktiv befüllt werden und sind bei irregulären Zugriffsmustern suboptimal. Ein Ausweg aus dieser Situation ist es zukünftig alle Anwendungsdaten permanent im Hauptspeicher zu halten.
DXRAM ist ein RAM-basierter Speicherdienst, der schnellen Zugriff auf Milliarden sehr kleiner Objekte gewährleistet, indem alle Objekte permanent im Hauptspeicher vieler zusammengeschlossener Rechner eines Rechenzentrums gehalten werden. Ein transparentes Hintergrundprotokoll sorgt für die Persistenz der Daten. Hierfür werden Replikate auf SSDs von mehreren Backupknoten geschrieben, die im Fehlerfall sehr schnell wiederherstellbar sind.
Ein fehlertolerantes Super-Peer-Overlay wird zur Strukturierung der Knoten und das Auffinden von Objekten eingesetzt. Jeder Super-Peer ist für eine Gruppe von Knoten zuständig, die er überwacht und um deren Metadaten er sich kümmert. Fällt ein Knoten aus, kontaktiert der zuständige Super-Peer die entsprechenden Backupknoten und koordiniert die Datenwiederherstellung.
Auf den einzelnen Knoten kommt eine eigene Adressübersetzung und Speicherverwaltung zum Einsatz. Die Adressübersetzung nutzt hierarchische Tabellen, um für eine globale ID die virtuelle Speicheradresse zu ermitteln, an der sich die zugehörigen Objektdaten befinden. Die Tabellen sind zusammen mit den Objektdaten in einem großen zusammenhängenden Speicherbereich abgelegt, der durch die Speicherverwaltung organisiert wird. Der zusammenhängende Speicherbereich und der konsequente Einsatz von Offset-Zeigern ermöglicht dabei eine Reduzierung des Speicherverbrauchs für die Metadaten.

Kapitel 3

Lokale Metadaten-Verwaltung

Die große Menge an zu verwaltenden Objekten stellt besondere Anforderungen an die Speicherverwaltung. So muss nicht nur der Zugriff auf die Daten sehr schnell erfolgen, sondern auch der Speicheraufwand minimiert werden. Bei bis zu einer Milliarde lokaler Objekte erhöht sich für jedes zusätzliche Byte (für Datenstrukturen, Metadaten, etc.) der insgesamte Speicherverbrauch um 1 GB. Aus diesem Grund wird darauf verzichtet Objekte an Cachezeilen oder Wortgrenzen auszurichten. Ein großer zusammenhängender Speicherbereich von mehreren Gigabyte (beispielsweise 32 GB) wird durch einen eigens entwickelten Speicherallokator verwaltet und beinhaltet alle Objekte und Verwaltungsstrukturen der lokalen Metadaten-Verwaltung. Dies ermöglicht den Einsatz von Offset-Zeigern und eine weitere Reduzierung des Speicheraufwands.
Das im folgenden vorgestellte Konzept für die lokale Metadaten-Verwaltung erlaubt eine schnelle Adressübersetzung mit konstantem Zeitaufwand und eine sehr effiziente Speicherung von Milliarden sehr kleiner Objekte und wurde in [59] und [67] publiziert.

3.1 Architektur

Nachfolgend wird die Architektur der lokalen Metadaten-Verwaltung betrachtet, die sich im Wesentlichen mit drei Aufgaben befasst. Die Speicherverwaltung erlaubt den schnellen Zugriff auf gespeicherte Chunks. Die Verwaltung von CIDs beschäftigt sich zum einen mit der Erzeugung der sequentiellen CIDs und zum anderen mit der Wiederverwendung von freigegebenen CIDs. Schließlich dienen die Zuordnungen von CIDs zu virtuellen Speicheradressen zum Auffinden von Chunks in der Speicherverwaltung.

3.1.1 Komponenten der lokalen Metadaten-Verwaltung

Die lokale Metadaten-Verwaltung besteht aus mehreren Komponenten, die die oben beschriebenen drei Aufgaben übernehmen. Die Komponenten sind eng miteinander verzahnt und greifen an verschiedenen Stellen aufeinander zu.
Die Zuordnung von CIDs zu virtuellen Speicheradressen erfolgt mit Hilfe von hierarchischen Tabellen. Neben der Adressübersetzung werden die Tabellen zusätzlich für die Wiederverwendung von freigegebenen CIDs benutzt. Eine detaillierte Betrachtung erfolgt in Abschnitt 3.2.
Chunks bestehen aus reinen Binärdaten und besitzen keine Zeiger oder Offsets auf andere Chunks. Aus diesem Grund arbeiten auch die Lese- und Schreibzugriffe auf Basis von Binärdaten. Die Speicherverwaltung agiert auf einem großen zusammenhängendem logischen Speicherblock (VMB), in dem neben den Chunks auch alle Verwaltungsstrukturen der lokalen Metadaten-Verwaltung abgelegt werden. Ein eigener Allokator wird verwendet, um Speicher zu allozieren und freizugeben. Abschnitt 3.3 beschäftigt sich ausführlich mit der Speicherverwaltung.

3.1.2 Funktionen der lokalen Metadaten-Verwaltung

Für die Verwendung der lokalen Metadaten-Verwaltung werden nach außen nur vier Funktionen zur Verfügung gestellt. Mit Hilfe dieser Funktionen ist es möglich Chunks zu schreiben, zu lesen und zu löschen. Eine weitere Funktion ermöglicht es eine oder mehrere (hintereinander liegende) freie CIDs zu beziehen.

Chunk schreiben

Beim Schreiben eines Chunks muss zwischen den beiden Fällen unterschieden werden, ob der Chunk bisher bereits existiert oder neu angelegt wird. Dafür wird überprüft, ob schon eine Zuordnung der CID zu einer virtuellen Speicheradresse vorhanden ist.
Wenn ein Eintrag existiert, liegen an der entsprechenden Speicheradresse die Chunkdaten der vorherigen Chunkversion, die anschließend überschrieben werden. Dies ist möglich, da ein Chunk nach der Erzeugung immer die gleiche Größe hat und dementsprechend alle Chunkversionen gleich viel Speicher benötigen. Die Vergrößerung (oder Verkleinerung) eines Chunks ist zwar grundsätzlich möglich, in diesem Fall wird der vorherige Chunk allerdings gelöscht und ein neuer Chunk mit der gleichen CID und der neuen Größe angelegt.
Wenn noch kein Eintrag existiert, ist auch noch kein Speicher für den Chunk vorhanden. Als Erstes muss daher Speicher durch die Speicherverwaltung alloziert werden. Die Speicherverwaltung ermittelt eine passende Adresse, an der genügend Speicherplatz zur Aufnahme des

Chunks besteht, und gibt die zugehörige virtuelle Speicheradresse zurück. Die Speicheradresse wird anschließend zusammen mit der CID in den Zuordnungen gespeichert. Zum Schluss werden die Chunkdaten an die ermittelte Speicheradresse geschrieben.

Chunk lesen

Die CID des zu lesenden Chunks wird genutzt, um die zugehörige Zuordnung der CID zu einer virtuellen Speicheradresse zu erhalten. Wenn kein Eintrag existiert, ist der Chunk nicht vorhanden und es wird `null` zurück gegeben. Andernfalls werden die Daten an der virtuellen Speicheradresse von der Speicherverwaltung ausgelesen. Die Menge der zu lesenden Daten ist in einem Kopf (Header) für die Speicheradresse hinterlegt. Die gelesenen Daten werden entweder direkt oder zusammen mit der CID als neues Chunk-Objekt zurück gegeben.

Chunk löschen

Wie beim Schreiben und Lesen eines Chunks wird auch beim Löschen zuerst die Zuordnung der CID zu einer virtuellen Speicheradresse ermittelt. Wenn kein Eintrag existiert, ist der Chunk nicht vorhanden und kann daher auch nicht gelöscht werden. Ansonsten wird zuerst der Eintrag in der Adressübersetzung gelöscht und anschließend die Daten an der ermittelten Speicheradresse. Die Speicherverwaltung kümmert sich schließlich um die Wiederverwendung des freigegebenen Speichers.

Freie CID anfordern

Bei jeder Chunkerzeugung muss dem Chunk eine eindeutige freie CID zugeordnet werden. Wie bereits in Abschnitt 2.1 vorgestellt, werden CIDs sequentiell erzeugt, wodurch die lokale Eindeutigkeit gewährleistet wird. Eine atomare Long-Variable wird in der Schnittstelle genutzt, die bei jeder Abfrage inkrementiert wird.
Abgesehen von den sequentiell erzeugten CIDs, können auch freigegebene CIDs (von gelöschten Chunks) wiederverwendet werden. Die Wiederverwendung von CIDs sorgt dafür, dass der ID-Raum kompakt bleibt und die Nutzung von hierarchischen Tabellen effizient ist. Allerdings ist eine Wiederverwendung nicht für jede Anwendung geeignet, da kürzlich gelöschte CIDs neuen Chunks zugewiesen werden. Wenn die Anwendung eine Lösch-Operation intern nicht propagiert, kann es sein, dass ein anderer Knoten auf den neuen Chunk zugreift, in der Absicht den alten Chunk zu erhalten. Die Wiederverwendung von CIDs lässt sich einschränken oder komplett abschalten, was aber nicht sinnvoll ist, da dadurch die Performanz beeinträchtigt werden kann.

Wenn CIDs wiederverwendet werden (Standard), koordiniert die Schnittstelle den Zugriff. Wird eine freie CID angefordert, wird zuerst geprüft, ob eine CID wiederverwendet werden kann. Nur wenn keine freigegebene CID zur Verfügung steht, wird eine neue sequentielle CID erzeugt. Werden mehrere hintereinander liegende CIDs angefordert, werden grundsätzlich neue sequentielle CIDs eingesetzt, da es unwahrscheinlich ist, mehrere hintereinander liegende freigegebene CIDs zu finden.
In Abschnitt 3.2 wird ausführlich beschrieben, wie freigegebene CIDs ermittelt und verwaltet werden.

3.2 Adressübersetzung (CID -> VMA)

Die Hauptaufgabe der Adressübersetzung ist das Ermitteln der virtuellen Speicheradresse für eine gegebene CID. In den ersten beiden Abschnitten werden mit Hashtabellen und Indizes die zwei gängigsten Ansätze für Abbildungen näher betrachtet. Anschließend werden die in DXRAM für die Adressübersetzung eingesetzten CID-Tabellen beschrieben. Zum Schluss wird das Verfahren erklärt, das zum Ermitteln und Verwalten von freigegebenen CIDs eingesetzt wird.

3.2.1 Hashtabellen

Viele RAM-basierte Speichersysteme, wie beispielsweise RAMCloud [80] und Trinity [63] nutzen Hashtabellen zur Adressübersetzung. Hashtabellen verwalten Schlüssel-Wert-Paare, wobei ein Schlüssel mit Hilfe einer Hashfunktion auf einen Index abgebildet wird. An der Indexposition in der Tabelle befindet sich der zugeordnete Wert. In diesem Kontext ist der Schlüssel die globale ID und der Wert die virtuelle Speicheradresse.
Der größte Vorteil von Hashtabellen ist die Zeitkomplexität der Operationen zum Lesen, zum Schreiben und zum Löschen eines Schlüssel-Wert-Paares. Alle drei Operationen können in $\mathcal{O}(1)$ durchgeführt werden, womit Hashtabellen eine sehr performante Adressübersetzung ermöglichen. Da Hashfunktionen jeden Schlüssel auf einen gültigen Tabellenindex abbilden, spielt es keine Rolle, ob der globale ID-Raum kompakt oder weit verstreut ist. Das ist besonders dann von Vorteil, wenn benutzerdefinierte IDs verwendet werden und das Speichersystem daher keinen Einfluss auf den ID-Raum hat.
Neben den genannten Vorteilen haben Hashtabellen allerdings auch einige Nachteile. Eine Hashfunktion bildet in der Regel eine große Eingabemenge (ID-Raum) auf eine kleine Zielmenge (Tabellenindizes) ab. Sie kann surjektiv sein, aber auf keinen Fall injektiv. Das bedeutet, dass mehrere Schlüssel auf den gleichen Index abgebildet werden können. Tritt solch ein Fall

auf, muss eine Kollisionsauflösung durchgeführt werden. Die Wahrscheinlichkeit einer Kollision ist relativ hoch. Schon 1939 zeigte Richard von Mises, dass selbst für wenige Eingabewerte und eine relativ große Zielmenge die Wahrscheinlichkeit einer Kollision sehr hoch ist [92]. Dieses Phänomen ist weithin als Geburtstagsparadoxon bekannt:

„Befinden sich in einem Raum mindestens 23 Personen, dann ist die Chance, dass zwei oder mehr dieser Personen am gleichen Tag (ohne Beachtung des Jahrganges) Geburtstag haben, größer als 50%.“

Für die Kollisionsauflösung gibt es eine ganze Reihe unterschiedlicher Algorithmen, die sich in zwei Hauptgruppen unterteilen lassen. Beim *Separate Chaining* wird jeder Tabellenplatz durch eine Datenstruktur repräsentiert, die mehrere Werte aufnehmen kann, beispielsweise eine Liste [93], ein Baum [94] oder ein dynamisches Array [95]. Im Gegensatz dazu werden beim *Open Addressing* die Werte direkt im entsprechenden Tabellenplatz gespeichert. Ist ein Tabellenplatz bereits belegt, wird eine Sondierungsfunktion ausgeführt, beispielsweise Lineares Sondieren, Quadratisches Sondieren oder Doppeltes Hashing [96]. Neben diesen klassischen Sondierungsfunktionen existieren weitere Varianten, die mit mehreren Tabellen und / oder mehreren Hashfunktionen arbeiten, zum Beispiel Cuckoo-Hashing [97, 98] und 2-Choice-Hashing [99]. Auch die Kombination mehrerer dieser Ansätze oder Abwandlungen sind verbreitet. Robin-Hood-Hashing ist eine Abwandlung des Doppelten Hashings und verbessert die Zeitkomplexität im schlechtesten Fall [100]. Hopscotch-Hashing vereint Cuckoo-Hashing und Lineares Sondieren und verbessert die Funktionsweise bei hohem Füllgrad [101]. Auch eine Kombination von Separate Chaining und Open Addressing ist möglich und wurde in [102] vorgestellt.
All diese Verfahren haben eine Gemeinsamkeit. Um eine Kollision überhaupt erkennen zu können, muss für jeden Wert auch der zugehörige Schlüssel gespeichert werden, wodurch zusätzlicher Speicherbedarf entsteht. Im konkreten Fall von DXRAM ist der Schlüssel eine CID mit 8 Byte und der Wert eine virtuelle Speicheradresse mit ebenfalls 8 Byte. Der zusätzliche Speicherverbrauch beträgt demnach 8 Byte beziehungsweise 100%. Wenn statt einer vollständigen Speicheradresse nur ein Offset mit weniger als 8 Byte verwendet wird, erhöht sich der prozentuelle Speichermehrverbrauch noch weiter.
Die Kollisionsauflösung ist ferner abhängig vom Füllgrad. Je größer der Füllgrad der Hashtabelle ist, desto wahrscheinlicher ist eine Kollision und desto aufwändiger ist die Kollisionsauflösung [95]. Im schlimmsten Fall muss die Hashtabelle vergrößert oder zumindest die Hashfunktion angepasst werden. Im Anschluss müssen alle Einträge neu gehasht und eingetragen werden. Die Kollisionswahrscheinlichkeit kann durch größere Tabellen verringert werden, erhöht aber gleichzeitig den Speicherverbrauch. Selbst die besten Implementierungen arbeiten mit einem maximalen Füllgrad von ungefähr 90%, was gleichzeitig einen ungenutzten Spei-

cherverbrauch von mindestens 10% bedeutet. Die meisten Implementierungen benötigen jedoch einen weitaus geringeren Füllgrad, um effizient zu arbeiten.
Zusammenfassend lässt sich festhalten, dass Hashtabellen eine exzellente Datenstruktur für die Adressübersetzung in weit verstreuten ID-Räumen sind. Auf Grund ihres Speicheraufwands sind sie jedoch ungeeignet für die Verwaltung von bis zu einer Milliarde kleiner Datenobjekte in einem kompakten ID-Raum. Bei einer Schlüssellänge von 8 Byte und einem Füllgrad von 90% beträgt der Mehraufwand für eine Milliarde Einträge mehr als 8 GB.

3.2.2 Indizes

Die meisten Datenbanken und tabellenbasierten Speichersysteme ermöglichen die Erzeugung von Indizes für Spalten oder Zeilen, um die Ausführung von Abfragen zu beschleunigen. Als Indizes werden verschiedene Datenstrukturen verwendet, die einen schnelleren Zugriff als das sequentielle Durchsuchen aller Einträge ermöglichen [103, 104]. Zumeist kommen dafür aber Varianten von B- und T-Bäumen [105, 106] zum Einsatz, die einen logarithmischen Zugriff ermöglichen.
Index-Bäume sind im Regelfall sortiert und ermöglichen somit eine schnelle Möglichkeit Schlüssel in sequentieller Reihenfolge zu durchsuchen, beispielsweise für Bereichsanfragen. Darüber hinaus kann die Nutzung von nur teilweise gefüllten Blöcken das Einfügen und Löschen von Einträgen beschleunigen [107]. Die meisten für Indizes genutzten Baumstrukturen sind für Plattenspeicher oder Flash-Speicher optimiert [108, 109, 110]. Dabei werden die Knoten auf Seiten- oder Blockgrößen optimiert, um die Anzahl der Zugriffe auf den Speicher zu minimieren.
Die genannten Vorteile sind auf die lokale Metadaten-Verwaltung nur bedingt anwendbar, da DXRAM aktuell keine Bereichsanfragen ermöglicht und die Indexstrukturen im RAM und nicht auf Festplatte oder SSD speichert. Die Nachteile von Indizes betreffen allerdings auch die lokale Metadaten-Verwaltung. So ist der logarithmische Zugriff, gerade bei sehr vielen Einträgen, bedeutend schlechter als beispielsweise der konstante Zugriff bei Hashtabellen. Darüber hinaus erfordert die Implementierung von Bäumen in objektorientierten Sprachen zumeist einen hohen Speicherbedarf für Zeiger und Objekte. Sortierte Bäume sind häufig ausgeglichen und erfordern zusätzlich eine regelmäßige Ausbalancierung bei Einfüge- und Löschoperationen.
Zusammenfassend kann festgehalten werden, dass Indizes besonders für Speichersysteme geeignet sind, die Festplatten oder Flash-Speicher verwenden. Bei RAM-basierten Systemen kommen die Vorteile von Indizes allerdings kaum zum Tragen, weshalb ein Einsatz in der Adressübersetzung nicht sinnvoll ist. Sollte zukünftig der Bedarf für Indizes oder Bereichsanfragen entstehen, können Indizes optional zusätzlich durch den Benutzer angelegt werden.

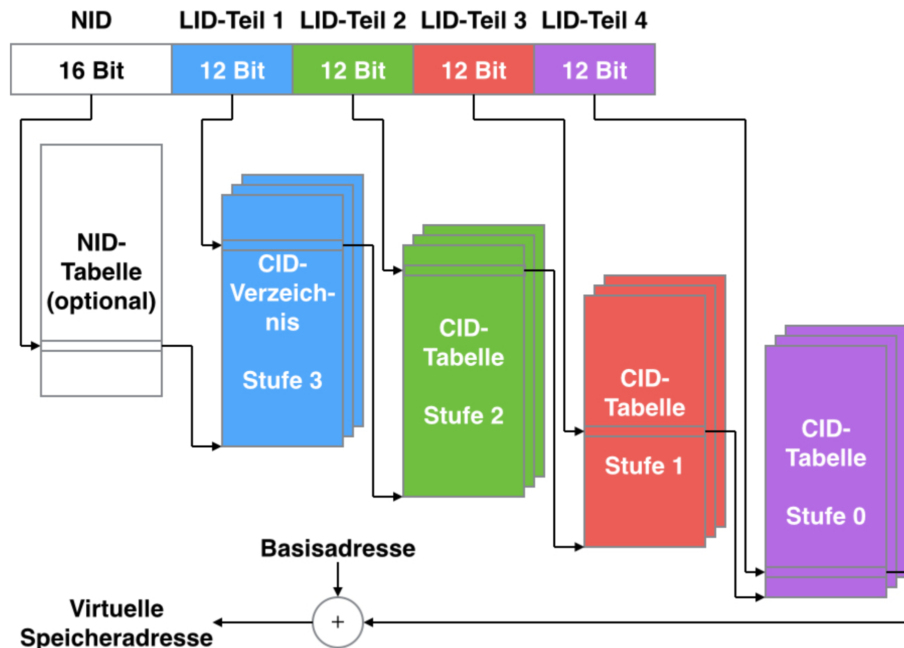


Abbildung 3.1: CID-Tabellen. *Die hierarchischen Tabellen ermöglichen eine effiziente Verwaltung von CIDs und virtuellen Speicheradressen.*

3.2.3 CID-Tabellen

Wie bereits in Abschnitt 2.1 erwähnt, werden die globalen IDs auf jedem Knoten in sequentieller Reihenfolge erzeugt. Die sequentielle Erzeugung ermöglicht die Nutzung von kompakten hierarchischen Übersetzungstabellen (hierarchischer Index). Die in Abbildung 3.1 vorgestellten *CID-Tabellen* haben Ähnlichkeit mit Paging-Tabellen aus der Speicherverwaltung klassischer PC-Betriebssysteme.
Bei der Verwendung der CID-Tabellen wird eine CID zuerst in NID und LID zerlegt und anschließend die LID noch einmal in n Teile gleicher Bit-Länge unterteilt (standardmäßig in vier Teile zu je 12 Bit). Der erste Teil der LID wird als Index in das CID-Verzeichnis verwendet, welches sich auf Stufe $n-1$ befindet. Das *CID-Verzeichnis* beinhaltet Zeiger, die jeweils auf eine CID-Tabelle verweisen. Mit Hilfe des ersten Teils der LID erhalten wir so eine CID-Tabelle auf Stufe $n-2$. In dieser Tabelle wird anschließend der zweite Teil der LID als Index genutzt. Auch in der CID-Tabelle sind Zeiger auf CID-Tabellen gespeichert, die sich auf Stufe $n-3$ befinden. In der, durch den Index bestimmten, CID-Tabelle wird der dritte Teil der LID wiederum als Index eingesetzt, wodurch wir eine CID-Tabelle auf Stufe $n-3$ erhalten. Dieser Ansatz wird so lange verfolgt, bis wir bei einer CID-Tabelle auf Stufe 0 ankommen. Die CID-Tabellen auf Stufe 0 enthalten keine Zeiger auf weitere CID-Tabellen, sondern beinhalten Offsets. Durch die Verwendung des letzten Teils der LID als Index, erhält man schließlich (mit dem Offset und

der Basisadresse des VMB) die virtuelle Speicheradresse des Chunks für die gegebene CID. Im weiteren Verlauf können die Chunkdaten, mit Hilfe der Speicherverwaltung, an der virtuellen Speicheradresse ausgelesen werden. Der hier beschriebene Ansatz erlaubt die Ermittlung virtueller Speicheradressen in $\mathcal{O}(1)$.
Die Größe einer CID-Tabelle hängt von der Anzahl der Einträge und der Länge eines Eintrags ab. Sei im Folgenden ein Index 12-Bit lang und eine Speicheradresse beziehungsweise ein Zeiger benötigt 8 Byte Speicher, dann enthält eine CID-Tabelle $2^{12} = 4.096$ Einträge und hat einen Speicherbedarf von $4.096 * 8$ B $= 32768$ B $= 32$ KB. Durch die Verkleinerung der Zeiger und Speicheradresse kann der Speicherbedarf der Tabellen erheblich reduziert werden. Aus diesem Grund werden in DXRAM statt vollständiger 64-Bit Zeiger nur 39-Bit Offset-Zeiger verwendet und die Einträge in den CID-Tabellen auf 5 Byte verkürzt. Mit 39-Bit Zeigern lassen sich 512 GB adressieren, was beim aktuellen Stand der DRAM-Technik ausreichend ist. Sollte jedoch zukünftig mehr Hauptspeicher zur Verfügung stehen, lassen sich die Zeiger auch vergrößern. Durch die Reduzierung lassen sich die CID-Tabellen auf $4.096 * 5$ B $= 20480$ B $= 20$ KB verkleinern.
Um den Speicherbedarf für einen einzelnen Eintrag zu ermitteln, muss die Größe einer CID-Tabelle durch die Anzahl der CIDs geteilt werden, die durch die CID-Tabelle verwaltet werden. Auf Stufe 0 ist eine CID-Tabelle für 4.096 CIDs zuständig. Mit zunehmender Stufe erhöht sich allerdings die Anzahl der CIDs. So ist eine CID-Tabelle auf Stufe 1 für $4.096 * 4.096 = 4.096^2 = 16.777.216$ CIDs, eine CID-Tabelle auf Stufe 2 für $4.096^3 = 68.719.476.736$ CIDs und eine CID-Tabelle auf Stufe 3 für $4.096^4 = 281.474.976.710.656$ CIDs zuständig. Allgemein ist eine CID-Tabelle der Stufe i für 4.096^{i+1} CIDs verantwortlich. Die Größe eines Eintrags lässt sich mit der folgenden Formel bestimmen:

$$\begin{aligned}
&\frac{20480}{4096}B + \frac{20480}{4096^2}B + \frac{20480}{4096^3}B + \frac{20480}{4096^4}B \\
&= 5B + \frac{5}{4096}B + \frac{5}{4096^2}B + \frac{5}{4096^3}B \\
&\approx 5B
\end{aligned}$$

Einen Eintrag umfasst demzufolge nur minimal mehr als 5 Byte. Da 5 Byte für die Speicherung der virtuellen Speicheradresse benötigt werden, erfordern die CID-Tabellen kaum zusätzlichen Speicher. Abgesehen davon bietet der hier vorgestellte Ansatz offensichtlich einen Zugriff in $\mathcal{O}(1)$.
Die CID-Tabellen werden durch die in Abschnitt 3.3 vorgestellte Speicherverwaltung erzeugt und verwaltet und befinden sich damit im gleichen Speicherbereich, in dem auch die Chunkdaten abgelegt sind. Anzumerken ist außerdem, dass CID-Tabellen nach Bedarf erzeugt und gelöscht werden und immer nur die benötigten CID-Tabellen existieren.
Wenn nicht nur selbst-erzeugte Chunks lokal gespeichert sind, sondern auch Chunks anderer

Knoten vorhanden sind, muss der Ansatz erweitert werden. In diesem Fall wird eine *NID-Tabelle* dynamisch erzeugt, in der die NID einer CID als Index verwendet wird (äquivalent zu den CID-Tabellen). Die NID-Tabelle beinhaltet Zeiger auf unterschiedliche CID-Verzeichnisse, für jeden Knoten, von dem Chunks gespeichert werden, eine. Die NID-Tabelle benötigt $65.536 * 5$ B $= 327.680$ B $= 320$ KB Speicher und existiert ebenfalls nur so lange, wie sie gebraucht wird und kann auch wieder gelöscht werden, sobald nur noch Chunks von einem Knoten existieren. Da Migrationen selten sind, ist im Regelfall eine NID-Tabelle nicht notwendig. Grundsätzlich lässt sich die NID-Tabelle auch durch eine kleine Hashtabelle ersetzen, da in der Regel nur Objekte von wenigen Knoten gespeichert werden. Da die Hashtabelle aber bei Bedarf vergrößert oder neu berechnet werden muss, ist die NID-Tabelle besser geeignet und kann problemlos eingesetzt werden.
Neben den bereits aufgezeigten Punkten, bietet der Einsatz der CID-Tabellen auch einen weiteren entscheidenden Vorteil. Durch die konsequente Verwendung von Offset-Zeigern und die Speicherung der CID-Tabellen zusammen mit den Chunks wird die in Abschnitt 2.4 vorgestellte lokale Wiederherstellung erheblich vereinfacht. Die lokale Wiederherstellung wird verwendet, wenn das System kontrolliert heruntergefahren und später fortgeführt wird, beispielsweise um Wartungsarbeiten durchzuführen. In diesem Fall wird der Speicherbereich, in dem die CID-Tabellen und die Chunkdaten gespeichert sind, komplett auf die lokale SSD geschrieben. Bei der Fortführung des Systems wird der Speicherbereich vollständig von der SSD gelesen und in den RAM geladen. Da in den CID-Tabellen nur Offset-Zeiger verwendet werden und Chunks keine Zeiger beinhalten kann der komplette Speicherbereich direkt wiederverwendet werden, selbst wenn sich die Basisadresse geändert hat.

Im nächsten Abschnitt wird erläutert, wie sich die CID-Tabellen abgesehen von der Adressübersetzung auch für die Ermittlung und Verwaltung von freigegebenen CIDs verwenden lassen.

3.2.4 Freie CIDs

In Abschnitt 2.3 und 3.1 wurde bereits erläutert, wann und warum freigegebene CIDs wiederverwendet werden. Im Folgenden wird beschrieben, wie freigegebene CIDs ermittelt und verwaltet werden. Zu diesem Zweck wird zuerst ein grundlegendes Konzept vorgestellt, das anschließend schrittweise erweitert wird.
Um zu wissen, ob eine CID freigegeben wurde, muss ein gelöschter Chunk erkannt werden. Wenn ein Chunk gelöscht wird, wird dies in der zugehörigen CID-Tabelle auf Stufe 0 vermerkt. Jeder Eintrag in einer CID-Tabelle umfasst 5 Byte beziehungsweise 40 Bit, wovon 39 Bit durch den Offset-Zeiger belegt sind. Das verbliebene Bit speichert, ob der zugehörige Chunk gelöscht

wurde (Lösch-Marker). Werden alle CID-Tabellen, angefangen beim CID-Verzeichnis, durchlaufen, lassen sich so gelöschte Chunks feststellen, deren CIDs wiederverwendet werden können. Die Indizes der CID-Tabellen auf den verschiedenen Stufen ergeben dabei die CID des gelöschten Chunks.
Dieses grundlegende Verfahren ist nicht besonders effizient, da im schlechtesten Fall alle CID-Tabellen durchlaufen werden müssen. Erschwerend kommt hinzu, dass die Wiederverwendung von CIDs der Erzeugung einer neuen sequentiellen CID vorzuziehen ist. Als Folge wird bei jeder Chunkerzeugung überprüft, ob eine freigegebene CID existiert.

CID-Zähler

Mit Hilfe eines simplen Zählers lässt sich die unnötige Suche nach einer freigegebenen CID vermeiden. Bei jedem Löschvorgang wird der Zähler inkrementiert. Wenn der Zähler größer 0 ist, existiert eine freigegebene CID die wiederverwendet werden kann und nur dann wird nach einer freigegebenen CID gesucht. Wird eine CID gesucht und gefunden, wird der Zähler anschließend dekrementiert.

Voll-Marker

Auf Stufe 0 wird das freie Bit in den Einträgen der CID-Tabellen genutzt um einen gelöschten Eintrag zu kennzeichnen. Auf den anderen Stufen wird das freie Bit für einen anderen Zweck verwendet. Wenn das Bit gesetzt ist, ist die verknüpfte (untergeordnete) CID-Tabelle voll besetzt. Das bedeutet, dass die CID-Tabelle und alle ihr untergeordneten CID-Tabellen keine gelöschten Einträge beinhalten. Wenn bei der Suche nach einer freigegebenen CID festgestellt wird, dass eine CID-Tabelle voll besetzt ist, wird das Bit in der darüber liegenden CID-Tabelle gesetzt. Beim Löschen eines Eintrags wird das Bit in allen darüber liegenden CID-Tabellen wieder gelöscht. Der beste und der schlechteste Fall bleiben zwar unverändert, aber durchschnittlich muss durch die Voll-Marker bei einer Suche nur noch das CID-Verzeichnis und eine CID-Tabelle je Stufe betrachtet werden.

CID-Cache

Eine weitere Optimierung ist der Einsatz eines Caches. Der Cache verwendet zum Speichern von freigegeben CIDs ein Array oder eine Liste fester Größe. Die Größe des Caches ist beschränkt, damit der Cache nicht unkalkulierbar viel Speicherplatz belegt. Beim Löschen eines Chunks wird der zugehörige Eintrag in den CID-Tabellen gelöscht und die Bits in den jeweiligen CID-Tabellen angepasst. Anschließend wird die CID des gelöschten Chunks in den Cache

eingefügt, solange noch Platz im Cache ist. Wird eine freie CID angefordert, wird zuerst überprüft, ob im CID-Cache eine freigegebene CID vorhanden ist. Nur wenn der Cache leer ist und noch freigegebene CIDs (CID-Zähler) vorhanden sind, wird eine Suche in den CID-Tabellen durchgeführt. Die Suche wird dabei so modifiziert, dass nicht bei der ersten gefundenen CID abgebrochen wird, sondern die CID-Tabellen komplett durchsucht werden. Die Suche wird nur dann frühzeitig beendet, wenn der CID-Cache wieder voll gefüllt ist oder keine weiteren freigegebenen CIDs (CID-Zähler) vorhanden sind. Dadurch wird verhindert, dass beim nächsten Zugriff auf den CID-Cache die CID-Tabellen erneut durchsucht werden müssen. Bei der Suche wird der Voll-Marker gelesen und gegebenenfalls gesetzt.

3.3 Speicherverwaltung für sehr viele sehr kleine Objekte

Für die lokale Metadaten-Verwaltung wurde eine eigene Speicherverwaltung inklusive Speicherallokator entwickelt, der speziell für die große Menge an sehr kleinen Objekten entworfen wurde. Bevor das Konzept der Speicherverwaltung präsentiert wird, wird zuerst auf traditionelle Speicherallokatoren eingegangen und warum deren Einsatz nicht sinnvoll ist.

3.3.1 Traditionelle Speicherallokatoren

Die Verwaltung von Millionen kleiner Objekte bedeutet für jeden Speicherallokator einen hohen Zeit- und Platzaufwand [111, 80]. Der hohe Platzaufwand ist durch die Menge der zu verwaltenden Allokationen bedingt. Der hohe Platzaufwand ergibt sich aus den Metadaten, die für jedes Objekt zusätzlich gespeichert werden müssen. Jeder Allokator muss für jede Allokation entsprechende Metadaten anlegen. Die Metadaten beinhalten beispielsweise die Länge des allozierten Speichers und werden als Header vor dem Objekt oder in einer separaten Datenstruktur abgelegt. Abhängig von Programmiersprache und Laufzeitumgebung umfassen die Metadaten pro Objekt zwischen 4 und 64 Byte und aus Performanzgründen werden Allokationen meist an Cachezeilen ausgerichtet (4- oder 8-Byte Grenzen). Viele Allokatoren, wie beispielsweise Hoard [112], TCMalloc [113] und jemalloc [114], sind besonders speichereffizient für Objekte mit der Größe von bestimmten Zweierpotenzen, schneiden aber schlecht ab, wenn die Objekte nur etwas größer als diese Zweierpotenzen sind.
Bei Objektgrößen von 16-64 Byte ist es evident, dass dieser Overhead (Metadaten + Ausrichtung an der Cachezeile) zu groß ist. Ein 16-Byte Mehraufwand pro Objekt resultiert in 16 GB Mehraufwand für eine Milliarde Objekte. Aus diesem Grund wird ein eigener Speicherallokator entworfen, der den Speicheraufwand besonders für die sehr kleinen Objekte minimiert.

Tabelle 3.1: *Zustände der Marker Bytes*

Zustand	Beschreibung
0000	freier Speicher (kleiner als 12 Byte)
0001	freier Block (Längenfeld ist 1 Byte groß)
0010	freier Block (Längenfeld ist 2 Byte groß)
0011	freier Block (Längenfeld ist 3 Byte groß)
0100	freier Block (Längenfeld ist 4 Byte groß)
0101	freier Block (Längenfeld ist 5 Byte groß)
0110	*aktuell nicht genutzt*
0111	*aktuell nicht genutzt*
1000	*aktuell nicht genutzt*
1001	belegter Block (Längenfeld ist 1 Byte groß)
1010	belegter Block (Längenfeld ist 1 Byte groß)
1011	belegter Block (Längenfeld ist 1 Byte groß)
1100	*aktuell nicht genutzt*
1101	*aktuell nicht genutzt*
1110	*aktuell nicht genutzt*
1111	freier Speicher (genau 1 Byte groß)

3.3.2 Initialisierung der Speicherverwaltung

Bei der Initialisierung wird ein großer logisch zusammenhängender virtueller Speicherblock (VMB) alloziert, der mehrere Gigabyte umfasst. Die Größe des VMB ist konfigurierbar, aber abhängig vom physikalischen Speicher. Verfügt der Knoten beispielsweise über 32 GB Hauptspeicher, so sollte der VMB nicht größer als 30 GB sein, damit ein Gigabyte für den Kernel und Systemdienste und ein weiteres Gigabyte als Puffer für weitere Datenstrukturen und temporäre Daten von DXRAM zur Verfügung steht. Durch diese Beschränkung soll Paging vermieden werden, welches die Performanz erheblich beeinträchtigen würde. Innerhalb des VMB arbeitet der im weiteren vorgestellte Allokator. Die Nutzung eines großen zusammenhängenden Speicherblocks hat zwei wesentliche Vorteile. Erstens muss der untergeordnete Speicherallokator nur einen einzigen Block verwalten und dementsprechend nur einmal Metadaten anlegen. Zweitens können innerhalb des VMB Offset-Zeiger verwendet werden, wodurch die Zeigergröße reduziert werden kann. Im hier vorgestellten Konzept kommen Zeiger mit einer Größe von 5 Byte zum Einsatz, womit ein Speicherblock von maximal 1 TB pro Knoten adressiert werden kann.

Innerhalb des VMB wird mit Blöcken gearbeitet. Die Blöcke können variable Größen ($\geq$ 12 Byte) haben und werden in frei und belegt unterteilt. Jeweils zwischen zwei Blöcken befindet sich ein einzelnes Byte, das gleichzeitig als Header für den nachfolgenden und Trailer für den vorherigen Block fungiert. Die oberen 4 Bits des so genannten Marker-Bytes sind der

Trailer und die unteren 4 Bits der Header. Header und Trailer können durch die Belegung der 4 Bits jeweils 16 Zustände annehmen, von denen aktuell 10 genutzt werden (siehe Tabelle 3.1). Die Zustände 1 bis 5 kennzeichnen einen freien Block, die Zustände 9 bis 11 werden für belegte Blöcke verwendet und die Zustände 0 und 15 werden für freien Speicher (kleiner als 12 Byte) genutzt.

Zum schnellen Auffinden eines passenden Blocks sind die freien Blöcke in doppelt verketteten Freispeicherlisten organisiert. Jede Liste enthält nur Blöcke größer oder gleich groß einem definierten Wert, beispielsweise $(12, 24, 36, 48, 64, 2^7, \ldots, 2^i)$. Die Zeiger für die Verkettung sind in den freien Blöcken selbst abgelegt und belegen daher keinen zusätzlichen Speicherplatz. Lediglich für die Anker der Listen muss weiterer Speicher genutzt werden. Neben den Zeigern befindet sich in den freien Blöcken am Anfang und am Ende ein repliziertes Längenfeld dynamischer Größe (1-5 Byte). Die minimale Größe eines freien Blocks von 12 Byte ergibt sich durch die beiden Listenzeiger (jeweils 5 Byte) und die beiden Längenfelder (jeweils mindestens 1 Byte). Ein Speicherbereich, der kleiner als 12 Byte ist, entspricht keinem freien Block und wird nicht in den Listen verkettet. Die Zustände 0 und 15 des Marker-Bytes werden verwendet, um dies kenntlich zu machen. Wenn der freie Speicher nur 1 Byte groß ist, nimmt das Marker-Byte den Zustand 15 an. Damit wird gekennzeichnet, dass nach dem Marker-Byte ein weiteres Marker-Byte folgt. Bei einem freien Speicherbereich von mehr als einem und weniger als zwölf Byte nimmt das Marker-Byte den Zustand 0 an. Die Länge des freien Speicherbereichs wird durch ein Längenfeld von einem Byte am Anfang und Ende festgelegt. Abbildung 3.2 zeigt das Layout eines freien Blocks beziehungsweise freien Speichers in den drei möglichen Ausprägungen.

Um den Speicherverbrauch zu reduzieren, sind belegte Blöcke nicht verkettet. Ein dynamisches Längenfeld (1-3 Byte) am Anfang (zwischen Marker-Byte und den eigentlichen Daten) enthält die Länge des Blocks. Damit lassen sich bis zu 16 MB große Blöcke anlegen, was für die Zielanwendungen, mit hauptsächlich sehr kleinen Objekten, mehr als ausreichend ist. Das Längenfeld lässt sich grundsätzlich auch vergrößern oder mit einem Offset in der Größe einer Zweierpotenz ergänzen. Wenn zum Beispiel alle Objekte mindestens 8 Byte groß sind, kann als Offset der Wert $2^3 = 8$ verwendet werden, womit es dann möglich ist Objekte zwischen 8 Byte und 256 MB zu verwalten.

Zu Beginn besteht der VMB aus einem einzigen freien Block, der den gesamten VMB ausfüllt. Auf eine Ausrichtung an Cachezeilen wird bewusst verzichtet. Zum einen wird dadurch interne Fragmentierung verhindert, zum anderen sind Cacheverzögerungen unkritisch, da die Anfragezeiten von den Übertragungszeiten im Netzwerk dominiert werden.

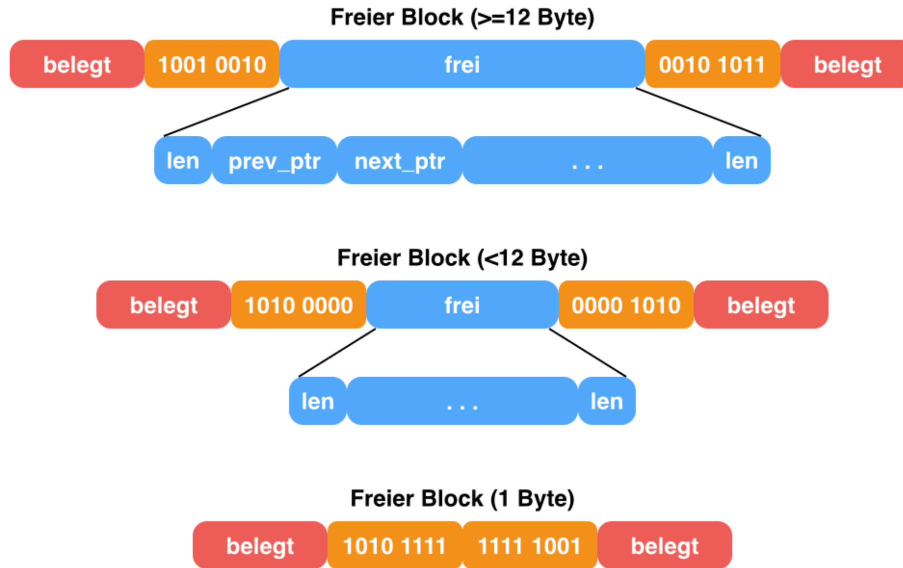


Abbildung 3.2: Layout von freiem Speicher. *Freier Speicher kommt in drei Ausprägungen vor. Ist der Speicher* ≥ 12 *Byte, wird ein freier Block erzeugt und in eine der Freispeicherlisten eingefügt. Ist der freiwerdende Speicher kleiner als 12 Byte kann kein Block erzeugt werden. Die Marker-Bytes zeigen den Fall an, indem die entsprechenden Bits auf 0 gesetzt werden. Beim dritten Fall ist der Speicher nur ein Byte groß. Hierbei handelt es sich um einen Spezialfall des Zweiten, der durch einen Wert von 15 in den Marker-Bytes erkennbar ist.*

3.3.3 Speicher allozieren (Malloc)

Bei der Speicherallokation müssen mehrere Schritte durchgeführt werden. Algorithmus 3.1 stellt die einzelnen Schritte vereinfacht dar.
Als erstes wird die Liste mit freien Blöcken der kleinsten passenden Größe betrachtet. Wenn die Liste leer ist, wird die Liste mit den nächstgrößeren Blöcken betrachtet, und so weiter. Der erste Block aus der gewählten Liste wird für die Allokation verwendet und aus der Liste entfernt. Durch diesen Auswahlprozess ist sicher gestellt, dass der gewählte Block auf jeden Fall groß genug ist. Das Entfernen des Blocks aus der Liste wird durch Umbiegen des Listenankers auf den nächsten freien Block der Liste erreicht. Der Vorgängerzeiger des jetzt ersten Blocks der Liste muss anschließend noch auf `0` gesetzt werden. Existiert kein Nachfolger, wird der Anker ebenfalls auf `0` gesetzt. Wenn der freie Block exakt so groß ist wie die angeforderte Allokation plus das Längenfeld, kann der freie Block direkt genutzt werden und es müssen lediglich die Marker-Bytes vor und hinter dem Block angepasst werden. Das Längenfeld muss nicht geschrieben werden, da der freie Block die gleiche Länge hatte. Wenn der freie Block jedoch größer ist, wird der Block geteilt und der erste Teil für die Allokation benutzt. Dabei wird das

Algorithmus 3.1 Speicher allozieren (vereinfachte Darstellung)

```
1:  function MALLOC(int size)
2:      int list = 0;
3:      long address;
4:      long freeSize;
5:
6:      // get not empty list with large enough free blocks
7:      while (list < List_Count && hasEntries(list)) do
8:          list++;
9:      end while
10:     if (list < List_Count) then
11:         // create new block
12:         address = getFreeBlock(list);
13:         unhookFreeBlock(list);
14:         createBlock(address, size);
15:
16:         freeSize = readSize(address);
17:         if (freeSize == size) then
18:             // case 1: exact size → nothing more to do
19:         else if (freeSize == size + 1) then
20:             // case 2: one byte to big → create additional marker
21:             createAdditionlMarker(address + size);
22:         else
23:             // case 3: more bytes to big → create additional free block
24:             createFreeBlock(address + size, freeSize − size);
25:         end if
26:     else
27:         address = 0;
28:     end if
29:
30:     return address;
31: end function
```

Längenfeld neu geschrieben und das Marker-Byte vor dem Block aktualisiert. Anschließend muss ein neues Marker-Byte hinter dem ersten Teil erzeugt werden. Das Marker-Byte belegt damit genau das erste Byte des zweiten Teils. Für den restlichen zweiten Teil muss zwischen drei Fällen unterschieden werden (siehe Abbildung 3.2):

1. Fall - Der restliche Speicher ist größer als 12 Byte: Der restliche Speicher ist groß genug für einen freien Block und wird in die passende Freispeicherliste eingefügt. Dafür werden die zugehörigen Zeiger und Längenfelder geschrieben und das Marker-Byte hinter dem ursprünglich verwendeten Block aktualisiert.

2. Fall - Der restliche Speicher ist kleiner als 12 Byte: Der restliche Speicher ist zu klein für einen freien Block und verbleibt als freier Speicher. Dieser Fall wird durch den Zustand 0 der beiden Marker-Bytes gekennzeichnet. Innerhalb des freien Speichers wird am Anfang und am Ende ein Längenfeld geschrieben.

3. Fall - Der restliche Speicher ist genau 1 Byte groß: Hierbei handelt es sich um einen Sonderfall des 2. Falls. In diesem Fall existiert noch nicht einmal genügend Speicher für ein Längenfeld. Das einzelne Byte wird durch ein zusätzliches Marker-Byte belegt, das den Zustand 15 annimmt. Der Zustand bestimmt, dass das nächste Byte ein weiteres Marker-Byte ist.

Der freie Speicher bei Fall 2 und 3 kann nicht in eine der Freispeicherlisten eingefügt werden und steht somit für weitere Allokationen nicht zur Verfügung. Trotz dessen ist der Speicher nicht verloren, sondern steht gegebenenfalls zu einem späteren Zeitpunkt wieder zur Verfügung. Im nächsten Abschnitt wird erläutert, wann der Speicher wieder genutzt werden kann.

3.3.4 Speicher freigeben (Free)

Bei der Freigabe von Speicher wird das Längenfeld an der übergebenen Speicheradresse ausgelesen und die Länge des freizugebenden Speichers bestimmt. Anschließend werden die benachbarten Blöcke betrachtet, ob sie ebenfalls frei sind (Marker-Bytes). Wenn einer oder beide benachbarten Blöcke frei sind, werden die Blöcke verschmolzen und ergeben einen größeren freien Block. Beim Verschmelzen wird auch freier Speicher < 12 Byte einbezogen. Anschließend werden die Blöcke aus ihren Freispeicherlisten entfernt und der resultierende neue freie Block in die passende Liste eingefügt. Das Verschmelzen erhöht die Chance bei Allokation freie Blöcke zu finden, die groß genug sind. Algorithmus 3.2 zeigt den vereinfachten Vorgang.

3.3.5 Speichereffizienz

Im Folgenden wird der Speicheraufwand, des hier vorgestellten Konzeptes, betrachtet. Zum Aufwand gehören der Platzbedarf für die Freispeicherlisten und der Speicher, der durch Marker-Bytes und Längenfelder der belegten Blöcke gefüllt ist.
Wie bereits erwähnt, beschränkt sich der Speicheraufwand der Freispeicherlisten auf den Platz für die Anker, da die Verknüpfung der freien Blöcke innerhalb der freien Blöcke selbst erfolgt. Pro Liste wird dementsprechend nur ein Zeiger benötigt. Die Anzahl der Freispeicherlisten ist abhängig von der Größe des VMB und kann darüber hinaus durch die Konfiguration angepasst werden. Die ersten vier Listen haben eine feinere Granularität als die Folgenden, die immer einer Zweierpotenz entsprechen. Diese Verteilung gewährleistet eine bessere Speicherzuteilung

Algorithmus 3.2 Speicher freigeben (vereinfachte Darstellung)

```
1: procedure FREE(long address)
2:     long size;
3:     long freeSize;
4:     int leftMarker;
5:     int rightMarker;
6:
7:     // read size and markers
8:     size = readSize(address);
9:     leftMarker = readLeftMarker(address);
10:    rightMarker = readRightMarker(address + size);
11:
12:    // check left marker
13:    if (leftMarker < 5 || leftMarker == 15) then
14:        // merge with free block or space
15:        freeSize = readLeftSize(address);
16:        address− = freeSize;
17:        size+ = freeSize;
18:    end if
19:
20:    // check right marker
21:    if (rightMarker < 5 || rightMarker == 15) then
22:        // merge with free block or space
23:        freeSize = readRightSize(address + size);
24:        size+ = freeSize;
25:    end if
26:
27:    // create new free block
28:    createFreeBlock(address, size);
29: end procedure
```

für Objekte kleiner als 64 Byte. Die letzte Liste wird so gewählt, dass der gesamte VMB in Form eines freien Blocks aufgenommen werden kann. Die Anzahl der Freispeicherlisten und ihr Speicherverbrauch können daher mit folgenden Formeln bestimmt werden:

Sei S_{VMB} die Größe des VMB, N_L die Anzahl der Listen und S_L die Größe der Listen, dann gilt für $2^{i-1} < S_{VMB} \leq 2^i$:

$N_L = c_{FG} + i - c_{ZP}$

$S_L = N_L * c_S$

Dabei ist c_{FG}, c_{ZP} und c_S wie folgt definiert:

$c_{FG}\hat{=}$Anzahl der Listen mit feinerer Granularität als eine Zweierpotenz
$c_{ZP}\hat{=}$kleinsten Wert für den eine Liste der Größe $2^{c_{ZP}+1}$ existiert,
die nicht zu den Listen mit feinerer Granularität zählt
$c_S\hat{=}$Größe eines Listenankers beziehungsweise eines Offset-Zeigers

Dabei entspricht c_{FG} der Anzahl der Listen mit feinerer Granularität als eine Zweierpotenz, c_{ZP} dem kleinsten Wert für den eine Liste der Größe $2^{c_{ZP}+1}$ existiert, die nicht zu den Listen mit feinerer Granularität zählt, und c_S der Größe eines Listenankers beziehungsweise eines Offset-Zeigers. Für das bisher betrachtete Beispiel mit den Listengrößen $(12, 24, 36, 48, 64, 2^7, \ldots, 2^i)$ und einem VMB mit 32 GB Speicher ergeben sich damit die folgenden Werte:

$$S_{VMB} = 32\text{ GB } = 34.359.738.368\text{ B}$$
$$2^{34} = 17.179.869.184 < 34.359.738.368 \leq 34.359.738.368 = 2^{35} \Rightarrow i = 35$$
$$c_{FG} = 4$$
$$64 = 2^6 = 2^{5+1} \Rightarrow c_{ZP} = 5$$
$$c_S = 5$$
$$N_L = 4 + 35 - 5 = 34$$
$$S_L = 34 * 5 = 170$$

Offensichtlich fällt der Speicherbedarf der Freispeicherlisten nicht ins Gewicht, insbesondere, da er unabhängig von der Anzahl der Blöcke ist.
Der Mehraufwand pro belegtem Block kann genauso einfach bestimmt werden. Für jeden Block fallen 1 Byte für das Marker-Byte (4 Bits Header + 4 Bits Trailer) und 1-3 Byte für das Längenfeld an. Das Längenfeld ist 1 Byte groß für Blöcke < 256 Byte, 2 Byte groß für Blöcke < 65.536 und 3 Byte groß für Blöcke $< 16.777.216$ Byte. Der zusätzliche Speicheraufwand pro Block entspricht daher 2-4 Byte. Für die Zielanwendungen von DXRAM beträgt dieser Mehraufwand in der Regel sogar nur 2 Byte.

3.3.6 Inkrementelle Defragmentierung

Jede Speicherverwaltung muss sich mit interner und externer Fragmentierung auseinander setzen. Ein stark fragmentierter Speicher kann dazu führen, dass Allokationen nicht durchgeführt werden können, obwohl insgesamt genügend Speicher zur Verfügung steht [115].

Durch die exakte Speicherallokation kann interne Fragmentierung gar nicht erst auftreten. Externe Fragmentierung ist allerdings möglich, da die Größe des VMB limitiert ist, Objekte gelöscht werden und Allokationsmuster sich dynamisch ändern können. Da DXRAM hauptsächlich für sehr viele sehr kleine Objekte konzipiert ist, kann davon ausgegangen werden, dass sich die Allokationsmuster nicht komplett hin zu großen Objekten verändern. Die Speicherverwaltung muss externe Fragmentierung aber schon alleine aus dem Grund behandeln, weil das System unter Umständen über lange Zeit läuft und der Speicher bestmöglich ausgelastet werden muss.
In der Speicherverwaltung wird aus diesem Grund eine inkrementelle Defragmentierung durchgeführt, die dynamisch die Fragmentierungsproblematik löst. Dabei wird ausgenutzt, dass im VMB keine Zeiger verwendet werden und die Datenblöcke somit leicht verschiebbar sind.

Fragmentierung

Bevor die eigentliche Defragmentierung durchgeführt werden kann, muss erst festgelegt werden, wann der Speicher als fragmentiert gilt. Da Fragmentierung kein festgelegter Wert ist, muss für jede Speicherverwaltung erst ein passender Wert definiert werden.
In der Speicherverwaltung wird ein freier Block als fragmentiert bezeichnet, wenn er kleiner als 64 Byte ist. In diesem Fall kann der Block nicht mehr alle Chunks der Zielanwendungen aufnehmen. Wenn beispielsweise der freie Block 40 Byte groß ist, kann er zwar noch für Chunks zwischen 16 und 40 Byte verwendet werden, aber nicht mehr für Chunks größer als 40 Byte. Der freie Block kann also nicht mehr alle Chunks zwischen 16 und 64 Byte aufnehmen. Aus diesem Grund gilt der Block zu diesem Zeitpunkt bereits als fragmentiert. Damit wird verhindert, dass viele kleine freie Blöcke existieren, aber zum Beispiel ein Chunk mit 64 Byte nicht angelegt werden kann.
Ein Speicherbereich ist fragmentiert wenn 75% der freien Blöcke fragmentiert sind. In einigen Fälle ist diese Bedingung allerdings unzureichend. Wenn der Speicherbereich nahezu komplett belegt ist und nur ein einziger fragmentierter Block existiert, würde der Speicherblock als fragmentiert gelten, obwohl dies offensichtlich nicht der Fall ist. Daher gilt ein Speicherbereich erst dann als fragmentiert, wenn eine Mindestzahl freier Blöcke existiert. Die Mindestanzahl entspricht dabei 1% der maximal möglichen 64-Byte-Blöcke. Kann ein Speicherbereich beispielsweise eine Millionen Chunks mit einer Größe von 64 Byte aufnehmen, entspricht die Mindestzahl dem Wert 10.000. Sowohl die 75% als auch die 1% Grenze wurden nicht wahllos bestimmt, sondern in mehreren Versuchen ermittelt. Sie erlauben den Status genau genug zu bestimmen, ohne einen Speicherbereich zu früh als fragmentiert zu kennzeichnen.
Dieser Sachverhalt wird noch einmal in den beiden nachfolgenden Definitionen festgehalten.

Definition 1 (Fragmentierungsgrad):
Sei FB die Anzahl der freien Blöcke (free blocks) und SB die Anzahl der freien Blöcke mit einer Größe kleiner als 64 Byte (small blocks).
*Dann ist der **Fragmentierungsgrad** definiert durch den Quotienten von FB und SB. Der Fragmentierungsgrad wird auch mit FD (fragmentation degree) bezeichnet.*

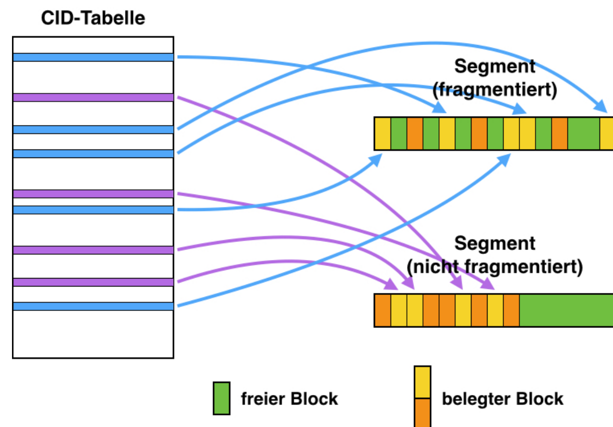
$$FD = \frac{FB}{SB}$$

Definition 2 (Fragmentiert):
*Sei MB die Anzahl der maximal möglichen 64-Byte-Blöcke (maximum blocks). Dann wird ein Speicherbereich als **fragmentiert** bezeichnet, wenn er mehr als $\frac{MB}{100}$ freie Blöcke enthält und der Fragmentierungsgrad größer als 75% ist.*

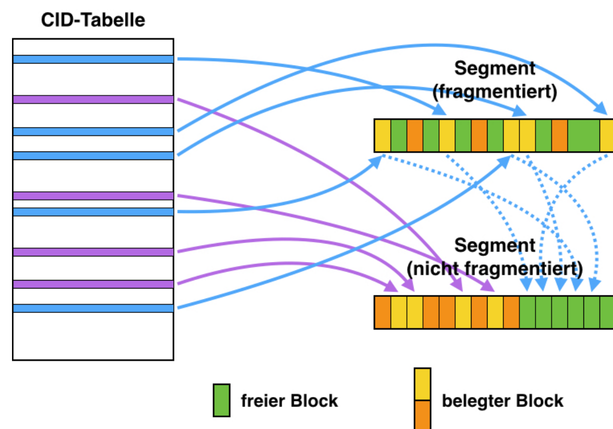
Um den Fragmentierungsgrad zur Laufzeit zu bestimmen, werden die Operation für die Allokation und die Freigabe von Speicher so angepasst, dass freie und belegte Blöcke direkt beim Erzeugen und Löschen protokolliert werden. Dieses Vorgehen ermöglicht den Fragmentierungsgrad mit geringem Aufwand aktuell zu halten.

Segmente

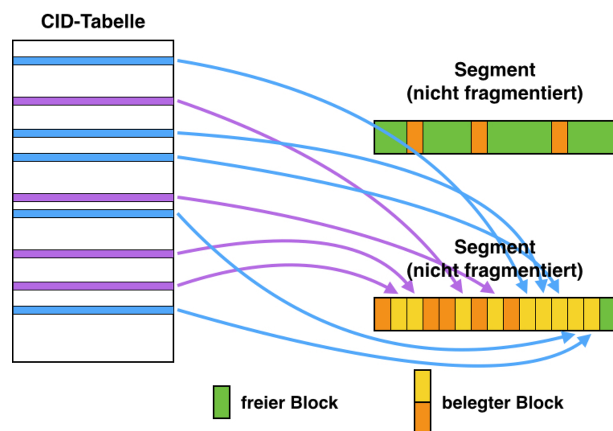
Der oben definierte Fragmentierungsgrad sagt lediglich aus, wann ein Speicherbereich fragmentiert und nicht wo die fragmentierten Blöcke sind. Wird beispielsweise der gesamte VMB als Speicherbereich betrachtet, ist der Fragmentierungsgrad relativ ungenau. Darüber hinaus ist eine Defragmentierung des gesamten VMBs sehr aufwändig. Um die Granularität zu verfeinern wird der VMB in mehrere logische *Segmente* gleicher Größe (beispielsweise 1 GB) unterteilt. Jedes Segment verwendet die zuvor beschriebene Speicherverwaltung und bestimmt seinen Fragmentierungsgrad nach den zuvor festgelegten Definitionen.
Die Aufteilung des VMB in Segmente ermöglicht es während der Defragmentierung Objekte aus einem fragmentierten Segment in ein nicht fragmentiertes Segment zu verschieben.



(a) Einträge überprüfen.



(b) Daten verschieben.



(c) Einträge anpassen.

Abbildung 3.3: Inkrementelle Defragmentierung. *Bei der inkrementellen Defragmentierung werden die Einträge einer CID-Tabelle auf Stufe 0 betrachtet. Zeigen Einträge in ein fragmentiertes Segment (a), werden die zugehörigen Daten in ein nicht fragmentiertes Segment verschoben (b). Anschließend werden die Einträge entsprechend angepasst (c).*

Inkrementelle Defragmentierungsstrategie

Während der Defragmentierung werden belegte Blöcke aus fragmentierten Segmenten verschoben. Das Problem ist das Auffinden der belegten Blöcke. Ohne Verkettung wäre es normalerweise notwendig den gesamten VMB oder zumindest einzelne Segmente komplett blockweise zu durchlaufen. Durch die Längenfelder in den freien und belegten Blöcken wäre dieses Vorgehen zwar möglich, aber auch sehr aufwendig. Nachfolgend wird ein inkrementeller Ansatz präsentiert, der es ermöglicht mehrere Segmente gleichzeitig zu defragmentieren.
Für die Defragmentierung wird auf die in Abschnitt 3.2 vorgestellte Verwaltungsstruktur der Adressübersetzung zugegriffen. In den CID-Tabellen auf Stufe 0 befinden sich die Speicheradressen der belegten Blöcke. In der Regel umfasst jede Tabelle 4.096 Einträge mit jeweils einer Speicheradresse. Die Speicheradressen liegen dabei nicht zwangsweise in einem Segment, sondern können sich in mehreren Segmenten befinden.
Während der Defragmentierung wird über eine CID-Tabelle der Stufe 0 iteriert und jede Speicheradresse überprüft. Wenn die Speicheradresse in einem fragmentierten Segment liegt, werden die dort gespeicherten Daten in ein nicht fragmentiertes Segment umkopiert und die Speicheradresse in der CID-Tabelle geändert (siehe Abbildung 3.3). Nachdem eine CID-Tabelle abgearbeitet wurde, wird die nächste CID-Tabelle gewählt und ebenso vorgegangen, und so weiter. Dieses Vorgehen erlaubt eine inkrementelle Ausführung, bei der in jedem Defragmentierungsschritt eine oder mehrere CID-Tabellen betrachtet werden. Die Anzahl lässt sich konfigurieren oder dynamisch bestimmen. Ein Defragmentierungsschritt kann in regelmäßigen Zeitintervallen, bei geringer Prozessorlast oder bei Bedarf ausgeführt werden. Auch eine komplette Defragmentierung kann, falls nötig, durchgeführt werden, indem alle CID-Tabellen der Stufe 0 nacheinander betrachtet werden. Die Defragmentierung wird von einem separaten Thread übernommen, der parallel zum eigentlichen Speichersystem auf jedem beliebigen CPU-Kern ausgeführt werden kann.

3.3.7 Besonderheiten bei Java

Viele Programmiersprachen und Laufzeitumgebungen erlauben keine direkte Allokation von Speicherbereichen oder den wahlfreien Zugriff innerhalb eines solchen Speicherbereichs.
In Java steht die Unsafe Klasse[1] zur Verfügung, die es ermöglicht aus dem Java Runtime Environment direkt auf die virtuelle Speicherverwaltung zuzugreifen. Java's Unsafe Klasse erlaubt es Speicherblöcken zu allozieren und, anhand ihrer Speicheradresse, wieder freizugeben. Weitere Methoden erlauben es Werte an gegebenen Speicheradressen zu lesen und zu schreiben. Durch den Einsatz der Unsafe Klasse für die Verwaltung von Objekten, kann der

[1] `http://www.docjar.com/docs/api/sun/misc/Unsafe.html`

Mehraufwand der Java Objektinstanzen vermieden werden. Grundsätzlich können die hier vorgestellten Konzepte aber unabhängig von der Programmiersprache eingesetzt werden.

3.4 Multi-Core-Optimierung

Lange Zeit wurde eine steigende CPU-Leistung durch eine Erhöhung der Taktfrequenz erreicht. Durch die damit verbundene Erhöhung der Spannung, des Energieverbrauchs und der notwendigen Kühlung, sind diesem Vorgehen physikalische Grenzen gesetzt [116, 117]. Die Verkleinerung des Fertigungsprozesses (*shrinking*) ist eine Möglichkeit zur Lösung, der allerdings auch Grenzen gesetzt sind und die sehr kostspielig ist.
Aus diesen Gründen werden seit dem Jahr 2005 Multicore-Porzessoren entwickelt und eingesetzt [116, 118]. Bei diesen Prozessoren werden mehrere Kerne auf demselben Die verbaut, wodurch eine parallele Verarbeitung auf einem einzelnen Rechner ermöglicht wird. Heutige Prozessoren verfügen über bis zu 18 Kerne [119] und erlauben damit die gleichzeitige Verarbeitung von mindesten 18 Prozessen beziehungsweise Threads. Mit Hilfe von Simultaneous Multithreading (SMT), beispielsweise Intels Hyper-Threading Technologie, kann die Parallelität weiter erhöht werden.
Um die vollständige Leistung dieser Prozessoren nutzen zu können, müssen Anwendungen für Nebenläufigkeit optimiert werden [120]. Greifen dabei mehrere Threads schreibend auf gemeinsame Speicherbereiche zu, müssen die Zugriffe synchronisiert werden, um einen deterministischen Ablauf der Anwendung zu gewährleisten. Zu diesem Zweck stehen Softwareentwicklern verschiedene Synchronisierungsmaßnahmen zur Verfügung.
In diesem Abschnitt wird erklärt, wie die in den vorherigen Abschnitten vorgestellte Adressübersetzung und Speicherverwaltung optimiert werden kann, um mit mehreren Threads gleichzeitig zugreifen zu können. In einem verteilten Speichersystem wird im Regelfall mit mehreren Threads (lokale Anfragen und Anfragen über das Netzwerk) auf die gespeicherten Daten zugegriffen, wodurch eine Synchronisierung unerlässlich ist. Hierfür werden nachfolgend verschiedene Synchronisierungsmaßnahmen und ihre Implementierungsaspekte betrachtet.

3.4.1 Synchronisierung

Bei der Synchronisierung geht es um die Koordinierung mehrerer konkurrierender Prozesse beziehungsweise Threads, die den gleichen kritischen Abschnitt ausführen wollen, in dem auf gemeinsame Speicherbereiche zugegriffen wird [121]. Die Worte Prozess und Thread werden im Weiteren gleichwertig verwendet.

Wenn ein Thread innerhalb des kritischen Abschnitts ist, müssen alle anderen Threads (die den kritischen Abschnitt ebenfalls betreten wollen) warten, bis der erste Thread den kritischen Abschnitt wieder verlässt. Dieses Vorgehen wird auch wechselseitiger Ausschluss genannt. Wenn Zugriffe mehrere Threads nicht korrekt synchronisiert werden, kann es zu einer Wettlaufsituation kommen, wodurch der Wert von Variablen unvorhersehbar wird [122, 123, 115]. Verändern beispielsweise zwei Threads den Wert einer Variablen und verwenden keine Synchronisierung, hängt der abschließende Wert der Variablen nicht mehr vom Programm, sondern von der Ausführungsreihenfolge der Threads ab, die sich bei jeder Ausführung ändern kann.
Nach Dijkstra muss der wechselseitige Ausschluss vier Bedingungen erfüllen [121]:

1. Führt ein Thread den kritischen Abschnitt aus, darf kein anderer Thread ebenfalls den kritischen Abschnitt ausführen.
2. Die relative Geschwindigkeit der CPU darf die Synchronisierung nicht beeinflussen.
3. Ein Thread außerhalb des kritischen Abschnitts darf einen anderen Thread nicht blockieren.
4. Wenn mehrere Threads den kritischen Abschnitt betreten möchten, muss irgendeiner der Threads in endlicher Zeit den kritischen Abschnitt betreten.

Die vierte Bedingung erlaubt allerdings, dass ein Thread unendlich lange darauf warten muss den kritischen Abschnitt zu betreten [124]. Dies wird als Starvation [125], Lockout [124] oder im Deutschen „Verhungern“ bezeichnet. Aus diesem Grund werden die vier oben genannten Bedingungen häufig durch ein fünfte (optionale) Bedingung ergänzt.

Kein Verhungern Ein Thread muss nicht unendlich lange auf den Eintritt in den kritischen Abschnitt warten [124, 126, 127].

Diese Bedingung kann noch weiter verschärft werden, so dass ein Thread den kritischen Abschnitt nicht mehrfach hintereinander durchlaufen darf, falls noch ein anderer Thread den kritischen Abschnitt betreten will [127].

Fairness Der Eintritt in den kritischen Abschnitt erfolgt entweder nach dem First-Come-First-Served-Prinzip (FCFS) [124, 128] oder nach dem schwächeren Konzept der linearen Wartezeit [127, 128].

Die häufigsten Probleme einer fehlenden oder fehlerhaften Synchronisierung sind nachfolgend kurz beschrieben:

Lost Update: Ein Lost Update kann auftreten, wenn mehrere Threads gleichzeitig eine Variable verändern. Bei einer ungünstigen Ausführung kann die Veränderung eines der Threads

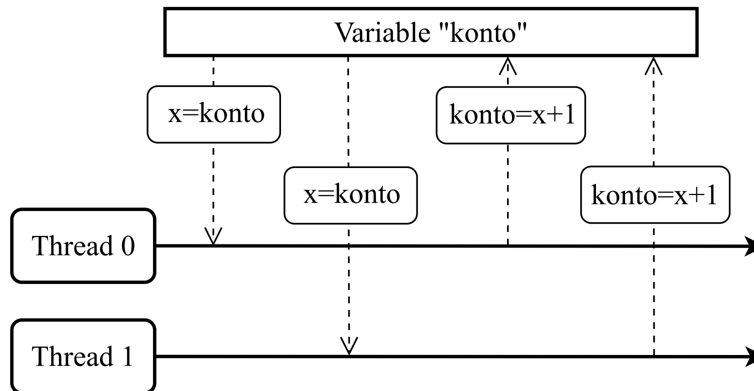


Abbildung 3.4: Lost Update (aus [129]). *Thread 0 und Thread 1 möchten gleichzeitig die Variable `konto` inkrementieren. Durch ein ungünstiges Threadscheduling, lesen beide Threads die Variable `konto` ein, bevor einer der Threads die Möglichkeit hatte den inkrementierten Wert zurückzuschreiben. Erst nachdem beide Threads gelesen haben, wird jeweils `konto + 1` geschrieben, wodurch eine Inkrementierung verloren geht.*

verloren gehen. Abbildung 3.4 zeigt die Ausführung zweier Threads, die jeweils die gleiche Variable inkrementieren wollen. Der erste Thread liest zunächst die Variable ein (Wert=0) und inkrementiert sie (Wert=1). Bevor allerdings der neue Wert zurückgeschrieben wird, kommt der zweite Thread an die Reihe, liest ebenfalls die Variable ein (Wert=0) und inkrementiert sie (Wert=1). Anschließend wird die Variable zurückgeschrieben (Wert=1). Erst jetzt kommt der erste Thread wieder an die Reihe und schreibt die Variable zurück (Wert=1). Trotz zweier Inkrementierungen hat die Variable am Ende den Wert 1. Es ist also eine Aktualisierung verloren gegangen.
Durch eine korrekte Synchronisierung kann dieses Problem vermieden werden.

Verklemmung: Bei einer Verklemmung sind zwei Threads jeweils in einem anderen kritischen Abschnitt und wollen den kritischen Abschnitt des jeweils anderen Threads betreten. Da beide Threads ihren kritischen Abschnitt nicht verlassen und folglich auch den kritischen Abschnitt des jeweils anderen Threads nicht betreten können, blockieren beide Threads und das Programm kann nicht weiter ausgeführt werden.

Verhungern: Wenn der Thread im kritischen Abschnitt diesen nicht mehr verlässt oder, nach dem Verlassen des kritischen Abschnitts, diesen sofort wieder betritt, kann es passieren, dass andere Threads nie den kritischen Abschnitt betreten können und verhungern.

Zur Synchronisierung mehrerer Prozesse können verschiedene Verfahren angewendet werden [130]. Diese Verfahren werden von der Hardware, vom Betriebssystem oder der Laufzeitumgebung zur Verfügung gestellt oder können softwareseitig umgesetzt werden. Nachfolgend werden die am häufigsten eingesetzten Verfahren vorgestellt.

Semaphore

Eine Semaphore besteht aus einem Zähler, einer Warteschlange und den zwei Operationen P und V [131]. Der Wert des Zählers entspricht, abstrakt gesehen, der Anzahl an Ressourcen, die zur Verfügung stehen. Die Anpassung des Zählers muss atomar erfolgen, wofür meist atomare Instruktionen der Hardware genutzt werden. Die P-Operation dekrementiert den Zähler (eine Ressource wird belegt) und blockiert den Thread, falls der Zähler anschließend kleiner als 0 ist. In diesem Fall wird der Thread in die Warteschlange eingefügt. Ist der Zähler nach der Dekrementierung größer oder gleich 0, wird der Thread einfach fortgesetzt. Die V-Operation inkrementiert den Zähler wieder und setzt einen blockierten Thread fort, falls die Warteschlange gefüllt ist. Bei einer Semaphore ist es möglich, dass ein Thread immer die P-Operation und ein anderer Thread immer die V-Operation ausführt.
Eine Semaphore, deren Zähler maximal den Wert 1 annehmen kann, wird als binäre Semaphore bezeichnet. Eine binäre Semaphore erlaubt immer nur einen Thread zur gleichen Zeit im kritischen Abschnitt.

Sperre

Die einfachste Form einer Sperre entspricht einer binären Semaphore und es kann sich immer nur ein Thread im kritischen Abschnitt befinden (exklusive Sperre). Im Gegensatz zu einer Semaphore muss allerdings der Thread, der die Sperre hält, diese auch wieder freigeben. Wie schon Semaphoren, nutzen meist auch Sperren atomare Instruktionen der Hardware.
Für den Fall, dass auf eine Variable überwiegend lesend und nur selten schreibend zugegriffen wird, gibt es spezielle Lese-Schreib-Sperren. Diese ermöglichen es, dass beliebig viele lesende Threads gleichzeitig in den kritischen Abschnitt eintreten. Ist allerdings ein schreibender Thread im kritischen Abschnitt, darf sich kein weiterer Thread (egal ob lesend oder schreibend) darin aufhalten.
Im Regelfall werden Threads blockiert, die in den kritischen Abschnitt nicht eintreten können, und geweckt, sobald der kritische Abschnitt wieder zur Verfügung steht. Anders verhält es sich bei Spinlocks, bei denen ein wartender Thread permanent die Sperre prüft und versucht zu erlangen (*busy waiting*).

Compare-and-Swap (CAS)

Bei Compare-and-Swap (CAS) handelt es sich um atomare Maschinenbefehl-Instruktionen, die eine Speicherstelle mit einem gegebenen Wert vergleichen. Sind beide Werte gleich, wird die Speicherstelle geändert. Wenn die beiden Werte unterschiedlich sind, passiert nichts. Der Erfolg oder Misserfolg der Operation kann auf zwei Weisen angezeigt werden. Entweder wird dazu ein Boolean-Wert oder der gelesene Wert der Speicherstelle verwendet.
Compare-and-Swap wird häufig für die Implementierung von Semaphoren und Sperren verwendet. Die CAS-Operation wird dabei benutzt, um den Zähler der Semaphore beziehungsweise den Status der Sperre zu verändern. Je nach Prozessorarchitektur gibt es unterschiedliche Befehle die ähnlich oder gleichwertig zu CAS Instruktionen sind, zum Beispiel Compare-and-Exchange, Test-and-Set, Fetch-and-Add, Load-Link/Store-Conditional.

Sperrfreie Algorithmen

Die Verwendung von atomaren Maschinenbefehl-Instruktionen, wie Compare-and-Swap, Test-and-Set, etc. ist vergleichsweise teuer und wird von einzelnen (älteren) Systemen nicht unterstützt. Aus diesem Grund wurden Algorithmen entwickelt, die ohne solche Instruktionen auskommen. Der Begriff sperrfreier Algorithmus ist dabei sehr missverständlich. Die hier vorgestellten Algorithmen dienen, genauso wie Sperren oder Semaphoren, dem wechselseitigen Ausschluss, verwenden jedoch keine atomarer Instruktionen, die eine Sperre auf Prozessor- oder Speicherebene erfordern, sondern sind komplett in der Anwendungsebene implementiert. In [129] werden drei der bekanntesten sperrfreien Algorithmen vorgestellt und miteinander verglichen. Die nachfolgend gezeigten Algorithmen sind den jeweiligen original Veröffentlichungen ([132, 133, 127]) entnommen und aufbereitet worden.

Petersons Algorithmus: Algorithmus 3.3 zeigt Petersons Algorithmus für `n` Prozesse [132]. Der Algorithmus verwendet ein Array `Q` der Länge `n` und ein Array `TURN` der Länge `n-1`, wobei `Q` mit Nullen und `TURN` mit Einsen initialisiert wird. Jeder Prozess muss `n-1` Level durchlaufen, bevor er in den kritischen Abschnitt eintreten kann. Gleichzeitig muss in jedem Level ein Prozess warten. Der Prozess der in Level `j` warten muss, wird in `TURN[j]` abgelegt und `Q[i]` speichert den höchsten Level, den Prozess `i` erreicht hat.
Die Level werden im Algorithmus durch die `for`-Schleife realisiert. In einem Schleifendurchlauf setzt jeder Thread sein höchstes erreichtes Level auf den aktuellen Schleifenzähler (`Q[i]:=j`) und schreibt seine Prozess-ID nach `TURN[j]`. Anschließend wartet der Prozess so lange, bis alle anderen Prozess in einem niedrigeren Level sind oder `TURN[j]` nicht mehr seiner Prozess-ID entspricht. Der „Gewinner“ im letzten Level tritt in den kritischen Abschnitt ein.

Nach dem Austritt aus dem kritischen Abschnitt setzt der Prozess sein Level zurück auf 0. Petersons Algorithmus ist nicht fair, es kann aber kein Prozess verhungern [126].

Algorithmus 3.3 Petersons Algorithmus für N Prozesse.

1: // `algorithm for process` P_i
2: **for** $j = 1$ **to** $N - 1$ **do**
3: $Q[i] = j$;
4: $TURN[j] = i$;
5: **wait until** $((\forall k \neq i, Q[k] < j)$ || $TURN[j] \neq i)$;
6: **end for**
7: *Critical Section*;
8: $Q[i] = 0$;

Lamports Bakery Algorithmus: Die Funktionsweise des Algorithmus ähnelt der Vorgehensweise in einer amerikanischen Bäckerei, woher er auch seinen Namen hat. Lamports Bakery Algorithmus löst das Problem des wechselseitigen Ausschlusses für `n` Prozesse fair und wird in Algorithmus 3.4 dargestellt.
Der Algorithmus verwendet zwei Integer-Arrays (`choosing` und `number`) der Länge `n`. Die Werte beider Arrays werden mit 0 initialisiert. Wenn ein Prozess `i` den kritischen Abschnitt betreten möchte, muss er erst eine Nummer ziehen, was er durch das Setzen von `choosing[i]:=1` ankündigt. Anschließend wird das Maximum der Werte im `number`-Array bestimmt und um Eins inkrementiert. Das Ergebnis ist die Nummer des Prozesses und wird in `number[i]` abgelegt und `choosing[i]` auf 0 gesetzt. Dieser erste Teil des Algorithmus wird als *Doorway* bezeichnet.
Da die Doorway nicht atomar ausgeführt wird, kann es passieren, dass mehrere Prozesse die gleiche Nummer haben. Die anschließende `for`-Schleife wird `n`-mal durchlaufen, wobei Prozess `i` jeden anderen Prozess `j` überprüft. Prozess `i` wartet solange, wie irgendein Prozess `j` noch eine Nummer zieht (`choosing[j]=1`) oder eine niedrigere Nummer hat. Wenn zwei Prozesse die gleiche Nummer haben, hat der Prozess mit der kleineren ID den Vorrang. Der Prozess, der am Ende die kleinste Nummer (ungleich 0) und (wenn notwendig) ID hat, darf in den kritischen Abschnitt eintreten. Nach dem Austritt wird die Nummer des entsprechenden Prozesses wieder auf 0 gesetzt.
Lamports Bakery Algorithmus ist fair (nach dem Wait-Free-Doorway-Prinzip [134]) und vermeidet Verhungern [133, 126].

Algorithmus 3.4 Lamports Backery Algorithmus für N Prozesse.

1: // `algorithm for process` P_i
2: $choosing[i] = 1$;
3: $number[i] = 1+$ *maximum(*$number[1], \ldots, number[N]$*)*;
4: $choosing[i] = 0$;
5: **for** $j = 1$ **to** N **do**
6:     **while** $(choosing[j] \neq 0)$;
7:     **while** $(number[j] \neq 0$ && $(number[j], j) < (number[i], i))$;
8: **end for**
9: *Critical Section*;
10: $number[i] = 0$;

Szymanskis Algorithmus: In Algorithmus 3.5 wird der von Szymanski in [127] beschriebene Algorithmus dargestellt. Er ermöglicht den wechselseitigen Ausschluss für `n` Prozesse, ist fair mit linearer Wartezeit und verhindert Verhungern.
Ein Prozess muss zuerst einen Warteraum (waiting room) passieren, bevor er den kritischen Abschnitt betreten kann. Der Warteraum wird durch eine erste Tür (door in) betreten und durch eine Zweite (door out) verlassen. Eine der Türen ist immer offen und die Andere geschlossen. Der Status jedes Prozesses wird in einem Byte-Array gehalten (`flag`), dessen Einträge zu Beginn 0 sind. Jeder Eintrag kann eine von fünf Ausprägungen haben:

0: Keine Absicht den kritischen Abschnitt zu betreten

1: Absicht den kritischen Abschnitt zu betreten

2: Warte auf andere Prozesse die erste Tür zu passieren

3: Soeben die erste Tür passiert

4: Soeben die zweite Tür passiert

Anfangs ist die erste Tür geöffnet und alle Prozesse, die die Tür passieren möchten, passieren diese und prüfen, ob es weitere Prozesse gibt, die die Tür passieren möchten. Solange noch Prozesse vorhaben die erste Tür zu passieren, bleibt diese geöffnet. Wenn kein Prozess mehr passieren möchte, wird die erste Tür geschlossen und die zweite Tür geöffnet. Alle Prozesse, die beabsichtigen den kritischen Abschnitt zu betreten, versammeln sich demnach zuerst im Warteraum.
Alle Prozesse im Warteraum betreten nun nacheinander (geordnet nach ihrer Prozessnummer) den kritischen Abschnitt. Erst, wenn alle Prozesse den kritischen Abschnitt durchlaufen haben, wird die zweite Tür geschlossen und die erste Tür wieder geöffnet, so dass die nächste Gruppe an Prozessen den Warteraum betreten kann.

Algorithmus 3.5 Szymanskis Algorithmus für N Prozesse.

1: // `algorithm for process` P_i
2: $flag[i] = 1$;
3: **wait until** $(\forall j, flag[j] < 3)$;
4: $flag[i] = 3$;
5: **if** $(\exists j, flag[j] = 1)$ **then**
6: $\quad flag[i] = 2$;
7: $\quad$ **wait until** $(\exists j, flag[j] == 4)$;
8: **end if**
9: $flag[i] = 4$;
10: **wait until** $(\forall j < i, flag[j] < 2)$;
11: *Critical Section*;
12: **wait until** $(\forall j > i, flag[j] < 2$ && $flag[j] > 3)$;
13: $flag[i] = 0$;

Threads, die aktuell den kritischen Abschnitt nicht betreten können, werden bei keinem der drei Algorithmen schlafen gelegt oder in eine Warteschlange eingefügt. Statt dessen wird aktives Warten (busy-waiting) eingesetzt, wobei kontinuierlich geprüft wird, ob der kritische Abschnitt betreten werden kann.
Das Verhungern eines Threads ist bei keinem der drei Algorithmen möglich, Fairness wird aber nur bei Szymanskis und Lamports Algorithmus gewährleistet.

Synchronisierung in Java

In der Java-Laufzeitumgebung existieren mehrere Möglichkeiten zur Synchronisierung von Threads. Die wichtigsten Umsetzungen sind die *ReentrantLock-Klasse*, die *Semaphore-Klasse* und das *Synchronized-Konzept*.

ReentrantLock: Seit Javaversion 1.5 existiert die Klasse ReentrantLock[2], mit dessen Hilfe sich exklusive Sperren realisieren lassen [135]. Für Lese-Schreib-Sperren ist darüber hinaus die Klasse ReentrantReadWriteLock[3] vorhanden.
Vor dem Betreten des kritischen Abschnitts muss der entsprechende Thread die `lock`-Methode aufrufen. Nach dem kritischen Abschnitt muss derselbe Thread die `unlock`-Methode ausführen. Wenn beim Aufruf der `lock`-Methode die Sperre bereits von einem anderen Thread gehalten wird, blockiert der aufrufende Thread bis die Sperre wieder frei ist.

[2] `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html`
[3] `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html`

Der Namenszusatz *reentrant* steht dafür, dass die Sperre von einem Thread auch mehrfach gehalten werden kann. Eine Zählvariable hält nach, wie oft der Thread die Sperre angefordert hat und erst, wenn die Zählvariable `0` ist, steht die Sperre für andere Threads wieder zur Verfügung. Es muss also für jeden `lock`-Aufruf auch ein `unlock`-Aufruf ausgeführt werden.
Bei der Erzeugung einer Instanz des ReentrantLock lässt sich angeben, ob die Sperre fair sein soll oder nicht. In beiden Fällen werden blockierte Threads in eine FIFO-Warteschlange eingefügt. Wenn die Sperre freigegeben wird, erhält im fairen Fall der erste Thread in der Warteschlange (mit der längsten Wartezeit) die Sperre, so dass ein Verhungern ausgeschlossen ist [135, 136]. Im nicht fairen Fall konkurriert der erste Thread mit einem möglichen Neuankömmling. Ein Neuankömmling darf dabei zuerst versuchen die Sperre zu erhalten (barging FIFO [135]). Ein Thread, der die Sperre freigibt und direkt wieder erhalten will, gilt in diesem Zusammenhang als Neuankömmling. Daher kann es bei einem ungünstigen Threadscheduling der JVM zum Verhungern kommen [135, 137].

Semaphore: Wie auch das ReentrantLock wurde die Semaphore-Klasse in Java 1.5 eingeführt. Die Klasse kann sowohl für reguläre Semaphoren, als auch für binäre Semaphoren verwendet werden. Dafür wird dem Konstruktor die Anzahl der Ressourcen (Permits) übergeben, die als Initialwert für den Semaphorenzähler verwendet werden.
Der Zugriff auf die Semaphore erfolgt über die Methoden `aquire` (P) und `release` (V). Die `aquire`-Methode erniedrigt den Semaphorenzähler und die `release`-Methode erhöht ihn. Im Gegensatz zum ReentrantLock können die Methoden von unterschiedlichen Threads aufgerufen werden.
Ist der Semaphorenzähler 0, wenn die `aquire`-Methode aufgerufen wird, blockiert der Thread und wird in eine FIFO-Warteschlange eingefügt. Ebenso, wie beim ReentrantLock, kann bei der Erzeugung der Semaphore angegeben werden, ob das Verhalten fair sein soll oder nicht [135]. Das Verhalten bezüglich der Fairness erfolgt äquivalent zum ReentrantLock.

synchronized: Das Schlüsselwort synchronized existiert seit der Javaversion 1.0 und verwendet Monitore für die Synchronisierung von Threads [138]. Mit einem Monitor kann ein Programmteil synchronisiert werden, ohne dass der Programmierer explizit Synchronisierungsoperationen aufrufen muss [139, 140]. In Java ist jedes Objekt und jede Klasse mit einem Monitor verknüpft.
Für synchronized gibt es drei Anwendungsfälle:

1. Instanz-Methode: Beim Einsatz von synchronized als Methodenattribut einer Instanz-Methode, wird der Monitor des Objektes verwendet, auf dem die Methode aufgerufen wird. Die gesamte Methode entspricht dabei dem kritischen Abschnitt. Das synchronized-Konzept geht aber noch weiter. Sind mehrere Methoden eines Objektes mit synchronized gekennzeichnet, erstreckt sich der kritische Abschnitt über all diese Methoden. Befindet sich ein Thread in einer

Tabelle 3.2: *Testsysteme (aus [129])*

	System i5	**System Opteron**
CPUs	1x Intel i5-2400 (Sandy Bridge)	2x AMD Opteron 6344 (Abu Dhabi)
Kerne pro CPU	4	12
Kerne Insgesamt	4	24
L2/L3 Caches pro CPU	L2: 4x 256 KB L3: 1x 6 MB	L2: 6x 2 MB L3: 2x 8 MB
Basistakt einer CPU	3.100 MHz	2.600 MHz
Turbotakt einer CPU	1 Kern: 3.400 MHz 2-3 Kerne: 3.300 MHz 4 Kerne: 3.200 MHz	1-6 Kerne: 3.200 MHz 7-12 Kerne: 2.900 MHz
RAM	16 GB	96 GB
OS	Ubuntu 14.04.1, 64 Bit	Linux Mint 17, 64 Bit
Java	1.7, 64 Bit	1.7, 64 Bit

gekennzeichneten Methode, kann kein anderer Thread in diese oder eine andere gekennzeichnete Methode eintreten.
2. Statische Methode: Das gleiche Verhalten, wie bei Instanz-Methoden, gilt auch für statische Methoden, allerdings wird der Monitor der Klasse und nicht der eines einzelnen Objektes verwendet. Alle statischen Methoden einer Klasse, die mit synchronized gekennzeichnet sind, bilden dabei einen kritischen Abschnitt.
3. Anweisungsblock: Wenn synchronized als Anwendungsblock eingesetzt wird, muss das Monitor-Objekt explizit angegeben werden [141]. Auch in diesem Fall bilden alle synchronized-Anweisungsblöcke, die dasselbe Monitor-Objekt verwenden, einen kritischen Abschnitt.
Die Verwendung von synchronized ist grundsätzlich nicht fair und es kann zusätzlich auch zum Verhungern kommen [137].

Vergleich unterschiedlicher Implementierungen

In [129] werden die zuvor vorgestellten Algorithmen miteinander verglichen. Zum Einsatz kommen zwei Testsysteme mit 4 und 24 Kernen (siehe Tabelle 3.2). Nachfolgenden werden die Ergebnisse wiedergegeben und in den vorliegenden Kontext eingeordnet.
Bei den Javaklassen ReentrantLock und Semaphore werden sowohl die fairen, als auch die nicht fairen Implementierungen verwendet. Für die sperrfreien Algorithmen werden jeweils zwei unterschiedliche Java-Implementierungen eingesetzt, die für den atomaren Zugriff auf

Tabelle 3.3: Speicherverbrauch der Synchronisierungsobjekte für n Threads ($n \geq 1$) (aus [129])

Synchronisierungsobjekt	Speicherverbrauch (in Byte)
Petersons Algorithmus (volatile)	$40 + 20n + 8(\lceil \frac{n}{8} \rceil) - 4(n \mod 2)$
Petersons Algorithmus (Unsafe)	$48 + 2n - 1$
Lamports Bakery Algorithmus (volatile)	$56 + 48n + 8(n \mod 2)$
Lamports Bakery Algorithmus (Unsafe)	$48 + 9n$
Szymanskis Algorithmus (volatile)	$40 + 20n + 4(n \mod 2)$
Szymanskis Algorithmus (Unsafe)	$48 + n$
ReentrantLock (fair, nicht fair)	48
Semaphore (fair, nicht fair)	48

Variablen einmal auf *volatile*[4] und einmal auf die Unsafe-Klasse zurückgreifen. Verglichen werden die Ausführungszeiten und der Speicherverbrauch. Während der Speicherverbrauch auf beiden Systemen identisch war (siehe Tabelle 3.3), unterscheiden sich die Ausführungszeiten auf den beiden Systemen erheblich (siehe Tabelle 3.4).
Zusammenfassend lässt sich sagen, dass die Javakonstrukte den geringsten Speicherverbrauch haben, der unabhängig von der Threadanzahl konstant ist, wohingegen die sperrfreien Algorithmen für jeden Thread zusätzlichen Speicher benötigen.
Die Performanz der Algorithmen und Implementierungen unterscheidet sich teilweise erheblich, die unfairen Javakonstrukte (besonders ReentrantLock und Semaphore) schneiden jedoch auf beiden Systemen am Besten ab. Wenn allerdings nur die fairen Algorithmen und Implementierungen betrachtet werden, zeigt sich, dass die sperrfreien Algorithmen eine bedeutend bessere Performanz zeigen, als die Javaimplementierungen. Systemübergreifend zeigt Lamports Bakery Algorithmus die beste Ausführungszeit und ist daher das Mittel der Wahl für eine faire Synchronisierung [129].

Sinnvoller Einsatz unterschiedlicher Synchronisierungsmöglichkeiten

Nachdem verschiedene Synchronisierungsmöglichkeiten vorgestellt wurden, wird nachfolgend die Frage beantwortet, wann welche Möglichkeit am sinnvollsten ist (siehe Tabelle 3.5). Dabei wird an dieser Stelle nur der Einsatz von exklusiven Sperren betrachtet, bei denen die Threads gleichberechtigt sind und grundsätzlich konkurrierende Zugriffe erfolgen. Führen hingegen die Threads überwiegend konfliktfreie Zugriffe aus, erlauben Lese-Schreib-Sperren eine wesentlich höhere Parallelität und sollten daher bevorzugt werden.

[4] Variablen, die als volatile gekennzeichnet sind, besitzen in Java Garantien bezüglich Ordnung, Sichtbarkeit und atomarer Eigenschaften [129] (siehe `http://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html`

Tabelle 3.4: *Zeit pro Sperre in Nanosekunden (Werte aus [129])*

Synchronisierungobjekt	System i5			System Opteron		
	2 Thr.	3 Thr.	4 Thr.	4 Thr.	12 Thr.	24 Thr.
Petersons Alg. (volatile)	78	122	175	1.244	5.948	11.590
Petersons Alg. (Unsafe)	63	88	127	832	5.902	10.147
Lamports Bakery Alg. (volatile)	109	123	150	668	1.035	1.259
Lamports Bakery Alg. (Unsafe)	75	101	126	717	1.992	2.226
Szymanskis Alg. (volatile)	191	210	221	1.040	1.819	1.870
Szymanskis Alg. (Unsafe)	135	152	157	922	1.882	2.367
ReentrantLock (fair)	2.292	2.383	2.416	17.637	18.841	28.302
ReentrantLock (nicht fair)	72	53	56	161	149	123
Semaphore (fair)	2.277	2.350	2.401	17.857	18.703	24.494
Semaphore (nicht fair)	99	52	50	119	122	86
synchronized-Block	48	51	50	301	566	649

Tabelle 3.5: *Einsatz unterschiedlicher Synchronisierungsmöglichkeiten*

	nicht fairer Algorithmus	fairer Algorithmus (Aktives Warten)	fairer Algorithmus (Schlafende Threads)
Anzahl Threads	begrenzt	nicht begrenzt	nicht begrenzt
Kritischen Abschnitts	kurz	kurz	lang
Anzahl Zugriffe	wenige	wenige	viele
Zeit zwischen Zugriffen	viel	viel	kurz
Entscheidender Durchsatz	Gesamtsystem	einzelner Thread	einzelner Thread

Entscheidend für die Auswahl einer passenden Synchronisierungsmöglichkeit ist die Frage nach der Fairness. Nicht faire Algorithmen haben eine wesentlich bessere Performanz und erlauben dadurch einen höheren Durchsatz des Gesamtsystems. Der größte Nachteil dieser Algorithmen ist jedoch, dass ein Thread verhungern kann oder zumindest lange nicht zum Zuge kommt, wodurch der Durchsatz eines einzelnen Threads erheblich sinken kann. Entscheidend dabei ist, wie viele Threads gegebenenfalls gleichzeitig auf die Sperre zugreifen können beziehungsweise sollen. Je mehr Threads potenziell auf eine Sperre zugreifen, desto höher ist die Wahrscheinlichkeit einer Wettlaufsituation und dementsprechend das Verhungern eines Threads. Die Wahrscheinlichkeit erhöht sich weiter, wenn derselbe kritische Abschnitt häufig schnell hintereinander ausgeführt wird oder sich ein Thread sehr lange im kritischen Abschnitt aufhält. Nicht faire Algorithmen eignen sich daher vor allem für die beiden Fälle, dass nur eine begrenzte Anzahl an Threads gleichzeitig in einen kritischen Abschnitt eintreten wollen oder der kritische Abschnitt nur einen kleinen Teil der jeweiligen Threadlaufzeit ausmacht. Von den oben vorgestellten Implementierungen sind die nicht faire Semaphore und das Schlüsselwort synchronized die besten Möglichkeiten.

Wenn die Anzahl der Threads, die potenziell in den gleichen kritischen Abschnitt eintreten wollen, nicht begrenzt ist oder der kritische Abschnitt während der Threadlaufzeit häufig kurz hintereinander ausgeführt wird, sollten faire Algorithmen bevorzugt werden. Dies gilt insbesondere, wenn das Gesamtsystem definierte Reaktionszeiten erfüllen muss. Durch die fairen Algorithmen wird sicher gestellt, dass ein Thread nur eine begrenzte Zeit auf den Eintritt in den kritischen Abschnitt warten muss. Bei fairen Algorithmen ist außerdem entscheidend, wie lange sich ein Thread im kritischen Abschnitt aufhält. Ist der kritische Abschnitt vergleichsweise kurz, ist der Einsatz eines Algorithmus mit aktivem Warten (zum Beispiel die vorgestellten sperrfreien Algorithmen) sinnvoll, da die Reaktionszeit minimiert wird. Das Aufwecken eines schlafenden Threads hingegen erfordert einige Zeit und daher eignen sich entsprechende Algorithmen eher für längere kritische Abschnitte, beispielsweise die fairen Versionen von ReentrantLock und Semaphore in Java. Der Vorteil dieser Algorithmen liegt darin, dass keine Threadlaufzeit für das aktive Warten verbraucht wird.

3.4.2 Synchronisierung der Adressübersetzung

Ein verteiltes RAM-basiertes Speichersystem erlaubt sowohl den lokalen Zugriff auf die Adressübersetzung und Speicherverwaltung, als auch Zugriffe über das Netzwerk, so dass gegebenenfalls viele Threads gleichzeitig auf die gespeicherten Daten zugreifen. Die parallelen Zugriffe der Threads müssen daher für eine korrekte Ausführung synchronisiert werden. In einem ersten Schritt werden die unterschiedlichen Zugriffsarten analysiert und auf eine notwendige Synchronisierung hin untersucht. Anschließend werden geeignete Maßnahmen erläutert.

Konkurrierende Zugriffe

Die meisten Zugriffe in der Adressübersetzung greifen nur lesend zu, um für eine gegebene CID die zugehörige Speicheradresse zu ermitteln. Dabei wird auf jeder Stufe auf eine einzelne CID-Tabelle lesend zugegriffen. Parallele Ermittlungen von Speicheradressen, durch mehrere Threads, sind daher unproblematisch. Neben den Lesezugriffen existieren aber vier Fälle, in denen schreibend zugegriffen wird und eine oder mehrere CID-Tabellen verändert werden.

1. Einen neuen Eintrag anlegen: Beim Anlegen eines neuen Antrags wird die Zuordnung einer CID zu einer virtuellen Speicheradresse in die CID-Tabellen eingetragen. Im besten Fall muss dabei nur auf die CID-Tabelle auf Stufe 0 schreibend und auf allen anderen Stufen auf jeweils eine CID-Tabelle lesend zugegriffen werden. Im schlechtesten Fall müssen mehrere CID-Tabellen neu erzeugt und auf jeweils eine CID-Tabelle je Stufe schreibend zugegriffen werden.

2. Einen Eintrag löschen: Wenn ein vorhandener Eintrag gelöscht und die entsprechende CID freigegeben werden soll, muss in jedem Fall in der passenden CID-Tabelle auf Stufe 0 der Eintrag geändert und der Lösch-Marker gesetzt werden. Gegebenenfalls muss in allen übergeordneten CID-Tabellen zusätzlich der Voll-Marker gelöscht werden (falls gesetzt). Im besten Fall wird nur auf eine CID-Tabelle, im schlechtesten Fall wird auf eine CID-Tabelle pro Stufe schreibend zugegriffen.

3. Freie CID suchen: Bei der Suche nach einer freien CID wird möglicherweise auf sehr viele Einträge in den CID-Tabellen auf allen Stufen lesend zugegriffen. Wenn eine freie CID gefunden wird, wird der Lösch-Marker des entsprechenden Eintrags entfernt und demzufolge auf eine CID-Tabelle schreibend zugegriffen. Erfolgt die Suche im Zusammenhang mit dem CID-Cache, wird die Suche so lange fortgesetzt, bis alle freien CIDs gefunden wurden oder der CID-Cache gefüllt ist. In diesem Fall wird gegebenenfalls auf mehrere CID-Tabellen der Stufe 0 zugegriffen.

4. Defragmentierung: Bei der Defragmentierung werden eine oder mehrere CID-Tabellen auf Stufe 0 durchlaufen und die adressierten Daten gegebenenfalls umkopiert. Vereinfacht kann angenommen werden, dass jeweils nur auf eine CID-Tabelle zugegriffen wird. Das Durchlaufen mehrerer Tabellen entspricht dem wiederholten Aufruf der Defragmentierung. Während der Defragmentierung wird auf jeden Eintrag der aktuellen CID-Tabelle lesend und auf Einträge, deren Daten umkopiert werden, schreibend zugegriffen. Abschließend wird auch noch geprüft, ob die CID-Tabelle selbst umkopiert werden muss. Dafür muss auf den passenden Eintrag, in der direkt darüber liegenden CID-Tabelle, lesend und schreibend zugegriffen werden.

In allen vier Fällen ist es notwendig die Zugriffe zu synchronisieren.

Granularität der Synchronisierung

Die Granularität ist für die Synchronisierung von erheblicher Bedeutung. Je feiner die Granularität ist, desto höher ist die mögliche Parallelität. Gleichzeitig erhöht sich aber auch die Anzahl der Synchronisierungsoperationen und damit der Aufwand für die Synchronisierung. Je grober die Granularität ist, desto geringer ist die mögliche Parallelität. Für die Synchronisierung der Adressübersetzung bestehen grundsätzlich vier sinnvolle Möglichkeiten für die Granularität.

A. Einzelner Eintrag: Die Synchronisierung auf Ebene eines einzelnen Tabelleneintrags erlaubt den höchsten Grad an Parallelität. Dabei muss allerdings bei jedem Zugriff auf einen Eintrag auch eine Synchronisierungsoperation ausgeführt werden. Dies bedeutet nicht nur einen großen zeitlichen Aufwand, sondern erfordert auch am meisten zusätzlichen Speicher, da jede Sperre, Semaphore oder Barriere in der Regel mindestens ein Byte belegt.

Tabelle 3.6: Synchronisierung in der Adressübersetzung

	Eintrag anlegen	Eintrag löschen	CID suchen	Defragmentierung
A: Einzelner Eintrag	++	++	-	-
B: Mehrere Einträge	+	+	+	+
C: Einzelne Tabelle	o	o	++	++
D: Zweig von Tabellen	- -	-	o	o

B. Mehrere Einträge: Bei dieser Möglichkeit werden mehrere Einträge zusammengefasst und als eine Einheit im Sinne der Synchronisierung betrachtet. Dieses Vorgehen schränkt die Parallelität etwas ein, vermindert aber den Mehraufwand für Laufzeit und Speicher. Das Verhältnis lässt sich gut durch die Anzahl der zusammenzufassenden Einträge steuern. Als obere Grenze für die Anzahl der Einträge dient die Größe einer CID-Tabelle, da es vergleichsweise aufwendig wäre Einträge benachbarter CID-Tabellen ausfindig zu machen.

C. Einzelne CID-Tabelle: Die Betrachtung einer einzelnen CID-Tabelle als Einheit der Granularität entspricht dem Spezialfall der vorherigen Möglichkeit, bei dem alle Einträge einer Tabelle zusammengefasst werden. Die Anzahl der Synchronisierungsoperationen wird dadurch stark reduziert, ermöglicht aber auch theoretisch weniger Parallelität. In der Praxis wird die Parallelität nicht sehr stark eingeschränkt, da zur Laufzeit mehrere zehn- bis hunderttausende CID-Tabellen existieren und es sehr wahrscheinlich ist, dass sich die meisten parallelen Zugriffe auf unterschiedliche CID-Tabellen verteilen. Das CID-Verzeichnis ist dabei allerdings ein Flaschenhals, da bei jedem Zugriff auf die Adressübersetzung zuerst auf das CID-Verzeichnis zugegriffen wird.

D. Zweig von CID-Tabellen: Die hierarchische Struktur der CID-Tabellen kann auch als Baumstruktur betrachtet werden. Die Wurzel ist das CID-Verzeichnis und die Blätter sind die CID-Tabellen auf Stufe 0. In diesem Fall bildet eine CID-Tabelle mit all ihren untergeordneten CID-Tabellen einen Zweig des Baumes. Geht man vom CID-Verzeichnis aus, umfasst der Zweig sogar den gesamten Baum. Offensichtlich werden bei dieser Möglichkeit die wenigsten Synchronisierungsoperationen benötigt, die Parallelität wird aber auch massiv eingeschränkt.

Zusammenfassend lässt sich feststellen, dass für die Fälle 1 und 2 die Möglichkeiten A und B und für die Fälle 3 und 4 die Möglichkeiten B und C am sinnvollsten sind (siehe Tabelle 3.6). Dabei ist Möglichkeit C für alle Fälle gut geeignet, hat einen geringen Speicherverbrauch und die Einschränkung der Parallelität fällt durch die schiere Anzahl an CID-Tabellen nicht ins Gewicht. Auch bei der eigentlichen Adressübersetzung (der häufigsten Operation) ist diese Möglichkeit gut, da auf jeder Ebene nur eine Sperre angefordert werden muss. Eine gute Alternative ist der kombinierte Einsatz der Möglichkeiten A bis C. Dabei wird das CID-Verzeichnis auf Basis einzelner Einträge und die CID-Tabellen auf Stufe 0 jeweils im

Gesamten synchronisiert. Die CID-Tabellen zwischen dem CID-Verzeichnis und der Stufe 0 werden auf Ebene mehrerer Einträge synchronisiert, wobei sich die Anzahl der zusammengefasster Einträge mit abnehmender Stufe erhöht.

Lese-Schreib-Sperre

Wie anfangs bereits erwähnt, ist die überwiegende Anzahl der Zugriffe auf die CID-Tabellen lesend. Schreibzugriffe sind nur selten und betreffen im Regelfall einzelne Tabellen. Aus diesem Grund bietet sich der Einsatz einer Lese-Schreib-Sperre an, die das parallele Lesen mehrerer Threads ermöglicht. Zur Synchronisierung wird für jede CID-Tabelle eine solche Sperre erzeugt.
Lese-Schreib-Sperren werden von den meisten Systemen zur Verfügung gestellt und können einfach verwendet werden. Der Einsatz einer solchen Sperre ist für die CID-Tabellen allerdings nicht geeignet, da die CID-Tabellen im VMB liegen und nur Offset-Zeiger in den VMB besitzen können. Ein Zeiger auf eine Sperre ist daher nicht praktikabel. Die Sperren können zwar in einer anderen Datenstruktur verwaltet werden, beispielsweise in einer Hashtabelle (mit der Speicheradresse der CID-Tabelle als Schlüssel und dem Zeiger auf die Sperre als Wert), allerdings erhöht sich dadurch der Speicheraufwand für die Synchronisierung. Einige Laufzeitumgebungen erlauben aber auch den Zugriff auf Speicheradressen mit atomaren Instruktionen. Java gehört nicht zu diesen Laufzeitumgebungen, ermöglicht aber mit Nutzung des Java Native Interface[5] auf entsprechende Instruktionen in der Sprache C zuzugreifen. Die Größe der CID-Tabellen wird um jeweils 4 Byte vergrößert und die zusätzlichen 4 Bytes für die Sperre verwendet.
In Algorithmus 3.6 und 3.7 wird gezeigt, wie sich eine entsprechende Lese-Schreib-Sperre mit Hilfe von CAS-Instruktionen auf einer 4-Byte Speicherstelle umsetzen lässt. Von den 4 Byte werden 31 Bit (Zähler) für die Lesesperre und 1 Bit für die Schreibsperre verwendet. Um die Sperren auseinander zu halten, werden eine Bitmaske und ein Bitmarker (Flag) eingesetzt. Die einzelnen Methoden der Implementierung werden nachfolgend erläutert.

readLock-Methode: In der readLock-Methode wird die Lesesperre angefordert, die sich an der übergebenen Adresse befindet. Zuerst wird in Zeile 11 geprüft, ob die Schreibsperre gesetzt ist. Solange sie gesetzt ist, kann die Lesesperre nicht angefordert werden. Wenn die Schreibsperre nicht gesetzt ist oder wieder freigegeben wurde, wird innerhalb einer Schleife (Zeile 14-17) der aktuelle Wert der Lesesperre ausgelesen. Der Wert der Lesesperre gibt an, wie oft die Lesesperre aktuell erteilt wurde. Der neue Wert der Lesesperre ist daher der aktuelle Wert plus eins. Am Ende wird eine atomare CAS-Operation mit dem alten und dem neuen Wert

[5] `http://docs.oracle.com/javase/7/docs/technotes/guides/jni/`

Algorithmus 3.6 Lese-Schreib-Sperre (Teil 1)

```
1: #define Reader_Bitmask 0x7FFFFFFF
2: #define Writer_Flag 0x80000000
3:
4: procedure READLOCK(long address)
5:     // use address as a pointer to an integer
6:     int *lock =(int*)address;
7:     int oldValue;
8:     int newValue;
9:
10:    // wait for write lock to release
11:    while ((*lock & Writer_Flag) != 0);
12:
13:    // busy waiting until read lock acquiring was successful
14:    repeat
15:        oldValue = *lock & Reader_Bitmask;
16:        newValue = oldValue + 1;
17:    until compare_and_swap(lock, oldValue, newValue) == true
18: end procedure
19:
20: procedure READUNLOCK(long address)
21:    // use address as a pointer to an integer
22:    int *lock =(int*)address;
23:    int oldValue;
24:    int newValue;
25:
26:    // busy waiting until read lock releasing was successful
27:    repeat
28:        oldValue = *lock;
29:        newValue = oldValue − 1;
30:    until compare_and_swap(lock, oldValue, newValue) == true
31: end procedure
```

durchgeführt. Schlägt die Operation fehl, wird die Schleife wiederholt. Andernfalls wurde die Lesesperre erteilt und die Methode wird beendet.

readUnlock-Methode: Die readUnlock-Methode läuft ähnlich ab wie die readLock-Methode. Allerdings ist eine Prüfung der Schreibsperre hier nicht notwendig, da der kritische Abschnitt verlassen wird. Die Schleife in Zeile 27-30 funktioniert genauso wie bei der readLock-Methode, der neue Wert der Lesesperre ist aber diesmal der aktuelle Wert minus eins.

writeLock-Methode: Die writeLock-Methode wird zum Anfordern der Schreibsperre verwendet. Auch in dieser Methode ist in Zeile 39-42 das selbe Schleifenkonstrukt, wie in den anderen

Algorithmus 3.7 Lese-Schreib-Sperre (Teil 2)

```
32: procedure WRITELOCK(long address)
33:     // use address as a pointer to an integer
34:     int *lock =(int*)address;
35:     int oldValue;
36:     int newValue;
37:
38:     // busy waiting until write lock acquiring was successful
39:     repeat
40:         oldValue = *lock & Reader_Bitmask;
41:         newValue = oldValue − 1 | Writer_Flag;
42:     until compare_and_swap(lock, oldValue, newValue) == true
43:
44:     // wait for read lock to release
45:     while ((*lock & Reader_Bitmask) != 0);
46: end procedure
47:
48: procedure WRITEUNLOCK(long address)
49:     // use address as a pointer to an integer
50:     int *lock =(int*)address;
51:
52:     // release write lock
53:     compare_and_swap(lock, Writer_Flag, 0);
54: end procedure
```

beiden Methoden zu finden. Der neue Wert der Sperre setzt sich diesmal aus dem aktuellen Wert und dem Bit-Marker verknüpft durch eine exklusives Oder zusammen. Im Anschluss an die erfolgreich durchgeführte CAS-Operation muss allerdings noch darauf gewartet werden, dass alle lesenden Threads den kritischen Abschnitt verlassen. Sobald die Lesesperre den Wert 0 hat (Zeile 45), befindet sich kein lesender Thread mehr im kritischen Abschnitt und der schreibende Thread kann eintreten.

writeUnlock-Methode: Die writeUnlock-Methode wird beim Verlassen des kritischen Abschnitts durch den schreibenden Thread aufgerufen. In dieser Methode wird lediglich der Wert der Sperre auf 0 gesetzt (Zeile 53). Es befindet sich anschließend kein einziger Thread mehr im kritischen Abschnitt.

CID-Cache

Gerade beim Anlegen vieler Chunks ist es unvermeidlich, dass mehrere Threads gleichzeitig auf den CID-Cache zugreifen. Dementsprechend müssen auch diese Zugriffe synchronisiert

werden.
Für die Funktionsweise und die Performanz der lokalen Metadaten-Verwaltung ist es unerheblich, ob eine freigegebene CID bei der nächsten, der übernächsten oder auch erst für die hundertste nachfolgende Chunkerzeugung verwendet wird. Aus diesem Grund ist eine verzögerte Aktualisierung (Lazy Update) des CID-Caches eine geeignete Möglichkeit.
Bei einem Zugriff auf den CID-Cache wird zuerst lesend auf den CID-Zähler zugegriffen. Dieser Zugriff erfolgt ohne Synchronisierung, weshalb nicht garantiert werden kann, dass der aktuellste Wert des Zähler gelesen wird. Wenn der Zähler den Wert 0 hat, wird der Zugriff auf den CID-Cache beendet und eine neue sequentielle CID erzeugt und verwendet. Wenn der CID-Zähler > 0 ist, wird eine exklusive Sperre angefordert und anschließend eine freie CID vom Cache verwendet. Dabei wird auch der CID-Zähler dekrementiert. Nach dem Anfordern der Sperre muss allerdings der CID-Zähler erneut geprüft werden, da möglicherweise ein veralteter Wert gelesen wurde. Ist der aktuelle Wert 0, wird die Sperre wieder freigegeben und verfahren, als wenn der CID-Zähler direkt 0 gewesen wäre.
Weil davon auszugehen ist, dass wesentlich mehr Chunks erzeugt als gelöscht werden, wird die meiste Zeit der CID-Cache leer und somit ein sperrfreier Zugriff möglich sein.
Beim Löschen eines Eintrags wird die freigegeben CID nach Möglichkeit in den CID-Cache eingefügt. Dafür wird ebenfalls die exklusive Sperre angefordert und nach dem Eintragen der CID und dem Inkrementieren des CID-Zählers wieder freigegeben.

3.4.3 Synchronisierung der Speicherverwaltung

Auch die Speicherverwaltung muss für die Verwendung mit mehreren Threads angepasst werden. Die Verwendung mehrerer Threads kann im besten Fall den Durchsatz steigern und insgesamt zu kürzeren Antwortzeiten führen.

Arenen

Ein Arenamanager wird eingesetzt um die Threads zu synchronisieren. Jeder Arena ist ein Thread und ein Segment zugewiesen, auf das nur der zugeordnete Thread schreibend zugreifen darf. Durch die Arenen werden die Threads und die Segmente voneinander abgeschottet. Beim ersten Zugriff eines Threads erzeugt der Arenamanager eine neue Arena und weist ihr den Thread und ein freies Segment zu. Sobald das Segment voll ist, wird der Arena ein neues Segment zugewiesen. Um ein Passendes zuzuweisen, müssen die existierenden Segmente überprüft werden. Damit die Zuweisung eindeutig ist und nicht zwei Arenen das gleiche Segment zugewiesen bekommen, verwendet der Arenamanager eine Sperre. Diese Sperre wird nur bei der Zuweisung eines neuen Segmentes benutzt. Durch die eineindeutige Zuordnung von

Thread und Segment, können sich Schreibzugriffe unterschiedlicher Threads nicht beeinflussen.
Lesezugriffe darf grundsätzlich jeder Thread auf jedem Segment durchführen. Dabei kann durchaus ein Konflikt zwischen einem Lese- und einem Schreibzugriff auf den gleichen Chunk auftreten. DXRAM garantiert aus Gründen der Skalierbarkeit keine Konsistenz, weswegen es sich dabei nicht um einen Fehler handelt. Um dieses Verhalten zu verhindern, kann die Anwendung einen Chunk explizit sperren, bevor ein lesender oder schreibender Zugriff erfolgt.

Segment-Stealing-Mechanismus

In bestimmten Fällen kann einer Arena kein neues Segment mehr zugewiesen werde. Zum Beispiel wenn mehr Threads auf die Speicherverwaltung zugreifen als Segmente existieren. Das Gleiche kann auch passieren, wenn fast alle Segmente voll sind und die nicht vollen Segmente schon einer Arena zugeordnet sind. Um dieses Problem zu lösen, kommt ein Segment-Stealing-Mechanismus zum Einsatz (siehe Abbildung 3.5).
Kann einer Arena kein freies Segment zugewiesen werden, wird geprüft, ob eine andere Arena ein passendes Segment hat. In diesem Fall wird der Arena das Segment entzogen (Stealing) und der Arena, des aktuell anfragenden Threads, zugewiesen. Um ein Segment nicht während eines laufenden Zugriffs zu entziehen, wird jedes Segment mit einer Sperre versehen. Wenn ein Thread auf seine Arena zugreift, wird geprüft, ob der Arena ein Segment zugewiesen ist und falls ja, ob die Sperre angefordert werden kann. Schlägt eine der beiden Prüfungen fehl, fordert der Thread den Arenamanager auf, ihm ein neues Segment zuzuweisen. Für die Zuweisung stehen nur Segmente zur Verfügung, die aktuell nicht gesperrt sind. Für das neu zugewiesene Segment ist die Sperre schon angefordert, so dass eine neue Überprüfung des Segmentstatus nicht nötig ist. Sobald der Schreibzugriff beendet wurde, wird die Sperre wieder freigegeben. Die für den Segment-Stealing-Mechanismus verwendeten Sperren besitzen, neben der lock- und unlock-Methode, eine weitere Methode. Diese Methode ermöglicht es die Sperre anzufordern, wenn sie genau zum Zeitpunkt der Anfrage frei ist (tryLock). Andernfalls wird der Thread nicht blockiert oder wartet auf die Freigabe der Sperre, sondern kehrt sofort zum Aufruf zurück und der Programmablauf wird fortgesetzt. Diese Verfahrensweise erlaubt einen sehr effizienten Umgang mit den Segment-Sperren.
Es wird erwartet, dass der Segment-Stealing-Mechanismus nur selten zum Einsatz kommt. In der Default-Konfiguration ist jedes Segment 1 GB groß, so dass zum Beispiel auf einem Knoten mit 32 GB Hauptspeicher und einem VMB von 30 GB 30 Segmente zur Verfügung stehen. Bei Objektgrößen zwischen 16 und 64 Byte können in jedem Segment mehrere Millionen Objekte abgelegt werden. Die inkrementelle Defragmentierung sorgt darüber hinaus dafür, dass der Speicher kompakt ist. Die Anzahl der Arenen kann beschränkt werden, indem die Anzahl

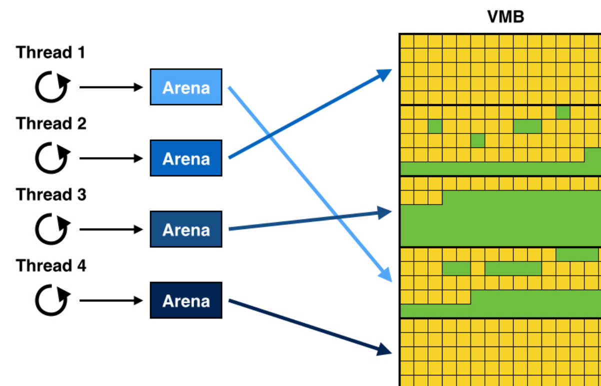
Algorithmus 3.8 Segmentzuweisung im Arenamanager

```
1:  procedure ASSIGNNEWSEGMENT(long threadID, Segment current, int minSize)
2:      Segment newSegment = null;
3:
4:      lock manager
5:      if (current != null) then
6:          remove current from arena
7:      end if
8:
9:      for all segment in segments do
10:         if (segment.getFreeSpace() > minSize && segment.tryLock()) then
11:             if (hasBetterStatus(segment, newSegment)) then
12:                 if (newSegment != null) then
13:                     newSegment.unlock();
14:                 end if
15:                 newSegment = segment;
16:             else
17:                 segment.unlock();
18:             end if
19:         end if
20:     end for
21:
22:     add newSegment to arena
23:     unlock manager
24:
25:     return newSegment
26: end procedure
```

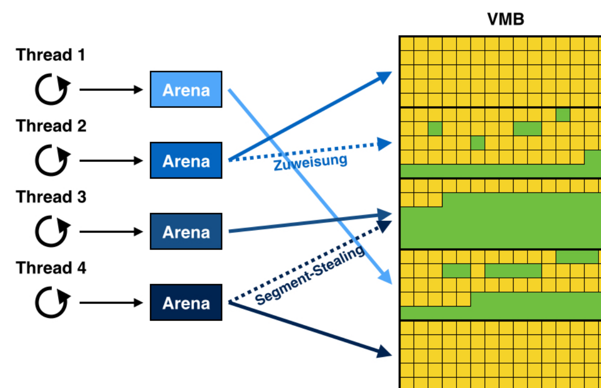
der Threads beschränkt wird. Zu diesem Zweck kann ein Threadpool eingesetzt werden. In der Netzwerkschnittstelle kommt ein entsprechender Threadpool bereits aus anderen Gründen zum Einsatz. All diese Maßnahmen sorgen dafür, dass der vorgestellte Mechanismus nur in Ausnahmefällen zum Tragen kommt.

Malloc

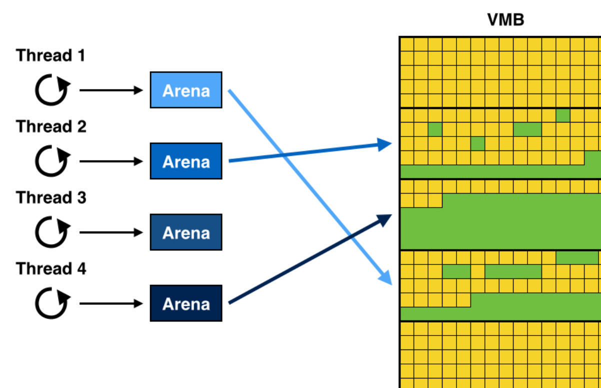
Jeder Thread führt Malloc-Operationen auf einem eigenen Segment durch. Ist das Segment voll, wird der Arena des Threads ein neues Segment zugewiesen. Algorithmus 3.8 zeigt vereinfacht, wie ein neues Segment für die Zuweisung ermittelt wird. Existiert kein passendes Segment, tritt der zuvor beschriebene Segment-Stealing-Mechanismus in Kraft. Der Zugriff auf das Segment und die Zuweisung eines Segmentes wird mit Sperren synchronisiert. Die Sperren werden durch den Arenamanager organisiert und für den Thread transparent beim Be-



(a) *Aktuelle Zuordnung von Segmenten.*



(b) *Der Arena von Thread 2 wird ein neues Segment zugewiesen. Die Arena von Thread 4 bezieht ein Segment über den Segment-Stealing-Mechanismus.*



(c) *Neue Zuordnung von Segmenten. Die Arena von Thread 3 hat temporär kein zugeordnetes Segment.*

Abbildung 3.5: Arenen. *Einer Arena ist genau ein Thread und ein Segment zugewiesen (a). Ist das Segment voll, wird ein freies Segment zugewiesen oder über den Segment-Stealing-Mechanismus bezogen (b). Anschließend kann es sein, dass temporär einer Arena kein Segment zugewiesen ist (c).*

Algorithmus 3.9 Zugriff auf die Arena

1: **procedure** ENTERARENA(**long** $threadID$, **int** $minSize$)
2: Segment ret;
3:
4: ret = *get current Segment for the arena*
5: **if** (ret == **null** || !ret.tryLock()) **then**
6: ret = assignNewSegment($threadID$, **null**, $minSize$);
7: **end if**
8:
9: **return** ret;
10: **end procedure**
11:
12: **procedure** LEAVEARENA(**long** $threadID$, Segment $segment$)
13: $segment$.unlock();
14: **end procedure**

treten der jeweiligen Arena gesetzt und beim Verlassen wieder freigegeben. Das Betreten und Verlassen der Arena wird in Algorithmus 3.9 dargestellt.

Free

Im Gegensatz zur Malloc-Operation kann eine Free-Operation nicht in einem beliebigen Segment durchgeführt werden, sondern muss in dem Segment ausgeführt werden, in dem die Speicheradresse liegt. Da das Segment zur Arena eines anderen Threads gehören kann, wird in diesem Fall von der Sperre des Segments Gebrauch gemacht. Das entspricht im Prinzip dem Segment-Stealing-Mechanismus, allerdings wechselt in diesem Fall das Segment nicht die Arena, sondern ist nur für kurze Zeit blockiert. Nach der Free-Operation kann der Thread, zu dessen Arena das Segment eigentlich gehört, weiterhin auf das Segment zugreifen.

3.4.4 Eingesetzte Threads und Sperren

Tabelle 3.7 zeigt die Threads und verwendeten Sperren für die einzelnen Komponenten in der lokalen Metadaten-Verwaltung. In der Speicherverwaltung müssen die Threads für lokale Zugriffe, die Threads für die Netzwerkzugriffe und der Thread für die Defragmentierung synchronisiert werden. Dabei wird auf Ebene von einzelnen Segmenten eine exklusive Sperre eingesetzt. Die Zugriffe auf diese Sperren wird allerdings nicht blockierend durchgeführt, so dass der Thread warten muss, bis die Sperre frei ist, sondern es wird nur einmal versucht die Sperre zu bekommen und bei Misserfolg die Sperre des nächsten passenden Segments angefordert (siehe 3.4.3). Die Koordinierung des Segment-Stealing-Mechanismus erfolgt im

Tabelle 3.7: *Threads und Sperren*

Komponente	**Threads** (Anzahl)	**Sperren**
Speicherverwaltung	Lokale Zugriffe (n) Netzwerkzugriffe (m) Defragmentierung (1)	Exklusive Sperre
Arenamanager	Lokale Zugriffe (n) Netzwerkzugriffe (m)	Exklusive Sperre
CID-Tabellen	Lokale Zugriffe (n) Netzwerkzugriffe (m) Defragmentierung (1)	Lese-Schreib-Sperre
CID-Cache	Lokale Zugriffe (n) Netzwerkzugriffe (m)	Exklusive Sperre

Arenamanager, der einem Thread ein Segment (über eine Arena) zuordnet. Nur bei einer neuen Zuordnung müssen lokale Zugriffe und Netzwerkzugriffe synchronisiert werden, wozu eine exklusive Sperre verwendet wird.

Wie schon in der Speicherverwaltung, müssen auch in den CID-Tabellen lokale Zugriffe, Netzwerkzugriffe und die Defragmentierung synchronisiert werden. In diesem Fall werden jedoch Lese-Schreib-Sperren verwendet, deren Anzahl von der Granularität abhängt (siehe 3.4.2). Aktuell ist nur der Standardfall implementiert, bei dem jede CID-Tabelle über eine eigene Lese-Schreib-Sperre verfügt.

Bei der Verwaltung der freien CIDs im CID-Cache werden lokale Zugriffe und Netzwerkzugriffe mithilfe einer exklusiven Sperre synchronisiert. Beim Hinzufügen einer freien CID wird die Sperre immer angefordert. Beim Anfordern einer freien CID wird jedoch ein Lazy-Update-Verfahren eingesetzt und die Sperre nur angefordert, wenn eine freie CID existiert (siehe 3.4.2).

In Abbildung 3.6 wird der Ablauf der Threads über die einzelnen Komponenten hinweg noch einmal veranschaulicht. Für die Implementierung der Sperren bieten sich zwei Alternativen an, je nach Anwendungszweck des Speicherdienstes.

Wird der Speicherdienst als performanter Backend-Speicher verwendet, bietet Java's ReentrantLock[6] (in der nicht fairen Version) den schnellsten Zugriff [129]. Da die Anzahl der Netzwerkthreads durch einen Threadpool limitiert ist und die überwiegende Zeit eines Netzwerkzugriffs in der Netzwerkschnittstelle verbracht wird, besteht nicht die Gefahr, dass durch die fehlende Fairness ein Thread verhungert. Statt des ReentrantLocks von Java können auch die CAS-Operation der Unsafe-Klasse für ein Lock verwendet werden (siehe Algorith-

[6] `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html`

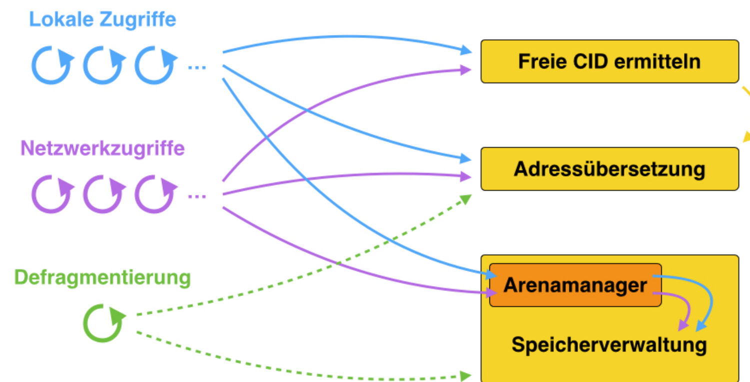


Abbildung 3.6: Threadverlauf. *In allen Komponenten müssen lokale Zugriffe und Netzwerkzugriffe synchronisiert werden. In der Speicherverwaltung und den CID-Tabellen muss darüber hinaus auch die Defragmentierung synchronisiert werden.*

mus 3.10).

Wird der Speicherdienst allerdings lokal angewendet und dabei mit vielen Threads gleichzeitig zugegriffen, ist der Einsatz einer fairen Sperrimplementierung angebracht. In [129] wird gezeigt, dass sperrfreie Algorithmen eine bessere Performanz besitzen, als die fairen Synchronisierungsmöglichkeiten in Java. Ein Nachteil beim Einsatz von sperrfreien Algorithmen ist allerdings, dass zumindest die maximale Anzahl der Threads bekannt sein muss. Entweder wird in diesem Fall von einer ausreichend hohen Anzahl Threads ausgegangen oder die Anzahl der Threads, mit Hilfe eines Thread-Pools, begrenzt. Wie in Abschnitt 3.4.1 beschrieben bietet Lamports Bakery Algorithmus die beste Performanz. Szymanskis Algorithmus benötigt allerdings wesentlich weniger Speicherplatz, so dass dieser Algorithmus bevorzugt wird.

Algorithmus 3.10 Exklusive CAS-Sperre

```
1:  procedure LOCK()
2:      repeat
3:      until (compareAndSwap(0, 1))
4:  end procedure
5:
6:  procedure UNLOCK()
7:      repeat
8:      until (compareAndSwap(1, 0))
9:  end procedure
10:
11: procedure TRYLOCK()
12:     return compareAndSwap(0, 1)
13: end procedure
```

3.5 Verwandte Arbeiten

Ebenso, wie Speichersysteme im Allgemeinen, ist auch Speicherverwaltung ein weit verbreitetes Themenfeld, in dem viele Lösungen und Publikationen existieren. Die Speicherverwaltung für sehr viele sehr kleine Objekte stellt jedoch eine besondere Herausforderung dar, die bisher kaum untersucht wurde.
Die meisten hier aufgeführten Systeme wurden schon grundlegend in Abschnitt 2.5 betrachtet. Für diese Systeme werden nachfolgend nur die Unterschiede zu den in diesem Kapitel vorgestellten Konzepten beschrieben.

RAMCloud
RAMCloud ist ein RAM-basiertes Speichersystem, dass Daten in Form von Schlüssel-Wert-Paaren in Tabellen verwaltet [16]. Die Schlüssel und Werte haben eine variable Länge und sind bis zu 64 KB beziehungsweise bis zu 1 MB groß [80]. Sowohl Schlüssel-Wert-Paare als auch Tabellen können über mehrere Knoten verteilt sein, wobei ein zentraler Koordinator die Zuordnung von Objekten und Tabellen zu Knoten verwaltet. Der RAM jedes Knotens ist in 64 KB große Seglets aufgeteilt, die bis zu 8 MB großen Segmenten zugeordnet werden. Die Segmente formen insgesamt ein Protokoll, bei dem immer nur das Kopf-Segment schreibbar ist (siehe auch LSF [142]). Damit Daten auch nach einem Knotenausfall zur Verfügung stehen, werden die Segmente auf Festplatte mehrerer Backupknoten repliziert, die die Segmente in einem gleich aufgebauten Protokollen speichern. Damit die Daten bei einem Knotenausfall wiederhergestellt werden können, müssen die Segmente selbstbeschreibend sein und beinhalten deswegen für jedes Objekt einen Entry Header (2-5 Byte) und einen Object Record (26 Byte) [81]. Gelöschte Objekte werden durch einen 32-Byte großen Tombstone repräsentiert. Neben den Metadaten pro Objekt, enthält auch jedes Segment einige Metadaten: Eine Segment-ID, eine Segment Digest (Liste aller zum Protokoll gehörender Segment-IDs), ein Zertifikat (Byte-Zähler und Prüfsumme), Segment Statistiken und eine Liste, der zum Segment zugeordneten Seglets. Eine Hashtabelle speichert die Zuordnung von globalen IDs zu Segment und Versatz der aktuellen Version des Objektes. Die Hashtabelle verwendet dazu ein Bucket-Hashing mit Verkettung, wobei die Anzahl der Buckets bei Systemstart festgelegt werden. Ein Bucket ist in 8-Byte große Einträge unterteilt und exakt so groß wie eine Cachezeile. Die Einträge umfassen einen 47-Bit großen Zeiger auf das Objekt, einen 16-Bit großen Teilhash und ein Bit-Flag, das angibt, ob der Zeiger auf einen weiteren Bucket verweist (statt auf ein Objekt). Da ein Bucket mehrere Einträge umfasst, muss bei der Suche überprüft werden, welcher Eintrag der Gesuchte ist. Der Teilhash ist zwar nicht eindeutig, ermöglicht aber einen ersten Hinweis, so dass nicht bei jedem Eintrag die Daten des Segmentes gelesen werden müssen, um die ID zu überprüfen. Wird ein (möglicherweise) passender Eintrag gefunden, wird das referenzierte Objekt

eingelesen und der Schlüssel überprüft. Stimmt der Schlüssel des Objektes mit dem gesuchten Schlüssel überein, wurde das passende Objekt gefunden. Stimmen die Schlüssel nicht überein, muss der Bucket weiter durchsucht werden.
Da immer nur das Kopf-Segment beschreibbar ist, sammeln sich über die Zeit mehrere Versionen der Objekte an und eine Defragmentierung des Protokolls wird notwendig. Dafür kommen zwei unterschiedliche Verfahren zum Einsatz, die eine inkrementelle Defragmentierung ermöglichen. Die Segment Compaction arbeitet nur auf dem Protokoll im RAM, kopiert die noch aktuellen Objekte eines Segmentes zusammen, löscht veraltete Objekte und verkleinert dadurch das Segment. Das Combined Cleaning wird auf dem Protokoll im RAM und auf den Protokollen auf den Backupknoten angewendet. Dabei werden die noch aktuellen Objekte mehrerer Segmente zu einem neuen Segment zusammengefügt, als neues Kopf-Segment an den Anfang des Protokolls kopiert und die alten Segmente komplett gelöscht. Tombstones stellen bei beiden Verfahren eine besondere Schwierigkeit dar, da sie erst gelöscht werden dürfen, wenn alle Versionen des zugehörigen Objektes bereits gelöscht wurden. Die Defragmentierung kann durch mehrere Threads parallel zum Gesamtsystem ausgeführt werden. Dabei wählen die Threads die zu defragmentierenden Segmente basierend auf einer Kosten-Nutzen-Formel aus. Um die Zugriffe mehrerer Threads zu synchronisieren, kommen feingranulare Sperren auf Ebene der Buckets in der Hashtabelle zum Einsatz.
Im Gegensatz zu RAMCloud ist DXRAM für sehr viele sehr kleine Objekte konzipiert, bei denen es auf eine sehr speichereffiziente Verwaltung ankommt. In DXRAM werden pro Objekt nur 7 Byte Metadaten benötigt, wohingegen in RAMCloud (alleine für den Entry Header und Object Record) mindestens 28 Byte benötigt werden. Die zusätzlichen Segment-Metadaten und die Tombstones erhöhen den Aufwand noch weiter. Der Einsatz einer Hashtabelle für die Zuordnung von globalen IDs zu Speicheradressen, wie in RAMCloud, ist für die sehr vielen Objekte auf Grund des hohen Speicherbedarfs nicht effizient genug. Aus diesem Grund werden hierarchische Tabellen eingesetzt, die durch die sequentielle ID-Erzeugung mit sehr wenig Speicher auskommen und trotzdem eine Zugriffszeit in $\mathcal{O}(1)$ besitzen.

Trinity Graph Engine

Trinity Graph Engine ist ein verteilter, RAM-basierter Schlüssel-Wert-Speicher für die Online-Anfragebearbeitung und Offline-Analyse von großen Graphen [15]. Die Graphen können dabei aus Milliarden von Knoten bestehen, wie zum Beispiel in sozialen Netzwerken oder anderen Webgraphen [63]. Die Daten des Graphen werden in einer Speicher-Cloud im RAM vieler verteilter Knoten gespeichert. Daten-Objekte in der Speicher-Cloud werden in Schlüssel-Wert-Paaren verwaltet, wobei der Schlüssel 64 Bit lang ist und die Werte von ein paar Bytes bis zu mehreren Kilobytes umfassen können. Der Speicher ist in 2^p Memory Trunks zu je 2 GB aufgeteilt und auf m Knoten verteilt, wobei in der Regel $2^p > m$ ist [63]. Jeder Memo-

ry Trunk wird im Trinity File System (TFS) repliziert (ähnlich zu HDFS) und steht dadurch auch bei einem Knotenausfall weiter zur Verfügung. Zum Auffinden eines Objektes wird der 64-Bit Schlüssel gehasht und damit der zugehörige Memory Trunk ermittelt. Jeder Memory Trunk verfügt über eine Hashtabelle, die Informationen zu den gespeicherten Objekten beinhaltet. Der Schlüssel wird erneut gehasht und so der Versatz und die Länge des Objektes aus der Hashtabelle ermittelt. Jeder Memory Trunk ist als Ring-Speicher organisiert und umfasst einen 2 GB großen virtuellen Adressraum, belegt aber physikalisch potentiell weniger Speicher. Beim häufigen Verändern von Objekten kann es zu Lücken im Ring-Speicher kommen, die durch einen Defragmentierungsprozess in regelmäßigen Abständen beseitigt werden. Dabei ist ebenfalls entscheidend, dass Objektgrößen veränderbar sind und Trinity zwischen zwei Objekten kleine Puffer anlegt, um häufiges Umkopieren zu vermeiden. Beispielsweise wird bei der Vergrößerung eines Objektes um 16 Byte von Trinity weitere 16 Bytes reserviert [63]. Speicherreservierungen sind zwar nur von kurzer Dauer, für diese Zeitspanne erhöht sich jedoch der Speicherverbrauch des Objektes.
In Trinity werden die 2 GB großen Memory Trunks jeweils als Ring-Speicher verwaltet. Aktualisierungen und neue Daten werden in der Regel an den Kopf des Ring-Speichers geschrieben. Eine regelmäßige Defragmentierung bereinigt die durch Aktualisierung entstehenden Lücken (veraltete Daten) und sorgt für die Kompaktifizierung des Speichers. Die Memory Trunks reservieren zwar 2 GB Adressraum, belegen diesen aber nicht zwangsweise komplett. In DXRAM hingegen wird der gesamte zugewiesene RAM durch die Speicherverwaltung genutzt und bei Systemstart in den physikalischen Arbeitsspeicher geladen, wodurch Seitenfehler und Zeiten für die Allokation weiteren Speichers in der Regel vermieden werden. Der Speicher ist in gleich große Segmente unterteilt, die eine feinere Fragmentierungsgranularität bieten und in Kombination mit den CID-Tabellen eine parallele, inkrementelle Defragmentierung ermöglichen. Die temporären Speicherreservierungen in Trinity sorgen für einen stetigen Speichermehrbedarf, da bei sehr vielen Objekten davon auszugehen ist, dass immer für einige Objekte eine Reservierung besteht. Da bisher jedoch zur Speicher-Cloud nur wenige Details veröffentlicht wurden ist eine genaue Betrachtung des Speicherbedarfs nicht möglich. Im Gegensatz zu den Hashtabellen in Trinity verwendet DXRAM hierarchische Tabellen für die Verwaltung der Datenzeiger.

FaRM

FaRM ist eine RAM-basierte verteilte Berechnungsplattform, die RDMA für den rechnerübergreifenden Datenzugriff nutzt [61]. Gespeicherte Schlüssel-Wert-Paare sind von 64 Byte bis mehreren Megabyte groß. Der verteilte Speicher ist in 2 GB große, gemeinsam genutzte Speicherregionen aufgeteilt, die die Basis für die Adressierung, die Datenwiederherstellung und die RDMA-Registrierung bilden. Jede globale ID besteht aus einer 32-Bit großen Regions-ID und einem 32-Bit-Offset. Speicher kann in Form von Slabs, Blöcken oder Regionen angefordert

werden, wobei ein Slab zwischen 64 Byte und 1 Megabyte und ein Block mehrere Megabyte groß ist. Jeder Slab ist einer von 256 eindeutigen Größenklassen zugeordnet und wird auf die entsprechende Größe aufgerundet. Die Objektdaten werden zusammen mit mehreren Metadaten abgespeichert. Die Metadaten umfassen eine 64-Bit Versions- und eine 64-Bit Inkarnationsnummer. Die Versionsnummer wird zusätzlich in gekürzter Form am Anfang jeder Cachezeile wiederholt. Die Inkarnationsnummer speichert, wie oft die entsprechende Speicherstelle schon freigegeben wurde. Versions- und Inkarnationsnummer werden für den Erhalt der Konsistenz bei sperrfreien Lesezugriffen über RDMA benötigt. Die Wiederverwendung von Speicherbereichen für verschiedene Objektgrößen ist schwierig und wird mit Hilfe von Epochen gelöst. Wird eine neue Epoche gestartet, werden alle gecachten Datenzeiger gelöscht und nicht mehr benötigte Speicherbereiche freigegeben, die damit für neue Allokationen wieder zur Verfügung stehen. Bei der Erzeugung eines Objektes kann FaRM mitgeteilt werden, dass das Objekt in der Nähe eines bereits bestehenden Objektes alloziert werden soll. Diese Datenlokalität kann nicht garantiert werden, FaRM versucht aber das neue Objekt im selben Block, in der selben Region oder aber in einer Region in der Nähe anzulegen (in dieser Reihenfolge).
In FaRM werden Allokationen in Form von Slabs verschiedener Größenklassen durchgeführt, wobei die kleinste Größenklasse 64 Byte umfasst. Demgegenüber alloziert DXRAM immer exakt den für die Daten benötigten Speicher, wodurch interne Fragmentierung vermieden wird, und ist für kleinere Objekte (16-64 Byte) ausgelegt. Die in FaRM eingesetzte Speicherverwaltung kann auf sich ändernde Allokationsmuster nur schwerfällig reagieren (Epochen) und benötigt für die sperrfreien Lesezugriffe besondere Maßnahmen zur Sicherstellung der Konsistenz (zum Beispiel Versions- und Inkarnationsnummern), wodurch auch der Speicherverbrauch pro Objekt steigt. Durch die exakte Speicherallokation und die inkrementelle Defragmentierung kann DXRAM sehr flexibel auf Änderungen der Allokationsmuster reagieren und benötigt nur wenig Speicher für notwendige Metadaten. Chunks werden auf dem Knoten gespeichert, auf dem sie erzeugt werden und besitzen dadurch einen Lokalitätsvorteil. Eine Multi-Create-Operation ermöglicht es dabei sequentiell hintereinander liegende Chunks zu erzeugen, die in der Regel auch im RAM hintereinander gespeichert werden.

TAO

TAO ist ein für Lesezugriffe optimierter Graph-Speicher, der bei Facebook die bisher genutzten Memcached-Instanzen ablöst [5]. TAO ist ein Cache oberhalb vielzähliger MySQL-Server. TAO cacht Objekte, Verknüpfungslisten und Verknüpfungszähler und wird nach Bedarf gefüllt. Dabei werden gegebenenfalls bereits gecachte Daten nach einer LRU-Strategie entfernt. Ein Slab-Allokator verwaltet gleich große Speicherbereiche, die Knoten oder Kanten-Listen beinhalten. In der Speicherverwaltung kommen eine thread-sichere Hashtabelle, die LRU-Verdrängungsstrategie und ein dynamischer Ausgleich der Threads zum Einsatz. Ein detaillier-

terer Einblick in die Speicherverwaltung ist leider nicht möglich, da TAO nicht frei zugänglich ist.
DXRAM ist nicht nur ein Cache, sondern hält alle Daten permanent im RAM. Ferner wird, statt einer Hashtabelle, ein Ansatz mit hierarchischen Tabellen verwendet. Für Daten werden immer exakte Speicherbereiche alloziert und so eine interne Fragmentierung vermieden.

Speicher-Allokatoren
Ferner existieren viele unterschiedliche Speicher-Allokatoren, von denen allerdings keiner für sehr viele sehr kleine Objekte ausgelegt ist, da zumeist interne Fragmentierung für bessere Cache-Performanz in Kauf genommen wird. Als Beispiel können ptmalloc (glibc) [143], jemalloc [114], tcmalloc [113], hoard [112] und boehmgc [144] aufgeführt werden.
Besonders hervorzuheben ist der von Doug Lea in [145] vorgestellte Speicherallokator, der ebenfalls den Speicher in freie und belegte Blöcke (*Chunks*) aufteilt und die freien Blöcke in doppelt-verketteten Listen (*Bins*) organisiert. Die Bins für Blöcke kleiner als 512 Byte enthalten nur Blöcke exakt der gleichen Größe, wofür eine 8-Byte Ausrichtung verwendet wird. Der in diesem Kapitel vorgestellte Allokator hingegen verzichtet auf die Ausrichtung, um besonders bei sehr kleinen Objekten den Speicheroverhead zu minimieren. Auch der Ansatz mit den Marker-Bytes als gemeinsamer Header und Trailer unterscheidet sich von Leas Ansatz. Bei der Suche nach einem passenden Speicherbereich, wird in DXRAM eine *First-Fit*-Strategie eingesetzt, wohingegen Lea eine *Best-Fit*-Ansatz verfolgt. Die First-Fit-Strategie verhindert, dass eine Freispeicherliste komplett durchsucht werden muss, um den besten Speicherbereich zu finden, sondern verwendet den ersten passenden Bereich. Durch diese Strategie kann sich zwar der Verschnitt vergrößern, allerdings ist bei den sehr kleinen Objekten der Verschnitt fast immer noch nutzbar. In allen anderen Fällen sorgt die Defragmentierung für eine entsprechende Reorganisation, so dass der Verschnitt bei Allokationen wieder nutzbar wird.

3.6 Zusammenfassung

Große interaktive Applikationen und Graphanwendungen verwalten Milliarden kleiner Datenobjekte. Der einzige Weg den schnellen Zugriff auf diese Menge an Objekten zu gewährleisten, ist es alle Daten permanent im Hauptspeicher zu halten. Die pure Menge an Objekten und die begrenzte Kapazität des RAMs erfordert nicht nur den Zusammenschluss vieler Knoten zu einem verteilten Speicher, sondern viel mehr noch eine besondere Effizienz bei der Speicherung der Objekte. Jedes Byte zusätzlich bedeutet bei so vielen Objekten insgesamt mehrere Gigabyte an unnutzbaren Speicher. Abgesehen vom Speicherverbrauch muss auch der Zugriff auf

die Anwendungsdaten sehr performant sein, um dem Durchsatz der Anwendungen gerecht zu werden.

In diesem Kapitel wurden verschiedene Ansätze für die Adressübersetzung von globalen IDs in virtuelle Speicheradressen näher betrachtet und verglichen. Bestehende Ansätze sind für die große Menge an sehr kleinen Objekten ungeeignet, weswegen ein neuer Ansatz, auf Basis von hierarchischen Tabellen (CID-Tabellen), präsentiert wurde. Der Einsatz der hierarchischen Tabellen ist an das Paging-Verfahren klassischer PC-Betriebssysteme angelehnt und erlaubt eine Übersetzung in $\mathcal{O}(1)$. Die dynamische Erzeugung und Freigabe von Tabellen, sowie die Reduzierung von Zeigern auf 5-Byte, reduziert den Speicheraufwand der Tabellen auf ein Minimum. Ein besonderer Mehrwert der CID-Tabellen, neben der Adressübersetzung, ist die gleichzeitige Verwaltung von freigegebenen globalen IDs, wodurch der globale Adressraum möglichst klein gehalten wird. Eine hierarchische Suche wird verwendet, um entsprechende IDs zu ermitteln und wiederzuverwenden. Mehrere Optimierungen der hierarchischen Suche sorgen dafür, dass eine freigegebene ID im Regelfall in konstanter Zeit ermittelt werden kann und nur in Ausnahmefällen die eigentlichen CID-Tabellen durchsucht werden müssen.
Sowohl die CID-Tabellen als auch die Ermittlung freigegebener IDs ist für den parallelen Zugriff mit mehreren Threads optimiert. Für die CID-Tabellen werden Lese-Schreib-Sperren auf Basis von CAS-Instruktionen eingesetzt. Die Granularität der Sperren kann dynamisch angepasst werden und umfasst einzelne Einträge bis ganze Tabellen. Bei der Ermittlung von freigegebenen IDs wird eine verzögerte Aktualisierung verwendet, wodurch im Normalfall ein sperrfreier Zugriff möglich ist.

Für die Speicherung der Objekte wurde eine neuartige Speicherverwaltung inklusive Allokator entwickelt, der explizit für die große Anzahl sehr kleiner Objekte entworfen wurde. Im Gegensatz zu traditionellen Speicherallokatoren wird auf die Ausrichtung an Cachezeilen oder anderen Grenzen verzichtet, wodurch der Speichermehraufwand pro Objekt auf 2-4 Byte reduziert werden kann. Freie Speicherblöcke werden in mehreren doppelt-verketteten Freispeicherlisten verwaltet, die innerhalb der freien Blöcke selbst liegen. Eine First-Fit-Strategie sorgt dabei für einen konstanten Aufwand für die Allokation eines Speicherblocks. Bei der Freigabe von Speicher werden benachbarte freie Speicherblöcke verschmolzen, damit jederzeit ausreichend große Speicherbereiche für neue Allokationen zur Verfügung stehen. Darüber hinaus wird eine neuartige inkrementelle Defragmentierungsstrategie eingesetzt, die auf Informationen der Speicherverwaltung und der Adressübersetzung zugreift, um Objekte transparent von einem fragmentierten Speicherbereich in einen nicht-fragmentierten Bereich zu verschieben. Die inkrementelle Defragmentierung kann parallel zum laufenden Betrieb in regelmäßigen Intervallen, beim Überschreiten eines Grenzwertes oder bei Bedarf ausgeführt werden, wobei auch eine komplette Defragmentierung möglich ist.

Die gesamte Speicherverwaltung arbeitet auf einem großen zusammenhängenden Speicherbereich (VMB) und nutzt zur Optimierung des Speicheraufwandes intern Offset-Zeiger mit reduzierter Länge. Neben den Anwendungsdaten sind auch alle Verwaltungsstrukturen der Adressübersetzung und der Speicherverwaltung im VMB abgelegt. Dadurch kann beispielsweise der Zustand eines einzelnen Knotens auf Festplatte repliziert und später wiederhergestellt werden. Dieser Mechanismus wird eingesetzt, wenn das Gesamtsystem, zum Beispiel für Wartungszwecke, heruntergefahren und später wiederfortgesetzt werden soll.
Ebenso wie die Adressübersetzung ist auch die Speicherverwaltung und der Zugriff auf die gespeicherten Daten, für Multi-Core- beziehungsweise Multi-Threading-Systeme, optimiert. Ein innovatives Arenenkonzept sorgt dafür, dass jeder Thread exklusiven Schreibzugriff auf einen zugewiesenen Speicherbereich (Segment) besitzt. Ein Segment-Stealing-Mechanismus ermöglicht bei Bedarf den Zugriff auf andere Speicherbereiche, sollte der freie Speicherplatz zur Neige gehen oder eine Arena nicht weiter verwendet werden.

Neben DXRAM existieren mit RAMCloud, Trinity und FaRM drei weitere Systeme, die Anwendungsdaten permanent im Hauptspeicher halten.
Bei RAMCloud werden Schlüssel-Wert-Paare in Tabellen verwaltet und in einem selbstbeschreibenden Protokoll im Hauptspeicher abgelegt. Der hohe Speicherbedarf hierfür (mindestens 28 Byte pro Objekt) macht RAMCloud ungeeignet für die sehr vielen sehr kleinen Objekte, die mit DXRAM verwaltet werden können.
Trinity verwendet mehrere Ring-Speicher zur Verwaltung der Daten im RAM. Um häufiges Umkopieren bei Aktualisierungen zu vermeiden, wird bei Objektvergrößerungen temporär zusätzlicher Speicher reserviert. Dadurch wird zum einen der Speicherverbrauch erhöht und zum anderen entstehen zusätzliche Lücken, sobald die Reservierung erlischt. In DXRAM wird Speicher immer exakt alloziert, wodurch nicht so schnell Lücken entstehen und eine Defragmentierung seltener notwendig ist. Ferner erlaubt die sequentielle ID-Erzeugung kompakte hierarchische Tabellen für die Adressübersetzung, wohingegen Trinity mehrfache Hashtabellen verwendet.
Objekte in FaRM belegen immer mindestens 64 Byte Speicher, der in Form von Slabs organisiert ist. Da zum Übertragen von Objekten gegebenenfalls mehrere RDMA-Zugriffe notwendig sind, werden in die Objektdaten zusätzliche Metadaten eingefügt, und damit der Speicherverbrauch erhöht. FaRM eigenet sich darum nicht für die sehr vielen sehr kleinen Objekte in DXRAM.

Kapitel 4

Globale Metadaten-Verwaltung

Die große Menge an zu verwaltenden Daten ist nicht nur für die lokale Metadaten-Verwaltung eine Herausforderung, sondern auch auf globaler Ebene sind damit einige Probleme verbunden. Zu den Aufgaben der globalen Metadaten-Verwaltung gehört hauptsächlich die schnelle Beantwortung von Objektsuchen und die Integration der Backupknoten für die Koordinierung der Datenwiederherstellung. Für jede CID muss vorgehalten werden, welcher Knoten den zugehörigen Chunk speichert und welcher Backupknoten kontaktiert werden muss, wenn der Knoten mit dem zugehörigen Chunk ausfällt.
Nachfolgend wird ein dezentrales Super-Peer-Overlay vorgestellt, bei dem die Metadaten auf die Super-Peers verteilt wird. Ein neuartiger Ansatz ermöglicht es große Mengen an CIDs zu wenigen CID-Bereichen zusammenzufassen, womit der Speicherverbrauch um bis zu 99,99% reduziert wird. Darüber hinaus erlauben die CID-Bereiche und die sequentielle CID-Erzeugung eine Integration von lokaler und globaler Metadaten-Verwaltung zu einem Gesamtkonzept.
Die vorgestellten Konzepte wurden in [59, 68, 69] publiziert.

4.1 Architektur

Für die globale Metadaten-Verwaltung wird ein Super-Peer-Overlay eingesetzt, das eine einfache Lastverteilung auf mehrere Knoten ermöglicht und ein hohes Maß an Fehlertoleranz gegenüber Knotenausfällen (Peers und Super-Peers) bietet. Jeder Super-Peer verwaltet primär eine Gruppe von Knoten und ist für alle Metadaten dieser Gruppe zuständig. Wie in Abschnitt 2.1 bereits beschrieben wurde, werden CIDs sequentiell erzeugt und ein Namensdienst zur Verfügung gestellt, der es ermöglicht für einen Teil der Chunks benutzerdefinierte Schlüssel festzulegen. Die Einträge des Namensdienstes werden dabei über die einzelnen Super-Peers verteilt.
Äquivalent zu den Anwendungsdaten werden auch die Metadaten auf den Super-Peers permanent im RAM gehalten. Da der Hauptspeicher teuer und begrenzt ist, muss die Speicherung

sehr effizient erfolgen. Erschwert wird dies durch die enorme Menge an Objekten, für die Metadaten gespeichert werden müssen. Abgesehen von der Speichereffizienz muss der Zugriff auf die Metadaten sehr performant durchführbar sein, da im schlimmsten Fall bei jedem Datenzugriff zuerst der Knoten ermittelt werden muss, der die Daten speichert.
Die meisten verteilten RAM-basierten Speichersysteme verwenden Hashtabellen für die Zuordnung von globalen IDs zu Knoten, beispielsweise Trinity [63] and RAMCloud [16]. Diese Hashtabellen enthalten in der Regeln einen Eintrag pro Objekt und erlauben eine schnelle Übersetzung einer globalen ID in eine Knotenadresse (Knoten-ID, IP-Adresse, etc.). Offensichtlich ist dies keine gute Lösung für Milliarden kleiner Objekte, besonders da Hashtabellen einen großen Speicherverbrauch haben (siehe Abschnitt 3.2.1).
In den folgenden Abschnitten wird ausführlich erläutert, wie die Menge der Metadaten durch den Einsatz von CID-Bereichen reduziert werden kann. Darüber hinaus wird eine modifizierte B-Baum-Struktur vorgestellt, die einen schnellen Zugriff auf die Metadaten und gleichzeitig eine sehr effiziente Speicherung ermöglicht.

4.2 Super-Peer-Overlay

Ungefähr 5-10% der zur Verfügung stehenden Knoten sind dedizierte Super-Peers, die einen Ring formen, ähnlich der DHT Chord [76] (siehe Abbildung 4.1). Mit den Knoten-IDs (NID) wird die Positionen der Super-Peers auf dem Ring bestimmt. Für die Kommunikation untereinander wird allerdings keine Fingertable (wie in Chord) verwendet, sondern die Super-Peers kennen sich alle untereinander (durch eine Konfigurationsdatei in ZooKeeper) und können bei Bedarf eine Verbindung zueinander aufbauen. Die begrenzte Anzahl an Super-Peers und die schnelle Netzwerkinfrastruktur in Rechenzentren ermöglichen dieses Vorgehen (im Gegensatz zu DHTs im Internet), wodurch die Suche in $\mathcal{O}(1)$ erfolgen kann. Jeder Super-Peer ist für einen Bereich von NIDs zuständig, der sich zwischen seiner eigenen NID und der NID seines Vorgängers befindet. Alle regulären Knoten (Peers) werden dem Super-Peer zugeordnet in dessen Bereich die eigene NID liegt. Der entsprechende Super-Peer verwaltet die Metadaten seiner zugeordneten Peers, kümmert sich um die Überwachung seiner Peers und um die Koordinierung der Datenwiederherstellung für den Fall, dass ein zugeordneter Peer ausfällt. Im Normalfall kennt ein Super-Peer nur seine eigenen Peers, die Peers hingegen kennen alle Super-Peers, wie im nächsten Abschnitt beschrieben wird.

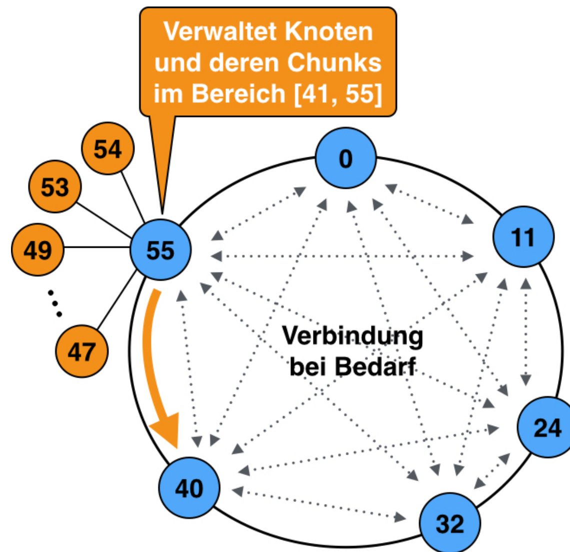


Abbildung 4.1: Super-Peer-Ring. *Das Beispiel zeigt sechs Super-Peers (0, 11, 24, 32, 40, 55) und vier Peers (47, 49, 53, 54), die dem Super-Peer zugeordnet sind. Die IDs der Knoten sind in diesem Beispiel stark vereinfacht.*

4.2.1 Objektsuche

Bevor auf einen Chunk zugegriffen werden kann, muss zuerst der Speicherort des Chunks anhand seiner CID ermittelt werden. Auf Grund von Chunkmigrationen oder Knotenausfällen kann sich der Speicherort im Laufe der Zeit ändern. Der NID-Teil der gegebenen CID wird verwendet, um den für den Chunk zuständigen Super-Peer zu identifizieren. Dafür wird überprüft, in welchem NID-Bereich die NID liegt. Anschließend wird der zugehörige Super-Peer kontaktiert und eine Anfrage nach dem Speicherort des Chunks gesendet.
Damit jeder Knoten die NID-Bereiche und die zugehörigen Super-Peers kennt, sendet jeder Super-Peer periodisch eine Liste mit allen aktuellen Super-Peers an seine Peers. Wenn das Super-Peer-Overlay sich ändert, beispielsweise wenn eine Super-Peer ausfällt oder beitritt, kann es passieren, dass der falsche Super-Peer kontaktiert wird. In diesem seltenen Fall teilt der kontaktierte Super-Peer dem anfragenden Knoten den korrekten Ansprechpartner mit.
Da Migrationen nur sehr selten zu erwarten sind und Knotenausfälle in einem Rechenzentrum nur vereinzelt auftreten, lässt sich der vorgestellte Ansatz so konfigurieren, dass grundsätzlich zuerst der Erzeuger des Chunks (erkennbar am NID-Teil der CID) kontaktiert wird. Im Normalfall ist der Erzeuger der Speicherort des Chunks und ändert sich nur bei einer Migration oder einem Ausfall des Erzeugers. Steht der Erzeuger nicht mehr zur Verfügung oder wurde der Chunk migriert, wird erst im zweiten Schritt der Super-Peer angesprochen. Es kann auch vorkommen, dass die IP-Adresse des Erzeugers nicht bekannt ist, wodurch ebenfalls der Super-Peer angesprochen werden muss.

4.2.2 Migrationen

Wie bereits erwähnt, werden Migrationen für die Lastverteilung genutzt, indem Hot-Spots von einem Knoten zu einem Anderen verschoben werden. Es werden nicht Millionen von Migrationen erwartet, es muss aber das potentielle Clustering von Hot-Spots auf einzelnen Knoten auflösbar sein. In einem Sozialen Netzwerk kann es beispielsweise auftreten, dass mehrere bekannte Stars (einige Stars haben bis zu 40 Millionen Likes oder Followers [63]) auf ein und demselben Knoten gespeichert sind. Global gesehen existieren aber nicht Millionen Stars.
Bei einer Migrationen sind immer drei Knoten involviert. Zuerst der Quellknoten, der einen Chunk verschieben möchte, dann der Zielknoten, der den Chunk aufnimmt, und schließlich der für den Chunk zuständige Super-Peer, der die Metadaten für den Chunk aktualisieren muss, damit auch weiterhin bei einer Objektsuche der korrekte Speicherort ermittelt werden kann. Der Datenaustausch zwischen Quell- und Zielknoten erfolgt dabei auf direktem Weg, der Super-Peer wird lediglich über die Migration informiert. Während der Migration kann der Super-Peer Objektsuchen bereits mit dem neuen Speicherort (Zielknoten) beantworten.
Bei einer Migration handelt es sich in den meisten Fällen nicht um einzelne, sondern um mehrere zusammenhängende Chunks. Beispielsweise besteht ein Profil in einem sozialen Netzwerk häufig aus mehreren Objekten (Ressourcen, Kommentare, Freundesliste, etc.), wobei Zugriffe überwiegend auf mehrere dieser Objekte erfolgen. Wenn ein Profil migriert wird, werden daher alle zu diesem Profil gehörenden Chunks mit migriert.

4.2.3 Ausfall eines Peers

Der Ausfall eines Peers wird durch ein periodisches Heart-Beat-Protokoll des zugehörigen Super-Peers oder durch eine Zeitüberschreitung bei einer Anfrage eines anderen Peers erkannt. Die Zeitspanne bevor eine Zeitüberschreitung eintritt ist abhängig vom Netzwerk und beträgt in der Regel nur weniger Millisekunden. Im zweiten Fall wird der Super-Peer vom anfragenden Peer über den Ausfall informiert. Die Ausfallerkennung und der weitere Ablauf wird in Abbildung 4.2 dargestellt.
Nachdem der Ausfall eines Knotens erkannt wurde, kontaktiert der Super-Peer alle Backupknoten die Replikate des ausgefallenen Knotens besitzen und fordert sie auf die Daten lokal wiederherzustellen. Eine explizite Ordnung der Backupknoten und -replikate (siehe Abschnitt 2.4) garantiert, dass zu jederzeit klar ist, welcher Backupknoten welche Replikate wiederherstellen muss, so dass die Wiederherstellung dezentral durch die kontaktierten Backupknoten durchgeführt werden kann. Zusätzlich erlaubt dies dem Super-Peer schon während der Laufzeit der Datenwiederherstellung die Metadaten zu aktualisieren und Objektsuchen (je nach Konfiguration) bis zur Fertigstellung der Wiederherstellung zu puffern oder direkt mit dem neuen Besitzer

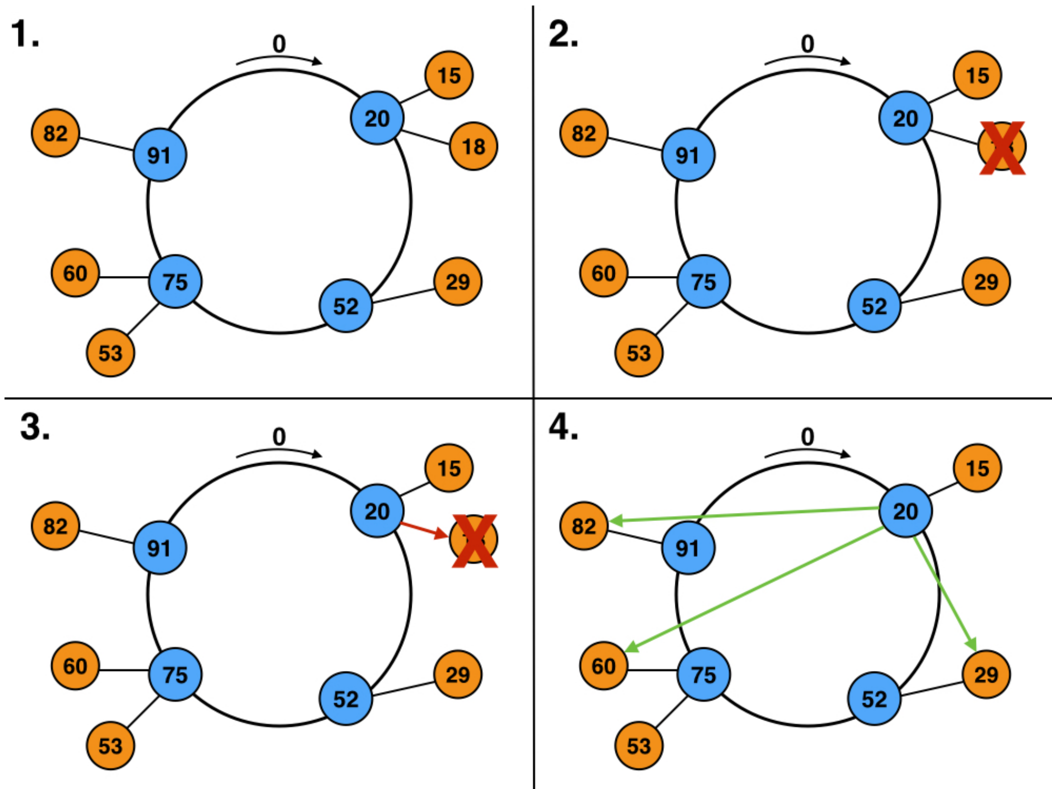


Abbildung 4.2: Ausfallerkennung. *1. Funktionierendes System. 2. Ausfall von Peer 110. 3. Ausfall wird erkannt. 4. Super-Peer kontaktiert die zuständigen Backupknoten.*

zu beantworten.

Auf dem Super-Peer entspricht die Aktualisierung der Metadaten prinzipiell der gleichzeitigen Migration einer großen Menge von Chunks. Um im weiteren Verlauf die Anzahl der Migrationen zu reduzieren und die Lokalität der Daten zu rekonstruieren werden zwei Strategien unterstützt.

Bei der ersten Strategie werden alle wiederhergestellten Chunks über eine bestimmte Zeit asynchron von den Backupknoten zu einem einzigen Knoten migriert. Offensichtlich wird dadurch die Lokalität der Daten rekonstruiert. Die Reduzierung der Migrationen erfolgt durch den Einsatz der CID-Bereiche, die in Abschnitt 4.3 vorgestellt werden. Die CID-Bereiche sorgen dafür, dass nur ein einziger Migrationseintrag existiert.

Die zweite Strategie geht noch einen Schritt weiter und weist einem neuen Knoten die NID des ausgefallenen Knotens zu. Anschließend wird wie in der ersten Strategie verfahren. Durch die Wiederverwendung der NID wird nicht nur die Lokalität vollständig rekonstruiert, sondern auch alle Migrationen beseitigt. Für die Wiederverwendung von NIDs wird ZooKeeper eingesetzt, um strikte Konsistenz bei der Zuweisung der NID zu gewährleisten.

4.2.4 Ausfall eines Super-Peers

Der Ausfall eines Super-Peers kann mehrere Konsequenzen haben. Die auf dem Super-Peer gespeicherten Metadaten sind verloren, der Super-Peer-Ring ist beschädigt und die zum Super-Peer gehörenden Peers sind vom System getrennt.
Der Verlust der Metadaten kann durch die Replikation der Metadaten auf eine vorbestimmte Anzahl an Nachfolgern im Overlay-Ring verhindert werden. Dadurch, dass die Super-Peers sich alle gegenseitig kennen, ist der zweite Punkt nicht kritisch. Darüber hinaus ist diese Situation umfangreich erforscht [76, 146, 147] und stellt selbst im Falle einer Netzwerkpartitionierung kein Problem dar [148]. Beispielsweise könnte ZooKeeper für die Erkennung einer Netzwerkparitionierung verwendet werden. Da ZooKeeper nur als zentrale Instanz existiert, befindet sich ZooKeeper in genau einer Partition. Die Knoten, die sich untereinander noch erreichen können, aber nicht mehr ZooKeeper, bilden die eine Partition und die Knoten, die ZooKeeper immer noch erreichen, bilden die andere Partition. Im einfachsten Fall arbeitet nur noch das Teilsystem mit ZooKeeper weiter. Alternativ arbeiten beide Teilsysteme unabhängig voneinander weiter und sobald die Partitionierung aufgehoben ist, werden die Daten beider Teilsysteme (gegebenenfalls mit der Mitwirkung eines Benutzers) zusammen geführt.
Für den Fall, dass die Peers vom System getrennt sind, wird ein Beförderungsalgorithmus eingesetzt, der aus der abgetrennten Gruppe von Peers denjenigen mit der höchsten NID auswählt und zum Super-Peer befördert. Wenn die Gruppe leer ist oder kein geeigneter Peer ermittelt werden kann, wird der Peer mit der niedrigsten NID aus der Gruppe des nachfolgenden Super-Peers ausgewählt und befördert. Der neue Super-Peer übernimmt anschließend die abgetrennte Gruppe.
Da Super-Peers keine Anwendungsdaten speichern, müssen für die Beförderung eines Peers alle Daten auf andere Knoten migriert werden, wobei die Knoten der eigenen Gruppe bevorzugt werden. Nachdem der Peer befördert wurde, bezieht er von einem seiner Nachfolger seine Metadaten. Durch die Auswahl des Knotens mit der höchsten NID der abgetrennten Gruppe beziehungsweise des Knotens mit der niedrigsten NID der nachfolgenden Gruppe müssen keine Peers ihre Gruppe wechseln und das Overlay kann problemlos weiter agieren. Während des Beförderungsprozesses kann der Nachfolger im Super-Peer-Ring Objektsuchen, Migrationen, etc. ausführen.
Anstatt den Beförderungsalgorithmus zu verwenden, kann auch ein neuer Knoten dem System zugewiesen werden. Der neue Knoten übernimmt die NID des ausgefallenen Super-Peers und kann sofort die Rolle im Super-Peer-Overlay einnehmen. Auch in diesem Fall müssen die Metadaten von einem der nachfolgenden Super-Peers bezogen werden.
Ein besondere und seltene Problemstellung ergibt sich, wenn mehrere hintereinander befindliche Super-Peers gleichzeitig ausfallen (sehr selten [149]), wodurch gegebenenfalls die Metada-

ten und alle ihre Replikate verloren und die Peers des ausgefallenen Super-Peers dem System unbekannt sind. Die Peers der abgetrennten Gruppe kennen jedoch noch weitere Super-Peers, da den Peers periodisch eine Liste aller Super-Peers zugeschickt wird. Nachdem die Peers dem System über einen noch aktiven Super-Peer wieder beigetreten sind, erfragt der jetzt zuständige Super-Peer nach, welche Chunks die Peers gespeichert haben und rekonstruiert damit die verlorenen Metadaten. Aus diesem Grund ist auch eine kleine Anzahl an Metadaten-Replikaten ausreichend. Die CIDs können dabei gebündelt in einem großen Netzwerk-Paket an den Super-Peer transferiert werden.

4.2.5 Totalausfall

Im seltenen Fall, dass alle Knoten gleichzeitig ausfallen (beispielsweise bei einem Stromausfall), ist es trotzdem möglich die gespeicherten Daten wiederherzustellen. Sobald die Knoten neu gestartet werden, stellt jeder Knoten die Daten seines lokales Protokolls wieder her. Das lokale Protokoll ist allerdings nicht in jedem Fall vollständig oder aktuell, da das Kopieren des VMB auf SSD relativ zeitaufwändig ist und deswegen das lokale Protokoll nur in konfigurierbaren Zeitabständen (beispielsweise von mehreren Minuten) erzeugt beziehungsweise aktualisiert wird. Aus diesem Grund müssen anschließend die gespeicherten Backupprotokolle eingelesen und überprüft werden. Dabei wird den zugehörigen Knoten mitgeteilt, welche Chunks lokal gespeichert sind. Sind die Chunks neuer als die Chunks im lokalen Protokoll des zugehörigen Knotens, werden die Chunks aus dem Backupprotokoll wiederhergestellt und an den zugehörigen Knoten übertragen. Sind alle lokalen Protokolle und Backupprotokolle verarbeitet, ermitteln die Super-Peers die Metadaten indem sie ihre Peers nach den gespeicherten Chunks befragen. Sobald die Metadaten rekonstruiert wurden, befindet sich das System im zuletzt gültigen Zustand und ist wieder einsatzbereit.
Der beschriebene Ansatz ist sehr zeitaufwendig und erfordert einen hohen Leseaufwand von SSD und Netzwerkverkehr zur Überprüfung und gegebenenfalls Übertragung der Daten. Da ein Stromausfall oder ein ähnliches Ereignis, dass zum Totalausfall des Systems führen kann, jedoch sehr selten ist, ist dieser Aufwand akzeptabel.

4.3 CID-Bereiche

In Abschnitt 3.2.1 und 3.2.2 wurde bereits ausführlich erörtert, dass der Speicherverbrauch von Hashtabellen und Indizes für die lokale Adressübersetzung zu hoch ist. Auf Ebene der Super-Peers müssen allerdings noch wesentlich mehr Einträge verwaltet werden, da ein Super-Peer für mehrere Peers zuständig ist und für jeden Chunk Metadaten verwaltet. Wenn zum Beispiel

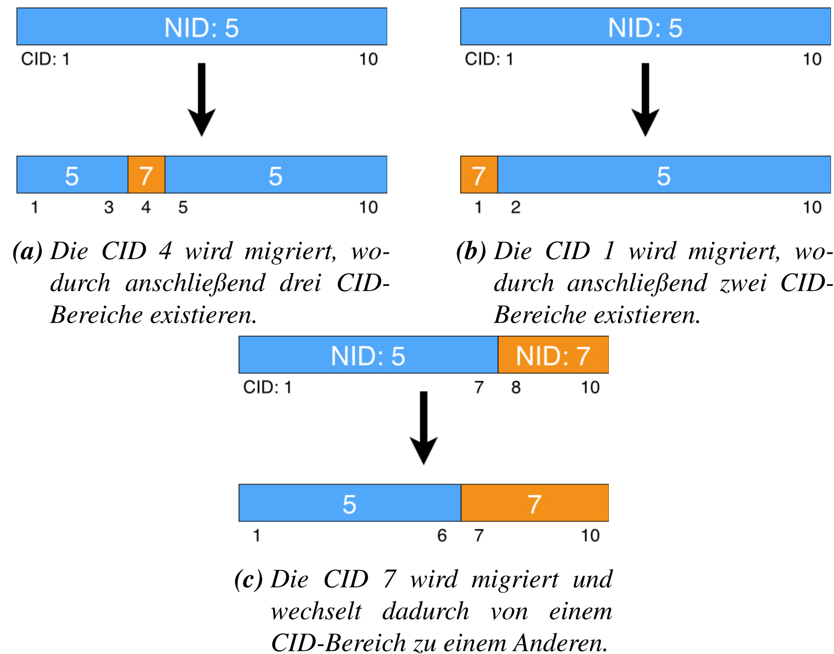
10 Peers, jeweils mit einer Milliarde Chunks, einem Super-Peer zugeordnet sind, müssen insgesamt 10 Milliarden Metadaten-Einträge verwaltet werden. Jeder Eintrag besteht dabei mindestens aus einer NID (2 Byte) und gegebenenfalls einer CID (8 Byte), wodurch der gesamte Speicherbedarf bereits 20 bis 100 GB beträgt. Der Speicherverbrauch der Verwaltungsstruktur ist dabei noch gar nicht einkalkuliert. Außerdem repliziert jeder Super-Peer die Metadaten mindestens eines Vorgängers (weitere 20-100 GB). Offensichtlich ist es unerlässlich die Anzahl der Metadaten-Einträge zu verringern, um diese Menge an Metadaten verwalten zu können. Dabei ist es erforderlich, dass der Zugriff trotz dessen schnell erfolgt und alle Metadaten jederzeit zur Verfügung stehen.
Die sequentielle CID-Erzeugung (siehe Abschnitt 2.1) kann für eine kompaktere Darstellung verwendet werden. Anstatt für jede CID einen einzelnen Metadaten-Eintrag zu speichern, werden mehrere CIDs zu einem CID-Bereich zusammen gefasst. Ein CID-Bereich besteht aus der Start- und End-CID (jeweils 8 Byte) sowie der NID des Knotens (2 Byte), der alle Chunks des CID-Bereichs speichert. Im besten Fall reicht ein CID-Bereich für alle CIDs eines Knotens. Für das vorherige Beispiel bedeutet dies, dass der Super-Peer im Idealfall nur 10 CID-Bereiche speichert. Die CID-Bereiche benötigen dabei lediglich 180 Byte Speicher, was einer Reduzierung des Speicherverbrauchs von mindestens 99,9999991% entspricht.
Dieser beste Fall ist allerdings nicht immer möglich, so können beispielsweise Migrationen zur Aufspaltung eines CID-Bereich führen. Abbildung 4.3 zeigt die drei Szenarios, die bei einer Migration betrachtet werden müssen.

Fall 1: Wenn die CID sich in der Mitte eines CID-Bereichs befindet, wird der ursprüngliche CID-Bereich in drei Bereiche aufgeteilt (siehe Abbildung 4.3 a). Der erste CID-Bereich enthält alle CIDs die kleiner als die migrierte CID sind. Der zweite CID-Bereich hat die Länge eins und beinhaltet lediglich die migrierte CID. Ein CID-Bereich der Länge eins bedeutet dabei, dass Start- und End-CID gleich sind. Der dritte CID-Bereich enthält schließlich alle CIDs die größer als die migrierte CID sind.

Fall 2: Wenn die CID sich am Beginn oder am Ende eines CID-Bereichs befindet, wird der ursprüngliche CID-Bereich um die entsprechende CID gekürzt (siehe Abbildung 4.3 b). Dafür wird die Start-CID des Bereichs inkrementiert beziehungsweise die End-CID dekrementiert. Anschließend wird für die migrierte CID ein neuer CID-Bereich der Länge eins erzeugt.

Fall 3: Beim dritten Fall existieren zwei CID-Bereiche und die zu migrierende CID befindet sich entweder am Ende des ersten Bereichs oder am Beginn des zweiten Bereichs und durch die Migration wechselt die CID von dem einen Bereich in den Anderen (siehe Abbildung 4.3 c). Nach der Migration existieren weiterhin nur die beiden CID-Bereiche, wobei der eine Bereich um eins verkürzt und der andere Bereich um eins verlängert wurde.

Wie zuvor bereits erwähnt, werden Migrationen nur selten erwartet, da nur eine begrenzte



(a) *Die CID 4 wird migriert, wodurch anschließend drei CID-Bereiche existieren.*

(b) *Die CID 1 wird migriert, wodurch anschließend zwei CID-Bereiche existieren.*

(c) *Die CID 7 wird migriert und wechselt dadurch von einem CID-Bereich zu einem Anderen.*

Abbildung 4.3: CID-Bereiche bei Chunkmigrationen. *Bei der Migration eines Chunks muss zwischen drei Fällen unterschieden werden, die durch die Position der zugehörigen CID innerhalb des CID-Bereichs bestimmt sind.*

Anzahl an Hot-Spots zu erwarten sind. Daher sind Aufspaltungen von CID-Bereichen eher selten und dementsprechend kein Problem. Selbst wenn pro Knoten 10 Millionen Chunks (1% der gespeicherten Chunks) migriert werden, würden insgesamt maximal 200 Millionen CID-Bereiche auf dem Super-Peer existieren. Jede Migration führt im schlechtesten Fall dazu, dass zwei zusätzliche CID-Bereiche entstehen. Bei 10 Millionen Migrationen existieren demnach $1 + 10^6 * 2 = 20.000.001$ CID-Bereiche pro Knoten und insgesamt 200 Millionen CID-Bereiche. Die Bereiche würden ungefähr 3,3 GB Speicher belegen, was immer noch einer Reduzierung des Speicherverbrauch von mindestens 83-96% gegenüber einer Liste oder Tabelle entspricht.

4.3.1 Datenstrukturen zur Verwaltung von CID-Bereichen

Hashtabellen und traditionelle Indexstrukturen (zum Beispiel auf ISAM basierende [150]) speichern einen Eintrag pro Schlüssel, wohingegen CID-Bereiche aus zwei Schlüsseln bestehen (Start-CID und End-CID). Solche Schlüssel-Schlüssel-Wert-Paare (*SSW-Paare*) sind bisher nur spärlich untersucht worden und die meisten Datenstrukturen sind dafür nicht optimiert. Nachfolgend werden zwei Datenstrukturen vorgestellt, die für die Verwaltung solcher SSW-Paare modifiziert wurden und damit eine effiziente Verwaltung der CID-Bereiche ermöglichen. Dabei

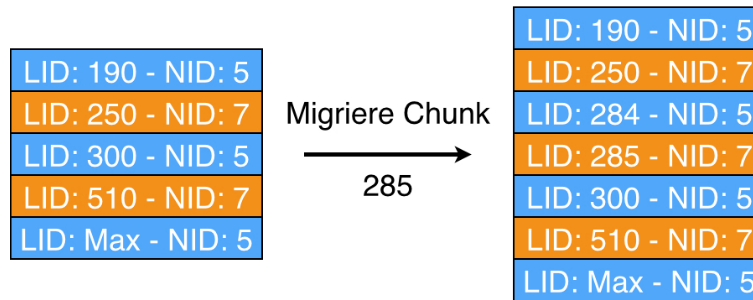


Abbildung 4.4: Beispiel einer Chunkmigration in einer CID-Liste. *Die Migration des Chunks mit der CID 285 resultiert in einem einzelnen Eintrag und einem neuen Bereichseintrag.*

sind vor allem die Operationen zum Einfügen, Löschen und Modifizieren von CID-Bereichen und die Suche nach Schlüsseln innerhalb der CID-Bereiche von Interesse.
Für jeden zu verwaltenden Peer wird eine Instanz der nachfolgend vorgestellten Datenstrukturen erzeugt. Somit ist es ausreichend für die CID-Bereiche LIDs und nicht die kompletten CIDs zu verwenden, wodurch der Speicherverbrauch eines CID-Bereichs noch weiter reduziert wird.

CID-Liste

Die CID-Liste basiert auf einem Array mit dynamischer Größe[1]. Die Einträge des Arrays sind Referenzen, die auf Objekte verweisen, die aus einer LID (6 Byte) und einer NID (2 Byte) bestehen. Die LID des Eintrags entspricht immer dem Ende des CID-Bereichs.
Nach der Initialisierung der CID-Liste existiert nur ein einzelner Eintrag, der aus der Maximalen LID ($2^{48} - 1$) und der NID des zugehörigen Peers (Erzeuger) besteht. Solange keine Migration durchgeführt wird, existiert nur der eine Eintrag und die Objektsuche erfolgt in $\mathcal{O}(1)$.
Wird ein Chunk migriert, muss mindestens ein neuer Eintrag in die CID-Liste eingefügt werden. Dazu wird die Position der LID, der zu migrierenden CID, in der Liste durch eine binäre Suche bestimmt. Hat der Eintrag an dieser Position die selbe LID wie die Einzufügende, wird der Eintrag ersetzt. Ansonsten wird an der ermittelten Position ein neuer Eintrag eingefügt und alle nachfolgenden Einträge um eine Position nach hinten verschoben. Befindet sich die zu migrierende CID in der Mitte eines CID-Bereichs muss ein weiterer Eintrag eingefügt werden. Dieser Eintrag umfasst alle CIDs die kleiner als die zu migrierende CID sind, und hat als LID die zu migrierende LID – 1. Ein Beispiel einer Migration wird in Abbildung 4.4 gezeigt. Wenn ein Chunk zurück zu seinem Erzeuger migriert wird, wird der zugehörige Eintrag gelöscht und

[1] Zum Beispiel eine ArrayList in Java.

gegebenenfalls die beiden jetzt benachbarten CID-Bereiche verschmolzen. Für das Verschmelzen wird überprüft, ob die beiden benachbarten Einträge in der CID-Liste die gleiche NID haben. In diesem Fall kann der erste Eintrag gelöscht und alle nachfolgenden Einträge um eine Position nach vorne verschoben werden.
Die Suche nach einer CID während der Objektsuche erfolgt mit der gleichen binären Suche, die auch bei einer Migration zum Einsatz kommt. Die ermittelte Position bestimmt den zugehörigen Eintrag, der wiederum die NID für die CID speichert. Die LID des Eintrags muss nicht überprüft werden, da sie nur das Ende des CID-Bereichs bestimmt und daher nicht mit der zu suchenden LID übereinstimmen muss.
Die CID-Liste ist eine sehr simple Datenstruktur, die eine Objektsuche in $\mathcal{O}(\log n)$ ermöglicht. Die Lösch- und Einfügeoperation sind sehr teuer, da gegebenenfalls viele Einträge verschoben werden müssen und mit zunehmender Anzahl an CID-Bereichen wird das Verschieben immer aufwändiger. Dabei kann es notwendig werden, dass das dynamische Array verkleinert oder vergrößert werden muss. Eine Größenänderung des Arrays erfolgt immer dann, wenn die Anzahl der Einträge einen definierten Schwellwert unter- beziehungsweise überschreitet. Dabei wird ein neues Array erzeugt, dass über entsprechend weniger oder mehr Platz verfügt. Anschließend werden alle Einträge des alten Arrays in das neue Array kopiert. Während der fehlerfreien Ausführung sind Lösch- und Einfügeoperation eher selten, während einer Datenwiederherstellung allerdings sehr häufig. Besonders im zweiten Fall sind CID-Listen zu langsam. Wie später bei den Messungen in Abschnitt 5.3 gezeigt wird, sind die Geschwindigkeitseinbußen so groß, dass ein produktive Einsatz nicht in Frage kommt.
Die CID-Liste ist theoretisch sehr speichereffizient, da jeder CID-Bereich nur 8 Byte belegt. In einem dynamischen Array sind in der Regel aber nicht alle Einträge gefüllt, wodurch gegebenenfalls viel Speicher verschenkt wird.

CID-Baum

Für den CID-Baum wurde ein B-Baum so angepasst, dass effizient CID-Bereiche verwaltet werden können. Die Start- und End-LIDs des Bereiches werden in den inneren Knoten und die NIDs in den Blättern gespeichert (siehe Abbildung 4.5). Ein B-Baum ist grundsätzlich schon in Bereiche unterteilt, die für die CID-Bereiche verwendet werden können. Beispielsweise besteht der B-Baum in Abbildung 4.5 aus insgesamt acht Bereichen. Dabei begrenzen die Einträge der inneren Knoten die Bereiche. Der erste Bereich ist zwischen 1 und 7, der zweite Bereich zwischen 7 und 31 und der dritte Bereich zwischen 31 und 56. Bereich vier befindet sich zwischen 56 und 100 und der fünfte Bereich liegt zwischen 100 und 120. Die Bereiche sechs und sieben befinden sich zwischen 120 und 183 beziehungsweise 183 und 202. Der letzte Bereich liegt schließlich zwischen 202 und der maximalen LID. Für jeden Bereich enthält der Baum einen

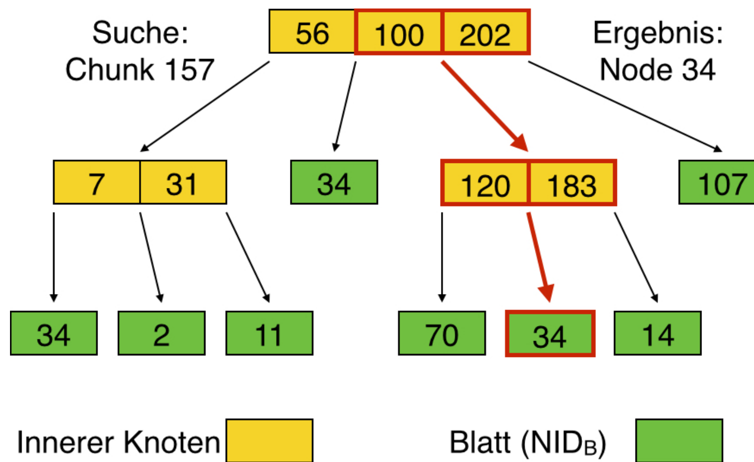


Abbildung 4.5: Suche in einem CID-Baum. *Im gegebenen CID-Baum wird nach der CID 157 gesucht und dabei die NID 34 ermittelt.*

Zeiger auf ein Blatt, welches im Falle eines CID-Baum die NID des CID-Bereichs enthält. Das Einfügen und Löschen verläuft äquivalent zur CID-Liste, es werden allerdings die normalen B-Baum-Operationen verwendet. Im Gegensatz dazu kann für die Suche nicht die reguläre Operation des B-Baumes verwendet werden, sondern eine modifizierte Suche ist notwendig (wie in Algorithmus 4.1 dargestellt). Bei der modifizierten Suche wird in der Wurzel und in den inneren Knoten eine binäre Suche durchgeführt und damit die Referenz auf den nächsten inneren Knoten beziehungsweise ein Blatt ermittelt. Ein B-Baum speichert auch in einem Blatt mehrere Einträge, wohingegen der CID-Baum nur eine einzelne NID in einem Blatt speichert. Sobald bei der Suche ein Blatt erreicht wird, wird die NID des Blattes zurück gegeben. Die Komplexität aller Operationen des CID-Baumes liegt sowohl im besten als auch im schlechtesten Fall in $\mathcal{O}(\log n)$ [151, 152] genau wie bei der CID-Liste, allerdings sind die Lösch- und Einfügeoperation bei vielen CID-Bereichen wesentlich effizienter, da kein Speicher umkopiert werden muss.

Der Speicherbedarf eines einzelnen Eintrags lässt sich beim CID-Baum nicht einfach kalkulieren. Da die LIDs der Einträge in den inneren Knoten und die NIDs in den Blättern des Baumes abgelegt sind und die inneren Knoten mehrere CID-Bereiche beinhaltet, kann die Größe eines Eintrags nur approximiert werden. Die nachfolgende Formel stellt diese Approximation dar:

$$\text{Größe eines Eintrags} = \frac{2 * \text{Ordnung} * (\Delta\text{Schlüssel} + (\Delta\text{Blatt} \oplus \Delta\text{Referenz}))}{\text{Anzahl der Einträge}}$$

$$\Delta\text{Schlüssel} = 6 \text{ Byte}$$

$$\Delta\text{Blatt} = 2 \text{ Byte}$$

$$\Delta\text{Referenz} = 4 \text{ Byte}$$

Der zugrunde liegende B-Baum garantiert dabei, dass jeder Knoten mindestens so viele Einträge besitzt, wie die Ordnung angibt, und maximal doppelt so viele. Der Speicherbedarf eines einzelnen Eintrags liegt somit bei ungefähr 8 bis 20 Byte. Ein Eintrag deckt aber einen ganzen Bereich ab, und nicht nur eine einzelne CID.

Algorithmus 4.1 Suche im CID-Baum.

```
1:  cid = die gesuchte CID;
2:  knoten = Wurzel des CID-Baumes;
3:  nid = −1;
4:
5:  // Durchlaufe Baum von der Wurzel bis zu einem Blatt
6:  while (nid == −1) do
7:      // Binäre Suche im Knoten
8:      pos = knoten.suche(cid);
9:      // Hole Sohn an ermittelter Position
10:     knoten = knoten.getSohn(pos);
11:     if (knoten.istBlatt()) then
12:         // Wenn ein Blatt erreicht wurde, ist die NID gefunden
13:         nid = knoten.getNID();
14:     end if
15: end while
16:
17: return nid;
```

4.3.2 Integration der Backupknoten

Wie bereits in Abschnitt 2.4 ausführlich erläutert, werden in der Regel für jeden Chunk drei Backupknoten verwendet, auf denen der Chunk auf SSD repliziert wird und die für die Wiederherstellung des Chunks im Fehlerfall zuständig sind. Die Datenwiederherstellung erfordert, dass der Super-Peer die Backupknoten seiner Peers kennt, um diese beim Ausfall des Peers zu benachrichtigen. Der im vorherigen Abschnitt vorgestellte CID-Baum ermöglicht eine einfache Integration der Backupknoteninformationen in die CID-Bereiche. Wie bereits beschrieben, sind die Backupknoten immer für eine Backupzone zuständig, die sich problemlos als CID-Bereich darstellen lässt.

Basisansatz

In den Blättern des CID-Baumes wird neben der NID des Besitzers (NID_B) zusätzlich die NIDs der drei Backupknoten (die Anzahl ist konfigurierbar) abgelegt. Dazu werden die Blätter von

einem Short-Wert (2 Byte) auf einen Long-Wert (8 Byte) vergrößert. In den niedrigsten zwei Byte (Bit 0-15) des Long-Wertes wird die NID_B des CID-Bereiches gespeichert. Die restlichen 6 Bytes nehmen die NIDs der drei Backupknoten auf, wobei sich der erste Backupknoten in den Bits 16-31, der zweite Backupknoten in den Bits 32-47 und der dritte Backupknoten in den Bits 48-63 befindet. Diese Konstruktion erlaubt eine schnelle und einfache Reparatur der Metadaten. Bei einem Knotenausfall wird der Long-Wert um 16 Bit nach rechts verschoben (shift), wodurch sich die NID des ersten Backupknotens anschließend in den Bits 0-15 befindet. Der erste Backupknoten entspricht damit dem Besitzer der entsprechenden Chunks, was exakt dem Vorgehen der Datenwiederherstellung entspricht. Der zweite Backupknoten befindet sich jetzt in den Bits 16-31 und stellt damit den neuen ersten Backupknoten. Äquivalent dazu wird der vormals dritte Backupknoten zum neuen zweiten Backupknoten. Ein neuer dritter Backupknoten wird erst nach der Datenwiederherstellung durch den Super-Peer bestimmt. Sobald er bestimmt wurde, bezieht dieser die Backupreplikate direkt vom neuen Besitzer oder von den anderen beiden Backupknoten.

Es existiert nur ein einziger CID-Bereich, wenn keine Chunks migriert wurden. Dabei handelt es sich um den besten Fall in Bezug auf den Speicherverbrauch, ist aber für die Backup-Lastverteilung ungeeignet, da in diesem Fall alle Chunks eines Knotens auf nur insgesamt drei Backupknoten gespeichert wären. Dies wiederum würde die Geschwindigkeit der Datenwiederherstellung erheblich beeinträchtigen, da ein einzelner Backupknoten im Fehlerfall alle Daten von seiner SSD wiederherstellen müsste. Aus diesem Grund wird die Limitierung, die für die Backupzonen gilt (beispielsweise 2,5 Millionen Chunks) auch auf die CID-Bereiche angewendet.
Statt eines einzigen CID-Bereichs würden in dem bisher betrachteten Beispiel mit 10 Milliarden Chunks 4.000 CID-Bereiche ($\frac{10.000.000.000}{2.500.000} = 4.000$) auf dem Super-Peer gespeichert werden, die insgesamt ungefähr 32-80 KB Speicher belegen. Die Größenlimitierung der CID-Bereiche bedeutet daher keine entscheidende Einschränkung der Speichereffizienz.

Optimierter Ansatz

Durch einzelne Migrationen kann es passieren, dass mehrere CID-Bereiche die gleichen Backupknoten besitzen, da sie zur gleichen Backupzone gehören. Beim Basisansatz können in diesem Zusammenhang unnötigen Redundanzen auftreten, die sich durch einen optimierten Ansatz vermeiden lassen. Die Backupknoten-Tripel werden dabei in einem separaten Long-Array gespeichert, anstatt direkt in den Blättern des CID-Baumes (siehe Abbildung 4.6). Die Blätter enthalten in diesem Fall wieder nur eine NID. Durch die feste Größe einer Backupzone kann der zu einer CID gehörige Array-Index berechnet werden. Dafür muss lediglich die LID durch die Größe einer Backupzone ganzzahlig dividiert werden. Die Shift-Operation, die bei einem

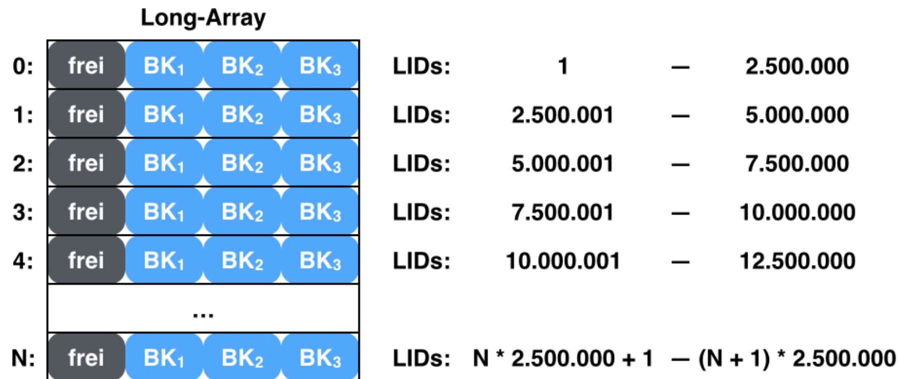


Abbildung 4.6: Optimierter Ansatz. *Die NIDs der Backupknoten werden in einem separatem Long-Array gespeichert. Eine Backupzone umfasst dabei 2,5 Millionen Chunks.*

Knotenausfall zur Reparatur der Metadaten verwendet wird, kann auch im optimierten Ansatz verwendet werden.
Diese Optimierung erlaubt grundsätzlich auch wieder den Fall, dass ein CID-Baum nur einen einzigen CID-Bereich mit allen CIDs eines Knotens enthält. Aus den genannten Gründen ist der optimierte Ansatz dem Basisansatz vorzuziehen.

4.3.3 Caching auf Clientseite

Um die Anfragen an die Super-Peers zu reduzieren und gleichzeitig den Durchsatz des Gesamtsystems zu erhöhen, werden die Ergebnisse der Objektsuchen auf den Peers gecacht. Bei einer Objektsuche wird zuerst die lokale Datenstruktur (Cache) überprüft und nur beim Fehlen eines passenden Eintrags der zuständige Super-Peer kontaktiert. Je nach Zugriffsmuster der Anwendung kann zwischen zwei Ansätzen gewählt werden.
Wenn das Zugriffsmuster zufällig ist und keine Datenlokalität vorliegt, erlaubt eine Hashtabelle mit beschränkter Größe den schnellsten Zugriff auf die gecachten Daten. Die Hashtabelle speichert nur einfache Einträge, bei denen eine CID auf eine NID abgebildet wird. Falls die Hashtabelle einen definierten Füllgrad erreicht, müssen alte Einträge gelöscht werden, um Platz für neue Daten zu machen. Zu diesem Zweck wird die Kollisionsauflösung so angepasst, dass keine neue Stelle gefunden, sondern der Eintrag an der gefunden Stelle überschrieben wird.
Wenn allerdings ein Zugriffsmuster vorliegt, bei dem nach dem Zugriff auf einen Chunk die Wahrscheinlichkeit hoch ist, dass als nächstes auf einen Chunk in der Nähe der vorherigen CID zugegriffen wird, ist eine andere Datenstruktur vorteilhafter. So ein Zugriffsmuster liegt beispielsweise vor, wenn auf Chunks, die nacheinander erzeugt wurden, in der Regel auch nacheinander zugegriffen wird. In diesem Fall wird auf Seite des Clients ebenfalls ein CID-

Baum eingesetzt. In diesem CID-Baum müssen allerdings für die Bereiche komplette CIDs verwendet werden, da ein Peer Antworten von mehreren Super-Peers cacht. Die Antworten der Super-Peers bestehen in diesem Zusammenhang auch nicht mehr nur aus einer NID, sondern aus dem kompletten CID-Bereich (Start-CID, End-CID und NID) in dem die CID der Anfrage liegt. Der Vorteil dieses Ansatzes lässt sich anhand eines Beispiels gut demonstrieren. Wenn Peer *A* den Chunk *B5* anfordern will, kontaktiert er den Super-Peer, der für den Peer *B* zuständig ist. Der Super-Peer antwortet mit dem Speicherort von *B5*, der falls *B5* nicht migriert wurde der Peer *B* ist. Werden als nächstes die Chunks *B6* bis *B10* benötigt, wird für jeden Chunk erneut der Super-Peer kontaktiert. Werden allerdings CID-Bereiche gecacht und befinden sich die Chunks *B5* bis *B10* im gleichen CID-Bereich, muss der Super-Peer nur ein einziges Mal kontaktiert werden. Alle weiteren Objektsuchen, können durch den lokalen Cache beantwortet werden.
Damit der CID-Baum alle Funktionen eines Caches übernehmen kann, müssen einige Modifikationen durchgeführt werden. Erstens muss die Anzahl der Einträge begrenzt werden, damit der CID-Baum nur eine bestimmte Menge an Speicher belegt. Zweitens muss es möglich sein CIDs zu erkennen, die nicht gecacht sind und gecachte CIDs zu invalidieren. Drittens muss es möglich sein Einträge zu löschen, um für neuere Einträge Platz zu schaffen.

Beschränkung der Einträge

Die Beschränkung der Einträge ist am einfachsten Umzusetzen. Dafür kann ein simpler Zähler verwendet werden, der die Anzahl der CID-Bereiche speichert. Die Integration des Zählers in den CID-Baum erlaubt es immer die korrekte Anzahl an CID-Bereichen zu ermitteln. Wie auch bei einer Migration ist es möglich, dass durch das Hinzufügen oder Löschen eines Eintrags beziehungsweise eines Bereiches kein neuer CID-Bereich entsteht sondern sich nur die Länge bestehender Bereiche verändert.

Erkennung nicht vorhandener Einträge und Invalidierung

Um nicht gecachte Einträge zu erkennen, wird bei der Initialisierung ein CID-Bereich eingefügt, der den gesamten CID-Raum umfasst. Als NID wird -1 verwendet, um zu kennzeichnen, dass der Bereich keinem Knoten zugeordnet ist. Wenn weitere CID-Bereiche hinzugefügt werden, wird der initiale CID-Bereich aufgesplittet.
Die Invalidierung von bereits gecachten CIDs erfolgt auf ähnlich Weise. Um eine CID zu invalidieren, wird die CID zur NID -1 „migriert“. Im Normalfall wird allerdings keine einzelne CID invalidiert, sondern direkt der ganze zugehörige CID-Bereich, da der wahrscheinlichste Grund

CID-Start	CID-Ende	Erzeugung
300	900	31.05.2015 08:49:07.896
17.700	18.000	31.05.2015 10:02:28.998
11.200	12.000	31.05.2015 18:27:53.456
12.300	12.600	01.06.2015 01:35:57.037
7.400	11.100	01.06.2015 04:15:15.764
1.000	2.000	01.06.2015 04:27:32.290
15.000	15.800	01.06.2015 21:58:54.686
13.000	14.500	02.06.2015 00:54:01.572
4.000	6.000	02.06.2015 03:59:39.760
2.200	2.300	02.06.2015 22:27:11.875

Tabelle 4.1: Löschliste (ältester Eintrag). *Die Liste ist sortiert nach dem Erzeugungsdatum. Es wird immer der CID-Bereich gelöscht, der am ältesten ist.*

CID-Start	CID-Ende	Zugriffe
1.000	2.000	33
4.000	6.000	102
300	900	211
7.400	11.100	220
11.200	12.000	337
2.200	2.300	355
13.000	14.500	483
15.000	15.800	602
12.300	12.600	678
17.700	18.000	797

Tabelle 4.2: Löschliste (Zugriffe). *Die Liste ist sortiert nach der Anzahl der Zugriffe. Es wird immer der CID-Bereich gelöscht, der die wenigsten Zugriffe hat.*

für eine Invalidierung ein Ausfall des entsprechenden Knotens ist. Im Gegensatz zu einer „normalen“ Migration, wird der CID-Bereich im Baum ermittelt und die NID auf -1 gesetzt. Dafür wird der Bereich eventuell mit den benachbarten Bereichen verschmolzen.

Einträge löschen

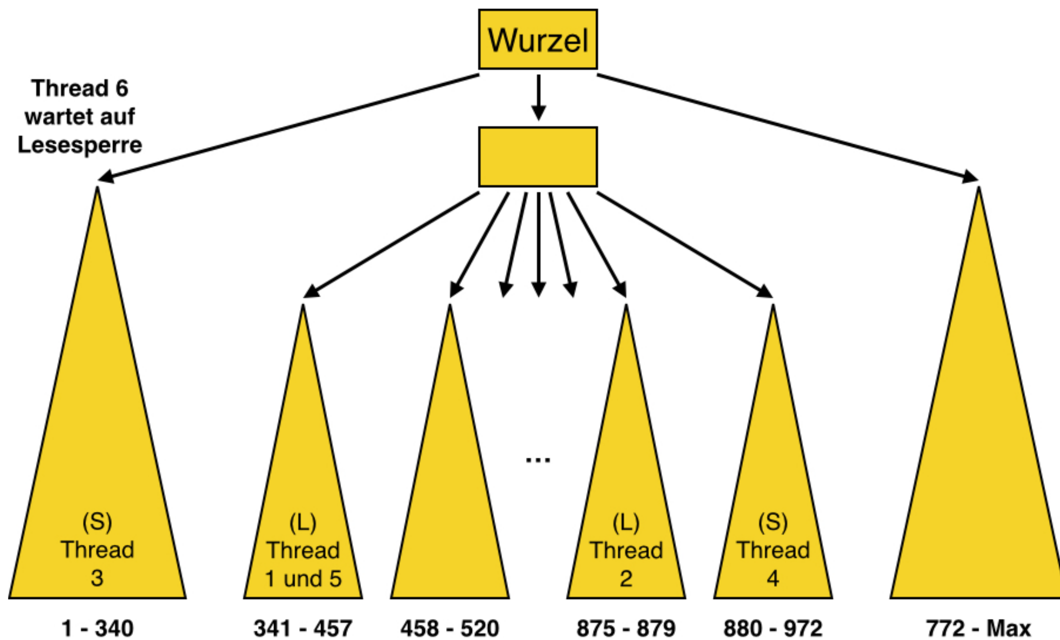
Das Löschen von Einträgen gestaltet sich am schwierigsten, da sinnvollerweise immer der Eintrag gelöscht werden sollte, der entweder am längsten vorhanden ist oder auf den am längsten nicht mehr zugegriffen wurde. Ein Zeitstempel wäre dafür am besten geeignet, verbraucht aber mindestens vier zusätzliche Bytes in jedem CID-Bereich. Bei sehr vielen Zugriffen sind gegebenenfalls sogar mehr Bytes nötig. Alternativ könnte für jeden CID-Bereich ein Zähler geführt

CID-Start	CID-Ende	Letzter Zugriff
17.700	18.000	31.05.2015 06:55:42.405
4.000	6.000	31.05.2015 10:38:05.867
300	900	31.05.2015 11:40:32.124
12.300	12.600	31.05.2015 19:35:56.548
11.200	12.000	01.06.2015 01:03:08.522
13.000	14.500	01.06.2015 10:33:10.310
15.000	15.800	02.06.2015 06:40:27.385
7.400	11.100	02.06.2015 07:07:38.843
1.000	2.000	02.06.2015 08:44:37.275
2.200	2.300	02.06.2015 16:32:07.704

Tabelle 4.3: Löschliste (letzter Zugriff). *Die Liste ist sortiert nach dem Datum des letzten Zugriffes. Es wird immer der CID-Bereich gelöscht, dessen letzter Zugriff am längsten her ist.*

werden, um die Anzahl der Zugriffe zu speichern. Bei beiden Alternativen, ist allerdings die Suche nach dem zu löschenden Bereich sehr aufwändig, da alle Bereiche überprüft werden müssen.

Gelöst wird das Problem mit einer zusätzlichen sortierten Liste, in der einige wenige CID-Bereiche vermerkt sind. Für das Löschen des ältesten Eintrags enthält die Liste die ältesten Einträge (siehe Tabelle 4.1). Die Liste ist anfangs leer und wird beim Hinzufügen von CID-Bereichen in den Baum gefüllt. Sobald alle Einträge der Liste gelöscht wurden, werden alle CID-Bereiche durchsucht und die Liste wieder gefüllt. Wird ein Zähler für die Zugriffe verwendet (siehe Tabelle 4.2) oder soll einer der Bereiche gelöscht werden, auf die am längsten nicht mehr zugegriffen wurde (siehe Tabelle 4.3), ist eine etwas andere Strategie sinnvoll. Die Liste ist anfangs leer und wird erst gefüllt, wenn der erste Eintrag gelöscht werden soll. Dabei werden alle CID-Bereiche durchsucht und in die Liste diejenigen Bereiche eingetragen, die beispielsweise den kleinsten Zählerwert besitzen. Anschließend wird der Bereich mit dem niedrigsten Zählerwert entfernt. Beim nächsten Löschen kann direkt ein Eintrag aus der Liste verwendet werden. Bei einem regulären Zugriff auf einen CID-Bereich wird auch der Wert in der Liste angepasst. Ist die Liste irgendwann leer, werden erneut alle CID-Bereiche durchsucht. Diese Strategie garantiert zwar nicht, das wirklich der CID-Bereich mit dem geringsten Zählerwert gelöscht wird, es wird aber zumindest der CID-Bereich gelöscht, der von einer kleinen Auswahl den geringsten Zählerwert hat.



Thread 1: Suche 422 **Thread 2: Suche 876** **Thread 3: Migriere 15**

Thread 4: Migriere 894 **Thread 5: Suche 431** **Thread 6: Suche 325 (wartend)**

Abbildung 4.7: Paralleler Zugriff im CID-Baum. *Die Lese-Schreib-Sperren in jedem Knoten erlauben den gleichzeitigen Zugriff mehrerer schreibender und lesender Threads in unterschiedlichen Zweigen des CID-Baumes.*

4.3.4 Paralleler Zugriff

Wie schon die lokale Metadaten-Verwaltung (siehe Abschnitt 3.4) müssen auch die hier vorgestellten Datenstrukturen für den parallelen Zugriff durch mehrere Threads angepasst werden. Im einfachsten Fall wird nur eine globale Lese-Schreib-Sperre eingesetzt, allerdings kann die Granularität noch verfeinert werden, so dass eine bessere Parallelität möglich ist. Dafür verfügt jeder Knoten des CID-Baumes über eine eigene Lese-Schreib-Sperre. Bei einem Lesezugriff wird auf jeder Ebene die Lesesperre des entsprechenden Knoten angefordert und freigegeben, sobald die nächste Ebene erreicht wird. Bei einem Schreibzugriff wird das gleiche Verfahren angewendet, bis der Knoten erreicht wird, der geändert werden soll. Für diesen Knoten wird die Schreibsperre angefordert und die Änderung durchgeführt. Beim Löschen oder Einfügen kann es jedoch erforderlich sein, dass der zugrunde liegende B-Baum ausgeglichen werden muss [151]. Dieser Fall liegt vor, wenn der Knoten nach dem Löschen eines Wertes weniger als die Mindestanzahl an Einträgen besitzt ($\widehat{=}$ Ordnung $-$ 1) oder wenn der Knoten nach dem Einfügen eines Wertes zu viele Einträge ($\widehat{=} 2 *$ Ordnung $+ 1$) besitzt.

Der Knoten wird aufgespalten, wenn er nach dem Einfügen zu viele Einträge besitzt. Dabei wird der mittlere Eintrag in den Vaterknoten verschoben und die restlichen Einträge auf zwei neue Knoten aufgeteilt. In diesem Fall muss zusätzlich zur aktuellen Schreibsperre auch die Schreibsperre des Vaterknotens angefordert werden. Hat der Vaterknoten nach der Aufspaltung ebenfalls zu viele Einträge, muss auch der Vaterknoten aufgespalten werden, usw. Dieser rekursive Vorgang ist relativ teuer, tritt allerdings selten auf und lässt sich durch die Anpassung der Ordnung beeinflussen.
Wenn der Knoten nach dem Löschen eines Eintrags zu wenige Einträge besitzt, muss eine Verschiebung oder eine Verschmelzung durchgeführt werden. Bei einer Verschiebung ersetzt der symmetrische Vorgänger (oder der symmetrische Nachfolger) den zu löschenden Eintrag. Der symmetrische Vorgänger ist der größte Eintrag im linken Teilbaum des gelöschten Knotens und entsprechend ist der symmetrische Nachfolger der kleinste Eintrag im rechten Teilbaum. Um den entsprechenden Eintrag zu verschieben, werden weitere Sperren benötigt. Ist eine Verschiebung nicht möglich, da keine weiteren Teilbäume existieren oder weil die entsprechenden Knoten in den Teilbäumen anschließend zu wenig Einträge hätten, muss eine Verschmelzung durchgeführt werden. Dabei wird der Knoten mit dem gelöschten Eintrag mit seinem vorausgehenden (oder nachfolgenden) Geschwisterknoten zusammengefügt. Als mittleren Eintrag wird der Eintrag aus dem Vaterknoten verwendet, der vorher zwischen den beiden Geschwisterknoten bestand. In diesem Fall muss also sowohl für den Vaterknoten als auch für einen der Geschwisterknoten die Schreibsperre angefordert werden. Anschließend kann es notwendig sein, dass weitere Ausgleichsmaßnahmen durchgeführt werden müssen. Offensichtlich ist das Löschen eines Eintrag sehr teuer, die Ausgleichsmaßnahmen können jedoch amortisiert in konstanter Zeit durchgeführt werden [153].
Abbildung 4.7 zeigt vereinfacht die parallele Ausführung von zwei schreibenden und vier lesenden Threads. Der erste Thread führt eine Suche nach der CID 422 durch und benötigt dafür die Lesesperre des zweiten Zweiges. Auch der zweite Thread führt eine Suche aus und akquiriert die Lesesperre des vierten Zweiges. Der dritte Thread greift schreibend auf den CID-Baum zu um die CID 15 zu migrieren. Dazu wird die Schreibsperre des ersten Zweiges angefordert. Thread vier führt ebenfalls eine Migration durch und fordert dazu die Schreibsperre des fünften Zweiges an. Der fünfte und sechste Thread führen wieder eine Suche aus, allerdings innerhalb von bereits gesperrten Zweigen. Thread fünf greift auf den zweiten Zweig zu und kann die Lesesperre anfordern, da Thread eins auch lesend auf den Zweig zugreift. Dagegen muss der sechste Thread warten, da er lesend auf den ersten Zweig zugreifen will, allerdings der dritte Thread bereits die Schreibsperre besitzt. Von den sechs parallelen Zugriffen können dementsprechend in diesem Fall fünf Zugriffe gleichzeitig durchgeführt werden.

4.4 Namensdienst

Manche Anwendungen erfordern, dass zumindest auf einen Teil der Objekte über benutzerdefinierte Schlüssel zugegriffen werden kann. In einem sozialen Netzwerk wird beispielsweise die E-Mail-Adresse mit dem Datensatz des Benutzers verknüpft. Alle weiteren Objekte, wie Freundesliste, Ressourcen, Kommentare, Bilder, etc. sind von diesem Datensatz aus erreichbar und müssen daher über keinen benutzerdefinierten Schlüssel verfügen. Zu diesem Zweck wird ein Namensdienst verwendet, der für die Umsetzung von benutzerdefinierten Schlüsseln auf die sequentiell erzeugten CIDs verantwortlich ist. Der benutzerdefinierte Schlüssel muss in diesem Fall schon bei der Erzeugung des entsprechenden Chunks angegeben werden. Der Chunk kann nach der Erzeugung sowohl mit dem benutzerdefinierten Schlüssel als auch mit seiner CID adressiert werden. Jeder Super-Peer ist für einen Teil dieser Zuordnungen zuständig und speichert diese in einer dafür angepassten Hashtabelle.
Nachfolgend wird ein Ansatz erläutert, welcher eine einfache Integration eines Namensdienstes erlaubt.

4.4.1 Zuordnung eintragen

Der Namensdienst lässt sich so konfigurieren, dass entweder Zahlenwerte oder Zeichenketten als benutzerdefinierte Schlüssel verwendet werden. Beim Einsatz von Zeichenketten wird ein Konvertierungsmechanismus verwendet, der aus einer Zeichenkette einen Zahlenwert erzeugt. Jedes Zeichen wird dabei in eine definierte Anzahl Bits umgeformt, wobei die Anzahl der Bits festlegt wie viele unterschiedliche Zeichen für den benutzerdefinierten Schlüssel verwendet werden können. So erlauben 6 Bit die Verwendung von 64 unterschiedlichen Zeichen, beispielsweise die Ziffern 0-9, Klein- und Großbuchstaben und ein Sonderzeichen. Wenn Groß- und Kleinschreibung ignoriert werden, können alternativ auch weitere Sonderzeichen unterstützt werden und so die gängigsten Zeichen für E-Mail-Adressen ebenfalls mit 6 Bit abgedeckt werden. Werden mehr unterschiedliche Zeichen benötigt, kann die Anzahl der Bits einfach erhöht werden. Wenn die zusätzlichen Zeichen nur sehr selten benötigt werden, kann die Anzahl der Bits auch gleich bleiben und die zusätzlichen Zeichen durch eine definierte Folge der bestehenden Zeichen maskiert werden.
Ein CRC16-Generator wird verwendet, um einen 2-Byte langen Hashwert für die erzeugten Zahlenwerte zu generieren. Der Hashwert wird auf den Super-Peer-Ring abgebildet und damit der zuständige Super-Peer ermittelt. Anschließend wird dem Super-Peer der Zahlenwert und die dazugehörige CID übermittelt und dort in die entsprechende Hashtabelle eingetragen.

4.4.2 CID ermitteln

Die Ermittlung einer CID mithilfe des Namensdienstes erfolgt auf ähnliche Weise. Dabei wird der benutzerdefinierte Schlüssel, wie bereits beschrieben in einen Zahlenwert umgewandelt. Dieser Wert wird anschließend mit Hilfe des CRC16-Generators gehasht und auf den Super-Peer-Ring abgebildet. Der für den Hashwert zuständige Super-Peer wird kontaktiert und die CID für den benutzerdefinierten Schlüssel angefragt. Der Super-Peer verwendet den Schlüssel für seine lokale Hashtabelle und schickt die damit ermittelte CID an den anfragenden Knoten zurück.

4.4.3 Modifizierte Hashtabelle

Als Hashtabelle wird ein Byte-, Short, Integer- oder Long-Array verwendet, wobei ein Eintrag in der Hashtabelle mehrere hintereinander liegende Plätze im Array belegt. Die Anzahl der Plätze hängt von der maximalen Größe der Zahlenwerte beziehungsweise von der maximalen Länge der Zeichenketten ab. Sind beispielsweise die Zeichenketten maximal 10 Zeichen lang und werden die Zeichen mit 6 Bit maskiert, verbrauchen die erzeugten Schlüssel maximal 60 Bit oder aufgerundet 8 Byte. Die CID (der Wert des Hashtabellen-Eintrags) benötigt ebenfalls 8 Byte. Wird in diesem Beispiel ein Long-Array eingesetzt, belegt ein Eintrag zwei Plätze im Array, Einen für den Schlüssel und Einen für den Wert. Wird ein Byte-Array verwendet, werden hingegen 16 Plätze benötigt. Beim Zugriff auf die Hashtabellen werden Schlüssel und Wert transparent zerlegt beziehungsweise zusammen gefügt.

4.5 Verwandte Arbeiten

Metadaten-Verwaltung und Objektsuche sind in jedem verteilten System unabdingbar und daher ein sehr wichtiges Forschungsgebiet. Die große Menge an Objekten die dezentral verwaltet werden, stellt allerdings neue Anforderungen an die Metadaten-Verwaltung, die bisher nur unzureichend betrachtet wurden.
Mehrere der hier aufgeführten Systeme wurden schon grundlegend in Abschnitt 2.5 betrachtet. Für diese Systeme werden nachfolgend nur die Unterschiede zu den in diesem Kapitel vorgestellten Konzepten beschrieben, fokussiert auf die globale Metadaten-Verwaltung.

RAMCloud
RAMCloud organisiert Anwendungsdaten in Tabellenform und verwendet für die Objektsuche eine Hashtabelle, die durch einen zentralen Koordinator in Form von Tablets (Bereiche des

Hashraums) verwaltet wird [9]. Tablets lassen sich unterteilen oder zusammen fügen und Daten eines Tablets können von einem Knoten zu einem anderen migriert werden [154]. Der zentrale Koordinator ist nicht zwangsweise ein einzelner Knoten, sondern besteht in der Regel aus drei bis sieben Knoten, die ein gemeinsames Konsensus-Protokoll nutzen. Beim ersten Zugriff auf eine Tabelle erhält der Client alle Tablets und cacht diese für zukünftige Anfragen.
In RAMCloud teilt jeder Knoten dem zentralen Koordinator seinen „letzen Willen“ mit [154]. Dazu werden so genannte Tablet-Profile erstellt, die dem Koordinator sagen, wie die Daten des Knotens bei einem Ausfall wiederhergestellt werden sollen. Der Knoten trägt selbst dafür Sorge, dass die einzelnen Tablet-Profile ungefähr gleich groß sind und gleich schnell wiederhergestellt werden können.
Die Datenwiederherstellung wird durch den Koordinator angestoßen und koordiniert. Wird ein Ausfall erkannt, schickt der Koordinator eine Broadcast-Nachricht an alle Knoten und erfragt die für den ausgefallenen Knoten gespeicherten Segmente. Sobald alle Knoten geantwortet haben, hat der Koordinator eine globale Sicht. Anschließend bestimmt der Koordinator die an der Wiederherstellung beteiligten Knoten und koordiniert den weiteren Vorgang.
Offensichtlich unterscheiden sich die Konzepte von RAMCloud von den in diesem Kapitel Vorgestellten. Im Gegensatz zu einer zentralen Instanz kommt ein dezentrales Super-Peer-Overlay zum Einsatz, wodurch die Anfragen und die wesentlich größere Menge an Metadaten auf viele Knoten verteilt werden. Auch die Integration der notwendigen Metadaten für die Datenwiederherstellung erfolgt auf sehr unterschiedliche Weise. In RAMCloud wird mit einer Broadcast-Nachricht ermittelt, welche Daten wiederhergestellt werden müssen und welche Knoten dafür zuständig sind. Das hier vorgestellte Konzept hingegen unterteilt alle Objekte in Backupzonen, die aus einer konfigurierbaren Anzahl an Chunks besteht. Für jede Backupzone sind drei Backupknoten zuständig. Durch eine explizite Backupreihenfolge ist jederzeit klar, welcher Backupknoten welche Daten wiederherstellt. Der Super-Peer verwaltet die Backupzonen in einem Array und muss im Fehlerfall die zuständigen Backupknoten nur noch kontaktieren. Die Backupknoten stellen anschließend selbstständig (ohne weitere Informationen des Super-Peers) die notwendigen Daten wieder her. Neben der Wiederherstellung der Daten können in DXRAM auch Metadaten rekonstruiert werden. Dies ist sogar möglich, wenn mehrere Super-Peers gleichzeitig ausfallen.

Trinity Graph Engine

Trinity Graph Engine ist ein verteilter, RAM-basierter Schlüssel-Wert-Speicher für die Online-Anfragebearbeitung und Offline-Analyse von großen Graphen [15]. Daten-Objekte werden zusammen mit einem 64-Bit Schlüssel in der Speicher-Cloud als Schlüssel-Wert-Paaren verwaltet. Zum Auffinden wird der 64-Bit Schlüssel auf einen p-Bit Wert gehasht und als Zeiger in eine Adresstabelle mit 2^p Plätzen verwendet. Die Adresstabelle wird auf einem definierten

Knoten (Leader) verwaltet und im Trinity File System repliziert. Darüber hinaus existiert eine Kopie der Tabelle auf jedem Knoten. Mit Hilfe der Tabelle wird der Knoten bestimmt, der den Memory Trunk speichert. Auf dem Knoten selber wird der Schlüssel wiederum gehasht und damit in der Hashtabelle des Memory Trunks der Versatz und die Länge des Objektes ermittelt. Das in diesem Kapitel vorgestellte Konzept verzichtet auf Hashtabellen zu Gunsten von Datenstrukturen, die CID-Bereiche effizient speichern können. Dieses Konzept erlaubt eine erhebliche Reduzierung des Speicherverbrauchs, was die Verwaltung der Metadaten einer so großen Anzahl von Objekten überhaupt erst ermöglicht. Neben dem Aufwand, um die lokalen Kopien der globalen Hashtabelle in Trinity konstant zu halten, ist auch eine Abstimmung notwendig, wenn der Leader ausfällt. Bei DXRAM hingegen müssen die Daten der Super-Peers nicht abgeglichen werden und Metadaten können selbst dann rekonstruiert werden, wenn mehrere Super-Peers gleichzeitig ausfallen. Da Trinity nicht als Open-Source zur Verfügung steht, ist ein genauerer Vergleich leider nicht möglich.

FaRM

FaRM ist eine RAM-basierte verteilte Berechnungsplattform, die RDMA für den rechnerübergreifenden Datenzugriff nutzt [61]. Der verteilte Speicher ist in 2 GB große, gemeinsam genutzte Speicherregionen (DSM) aufgeteilt, die die Basis für die Adressierung, die Datenwiederherstellung und die RDMA-Registrierung bilden. Jede globale ID besteht aus einer 32-Bit großen Regions-ID und einem 32-Bit-Offset. Zum Auffinden einer Region wird Consistent-Hashing mit mehreren virtuellen Ringen eingesetzt und die Metadaten der Region über RDMA bezogen und lokal gecacht. Für die Objektsuche kann neben der globalen ID auch ein Namensdienst verwendet werden. Für den Namensdienst setzt FaRM ein modifiziertes Hopscotch-Hashing mit Verkettung ein. Die verteilte Hashtabelle besteht aus „normalen“ Objekten, die selbst in FaRM gespeichert sind und ist so konzipiert, dass der passende Eintrag mit wenigen sperrfreien RDMA-Lesezugriffen bezogen werden kann. Die Einhaltung der Konsistenz erfordert einen hohen Aufwand (Objektversion, Verbundversion, Inkarnationsnummer), weswegen das Einfügen, Aktualisieren und Löschen von Einträgen mit Transaktionen erfolgt.
Im Unterschied zu FaRM erlauben die CID-Bereiche eine sehr speichereffiziente Möglichkeit zur Metadaten-Verwaltung und einfache Integration der Backup-Informationen. Der in diesem Konzept vorgestellte Namensdienst ist über lokale Hashtabellen auf allen Super-Peers verteilt und ermittelt den ID zu einem benutzerdefinierten Schlüssel in einem einzigen Netzwerkzugriff. In FaRM ist der Namensdienst selbst in FaRM gespeichert und benötigt gegebenenfalls mehrere RDMA-Lesezugriffe, um den passenden Eintrag zu finden.

TAO

TAO weist jedes Objekt eindeutig einem Shard zu, indem die entsprechende Shard-ID in die Objekt-ID zusammengefasst werden [5]. Leider ist TAO nicht frei zugänglich, so dass ein genauerer Einblick in die Metadaten-Verwaltung verwehrt bleibt. Die hier vorgestellten Konzepte zielen allerdings darauf ab, dass alle Anwendungsdaten permanent im Hauptspeicher gehalten werden, wohingegen TAO nur einen Teil der Daten vorhält und bei Bedarf auf den eigentlichen Sekundärspeicher zurückgreifen muss. Im Gegensatz zu den meisten Caches kümmert sich TAO selbstständig um die Konsistenz zwischen Cache und Sekundärspeicher. Dabei sind Schreibzugriffe für Clients des gleichen Cachebereichs sofort sichtbar, aber nicht sofort persistent. Die Persistenz wird erst nach einiger Zeit erreicht. Auch die Clients anderer Cachebereiche erhalten nicht sofort den aktuellen Zustand sondern können veraltete Daten lesen.

Cassandra

Cassandra ist ein verteiltes Speichersystem mit vielen Servern für eine sehr hohe Anzahl an strukturierten Daten [44]. Datenobjekte werden in Tabellen organisiert und durch eine beliebig lange Zeichenkette (Schlüssel) identifiziert. Consistent-Hashing wird eingesetzt um die Daten über die einzelnen Speicherknoten zu verteilen. Der Adressraum wird in Form eines Ringes abgebildet und jeder Knoten hat eine Position auf dem Ring und ist für einen Teil des Rings zuständig. Zur Lastverteilung ist es in Cassandra möglich, dass die Knoten ihre Ringposition anpassen und somit ihren zuständigen Bereich verkleinern oder vergrößern.
Cassandra ist ein festplatten-basiertes Speichersystem für strukturierte Daten, wohingegen DXRAM alle Daten permanent im RAM hält und unstrukturierten Binärdaten verwaltet. Statt Consistent-Hashing wird in dem in diesem Kapitel vorgestellten Konzept ein Super-Peer-Overlay für die Objektsuche eingesetzt.

Rangetable

In [155] wird für den Einsatz in skalierbaren Speichersystemen mit schreibintensiven Zugriffsmustern die Datenstruktur Rangetable präsentiert, die Daten auf Festplatte in Reihenfolge der Schlüssel hält und über mehrere große Dateien anhand von Schlüssel-Bereichen partitioniert. Dabei werden alle Daten eines Bereichs in einer einzelnen Datei gespeichert. Die Rangetable besteht aus drei Komponenten im RAM (Itemtable, Rangeindex, Chunkindex) und einer Komponente auf Festplatte (Rangefiles). Die Rangefiles entsprechen den Dateien auf der Festplatte, wobei jeder Rangefile in gleich große Chunks unterteilt ist und Daten eines bestimmten Schlüssel-Bereichs beinhaltet. Um die einzelnen Chunks zu lokalisieren wird für jeden Rangefile ein Chunkindex im RAM gehalten, der den Offset und den ersten Schlüssel jedes Chunks beinhaltet. Der Rangeindex ist eine Tabelle im RAM, deren Einträge auf die einzelnen Chunkindizes verweisen. Jeder Eintrag umfasst dabei einen Schlüssel-Bereich, der durch den

Chunkindex abgedeckt ist. Aktualisierungen landen nicht sofort in dem entsprechenden Rangefile, sondern werden im RAM in der Itemtable vorgehalten. Die Itemtable ist ein synchronisierter Rot-Schwarz-Baum, der neue Daten in sortierter Reihenfolge speichert und einen schnellen Zugriff für die Suche einzelner Einträge und Bereichsuchen ermöglicht. Wenn die Itemtable zu groß wird, führt ein Rangemerge-Algorithmus Teile der Itemtable und der Rangefiles zusammen und verkleinert damit die Itemtable.
Obwohl eine Verwaltung der Metadaten in Form von Schlüssel- beziehungsweise ID-Bereichen erfolgt, unterscheidet sich der Ansatz erheblich von dem in diesem Kapitel vorgestellten Konzept. Die Rangetable ist für Anwendungen konzipiert, bei denen Schreibzugriffe die Zugriffsmuster dominieren und die Anwendungsdaten überwiegend auf Festplatte gespeichert sind. Die Sortierung nach Schlüssel-Bereichen ist dabei auf den sequentiellen Zugriff von Festplatten abgestimmt. Demgegenüber ist DXRAM und die vorgestellte globale Metadaten-Verwaltung für Anwendungen mit überwiegend Lesezugriffen entwickelt worden. Die Daten werden permanent im RAM gehalten, der wahlfreien Zugriff ermöglicht und Daten daher nicht in sequentieller Reihenfolge organisiert werden müssen. Darüber hinaus handelt es sich bei dem Rangetable-Ansatz nur um eine lokale Datenstruktur, die durch einen globalen (nicht näher beschriebenen) Index ergänzt wird, wohingegen in dieser Arbeit ein integriertes Konzept aus lokaler und globaler Metadaten-Verwaltung beschrieben wird. Abschließend unterscheidet sich auch der in der Itemtable verwendete Rot-Schwarz-Baum von der modifizierten B-Baum-Struktur im vorgestellten Konzept.

BlobSeer

BlobSeer ist ein verteilter Speicher- und Verwaltungsdienst für große Binärobjekte (Binary Large Objects oder BLOBs) [156, 157]. BLOBs repräsentieren unstrukturierte Daten und können bis zu einem Terabyte groß sein [158]. Jeder BLOB wird in gleich große Blöcke einer (bei der BLOB-Erzeugung) definierten Größe unterteilt, die auf mehrere Speicherknoten verteilt werden. Zugriffe erfolgen immer auf einen Bereich (Offset, Länge) der aus mehreren Blöcken bestehen kann, und im Falle eines Schreibzugriffs eine neue Version der betroffenen Blöcke erzeugt. Für die Verwaltung der Blöcke eines BLOBs wird ein verteilter Segmentbaum verwendet, wobei für jede Version des BLOBs ein eigener Wurzelknoten existiert und nicht modifizierte Knoten von mehren Segmentbäumen gemeinsam genutzt werden [25]. Die verwendeten Segmentbäume basieren auf dem in [159] vorgestellten Konzept und sind in einer DHT gespeichert. Für die Objektsuche werden die ID des BLOBs, die spezifische Version und der Zugriffsbereich benötigt. Der Client kontaktiert zuerst den Versions Manager, überprüft die angegebene Version und erfragt die zuständigen Metadaten Provider. Die anschließende Anfrage an die Metadaten Provider liefert schließlich die Daten Provider, die die entsprechenden Blöcke des Bereichs speichern.

Im Gegensatz zu BlobSeer, mit bis zu 1 TB großen BLOBs, ist das präsentierte Konzept für die globale Metadaten-Verwaltung für sehr viele sehr kleine Objekte entwickelt worden. Die in BlobSeer verwendeten Segmentbäume verwalten zwar auch Bereiche, allerdings handelt es sich dabei um Blöcke eines einzelnen Objektes (beispielsweise einer Datei). Ein CID-Bereich hingegen umfasst mehrere IDs unterschiedlicher Objekte. Ferner ist die Größe der CID-Bereiche flexibel und kann problemlos angepasst werden. Im Gegensatz dazu unterteilen die Segmentbäume einen Bereich auf jeder Stufe in zwei gleich große Teilbereiche.

ISAM - Index Sequential Access Method

Ende der 60er Jahre entwickelte IBM die Zugriffsmethode Indexed Sequential Access Method (ISAM), die es ermöglicht auf Datensätze einer Datei mithilfe eines Index sowohl sequentiell als auch wahlfrei zuzugreifen [150]. Dabei können zu einer Datei durchaus mehrere Indizes existieren, um nach unterschiedlichen Merkmalen zu suchen. Die Datensätze innerhalb der Datei sind sortiert und werden zu Blöcken oder Seiten zusammengefasst. Der Index verweist dabei jeweils auf den niedrigsten Schlüssel eines Blockes. Bei der Suche nach einem Datensatz wird nicht die Datei sequentiell durchsucht, sondern lediglich der Index. Sobald der Schlüssel im Index gefunden wurde oder ein größerer Wert gelesen wird, ist der entsprechende Datenblock gefunden und wird anschließend durchsucht. Der verwendete Index kann auch mehrstufig sein, so dass beim Durchsuchen der ersten Stufe der passende Index der zweiten Stufe ermittelt wird. Beim Durchsuchen der zweiten Stufe wird er Index der dritten Stufe ermittelt, usw., bis zum Schluss der eigentliche Datenblock durchsucht wird. So ein mehrstufiger Index bei ISAM hat viel Ähnlichkeit mit heutigen B-Baum-Varianten. Der größte Nachteil bei ISAM zeigt sich beim Einfügen von Datensätzen. Sowohl Datei, als auch Index sind statisch, so dass Datensätze nicht immer an der richtigen Stelle eingefügt werden können und statt dessen in Überlaufblöcken eingefügt werden. Dadurch verlangsamt sich zum einen die Suche und zum anderen müssen Datei und Index regelmäßig zeitaufwändig reorganisiert werden.
Im Gegensatz zu ISAM basiert der CID-Baum auf einer B-Baum-Struktur, die durch mit Zeigern verknüpfte Knoten sehr flexibel ist und auf Überlaufblöcke verzichten kann. Zwar muss ein B-Baum in bestimmten Situationen ausbalanciert werden, diese Ausgleichsmaßnahmen können jedoch amortisiert in konstanter Zeit durchgeführt werden [153].

Verteilte Baumstrukturen

Es gibt eine ganze Reihe von verteilten Baumstrukturen, die zur Verwaltung von Metadaten verwendet werden können.

Der Distributed Segment Tree (DST) [159] ist ein verteilter Binärbaum, der es ermöglicht eine DHT um Bereichsabfragen zu ergänzen. Dazu bedient sich der DST den Eigenschaften des

zugrunde liegenden Segment Tree [160], der einen Bereich auf jeder Stufe des Baumes in zwei gleich große Teilbereiche aufteilt. Die Blätter des DST speichern alle IDs ihres Bereiches und die inneren Knoten replizieren einen Teil dieser IDs (begrenzter Platz pro Knoten). Wenn eine Bereichsabfrage gestellt wird, werden die benachbarten Knoten gesucht, die diesen Bereich abdecken, und anschließend alle dazugehörigen IDs zurück gegeben. Mit Hilfe eines Splitting-Algorithmus können alle notwendigen Knoten ermittelt und parallel kontaktiert werden.

Der Distributed B-Tree (DBT) [161] ist ein verteilter, skalierbarer und fehlertoleranter B-Baum, dessen Knoten auf mehreren Servern verteilt sind. Der Zugriff auf die Knoten erfolgt mit Mini-Transaktionen und Clients cachen innere Knoten für eine bessere Performanz. Die aktuellen Versionsnummern der inneren Knoten sind auf allen Servern repliziert und ermöglichen so eine schnelle Durchführung der Transaktionen auf einem beliebigen Server. Die Verteilung der Knoten auf die Server ist nicht vorgegeben, jeder Knoten verfügt aber über eine global eindeutige ID, über die er angesprochen werden kann.
Die globalen IDs, die Fehlertoleranz, die Transaktionen und die Speicherverwaltung werden vom zugrunde liegenden verteilten Speicherdienst (Sinfonia [90]) zur Verfügung gestellt.

Minuet ist ein RAM-basierter verteilter B-Baum mit der Unterstützung von Transaktionen, Copy-on-Write Schnappschüssen und modifizierbaren Klonen [162]. Zu diesem Zweck wird der Distributed B-Tree um eine Versionierung der Knoten, zusätzliche Knotenfelder und eine Garbage Collection erweitert.

Im Gegensatz zu den genannten Baumstrukturen ist der vorgestellte CID-Baum nicht verteilt, sondern befindet sich lokal auf jedem Super-Peer. Damit ist es möglich in einer einzigen Netzwerkanfrage an den zuständigen Super-Peer die gewünschten Metadaten zu erhalten. Bei den verteilten Baumstrukturen müssen potentiell mehrere Netzwerkanfragen an verschiedene Knoten gestellt werden. Caching von inneren Knoten kann zwar die Anzahl der Netzwerkanfragen reduzieren, der in diesem Kapitel präsentierte Ansatz für das Caching ermöglicht hingegen teilweise auf Netzwerkanfragen komplett zu verzichten. Grundsätzlich lassen sich die verteilten Baumstrukturen auch auf jedem Super-Peer lokal verwenden (also nicht verteilt), dabei entsprechen sie im Prinzip aber dem zugrunde liegendem Segmentbaum (DST) beziehungsweise B-Baum. (DBT und Minuet). Der CID-Baum basiert ebenfalls auf einem B-Baum, ist jedoch für die Verwaltung von Bereichen erweitert worden und unterscheidet sich damit vom DBT und Minuet. Im Gegensatz zu einem Segmentbaum, beim den alle Bereiche gleich groß sind, sind die CID-Bereiche sehr flexibel und lassen sich auch in der Größe anpassen.

Range-Queries

Die Suche in ID-Bereichen (Range Query oder Range-based Query) wird von vielen Systemen und Datenstrukturen unterstützt [163, 164, 165, 166, 167, 168, 169]. Dabei wird eine untere

und obere Grenze angegeben und alle IDs gesucht, die innerhalb des angegebenen Bereichs liegen.
Die Verwaltung von Metadaten in Form von ID-Bereichen (Range-based Metadata Management) ist davon klar abzugrenzen. Die CID-Bereiche ermöglichen es große Mengen an Metadaten zu bündeln und dadurch den Speicherverbrauch erheblich zu reduzieren. Die Suche von allen IDs innerhalb eines angegebenen Bereich ist dabei uninteressant. Falls zukünftig Bereichsanfragen in DXRAM umgesetzt werden sollten, lassen sich jedoch solche Anfragen leicht in das Konzept der CID-Bereiche integrieren.

4.6 Zusammenfassung

Verteilte Speichersysteme müssen für jedes Objekt Metadaten vorhalten, die es ermöglichen ein Objekt aufzufinden oder bei einem Knotenausfall wiederherzustellen. Die große Anzahl an Objekten in großen interaktiven Anwendungen führt zu einem sehr hohen Speicherbedarf für Metadaten. Wenn beispielsweise für jedes Objekt die IP-Adresse gespeichert würde, würden alleine die Metadaten für einen Knoten mit einer Milliarde Objekten 4 GB Speicher benötigen. Wenn weitere Metadaten hinzu kommen, erhöht sich dieser Speicherbedarf noch weiter. Diese Menge an Metadaten sind für klassische Ansätze, wie Hashtabellen und Index-Strukturen, auf Grund ihres zusätzlichen Speicherbedarfs ungeeignet. Daher ist es unerlässlich bestehende Ansätze zu erweitern und die Menge der Metadaten drastisch zu reduzieren.

In diesem Kapitel wurde ein dezentrales Super-Peer-Overlay vorgestellt. Jeder Super-Peer ist für eine Gruppe an Knoten zuständig, für die er die Metadaten speichert. Die Metadaten werden im Betrieb sowohl für Objektsuchen, als auch für die Koordinierung der Datenwiederherstellung nach einem Knotenausfall verwendet. Neben dem Speicherort jedes Objektes werden dafür auch Informationen zu den Backupknoten vorgehalten. Alle Daten der Super-Peers befinden sich permanent im RAM und werden auf mehreren weiteren Super-Peers repliziert. Durch das Overlay wird somit nicht nur die Last der Metadaten-Verwaltung auf mehrere Knoten verteilt, sondern auch ein hoher Grad an Fehlertoleranz erreicht. Ausfälle von Peers und Super-Peers werden dabei für die Anwendung transparent maskiert und der Betrieb des Systems durchgehend gewährleistet.

Für die Reduzierung der Metadaten werden mehrere globale IDs (CIDs) zu einem Bereich (CID-Bereich) zusammengefasst. Dieses neuartige Konzept erlaubt eine Reduzierung des Speicherverbrauchs von über 99,99%. Die globalen IDs der Objekte werden auf jedem Knoten sequentiell erzeugt, wobei in der Regel der erzeugende Knoten die Objekte auch speichert. Große Mengen hintereinander liegender CIDs befinden sich dementsprechend auf dem gleichen Knoten. Das Konzept der CID-Bereiche fasst solche globalen IDs zu großen Bereichen zusammen

und speichert nur noch die Zuordnung von Start- und End-CID zu einem Knoten (NID). Im besten Fall muss für jeden Knoten nur ein einziger CID-Bereich vorgehalten werden.
Um das Clustering von Hot-Spots aufzulösen, werden punktuelle Migrationen unterstützt, um Objekte auf andere Knoten zu verschieben und dadurch die Last besser zu verteilen. Bei einer Migration muss gegebenenfalls der zugehörige CID-Bereich aufgesplittet werden, wodurch sich die Anzahl der CID-Bereiche erhöht. Da Migrationen jedoch lediglich für die Lastverteilung bei Hot-Spots und bei der Datenwiederherstellung auftreten, sind die damit verbundenen Einschränkungen nur gering und beeinträchtigen den Einsatz von CID-Bereichen nicht.
Für die Verwaltung der CID-Bereiche wird ein modifizierter B-Baum (CID-Baum) eingesetzt, der die CID-Bereiche verwaltet. CID-Bereiche verfügen im Gegensatz zu einzelnen Objekten über zwei Schlüssel (Start-CID und End-CID). Der CID-Baum ist automatisch in Bereiche eingeteilt, wobei Start- und End-CIDs in den inneren Knoten und die zugehörigen NIDs in den Blättern gespeichert sind. Ein Super-Peer besitzt für jeden im zugeordneten Peer einen solchen CID-Baum. Der Speicherbedarf eines CID-Bereichs kann dadurch weiter reduziert werden, weil nur noch der lokale Teil der CIDs (die LIDs) gespeichert werden muss. Da ein Super-Peer oft viele Anfragen unterschiedlicher Peers gleichzeitig beantworten muss, ist die Datenstruktur des CID-Baumes für parallele Zugriffe optimiert. Die eingesetzte Synchronisierung erlaubt dabei prinzipiell das parallele Lesen in einem Zweig des Baumes während einer anderer Zweig modifiziert wird. Lediglich bei seltenen Ausnahmen müssen für die Ausgleichsoperationen, des zu Grunde liegenden B-Baumes, umfassendere Sperren verwendet werden.
Um das Super-Peer-Overlay zu entlasten, werden Antworten von Super-Peers von den Peers lokal gecacht, wofür eine modifizierte Form des CID-Baumes verwendet wird. Diese modifizierte Form ist in der Größe beschränkt und ermöglicht es CID-Bereiche zu invalidieren und zu löschen, um veraltete oder nicht mehr gültige Einträge zu entfernen. Die Antwort eines Super-Peers bei einer Objektsuche enthält dafür nicht nur den Speicherort des Objektes, sondern umfasst den kompletten CID-Bereich. Der lokale Cache enthält anschließend nicht nur den Speicherort für diese eine CID, sondern für möglicherweise sehr viele weitere CIDs und kann das Gesamtsystem dadurch erheblich beschleunigen.

Abgesehen von DXRAM existieren mit RAMCloud, Trinity und FaRM drei weitere Systeme, die Anwendungsdaten permanent im Hauptspeicher halten. Alle drei Systeme verenden hashbasierte Verfahren für die Objektsuche und die globale Metadaten-Verwaltung.
Bei RAMCloud werden Daten in Tabellen organisiert und die aggregierten Metadaten (Tablets) auf einem zentralen Koordinator gespeichert. Auf Grund der aggregierten Metadaten muss der Koordinator sich bei einem Knotenausfall allerdings erst eine konsistente globale Sicht verschaffen, was durch eine Broadcast-Nachricht und entsprechende Rückmeldungen erfolgt.
Trinity verwendet einen dezentralen Ansatz für die globale Metadaten-Verwaltung, bei der ein Leader die globale Adresstabelle verwaltet. Aus Gründen der Fehlertoleranz ist die Adres-

stabelle auf allen Knoten und in einem verteilten Dateisystem (TFS) repliziert. Die lokalen Kopien der Adresstabelle erfordern einen erhöhen Synchronisierungsaufwand bei Aktualisierungen und beim Ausfall des Leaders müssen die restlichen Knoten erst einen neuen Leader bestimmen, um die Metadaten-Verwaltung zu rekonstruieren.
In FaRM werden die Metadaten zu den RDMA-Regionen der einzelnen Knoten in einer Hashtabelle verwaltet und lokal gecacht. Für den Zugriff auf Daten müssen dabei gegebenenfalls mehrere RDMA-Zugriffe erfolgen. Ein Namensdienst erlaubt eine weitere Möglichkeit auf Objekte zuzugreifen, besonders bei Änderungen der Einträge ist jedoch erhöhter Aufwand notwendig, um den Namensdienst konsistent zu halten.

Ferner existieren mit Rangetable und BlobSeer zwei Systeme, die ebenfalls Metadaten in Form von Bereichen verwalten.
Rangetable ist für festplatten-basierte Anwendungen konzipiert und die Schlüssel-Bereiche sind für den sequentiellen Zugriff von Festplatten optimiert. Ein wahlfreier Zugriff auf Metadaten und Daten ist nicht bedacht und Rangetable daher für eine Verwaltung von Daten im Hauptspeicher nicht ausgelegt.
BlobSeer arbeitet mit sehr großen Dateien (bis zu 1 TB) und teilt diese in feste Bereiche (Offset, Länge) ein, die aus mehreren Dateiblöcken bestehen können. Für die Verwaltung der Datei-Bereich wird ein angepasster Segmentbaum verwendet, der dafür sorgt, dass die Bereiche eine definierte Größe haben und nicht flexibel sind. Die Datei-Bereiche sind fest einer Datei (einem Objekt) zugeordnet, wohingegen die CID-Bereiche in DXRAM mehrere Millionen Objekte umfassen können.

Kapitel 5

Messungen

In diesem Kapitel werden die in Kapitel 3 und 4 vorgestellten Konzepte evaluiert. Im Zuge dessen werden zuerst jeweils die vorgestellten Datenstrukturen mit klassischen Hashingverfahren verglichen, da diese in vielen RAM-basierten Speichersystemen verwendet werden. Für die lokale Metadaten-Verwaltungen werden zusätzlich die Speicherverwaltung und die Defragmentierung evaluiert und darauf folgend die Kombination von CID-Tabellen und Speicherverwaltung untersucht und mit RAMCloud verglichen. Da RAMCloud ebenfalls ein verteiltes RAM-basiertes Speichersystem ist und teilweise ähnliche Ziele wie DXRAM verfolgt, eignet es sich sehr gut für einen Vergleich. Bei der globalen Metadaten-Verwaltung wird abgesehen vom Vergleich des CID-Baumes mit klassischen Hashingverfahren auch der Einsatz des CID-Baumes als Cache auf Clientseite evaluiert. Abschließend werden die vorgestellten Ansätze als Gesamtsystem mit zwei weiteren Speichersystemen (MongoDB und Cassandra) verglichen. Der soziale Netzwerk-Benchmark BG wird dabei eingesetzt, um ein soziales Netzwerk mit vielen aktiven Benutzern zu emulieren.
Die in Kapitel 3 und 4 präsentierten Konzepte wurden in DXRAM implementiert und integriert. Daher wird im Folgenden der Begriff DXRAM sowohl verwendet, um die einzelnen vorgestellten Konzepte besser zu referenzieren als auch das Gesamtsystem (inklusive der in dieser Arbeit vorgestellten Konzepte) zu bezeichnen.
Die vorgestellten Messungen wurden größtenteils in [59, 68, 67, 69] publiziert.

5.1 Testumgebungen

Alle Evaluierungen wurden in einer von zwei Testunmgebungen durchgeführt. Dabei wird unterschieden zwischen lokalen Messungen, die auf einem einzelnen Knoten durchgeführt werden und verteilten Messungen, die mehreren Knoten in einer verteilten Umgebung auf einem privaten Cluster umfassen.

Lokale Testumgebung
Als lokale Testumgebung kommt ein einzelner Knoten mit einer Intel® Core™ i5-2400 CPU mit vier Kernen zu je 3.1 GHz und 32 GB RAM Arbeitsspeicher (DDR3) zum Einsatz. Ubuntu 11.04 wird als Betriebssystem zusammen mit dem OpenJDK Java Runtime Environment (Version 7) zur Ausführung der Messungen verwendet.

Verteilte Testumgebung
Für die verteilten Messungen wird ein kleiner privater Cluster mit 20 homogenen Knoten und einer Gigabit Ethernet Vernetzung eingesetzt. Jeder Knoten verfügt über eine Intel® Xeon® CPU E3-1220 V2 mit vier Kernen zu je 3.1 GHz, 16 GB RAM (DDR3) Arbeitsspeicher und eine 1 TB Festplatte (HUA722010CLA330). Die Knoten nutzen Debian 7.6 und das OpenJDK Java Runtime Environment (Version 7) für die Messungen.

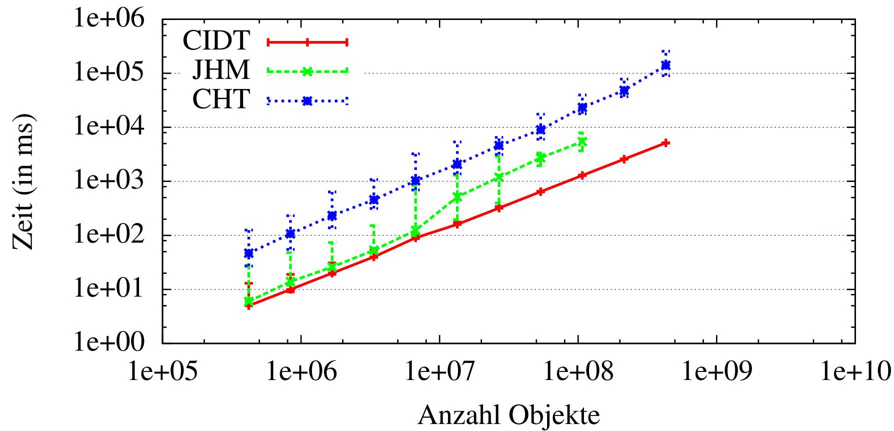
5.2 Evaluation der lokalen Metadaten-Verwaltung

Nachfolgend werden die Komponenten der lokalen Metadaten-Verwaltung evaluiert, beginnend mit einem Vergleich der vorgestellten CID-Tabellen mit zwei Hashtabellen. Für die Speicherverwaltung werden im Anschluss mehrere Speicherallokatoren mit dem vorgestellten Konzept verglichen und die Defragmentierung in einem Worst-Case-Szenario betrachtet. Zuletzt wird die Adressübersetzung zusammen mit der Speicherverwaltung evaluiert und mit RAMCloud gegenübergestellt. Da RAMCloud ähnliche Ziele wie DXRAM verfolgt, ist ein Vergleich sinnvoll.
Der Vergleich mit RAMCloud wurde in der verteilten Testumgebung durchgeführt, bei allen anderen Experimenten wurde die lokale Testumgebung verwendet (siehe Abschnitt 5.1).

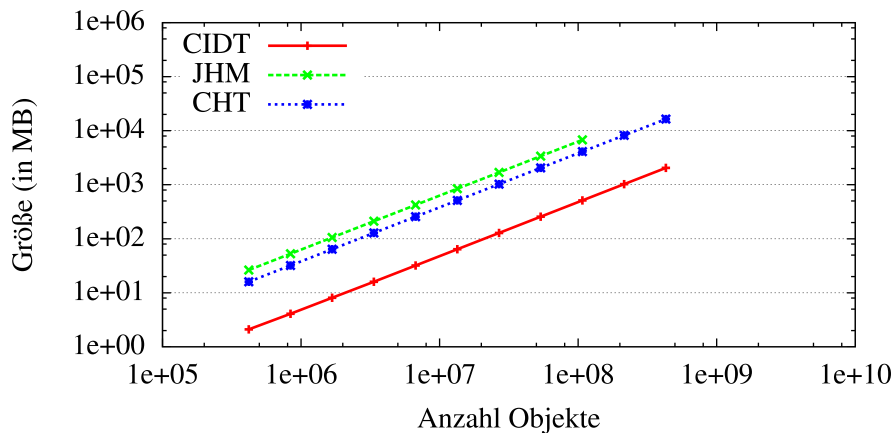
5.2.1 Vergleich von Hash- und CID-Tabellen

In der ersten Evaluierung wurden die in Abschnitt 3.2 vorgestellten CID-Tabellen mit zwei Implementierungen einer Hashtabelle verglichen. Als erste Implementierung kommt Java's HashMap[1] (JHM) aus dem OpenJDK zum Einsatz. Die Java HashMap verwendet Java Objekte für Schlüssel und Werte und speichert zugehörige Tupel in einem dynamischen Array. Jedes Array-Element ist ein so genannter Bucket und beinhaltet möglicherweise mehrere Schlüssel-Wert-Paare in einer verketteten Liste. Die zweiten Implementierung (CHT) verwendet primitive Long-Werte für Schlüssel und Werte und setzt Cuckoo-Hashing ein. Die Anzahl der dabei

[1] `http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html`



(a) Zeit für die Einfügeoperationen bei einem Füllgrad von 40%



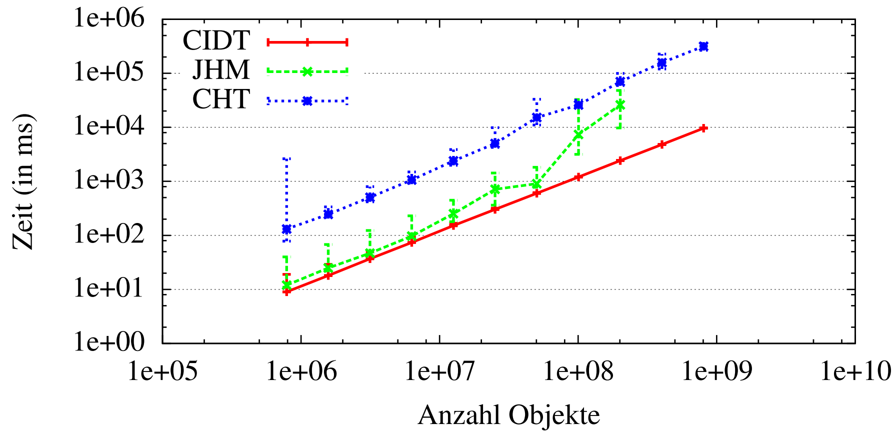
(b) Speicherverbrauch bei einem Füllgrad von 40%

Abbildung 5.1: Vergleich von Hash- und CID-Tabellen I. *Ausführungszeit und Speicherverbrauch für die Einfügeoperationen bei einem Füllgrad von 40%.*

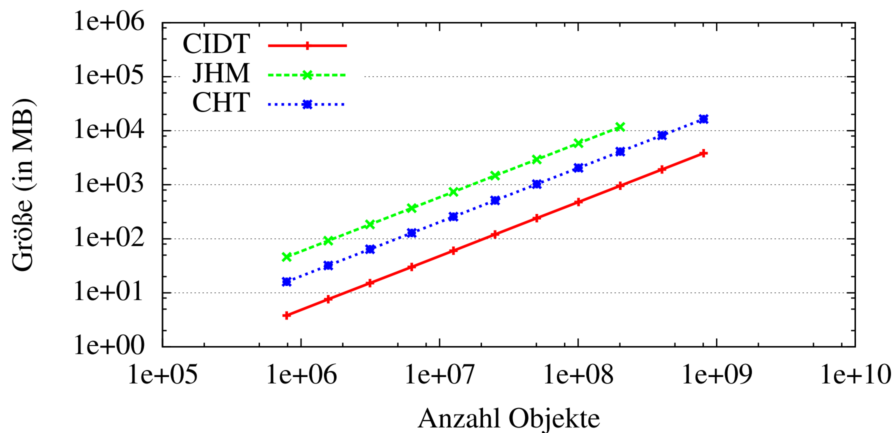
verwendeten Arrays und zugehörigen Hashfunktionen ist, wie in [98] empfohlen, konfigurierbar. Cuckoo-Hashing ist ein aktueller Hashing-Ansatz, der gute Performanz und einen hohen Füllgrad ermöglicht.

Alle Experimente wurden mehrfach (mindestens fünf Mal) mit einem anfangs festgelegten Füllgrad und darauf abgestimmten Anzahl an Einträgen durchgeführt. Die Anzahl der Einträge ist zugunsten der Hashtabellen auf den Füllgrad abgestimmt, so dass die beiden Hashtabellen voll ausgelastet sind. Das bedeutet, dass beim Hinzufügen eines weiteren Eintrags die zugrunde liegenden Arrays vergrößert werden müssen. Für die CHT werden nachfolgend immer nur die Ergebnisse der besten Konfiguration betrachtet.

Bei einem Füllgrad von 40% (siehe Abbildung 5.1) wächst die Zeit zum Einfügen linear mit der



***(a)** Zeit für die Einfügeoperationen bei einem Füllgrad von 75%*

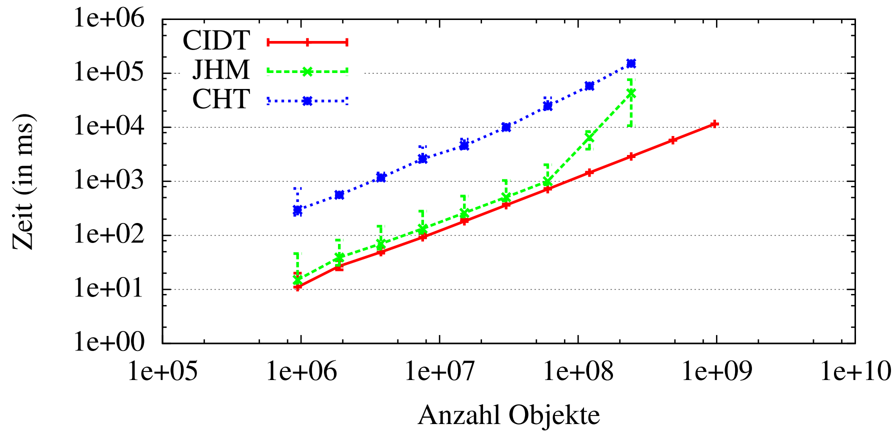


***(b)** Speicherverbrauch bei einem Füllgrad von 75%*

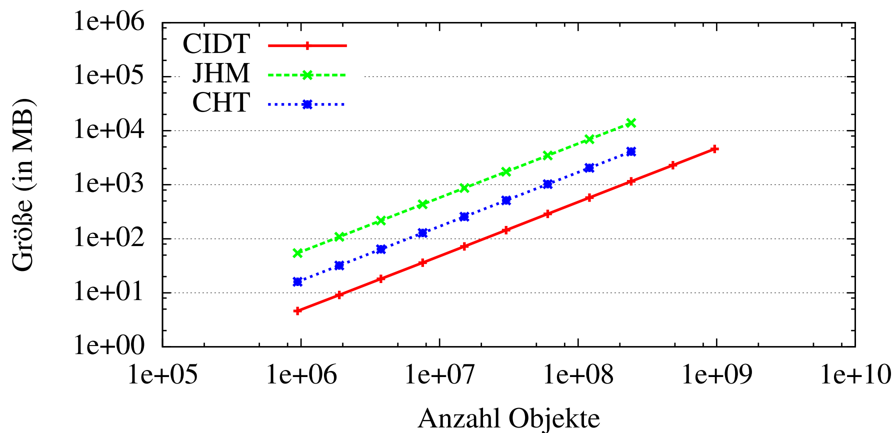
***Abbildung 5.2: Vergleich von Hash- und CID-Tabellen II.** Ausführungszeit und Speicherverbrauch für die Einfügeoperationen bei einem Füllgrad von 75%.*

Anzahl der Einträge für die CID-Tabellen und die Cuckoo-Hashtabelle. Bei der Java HashMap wächst die Zeit am Anfang und am Ende linear, in der Mitte ist allerdings ein Sprung zu erkennen, so dass die Java HashMap bei vielen Einträgen schlechter abschneidet. Bei beiden Hashtabellen lassen sich große Abweichungen nach unten, aber noch stärker nach oben feststellen. Die CID-Tabellen hingegen haben kaum Abweichungen und sind bei jedem Durchlauf nahezu gleich schnell. Insgesamt zeigt sich eindeutig, dass die CID-Tabellen sehr gut skalieren und ungefähr 4-mal schneller als die Java HashMap und 20-mal schneller als die Cuckoo-Hashtabelle sind. Beim Speicherverbrauch schneiden die CID-Tabellen ebenfalls am Besten ab und benötigt rund 92% weniger Speicher als die Java HashMap und 87% weniger Speicher als die Cuckoo-Hashtabelle.

Ein ähnliches Bild zeigt sich auch bei einem Füllgrad von 75% (siehe Abbildung 5.2). Alle drei



(a) Zeit für die Einfügeoperationen bei einem Füllgrad von 90%



(b) Speicherverbrauch bei einem Füllgrad von 90%

Abbildung 5.3: Vergleich von Hash- und CID-Tabellen III. *Ausführungszeit und Speicherverbrauch für die Einfügeoperationen bei einem Füllgrad von 90%.*

Datenstrukturen haben nahezu ein lineares Wachstum bei der Zeit zum Einfügen im Verhältnis zur Anzahl der Einträge. Bei der Java HashMap ist auch hier wieder einen Sprung zu verzeichnen, wodurch sich gerade bei vielen Einträgen die Zeit deutlich verschlechtert. Insgesamt sind die CID-Tabellen ungefähr 10-mal schneller als die Java HashMap und 30-mal schneller als die Cuckoo-Hashtabelle. Auch der Speicherverbrauch ist bei den CID-Tabellen weiterhin am geringsten. Im Vergleich zu den beiden Hashtabellen benötigen die CID-Tabellen ungefähr 92% weniger Speicher als die Java HashMap und 76% weniger Speicher als die Cuckoo-Hashtabelle.

Auch bei einem Füllgrad von 90% setzen sich die bisherigen Beobachtungen fort. Die Zeiten bei der Cuckoo-Hashtabelle verschlechtern sich leicht, bei der Java HashMap verändert sich das Wachstum allerdings zum Ende hin von linear zu exponentiell. Diese Veränderung

hängt voraussichtlich mit den verketteten Listen der Buckets zusammen. Wenn viele Einträge auf den gleichen Bucket gehasht werden, werden diese Einträge in einer verketteten Liste gespeichert, die nur in $\mathcal{O}(\log n)$ durchlaufen werden kann. Dabei steigt mit zunehmendem Füllgrad die Wahrscheinlichkeit, dass mehrere Einträge auf den gleichen Bucket gehasht und somit die verketteten Listen länger werden. Insgesamt sind bei einem Füllgrad von 90% die CID-Tabellen ungefähr 15-mal schneller als die Java HashMap und 52-mal schneller als die Cuckoo-Hashtabelle. Der Speicherverbrauch ist für die CID-Tabellen weiterhin sehr niedrig und erlaubt eine Reduzierung um 92% im Vergleich zur Java HashMap und 72% in Vergleich zur Cuckoo-Hashtabelle.

Zusammenfassend zeigt sich, dass die CID-Tabellen gut skalieren, einen schnellen Zugriff ermöglichen und extrem speichereffizient sind. Bei den Hashtabellen zeigt sich hingegen eine umgekehrte Abhängigkeit von Zeit und Füllgrad. Ein geringerer Füllgrad bedeutet einen schnelleren Zugriff, gleichzeitig jedoch auch einen höheren Speicherverbrauch. Soll der Speicherverbrauch gesenkt werden, muss der Füllgrad erhöht werden, wodurch sich allerdings gleichzeitig die Zugriffszeit verschlechtert.

5.2.2 Speicherallokation

Zunächst soll der Speicherverbrauch des in Abschnitt 3.3 vorgestellten Speicherallokators mit dem Speicherverbrauch verschiedener Speicherallokatoren verglichen werden. Zu diesem Zweck wurden mit jedem Allokator insgesamt 10 GB Anwendungsdaten erzeugt, die aus Objekten gleicher Größe bestehen. Die Größe betrug im ersten Test 16 Byte und wurde bei jedem weiteren Test um 4 Byte erhöht (bis zu einer maximalen Größe von 64 Byte). Beim letzten Durchlauf wurden Objekte unterschiedlicher Größe erzeugt, wobei die Objektgröße gleichverteilt zwischen 16 und 64 Byte lag. In jedem Durchlauf wurden dementsprechend mehrere Millionen Objekte erzeugt, so wie es bei den Zielanwendungen zu erwarten ist.

Bevor die eigentlichen Messergebnisse diskutiert werden, erfolgt eine kurze Vorstellung der verglichenen Speicherallokatoren.

Java Java verwendet sowohl Heap- als auch Non-Heap-Speicher für Allokationen [170]. Im Non-Heap befinden sich Metadaten, Java eigene Methoden, Thread-Stacks und Verwaltungsstrukturen, während im Heap alle Java Objekte gespeichert sind [171]. Der Heap wird ferner in zwei Bereiche die Young Generation und die Old Generation aufgeteilt. Neue Objekte werden im Regelfall in der Young Generation abgelegt und verweilen dort, bis sie freigegeben oder durch den Garbage-Collector in die Old-Generation umkopiert werden. Große Objekte (größer als 2-128 KB; abhängig von der Konfiguration) werden direkt in der Old Generation erzeugt. Für jeden Thread existiert eine Thread Local Area

(TLA) mit freiem Speicher, der exklusiv dem zugehörigen Thread zur Verfügung steht und den Synchronisierungsaufwand reduziert. Wenn eine TLA voll ist, wird dem Thread eine neue TLA zugewiesen. Speicher wird regelmäßig durch eine Garbage-Collection (GC) freigegeben, die normalerweise einen Mark-and-Sweep-Algorithmus verwendet. Dabei wird zwischen Minor- und Major-GC unterschieden. Eine Minor-GC betrifft nur die Young-Generation, wobei nicht mehr erreichbare Objekte freigegeben und überlebende Objekte in die Old Generation verschoben werden. Im Gegensatz dazu wird bei einer Major-GC der gesamte Heap betrachtet, was besonders zeitaufwändig ist. Um Fragmentierung vorzubeugen, vor allem in der Young Generation mit sehr vielen Allokationen, wird bei jeder GC ein Teil des Speichers kompaktifiziert. Weitere Details zur Speicherverwaltung und Garbage-Collection in Java können in [170, 171, 172] nachgelesen werden.

glibc Die GNU C Library (glibc) nutzt ptmalloc als universellen Speicherallokator [173]. Ptmalloc basiert auf dem in [145] beschrieben Speicherallokator von Doug Lea und erweitert diesen um eine bessere Multithreading-Unterstützung. Dazu wird jedem Thread eine eigene Arena zugewiesen, dessen Speicher alleine durch den zugehörigen Thread verwaltet wird und somit eine Synchronisierung entfallen kann. Neben einen Main-Heap existieren für jede Arena ein separater Heap und gegebenenfalls mehrere Sub-Heaps, die in verketteten Listen verwaltet werden. Speicher für die Heaps werden mittels mmap (Speicher > 32 Seiten) und sbrk (Speicher ≤ 32 Seiten) vom System angefordert. Die genau Implementierung des Speicherallokators ist relativ komplex und sprengt den Rahmen dieser Arbeit, weitergehende Informationen können jedoch [174, 175, 176, 177] entnommen werden.

hoard Hoard ist ein Speicherallokator für verschiedene Betriebsysteme und ist für die effiziente Verwendung von Multithreading-Anwendungen auf Multiprozessorsystemen entwickelt worden [112]. Hoard arbeitet speichereffizient, ist skalierbar und vermeidet False-Sharing zwischen den unterschiedlichen Prozessoren. Allokationsklassen festgelegter Größe erlauben eine niedrige Fragmentierung und ein separater Heap für jeden Prozessor minimiert die Synchronisierungskosten. Neuer Speicher wird in Form von Superblocks fester Größe (ein Vielfaches einer Seitengröße) von einem globalen Heap oder direkt vom System angefordert und einem Prozessor-Heap zugeordnet. Die Superblöcke jedes Heaps sind in so genannten „Fullness Groups“ in LIFO-Reihenfolge doppelt-verkettet und jeder Superblock enthält wiederum eine Liste freier Blöcke in LIFO-Reihenfolge. Wenn ein Prozessor-Heap nur zu einem festgelegten Bruchteil (Threshold) belegt ist (beispielsweise nur zu einem Viertel), wird ein Superblock der ebenfalls nur maximal zu diesem Bruchteil belegt ist, vom Prozessor-Heap in den globalen Heap verschoben und

steht damit auch anderen Prozessen zur Verfügung. Große Allokationen (größer als die Hälfte eines Superblocks) werden direkt mit mmap und munmap alloziert und freigegeben.

jemalloc Jemalloc ist eine malloc(3)-Implementierung, die zuerst in FreeBSD als libc-Allokator eingesetzt [114] und für skalierbare konkurrierende Allokationen in Multithreading-Anwendungen entwickelt wurde. Der Speicher wird vom Kernel in Vielfachen einer „Chunk“-Größe angefordert und jede Allokation auf die nächste Größenklasse gerundet. Die Metadaten der Allokationen werden dabei in einem Rot-Schwarz-Baum gespeichert. Jemalloc ist weit verbreitet und kommt beispielsweise in einer modifizierte Version bei Facebook für verschiedene Server-Anwendungen zum Einsatz [178].

tcmalloc Tcmalloc wurde von Google als schneller Speicherallokator mit sehr geringem Synchronisierungsaufwand entworfen [113]. Er ist besonders speichereffizient für kleine Objekte ($\leq$ 32 KB) aufgrund von bis zu 170 Allokationsklassen. Neben einem zentralen Heap verfügt jeder Thread über einen eigenen Thread-Cache mit einer verketteten Liste freier Objekte pro Allokationsklasse. Bei der Allokation schaut der Thread in die passende Liste und verwendet den ersten Eintrag. Ist die Liste leer wird der zentrale Heap geprüft und falls das ebenfalls fehlschlägt ein neuer Speicherbereich (ein Vielfaches einer Seitengröße) vom System angefordert und in gleich große Stücke der benötigten Größe unterteilt und in die passende Liste des eigenen Thread-Caches eingefügt. Große Objekte werden auf die nächste Seitengröße gerundet und direkt über das System alloziert. Die Speicherbereiche für kleine und große Objekte werden zentral in einem Patricia-Trie verwaltet [179]. Freigegebener Speicher wird bei Bedarf mittels einer periodischen Garbage-Collection von einzelnen Thread-Caches in den zentralen Heap überführt.

boehmgc Boehmgc ist ein Garbage-Collector für C und C++ Programme [144]. Beim Einsatz von boehmgc wird der klassische malloc-Aufruf ersetzt und ein explizites Aufrufen von free kann entfallen. Für Allokationen stehen mehrere Allokationsklassen zur Verfügung, für die jeweils eine eigene Freispeicherliste verwaltet wird. Eine Seite ist immer in gleich große Stücke unterteilt und damit direkt einer Allokationsklasse zugeordnet. Der Garbage-Collector verwendet einen Mark-and-Sweep-Algorithmus, um nicht mehr benötigte Objekte zu erkennen und den entsprechenden Speicher freizugeben. Programme können zu diesem Zweck boehmgc Informationen zu Objekten und Pointer mitteilen (müssen dies aber nicht) und können Finalization-Routinen registrieren.

Abbildung 5.4 zeigt die Ergebnisse der insgesamt 14 Durchläufe. Der speziell für die Verwaltung sehr vieler kleiner Objekte entworfene Speicherallokator schneidet in allen Durchläufen sehr gut ab. Der Speichermehrverbrauch für Metadaten und Verwaltungsstrukturen liegt dabei zwischen 3-12%. Der Java-eigene Speicherallokator schneidet bei fast allen Durchläufen

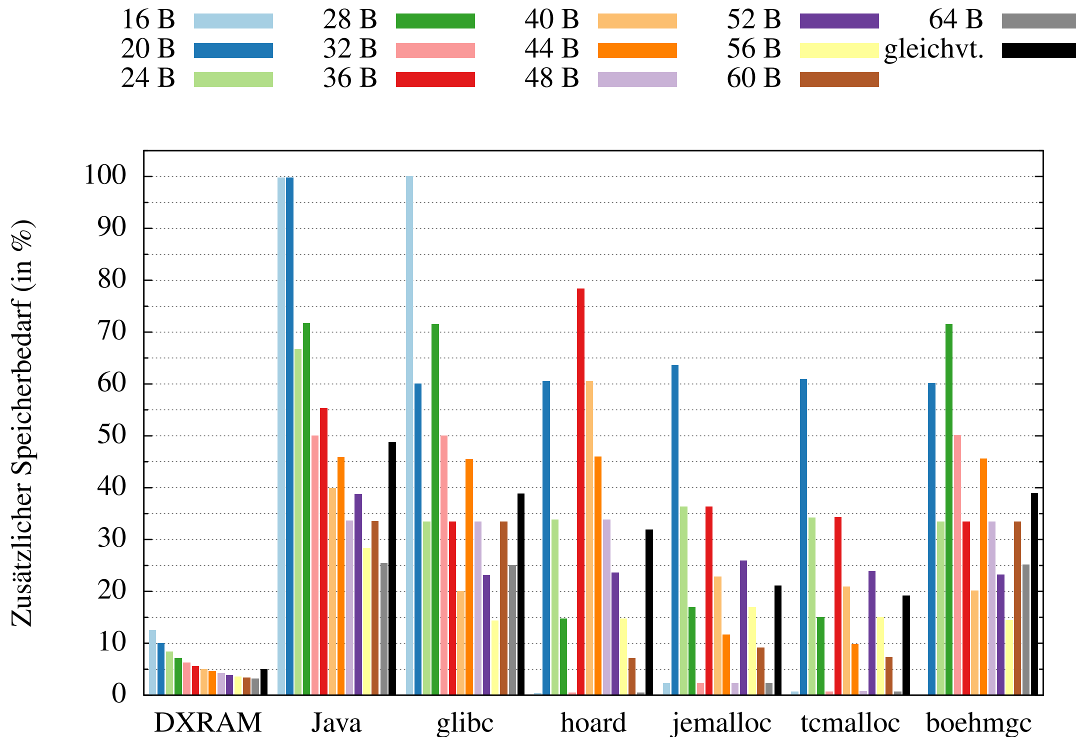


***Abbildung 5.4: Speicheroverhead verschiedener Allokatoren.** Für die Evaluierung wurden 10 GB Anwendungsdaten, bestehend aus Objekten gleicher Größe beziehungsweise gleichverteilter Größe, erzeugt.*

am schlechtesten ab und benötigt bis zu 100% mehr Speicher. Lediglich bei 36- und 40-Byte-Objekten schneidet hoard noch schlechter ab. Der Speicherallokator, der in der GNU C Bibliothek (glibc) eingesetzt wird, benötigt ebenso im schlechtesten Fall 100% mehr Speicher und ist auch bei den anderen Durchläufen kaum besser als der Java-Allokator. Hoard ist besonders für die Objektgrößen 16, 32 und 64 Byte extrem gut und benötigt nahezu keinen zusätzlichen Speicher. Bei Objektgrößen von 32 und 36 Byte kommt hoard allerdings sehr schlecht weg, mit einem zusätzlichen Speicherbedarf von bis zu 78%. Jemalloc und tcmalloc erzielen nahezu gleiche Ergebnisse, die besonders gut bei Objektgrößen von 16, 32, 48 und 64 Byte sind. Bei 20-Byte großen Objekten werden die schlechtesten Ergebnisse, mit ungefähr 60% zusätzlichem Speicherbedarf, erzielt. Tcmalloc schneidet durchweg etwas besser als jemalloc ab. Boehmgc erzielt ähnliche Ergebnisse wie der Allokator aus der glibc vermeidet allerdings den schlechtesten Fall bei 16-Byte großen Objekten. Der schlechteste Fall für boehmgc sind Objekte mit 28 Byte, wobei ungefähr 70% zusätzlicher Speicher benötigt wird.

Am aussagekräftigsten ist der Durchlauf mit gleichverteilter Objektgröße, da dieser Durchlauf am realistischsten ist. Bei diesem Durchlauf kommen sehr viele unterschiedliche Objektgrößen vor, was eine Optimierung erschwert. Der vorgestellte Speicherallokator (DXRAM) schneidet

Abbildung 5.5: Worst-Case-Szenario für die Defragmentierung. *Nach der Allokation wird jedes zweite Objekt gelöscht.*

hierbei mit Abstand am besten ab und benötigt lediglich 5% zusätzlichen Speicher. Alle anderen Allokatoren erfordern mindestens 20% mehr Speicherplatz, wobei der Java-Allokator sogar fast 50% benötigt und damit am schlechtesten abschneidet.

Zusammenfassend kann festgehalten werden, dass der speziell entwickelte Speicherallokator sehr speichereffizient arbeitet und den Speicherverbrauch im Vergleich zu klassischen Speicherallokatoren erheblich senken kann. Bei 10 GB Anwendungsdaten beträgt die Reduzierung im Vergleich zu den anderen Speicherallokatoren mindestens 3 GB beziehungsweise 75%.

5.2.3 Defragmentierung

Nachfolgend wurde die Defragmentierung (siehe Abschnitt 3.3) in einem Worst-Case-Szenario evaluiert. Dafür wurden 10 GB Daten, aufgeteilt in 50-Byte-Objekte, alloziert und anschließend jedes zweite Objekt gelöscht (siehe Abbildung 5.5). Die Speicherverwaltung stellt insgesamt 11,5 GB Speicher zur Verfügung, um die Objekte, die Metadaten (ca. 1,4 GB) und einen kleinen Puffer von ungefähr 100 MB aufzunehmen. Die Löschung jedes zweiten Objektes führt zu einem sehr stark fragmentiertem Speicher mit sehr vielen kleinen freien Blöcken. Nach dem Löschen wird eine vollständige Defragmentierung ausgeführt, die alle CID-Tabellen auf Stufe 0 umfasst. Während der Defragmentierung werden die Tabellen auf Stufe 0 gegebenenfalls verschoben, alle Tabellen auf höheren Stufen bleiben jedoch an ihrem ursprünglichen Speicherplatz bestehen. Offensichtlich handelt es sich um ein Worst-Case-Szenario, mit dem sich die Grenzen der Defragmentierung gut überprüfen lassen.
Tabelle 5.1 enthält die Anzahl der freien Blöcke unterschiedlicher Größe in den vier Phasen der Evaluierung: nach der Initialisierung, nach der Allokation der Objekte, nach dem Löschen jedes zweiten Objektes und nach der Defragmentierung. Nach der Initialisierung existieren elf

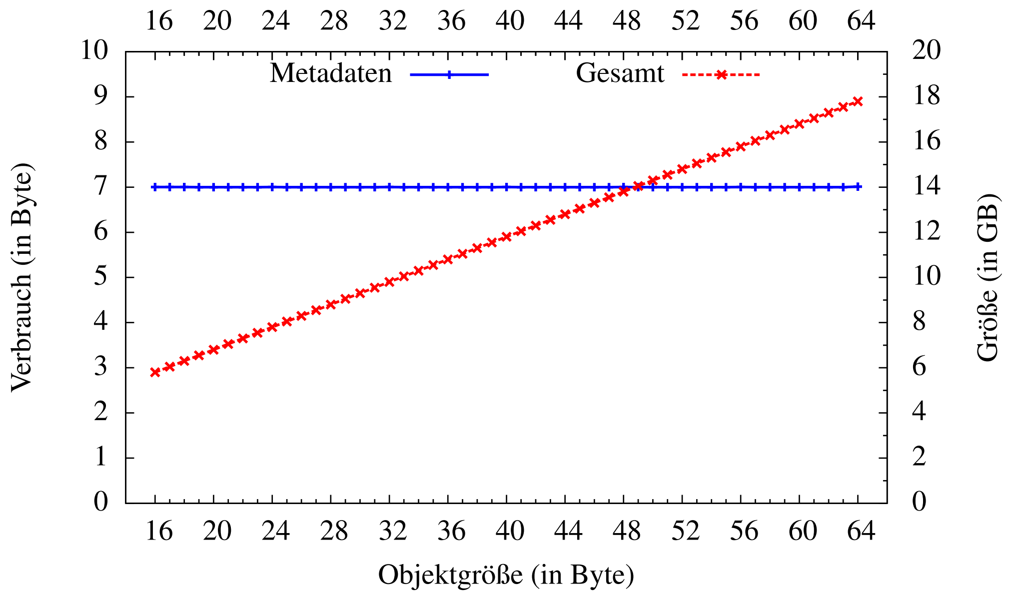
Größe	Initialisierung	Allokation	Löschvorgang	Defragmentierung
12 B – 24 B	0	1	0	0
48 B – 64 B	0	0	**107.374.176**	0
64 B – 128 B	0	0	1	0
16 KB – 32 KB	0	0	0	2
32 KB – 64 KB	0	0	0	2
64 KB – 128 KB	0	0	0	6
64 MB – 128 MB	0	1	1	0
256 MB – 512 MB	0	0	0	**1**
512 MB – 1 GB	1	0	0	**1**
1 GB	11	0	0	**4**

Tabelle 5.1: Defragmentierung von freien Blöcken. *Die Zahlen geben jeweils die Anzahl der freien Blöcke der jeweiligen Größe nach der Initialisierung, nach der Allokation, nach dem Löschvorgang und nach der Defragmentierung an.*

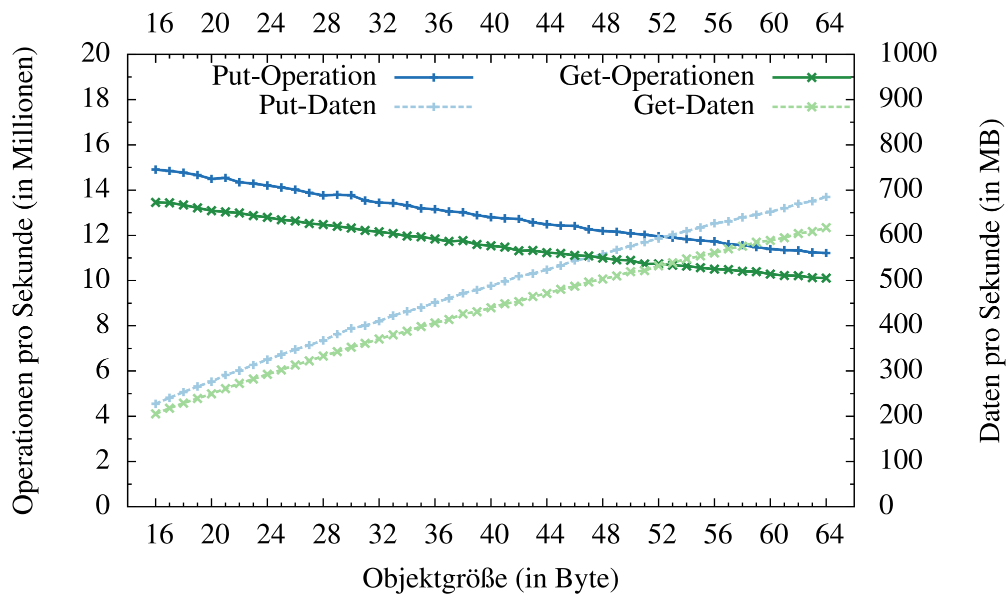
Segmente mit einem GB und ein Segment mit einem halben GB, entsprechend den 11,5 GB des zur Verfügung stehenden Speichers. Bei der Allokation werden insgesamt 214.748.354 belegte Blöcke erzeugt, die den initialen Speicherplatz nahezu komplett aufbrauchen. Lediglich ein kleiner Block mit 12-24 Byte und ein Block mit 64-128 MB (der kalkulierte Puffer) stehen noch zur Verfügung. Durch das anschließende Löschen jedes zweiten Objektes entstehen mehr als 100 Millionen Blöcke zwischen 48 und 64 Byte. Ferner wird der Block mit 12-24 Byte mit dem Block eines freigegebenen Objektes verschmolzen und es entsteht ein Block mit 64-128 Byte. Bei der Defragmentierung werden alle gespeicherten Objekte untersucht und größtenteils verschoben. Durch die Verschiebung können viele kleine freie Speicherblöcke zu wesentlich größeren Speicherblöcken verschmolzen werden. Als Ergebnis existieren nach der Defragmentierung zwei Blöcke mit 16-32 KB, zwei Blöcke mit 32-64 KB, sechs Blöcke mit 64-128 KB, sowie jeweils ein Block mit 256-512 MB und 512 MB bis 1 GB. Darüber hinaus konnten vier Segmente sogar komplett wieder freigegeben werden. Der kleinste so gewonnene Block kann anschließend für 200 bis 1.000 weitere Allokationen (16-64 Byte) verwendet werden.
Die Ergebnisse zeigen, dass die Defragmentierung sehr effizient arbeitet und selbst mit dem gegebenen Worst-Case-Szenario problemlos umgehen kann. Für weniger ungünstige Allokationsmuster ist die Defragmentierung noch deutlich effizienter.

5.2.4 Adressübersetzung und Speicherverwaltung

Bei der folgenden Evaluierung wurden der Durchsatz und der Speicherverbrauch der Metadaten pro Objekt für die Kombination von Adressübersetzung und Speicherverwaltung gemessen.



***(a)** Metadaten pro Objekt*



***(b)** Durchsatz (Operationen pro Sekunde)*

***Abbildung 5.6: Evaluierung der Adressübersetzung und Speicherverwaltung.** Der Speicherverbrauch für die Metadaten und der Durchsatz wurde mit jeweils 2^{28} Objekten ermittelt.*

Zu diesem Zweck wurden jeweils 2^{28} Objekte gleicher Größe erzeugt und in die Adressübersetzung und Speicherverwaltung eingetragen. Anschließend wurden alle Objekte abgefragt und aktualisiert. Dabei wurde sowohl der Speicherverbrauch als auch die Anzahl der Get- und Put-

Operationen pro Sekunde gemessen.
Abbildung 5.6a zeigt, dass für die relevanten Objektgrößen von 16-64 Byte der Speicherverbrauch für die Metadaten pro Objekt konstant 7 Byte beträgt, wobei 5 Byte auf die Adressübersetzung und 2 Byte auf die Speicherverwaltung entfallen. Der konstante Speicherbedarf der Metadaten pro Objekt zeigt sich ebenfalls im linearen Wachstum der gesamten Speichergröße von 5,8 auf 17,8 GB. In Abhängigkeit von der Objektgröße entspricht dies einem Mehraufwand von 10 bis maximal 44% und bei gleichverteilten Objektgrößen von 17,5%.
In Abbildung 5.6b wird der Durchsatz pro Sekunde für Get- und Put-Operationen dargestellt. Pro Sekunde können zwischen 10 und 13,5 Millionen Get-Operation und zwischen 11 und 15 Millionen Put-Operationen durchgeführt werden. Mit zunehmender Objektgröße müssen pro Operation mehr Daten gelesen oder geschrieben werden, so dass die Anzahl der Operation zwar leicht absinkt, der Datendurchsatz jedoch aufgrund der Objektgröße zunimmt. Bei den Get-Operation steigt der Datendurchsatz von 200 auf 650 MB pro Sekunde und bei Put-Operationen von 220 auf 700 MB pro Sekunde.

5.2.5 Vergleich mit RAMCloud

Als nächstes wurden Adressübersetzung und Speicherverwaltung im verteilten Fall evaluiert und mit RAMCloud verglichen. RAMCloud hat eine ähnliche Zielsetzung wie DXRAM und eignet sich daher gut für einen Vergleich. Sowohl für DXRAM als auch RAMCloud umfasst

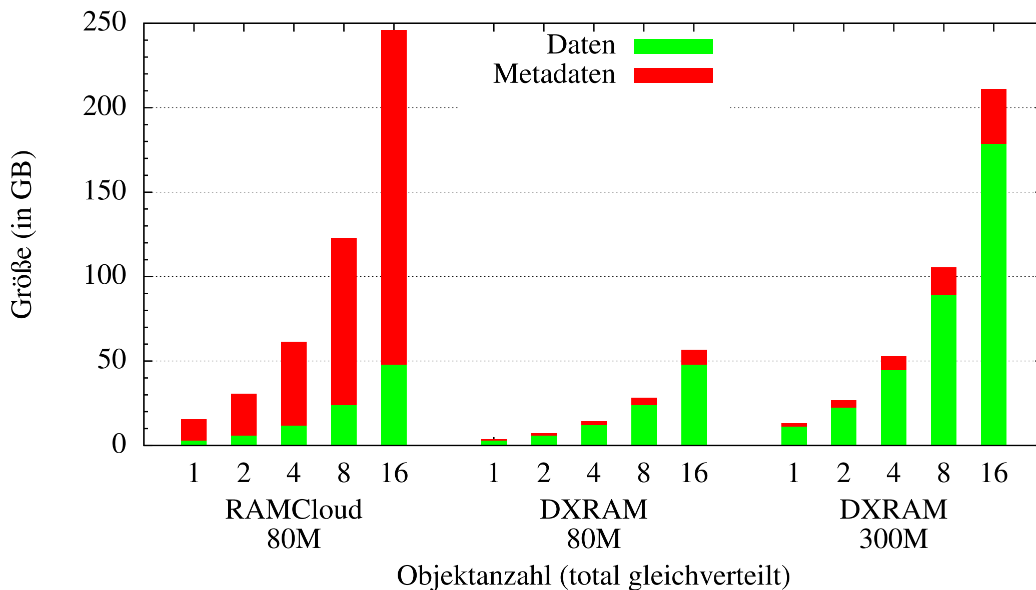


Abbildung 5.7: Speicherverbrauch im verteilten Fall. *Für die Evaluierung wurden RAMCloud und DXRAM mit jeweils 1, 2, 4, 8 und 16 Speicherknoten gestartet.*

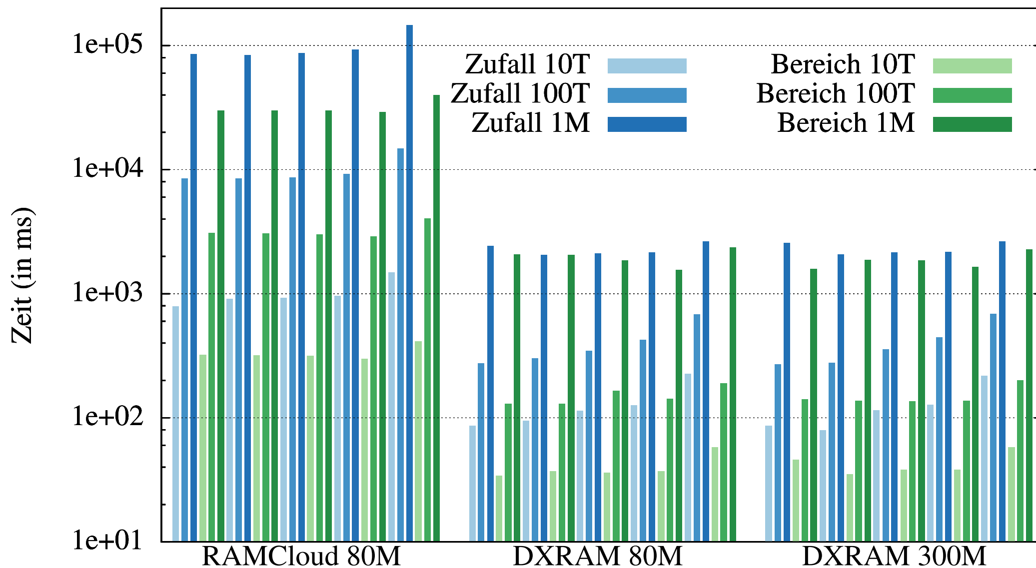


Abbildung 5.8: Performanz im verteilten Fall. *Für die Evaluierung wurden RAMCloud und DXRAM mit jeweils 1, 2, 4, 8 und 16 Speicherknoten gestartet und sowohl Mehrfach- als auch Bereichsabfragen durchgeführt.*

die Messumgebung einen Koordinator beziehungsweise Super-Peer, einen Klient und bis zu 16 Speicherknoten, mit jeweils 80 Millionen Objekten gleichverteilter Größe (16-64 Byte), was insgesamt ungefähr 3 GB Daten pro Knoten entspricht. Für die Messungen wurde in DXRAM und in RAMCloud die Replikation und der Backup-Dienst abgeschaltet, um die Messungen nicht zu beeinträchtigen.

Abbildung 5.7 zeigt den Gesamtspeicherverbrauch der Speicherknoten beider Systeme. Der Speicherverbrauch ist dabei aufgeteilt in die eigentlichen Daten und die notwendigen Metadaten. Beide Systeme skalieren gleich gut, was den gesamten Speicherverbrauch betrifft. Eine Verdopplung der Knoten führt zu einer Verdopplung des Speicherverbrauchs. Betrachtet man den Speicherbedarf für die Metadaten unterscheiden sich beide Systeme erheblich. Während in DXRAM der Speicherbedarf durch die Metadaten zusätzliche 20% beträgt, benötigen die Metadaten in RAMCloud über 400% zusätzlichen Speicher. Bei 16 Knoten führt dies dazu, dass RAMCloud insgesamt fast 250 GB Speicher benötigt und DXRAM einen Speicherbedarf von weniger als 60 GB hat. Zum Vergleich, wurde DXRAM zusätzlich mit 300 Millionen Objekten (ungefähr 11 GB) pro Speicherknoten aufgesetzt. Auch in diesem Fall benötigen die Metadaten nur ungefähr 20% zusätzlichen Speicher, so dass der gesamte Speicherbedarf bei 16 Knoten bei lediglich 210 GB liegt und damit immer noch weniger Speicher verbraucht, als RAMCloud mit 80 Millionen Objekten pro Knoten.

Die Performanz von Lesezugriffen wird in Abbildung 5.8 dargestellt. Für die Messungen wur-

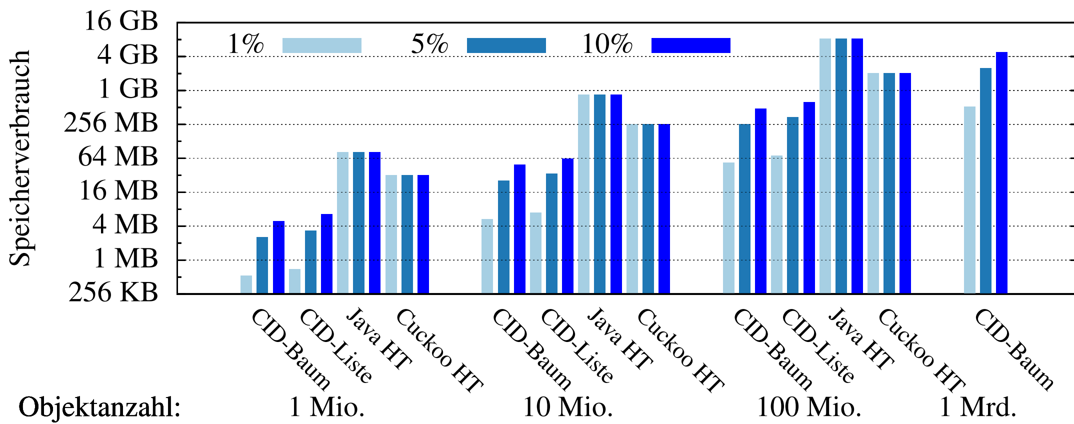
den RAMCloud und DXRAM mit bis zu 16 Speicherknoten mit jeweils 80 Millionen Objekten aufgesetzt. Die anschließenden Lesezugriffe wurden gebündelt und umfassen mehrere tausend Schlüssel, die entweder zufällig oder in sequentieller Reihenfolge gewählt wurden. Während zufällige Zugriffe einem allgemeinen Zugriffsmuster entsprechen, sind die Zugriffe in sequentieller Reihenfolge charakteristisch für lokalitäts-basierte Anwendungen. DXRAM bündelt dabei problemlos Millionen von Zugriffen zu einer einzelnen Netzwerkanfrage, wohingegen entsprechende Anfragen in RAMCloud auf 1.000 Zugriffe begrenzt sind. Die Ergebnisse der Messungen zeigen, dass alle Lesezugriffe in DXRAM ungefähr 10 bis 15-mal schneller durchgeführt werden können als in RAMCloud. Während bei einer kleinen Anzahl gebündelter Lesezugriffe ein merklicher Unterschied zwischen zufällig und sequentiell gewählten Schlüsseln existiert, gleicht sich der Unterschied bei größeren Anzahlen weiter an. Auch bei diesen Messungen wurde zusätzlich DXRAM mit 300 Millionen Objekten pro Speicherknoten aufgesetzt, wobei sich dieselben Messwerte ergeben. Dies zeigt, dass die Ergebnisse unabhängig von der Anzahl der gespeicherten Objekte sind und DXRAM gut skaliert.

5.3 Evaluation der globalen Metadaten-Verwaltung

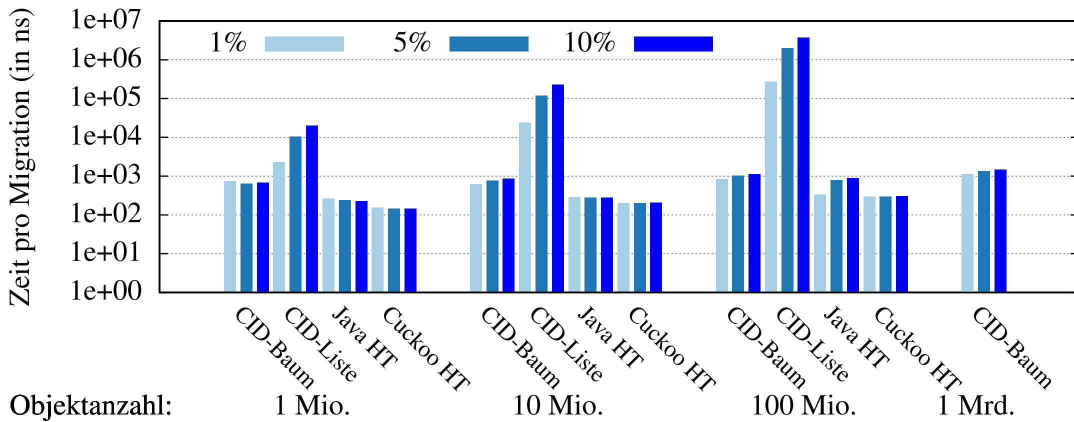
In diesem Abschnitt werden der vorgestellte CID-Baum zur Verwaltung von CID-Bereichen und das Caching von Objektsuchen zur Entlastung des Super-Peer-Overlays evaluiert. Für die Evaluierung des CID-Baumes wurde die lokalen Testumgebung verwendet und die Evaluierung des Caching in der verteilten Testumgebung durchgeführt (siehe Abschnitt 5.1).

5.3.1 CID-Baum

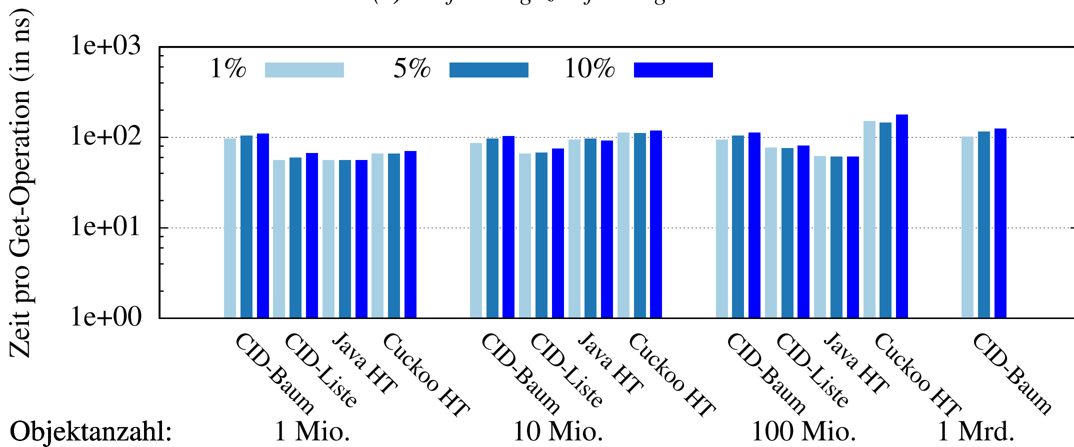
Bei der nachfolgenden Evaluierung wird der vorgestellte CID-Baum mit der CID-Liste, Java's HashMap und einer Cuckoo-Hashtabelle für primitive Datentypen verglichen. Jede dieser Datenstrukturen wird mit einer Millionen, zehn Millionen, hundert Millionen und einer Milliarden Metadaten-Einträgen befüllt und anschließend zufällig 1, 5 und 10% der Einträge migriert. Dabei wird der Speicherverbrauch und die Zeit pro Migration und Get-Operation (Objektsuche) gemessen. Die Messungen wurden für die CID-Liste 10-mal (auf Grund der sehr langen Laufzeit) und für alle anderen Datenstrukturen 100-mal durchgeführt. Die Messergebnisse waren wegen der hohen Anzahl an Operationen sehr dicht beieinander, so dass nur geringfügige Abweichungen feststellbar waren.
In Abbildung 5.9 werden die Durchschnittswerte aller Messungen dargestellt. Der Speicherverbrauch von CID-Baum und CID-Liste (siehe Abbildung 5.9a) liegt in allen Messungen nahezu gleich auf, wobei der CID-Baum immer etwas besser abschneidet. Im Gegensatz zu den beiden



***(a)** Speicherverbrauch*



***(b)** Ausführungszeit für Migrationen*



***(c)** Ausführungszeit für Get-Operationen*

***Abbildung 5.9: Evaluierung der Datenstrukturen.** Gemessen wurde der Speicherverbrauch und die Ausführungszeiten von Migrationen und Get-Operationen.*

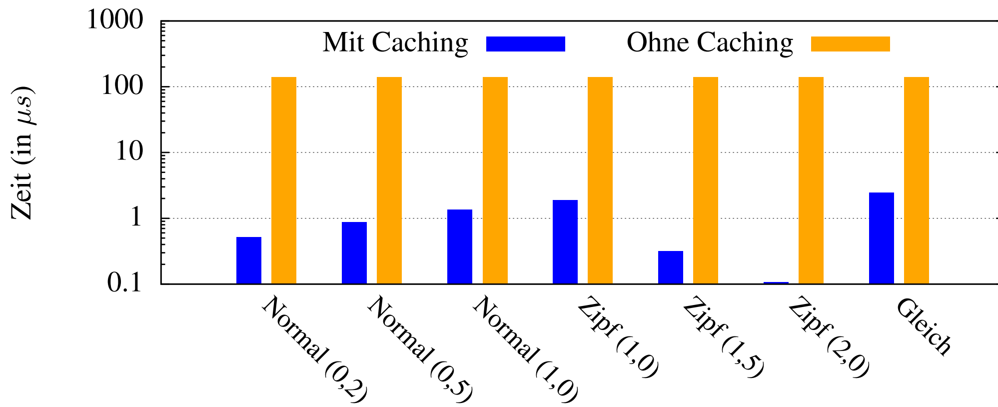
Hashtabellen nimmt der Speicherverbrauch von CID-Baum und -Liste mit mehr Migrationen zu, da zusätzliche CID-Bereiche gespeichert werden müssen. Bei den Hashtabellen führen Migrationen lediglich zu einer Anpassung der Einträge, so dass der Speicherbedarf gleich bleibt unabhängig von der Anzahl an Migrationen. Bei einer Migrationsrate von 10% verbrauchen CID-Baum und -Liste ungefähr 75-90% weniger Speicher und bei einer Migrationsrate von 1% sinkt der Speicherbedarf sogar um bis zu 99%. Die Messungen für eine Milliarden Objekte konnten auf dem Testsystem nur für den CID-Baum durchgeführt werden, da der Speicherbedarf und die Laufzeiten für die anderen Datenstrukturen zu hoch waren.
Bei den in Abbildung 5.9b gezeigten Migrationszeiten erzielen die beiden Hashtabellen die besten Ergebnisse. Sowohl die Migrationsrate als auch die Anzahl der Einträge beeinträchtigen die Ausführungszeit einer Migration dabei nur unwesentlich. Eine Migration im CID-Baum ist ungefähr fünfmal langsamer, als bei den Hashtabellen, benötigt bei zunehmender Migrationsrate allerdings nur geringfügig mehr Zeit. Die Anzahl der Einträge beeinflusst ebenfalls die Ausführungszeit, jedoch sind insgesamt die Ausführungszeiten über alle Messungen hinweg fast konstant. Die CID-Liste hat mit Abstand die schlechtesten Messergebnisse bei den Ausführungszeiten, die linear mit der Migrationsrate und der Eintragsanzahl ansteigen. Bei 100 Million Einträgen und einer Migrationsrate von 10% ist die CID-Liste bereits 1.000-mal langsamer als alle anderen Datenstrukturen.
Abbildung 5.9c zeigt schließlich die Ausführungszeiten für einzelne Get-Operationen. Die Messergebnisse aller vier Datenstrukturen liegen nur geringfügig auseinander. Bei einer Millionen Einträgen ist der CID-Baum am langsamsten, bei 10 und 100 Millionen Einträgen ist die Cuckoo-Hashtabelle jedoch noch langsamer. Die Java Hashtabelle ist in allen Messungen am schnellsten.
Insgesamt ist der CID-Baum am besten für die große Menge an Metadaten geeignet und ermöglicht eine hohe Reduzierung des Speicherverbrauchs für einen geringfügig langsameren Zugriff.

5.3.2 Caching von CID-Bereichen

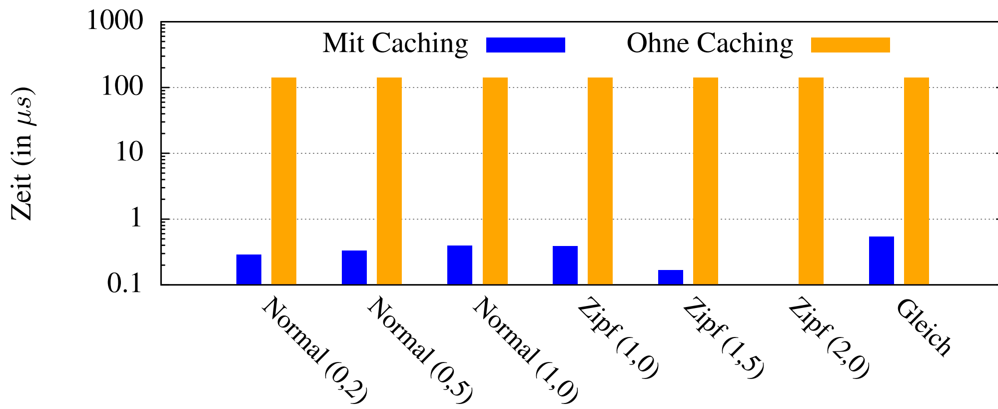
Für die Evaluierung der Caching-Leistung wurden drei Knoten verwendet. Der erste Knoten erzeugt insgesamt 100 Millionen Objekte und migriert 1, 5 und 10% der erzeugten Objekte. Der zweite Knoten agiert als Super-Peer und speichert die Metadaten für die erzeugten Chunks in Form von CID-Bereichen in einem CID-Baum. Die durchgeführten Migrationen umfassen jeweils 1, 10 oder 100 zusammenhängende Chunks pro migrierten Bereich (CpB). Werden beispielsweise aus Lastgründen Profile von Nutzern eines sozialen Netzwerkes migriert, müssen meist mehrere Objekte (Freundesliste, Kommentare, Ressourcen, etc.) gleichzeitig migriert werden (Aufgabe der Anwendung). Der dritte Knoten stellt schließlich eine Milliarden Objekt-

suchen an den Super-Peer, um den Speicherort der Objekte zu ermitteln. Auf diesem Knoten wird (wenn eingeschaltet) ebenfalls ein CID-Baum als Cache eingesetzt, der Antworten des Super-Peers speichert. Ein Messdurchlauf umfasst jeweils eine Messung mit eingeschaltetem Cache und eine Messung ohne Cache. Neben den Zeiten pro Operation wird auch die Größe des Caches gemessen. Während der Messungen werden Einträge nicht invalidiert, so dass die gemessenen Größen dem schlechtesten Fall entsprechen.
Die angefragten CIDs werden mit Hilfe verschiedener Verteilungen ausgewählt. Zum Einsatz kommen eine Normalverteilung mit einer Varianz von 0,2, 0,5 und 1,0, eine Zipf-Verteilung mit Rang 1,0, 1,5 und 2,0 und eine Gleichverteilung. Dabei stellt die Gleichverteilung den schlechtesten Fall dar, bei dem es unwahrscheinlich ist, dass die selbe CID mehrfach angefragt wird. Die Normal- und die Zipf-Verteilung stellen realistischere Verteilungen dar, da bestimmte CIDs häufiger angefragt werden, beispielsweise Profile von Popstars in einem sozialen Netzwerk. Der soziale Netzwerk-Benchmark BG verwendet beispielsweise eine Zipf-Verteilung zur Emulation von Nutzeranfragen [180]. In diesem Zusammenhang bedeutet eine geringere Varianz beziehungsweise ein größerer Rang, dass weniger Profile besonders häufig und eine größere Varianz beziehungsweise ein geringerer Rang, dass mehr Profile weniger häufig abgerufen werden.

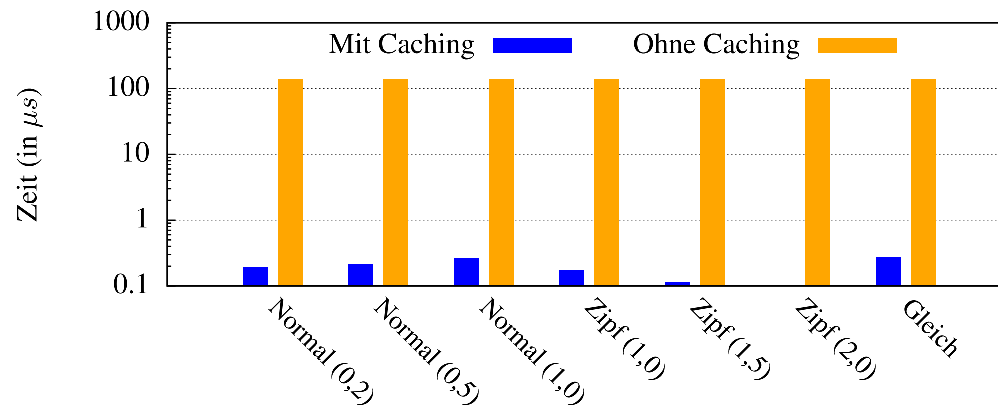
Abbildung 5.10a zeigt die Zeiten pro Operation bei 1% Migrationen und einem Chunk pro migriertem Bereich. Abhängig von der Verteilung ermöglicht der Einsatz des CID-Baumes als lokaler Cache bei 1 CpB eine Beschleunigung der Objektsuche um einen Faktor von 60 bis 1.300. Wie zu erwarten tritt bei der Gleichverteilung der schlechteste Fall ein, bei dem nur eine Beschleunigung um den Faktor 60 erreicht wird. Bei der Normalverteilung mit Varianz von 0,2 und der Zipf-Verteilung mit dem Rang 2,0 hingegen werden die größte Beschleunigungen mit dem Faktor 270 beziehungsweise 1.300 erzielt. Die Zeiten pro Operation ohne Caching sind nahezu unabhängig von der Verteilung und unterscheiden sich einzig im Nanosekundenbereich.
Bei 10 Chunks pro Bereich ist das Caching noch effektiver (siehe Abbildung 5.10b). Bei der Gleichverteilung ermöglicht der Cache eine Beschleunigung der Objektsuche um einen Faktor von 260 und bei der Normal- und Zipf-Verteilung um einen maximalen Faktor von 1.550. Auch bei dieser Messreihe unterscheiden sich die Zeiten ohne Caching nur geringfügig.
Die bisherigen Ergebnisse setzen sich auch bei 100 Chunks pro Bereich fort (siehe Abbildung 5.10c). Die Beschleunigung liegt bei einem Faktor von 520 bei der Gleichverteilung und maximal 1.500 bei Normal- und Zipf-Verteilung. Die Zeiten ohne Caching unterscheiden sich weiterhin nur geringfügig und sind auch nur minimal besser als bei 1 und 10 Chunks pro Bereich.
Abbildung 5.11 zeigt die Größe des Caches bei allen Messungen mit 1% Migrationen. Bei der Gleichverteilung ist der Cache jeweils am größten, dicht gefolgt von der Zipf-Verteilung



***(a)** 1 Chunk pro Bereich (CpB)*



***(b)** 10 Chunks pro Bereich (CpB)*



***(c)** 100 Chunks pro Bereich (CpB)*

***Abbildung 5.10: Caching von CID-Bereichen (1% Migrationen).** Die CIDs für die Anfragen wurden mit einer Normalverteilung, einer Zipf-Verteilung und einer Gleichverteilung erzeugt.*

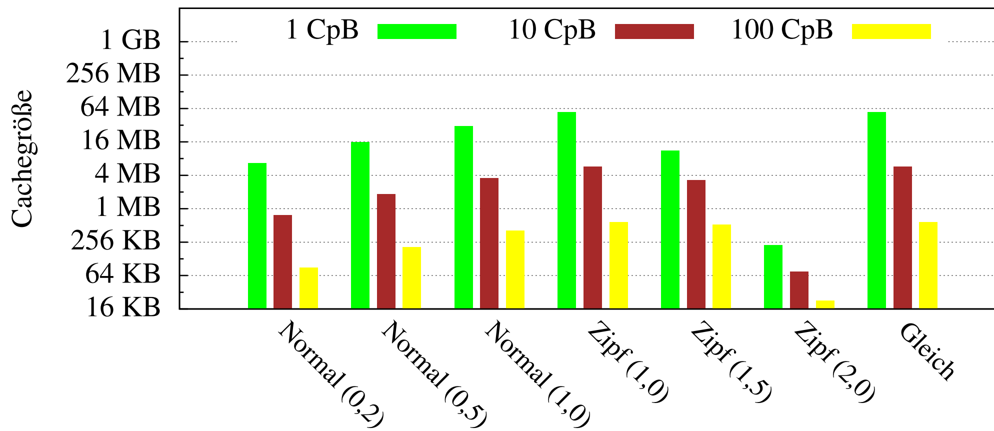
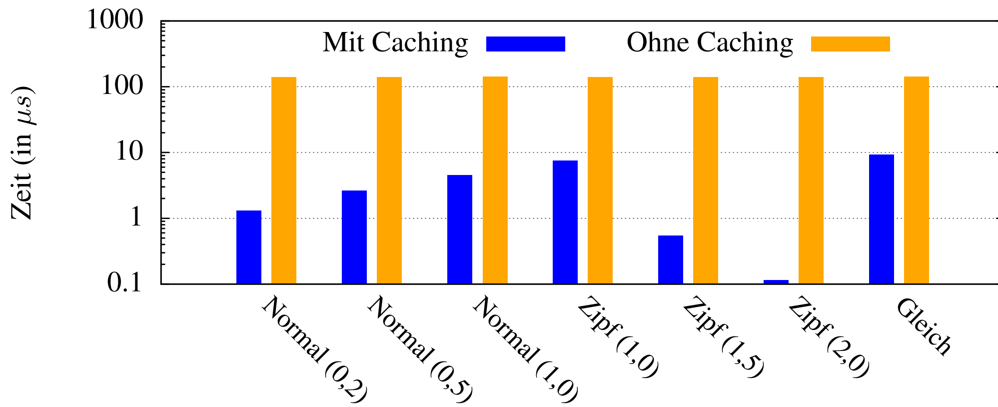


Abbildung 5.11: Cachegrößen (1% Migrationen). *Die angegebenen Cachegrößen sind jeweils für 1, 10 und 100 Chunks pro Bereich (CpB).*

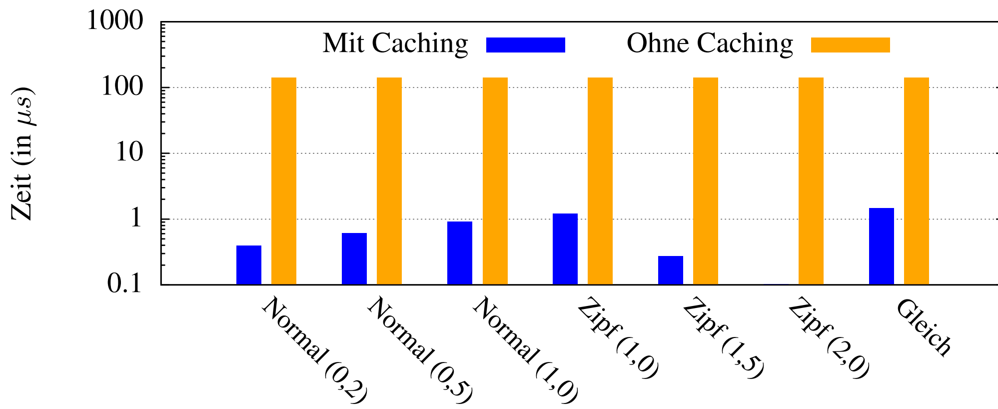
mit einem Rang von 1,0. Bei der Zipf-Verteilung mit einem Rang von 2,0 ist hingegen die Cachegröße am geringsten. Eindeutig erkennbar ist, dass die Größe der migrierten Bereiche die Größe des Caches erheblich beeinflusst. Bei der Gleichverteilung ist die Größe des Caches im Vergleich zu 1 CpB bei 10 CpB ungefähr 10 mal kleiner und bei 100 CpB ungefähr 100 mal kleiner. Bei der Normalverteilung (unabhängig von der Varianz) ist die Größe des Caches im Vergleich zu 1 CpB bei 10 CpB ungefähr 8 mal kleiner und bei 100 CpB ungefähr 75 mal kleiner. Bei der Zipf-Verteilung hängt die Cachegröße stark vom Rang ab. So ist bei einem Rang von 1,0 die Größe des Caches im Vergleich zu 1 CpB bei 10 CpB ungefähr 10 mal kleiner und bei 100 cpB ungefähr 100 mal kleiner. Bei einem Rang von 1,5 ist die Größe 3 mal beziehungsweise 21 mal kleiner und bei einem Rang von 2,0 ist die Größe nur noch 3 mal beziehungsweise 10 mal kleiner.

In Abbildung 5.12a werden die Zeiten für 5% Migrationen und 1 CpB dargestellt. Die Zeiten ohne Caching sind nahezu unverändert und unterscheiden sich kaum von den Zeiten bei 1% Migrationen. Bei eingeschalteten Cache lässt sich die Objektsuche erheblich beschleunigen, die Zeiten pro Operation sind allerdings höher als bei 1% Migrationen. Bei der Gleichverteilung wird eine Beschleunigung um den Faktor 15, bei der Normalverteilung maximal um den Faktor 110 und bei der Zipf-Verteilung maximal um den Faktor 1.200 erreicht. Im Vergleich zu 1% Migrationen sind die Zeiten bei der Gleichverteilung 3,7 mal höher, bei der Normalverteilung zwischen 2,5 und 3,3 mal höher und bei der Zipf-Verteilung zwischen 1,1 und 4 mal höher.

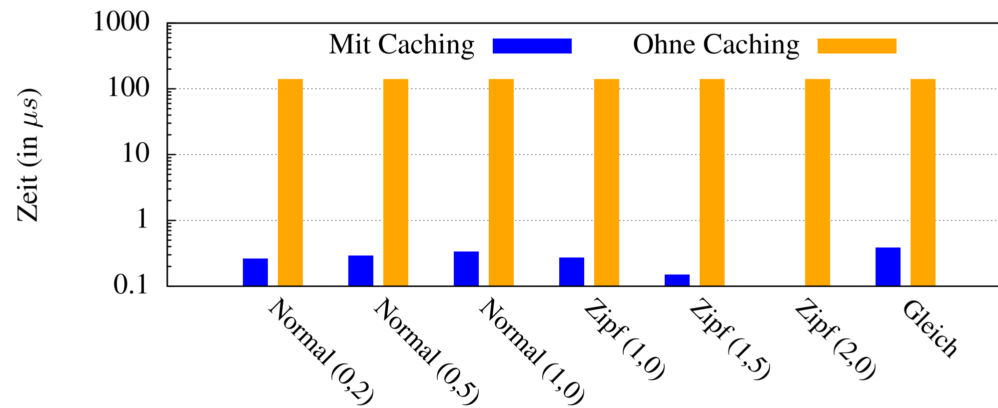
Für 10 CpB sehen die Messergebnisse ähnlich aus (siehe Abbildung 5.12b). Ohne Caching sind die Zeiten fast gleich und nur geringfügig verändert zu den Zeiten bei 1% Migrationen. Mit Caching sind die Zeiten höher als bei 1% Migrationen, erlauben aber eine Beschleuni-



***(a)** 1 Chunk pro Bereich (CpB)*



***(b)** 10 Chunks pro Bereich (CpB)*



***(c)** 100 Chunks pro Bereich (CpB)*

***Abbildung 5.12: Caching von CID-Bereichen (5% Migrationen).** Die CIDs für die Anfragen wurden mit einer Normalverteilung, einer Zipf-Verteilung und einer Gleichverteilung erzeugt.*

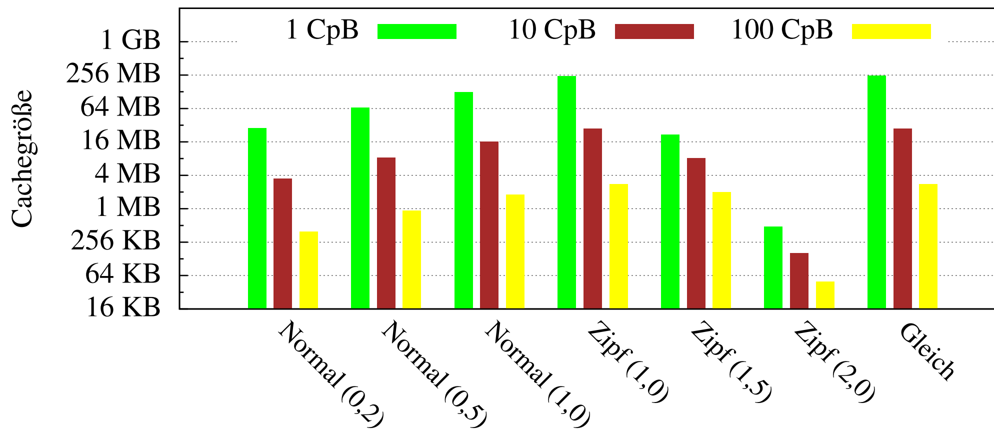
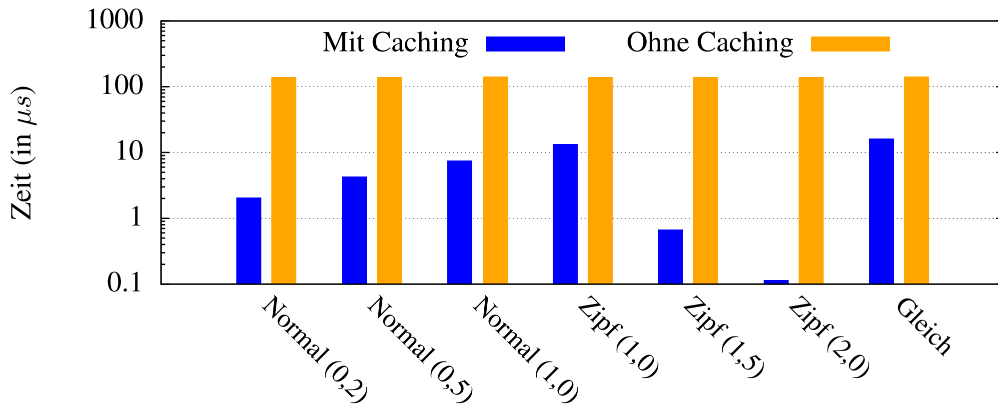


Abbildung 5.13: Cachegrößen (5% Migrationen). *Die angegebenen Cachegrößen sind jeweils für 1, 10 und 100 Chunks pro Bereich (CpB).*

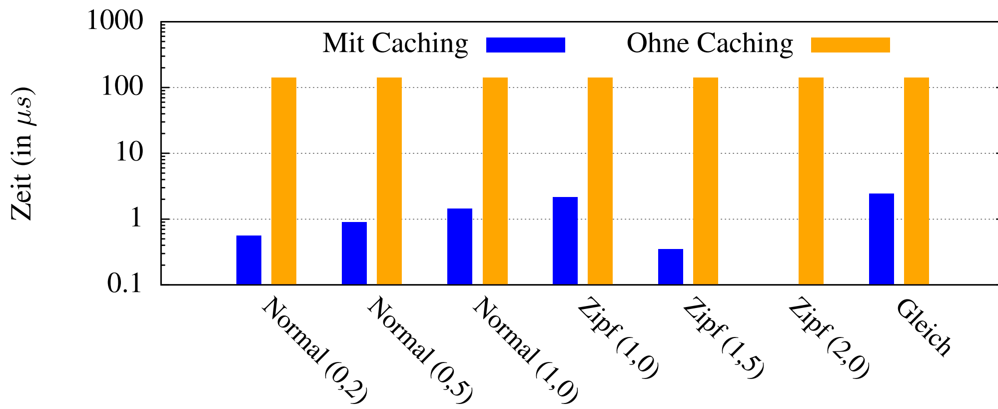
gung der Objektsuche um einen Faktor von bis zu 1.360. Die Gleichverteilung erlaubt eine Beschleunigung um den Faktor 95, die Normalverteilung maximal um den Faktor 350 und die Zipf-Verteilung maximal um den Faktor 1.360. Im Vergleich zu 1% Migrationen sind die Zeiten bei der Gleichverteilung 2,7 mal höher, bei der Normalverteilung zwischen 1,4 und 2,3 mal höher und bei der Zipf-Verteilung zwischen 1,1 und 3,1 mal höher.

Auch bei 100 CpB (siehe Abbildung 5.12c) sind die Zeiten bei ausgeschaltetem Cache fast gleich und nahezu unverändert zu 1% Migrationen. Das Caching ermöglicht eine Beschleunigung der Objektsuche bei der Gleichverteilung um den Faktor 360, bei der Normalverteilung maximal um den Faktor 530 und bei der Zipf-Verteilung maximal um den Faktor 1.380. Im Vergleich zu 1% Migrationen sind die Zeiten bei der Gleichverteilung 1,4 mal höher, bei der Normalverteilung zwischen 1,3 und 1,4 mal höher und bei der Zipf-Verteilung zwischen 1,1 und 1,5 mal höher.

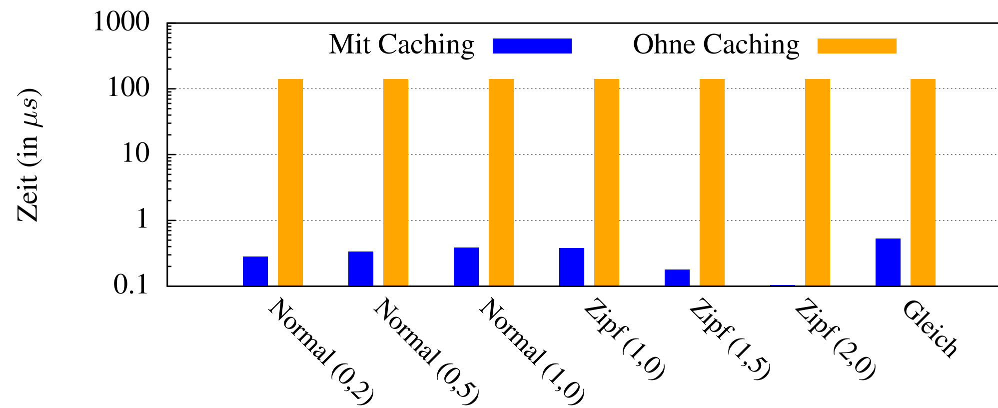
Abbildung 5.13 zeigt die Größe der Caches bei den Messungen. Bei der Gleichverteilung und der Zipf-Verteilung mit dem Rang 1,0 ist der Cache wieder am größten und bei der Zipf-Verteilung mit dem Rang 2,0 am kleinsten. Bei der Gleichverteilung ist die Größe des Caches im Vergleich zu 1 CpB bei 10 CpB ungefähr 9 mal kleiner und bei 100 CpB ungefähr 90 mal kleiner. Bei der Normalverteilung (unabhängig von der Varianz) ist die Größe des Caches im Vergleich zu 1 CpB bei 10 CpB ungefähr 8 mal kleiner und bei 100 CpB ungefähr 70 mal kleiner. Bei der Zipf-Verteilung hängt die Cachegröße stark vom Rang ab. So ist bei einem Rang von 1,0 die Größe des Caches im Vergleich zu 1 CpB bei 10 CpB ungefähr 9 mal kleiner und bei 100 cpB ungefähr 88 mal kleiner. Bei einem Rang von 1,5 ist die Größe 2,5 mal beziehungsweise 11 mal kleiner und bei einem Rang von 2,0 ist die Größe nur noch 3 mal beziehungsweise 10 mal kleiner.



(a) 1 Chunk pro Bereich (CpB)



(b) 10 Chunks pro Bereich (CpB)



(c) 100 Chunks pro Bereich (CpB)

Abbildung 5.14: Caching von CID-Bereichen (10% Migrationen). *Die CIDs für die Anfragen wurden mit einer Normalverteilung, einer Zipf-Verteilung und einer Gleichverteilung erzeugt.*

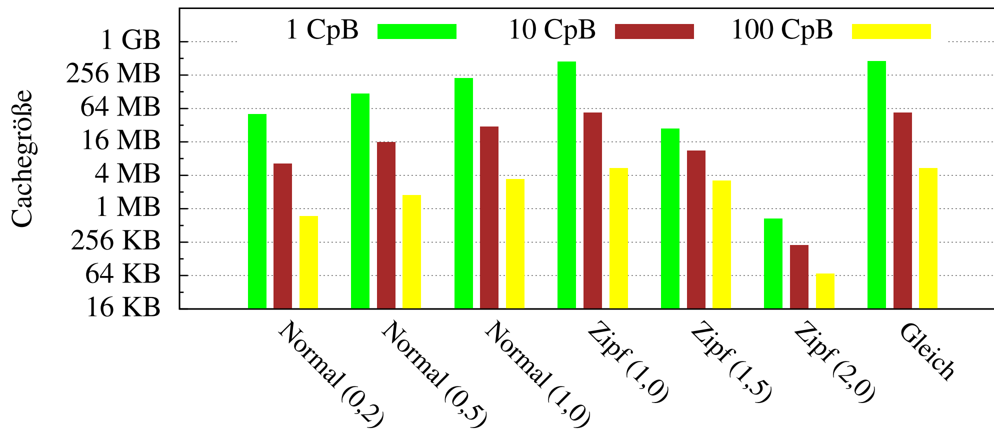


Abbildung 5.15: Cachegrößen (10% Migrationen). *Die angegebenen Cachegrößen sind jeweils für 1, 10 und 100 Chunks pro Bereich (CpB).*

Die Zeiten bei 10% Migrationen und 1 CpB werden in Abbildung 5.14a dargestellt. Auch bei dieser Migrationsrate sind die Zeiten ohne Caching nahezu identisch. Mit eingeschaltetem Cache wurde eine Beschleunigung der Objektsuche um einen Faktor zwischen 8,5 und 1.230 gemessen. Bei der Gleichverteilung wird mit dem Faktor 8,5 mit Abstand die geringste Beschleunigung erreicht. Bei der Normalverteilung kann eine maximale Beschleunigung um den Faktor 68 und bei der Zipf-Verteilung um den Faktor 1.230 erreicht werden. Im Vergleich zu 1% Migrationen sind die Zeiten bei der Gleichverteilung 6,6 mal höher, bei der Normalverteilung zwischen 3,9 und 5,5 mal höher und bei der Zipf-Verteilung zwischen 1,1 und 7 mal höher.

Bei 10 CpB und abgeschaltetem Caching unterscheiden sich die Zeiten nur geringfügig (siehe Abbildung 5.14b). Durch das Caching kann die Objektsuche um einen maximalen Faktor von 1.440 beschleunigt werden. Bei der Gleichverteilung wird ein Faktor von 57, bei der Normalverteilung ein maximaler Faktor von 253 und bei der Zipf-Verteilung ein maximaler Faktor von 1.440 erreicht. Im Vergleich zu 1% Migrationen sind die Zeiten bei der Gleichverteilung 4,5 mal höher, bei der Normalverteilung zwischen 1,9 und 3,6 mal höher und bei der Zipf-Verteilung zwischen 1,1 und 5,5 mal höher.

Abschließend werden die Messergebnisse bei 100 CpB in Abbildung 5.14c dargestellt. Auch in dieser letzten Messreihe unterscheiden sich die Zeiten ohne Caching kaum. Bei der Gleichverteilung lässt sich die Objektsuche um einen Faktor von 265, bei der Normalverteilung maximal um einen Faktor von 497 und bei der Zipf-Verteilung maximal um einen Faktor von 1.350 beschleunigen. Im Vergleich zu 1% Migrationen sind die Zeiten bei der Gleichverteilung 1,9 mal höher, bei der Normalverteilung zwischen 1,5 und 1,6 mal höher und bei der Zipf-Verteilung zwischen 1,1 und 2,1 mal höher.

Abbildung 5.15 zeigt die Größe der Caches bei den Messungen. Auch bei diesen Messungen ist der Cache bei der Gleichverteilung und der Zipf-Verteilung mit dem Rang 1,0 am größten und bei der Zipf-Verteilung mit dem Rang 2,0 am kleinsten. Bei der Gleichverteilung ist die Größe des Caches im Vergleich zu 1 CpB bei 10 CpB ungefähr 8,5 mal kleiner und bei 100 CpB ungefähr 84 mal kleiner. Bei der Normalverteilung (unabhängig von der Varianz) ist die Größe des Caches im Vergleich zu 1 CpB bei 10 CpB ungefähr 7,5 mal kleiner und bei 100 CpB ungefähr 67 mal kleiner. Bei der Zipf-Verteilung hängt auch bei dieser Messung die Cachegröße stark vom Rang ab. So ist bei einem Rang von 1,0 die Größe des Caches im Vergleich zu 1 CpB bei 10 CpB ungefähr 8 mal kleiner und bei 100 CpB ungefähr 83 mal kleiner. Bei einem Rang von 1,5 ist die Größe 2,5 mal beziehungsweise 9 mal kleiner und bei einem Rang von 2,0 ist die Größe nur noch 3 mal beziehungsweise 9,6 mal kleiner.

Zusammenfassend lässt sich sagen, dass sich der CID-Baum hervorragend für das Caching eignet und eine erhebliche Reduzierung der Zugriffszeiten ermöglicht. Für die realistischen Verteilungen (Normal- und Zipf-Verteilung) kann selbst mit einer geringen Cachegröße die Objektsuche problemlos um 90% reduziert werden. So hätte beispielsweise ein Cache von 64 MB für fast alle durchgeführten Messungen ausgereicht, um alle Anfragen aufzunehmen. Die notwendige Cache-Größe würde sich durch eine Cache-Verdrängung sogar noch steigern lassen (bisher nicht realisiert). Im besten Fall wird eine Reduzierung von über 99,99% erreicht. Bei der Gleichverteilung, bei der die schlechtesten Ergebnisse erzielt wurden, wird die Objektsuche immer noch um 88% reduziert.

5.4 Evaluation des Gesamtsystems mit BG-Benchmark

5.4.1 BG-Benchmark

BG-Benchmark ist ein verteiltes Testsystem, das die Performanz von Speichersystemen hinsichtlich interaktiver sozialer Netzwerke evaluiert [180] und wurde am Computer Science Department der USC entwickelt. Aktuell werden verschiedene moderne Datenspeicher, wie MongoDB [42] und Cassandra [44], unterstützt und eine Schnittstelle bereitgestellt, die es ermöglicht weitere Speichersysteme einzubinden.
BG emuliert Interaktionen in einem sozialen Netzwerke bekannt von Facebook[2], Twitter[3] und YouTube[4], indem bestimmte Aktionen ausgeführt werden, wie beispielsweise das Betrachten eines Benutzerprofils, das Abschicken und Bestätigen einer Freundschaftsanfrage, das Folgen

[2] http://www.facebook.com
[3] http://twitter.com
[4] http://www.youtube.com

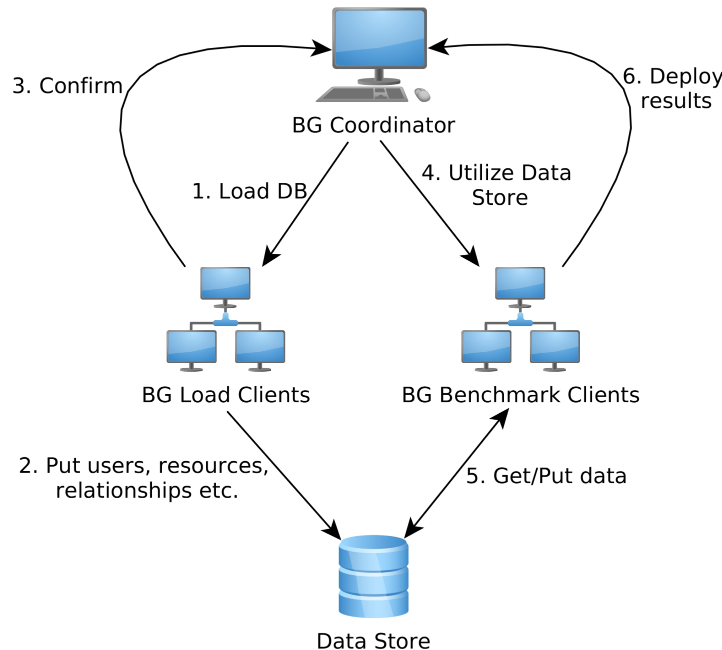


Abbildung 5.16: BG-Benchmark Aufbau. *Als Datenspeicher können unterschiedliche Systeme zum Einsatz kommen, beispielsweise MongoDB, Cassandra und auch DXRAM.*

eines Nutzers, das Abschicken eines Kommentars, etc. Zu diesem Zweck wird das Speichersystem mit einer vordefinierten Anzahl an Benutzern und optional Profilbildern, Freunden und Ressourcen (wie Bilder und Kommentare) befüllt. Durch die Verknüpfung von Benutzern untereinander und zwischen Benutzern und Ressourcen entsteht ein sehr großer Graph. Das zu Grunde liegende Speichersystem bestimmt dabei, wie der Graph genau gespeichert ist, beispielsweise in Form von Tabellen und Indizes im Fall von MongoDB und Cassandra. Nachdem alle Daten geladen wurden, emulieren Benchmark-Clients soziale Netzwerk-Aktionen, um den Datenspeicher zu evaluieren. Die Last für den Datenspeicher wird kontinuierlich durch einen zentralen Koordinator erhöht, bis das Speichersystem überlastet ist. Abbildung 5.16 zeigt einen Überblick einer Evaluierung mit dem BG-Benchmark.

Da der Durchsatz alleine meist nicht aussagekräftig genug für die Bewertung eines Speichersystems ist, sondern meist auch die Antwortzeiten relevant sind, erlaubt BG ein Service Layer Agreement (SLA) zu definieren. Das SLA legt fest, dass ein bestimmter Prozentsatz an Antworten innerhalb einer bestimmten Zeit erfolgen müssen. Beispielsweise müssen 95% aller Antworten innerhalb von 100 Millisekunden nach dem Senden der Anfrage eintreffen. Dadurch bestimmt BG den maximalen Durchsatz der ein definiertes SLA erfüllt, was einen wesentlich faireren Vergleich unterschiedlicher Speichersysteme ermöglicht, als der reine Durchsatz an sich.

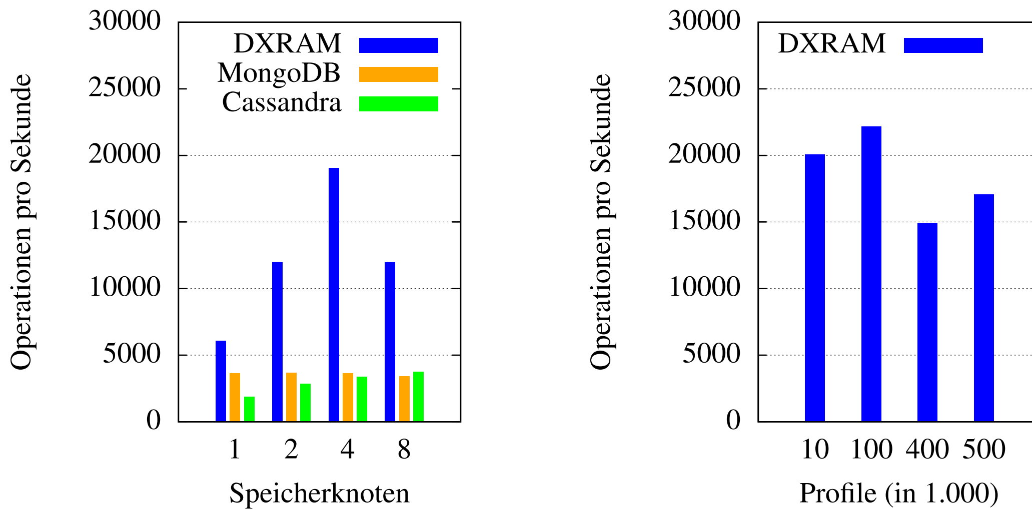
Die Verteilung der von den BG-Clients durchzuführenden Aktionen kann vor den Messungen konfiguriert werden. Für die nachfolgend durchgeführte Evaluierung wird eine Zipf-Verteilung verwendet bezüglich der Benutzer, für die entsprechende Aktionen ausgeführt werden. Des weiteren wird eine Verteilung gewählt, bei der nur selten Objekte aktualisiert werden und die Mehrzahl aller Zugriffe lesend sind, wie zum Beispiel das Betrachten eines Benutzerprofils. Das SLA ist wie zuvor beschrieben konfiguriert und sowohl das Speichersystem als auch der Benchmark werden auf einem privaten Cluster ausgeführt (Konfiguration siehe Abschnitt 5.1).

5.4.2 MongoDB

MongoDB ist eine schemafreie, dokumentenorientierte Datenbank für verschiedene Plattformen, die Sammlungen von BSON-Dokumenten verwaltet. Dokumente bestehen aus Schlüssel-Wert-Paaren, wobei Werte auch weitere Dokumente oder sogar Listen von Dokumenten sein können. Mehrere Dokumente lassen sich in Sammlungen bündeln und damit organisieren [42, 181].
MongoDB wurde entwickelt um möglichst einfach bedienbar, performant und skalierbar zu sein und verzichtet zu diesem Zweck auf einige Merkmale traditioneller relationaler Datenbanksysteme wie Transaktionen, ACID-Eigenschaften und eine umfangreiche Abfragesprache, die zu einem komplexen Programmcode führen würden.
Zur Lastverteilung werden die Daten mittels Sharding in Bereiche aufgespalten und auf mehrere Knoten verteilt. Ein Shard besteht aus einem Master und gegebenenfalls einem oder mehreren Slaves (Replikation). Die Shards können manuell oder automatisch konfiguriert werden und ermöglichen auch den Beitritt weiterer Knoten zur Laufzeit.

5.4.3 Cassandra

Apache Cassandra ist eine verteilte NoSQL-Datenbank für große strukturierte Datenmengen, die auf mehrere Rechner verteilt sind [44] und wurde ursprünglich von Facebook entwickelt [89]. Daten werden in Schlüssel-Wert-Paaren in zeilenorientierten Tabellen mit konfigurierbarer Konsistenz verwaltet. Cassandra ermöglicht hohe Skalierbarkeit und Verfügbarkeit ohne Single-Point-of-Failure.
Cassandra verfolgt einen dezentralen Ansatz ohne Single-Point-of-Failure. Jeder Knoten hat die gleiche Rolle inne und kann jede Anfrage beantworten, wodurch eine hohe Skalierbarkeit erreicht wird. Zu diesem Zweck werden Daten automatisch auf mehreren Knoten repliziert, wobei sich die Replikationsstrategien konfigurieren lassen und auch Replikation über mehrere Rechenzentren hinweg umsetzbar ist. Anfragen an das System werden in der zu SQL ähnlichen



(a) Vergleich des Gesamtsystems mit MongoDB und Cassandra.

(b) Das Gesamtsystem mit vielen Profilen.

Abbildung 5.17: BG-Benchmark Evaluierung. *Emulation eines sozialen Netzwerkes mit mehreren Speichersystemen.*

Abfragesprache Cassandra Query Language (CQL) gestellt, die für verschiedene Programmiersprachen zur Verfügung steht.

5.4.4 Performanz des Gesamtsystems

Bei der im Folgenden dargestellten Evaluierung wurde das in dieser Dissertation vorgestellte Gesamtsystem (DXRAM) mit MongoDB (2.6.4) und Cassandra (3.0.0 Snapshot) mit Hilfe des BG-Benchmark verglichen. Für das Gesamtsystem wurde eine eigene Implementierung der BG-Schnittstelle entwickelt und für MongoDB und Cassandra die bereits vorhandenen Implementierungen von BG verwendet. Die Implementierung der MongoDB-Schnittstelle wurde in [182] ausführlich beschrieben, wohingegen die Details zur Implementierung der Cassandra-Schnittstelle bisher nicht veröffentlicht wurden.
BG wurde so konfiguriert, dass insgesamt 10.000 Profile mit jeweils 100 Freunden und 100 Ressourcen auf 1 bis 8 Speicherknoten erstellt werden, wodurch ein Graph mit über einer Millionen Knoten und zwei Millionen Kanten entsteht. Anschließend führen vier BG-Knoten mit mehreren hundert Threads (jeder Thread emuliert einen Client) entsprechende Nutzeraktionen aus. Messungen mit mehr Daten konnten nicht durchgeführt werden, da bei größeren Datenmengen Probleme mit der Schnittstelle sowohl zwischen BG und MongoDB als auch zwischen BG und Cassandra existieren.

In diesem Szenario wird MongoDB mit einem Config-Server und einem Query-Router ausgeführt und die Daten mittels Sharding auf die bis zu acht Speicherknoten aufgeteilt. Cassandra verwendet den Standard-Partitionierer *Murmur3Partitioner*, der die Tabellenzeilen auf die Speicherknoten unter Zuhilfenahme von Consistent Hashing aufteilt. Normalerweise sorgt diese Aufteilung für eine gute Lastverteilung, da Cassandra zusätzlich virtuelle Knoten für die Partitionierung des Schlüsselraums einsetzt, für BG wird jedoch aus diesem Grund eine Menge von knotenübergreifendem Anfragen erwartet. Beispielsweise können so Einträge der Freundesliste über mehrere Knoten verteilt sein und zusätzlichen Netzwerkverkehr verursachen. Das in dieser Dissertation vorgestellte Gesamtsystem wird mit einem einzelnen Super-Peer gestartet, der alle Metadaten verwaltet.
Abbildung 5.17a zeigt die entsprechenden Messergebnisse. MongoDB führt durchgehend um die 3.500 Aktionen pro Sekunde aus. Alle Anfragen laufen dabei über den Query-Router, der in diesem Szenario der Flaschenhals des Systems ist. Bei Cassandra lässt sich die Skalierbarkeit gut beobachten. Mit zunehmender Knotenanzahl wächst der Durchsatz von 1.800 auf bis zu 3.700 Aktionen pro Sekunde. DXRAM agiert durchweg besser als die anderen beiden Systeme und skaliert sehr gut für bis zu vier Speicherknoten mit Ergebnissen von 6.000, 12.000 und 19.000 Aktionen pro Sekunde. Bei acht Speicherknoten ist der Durchsatz niedriger, da die vier BG-Knoten nicht genügend Last erzeugen können, um die Speicherknoten voll auszulasten. Wenn die Anzahl der BG-Knoten weiter erhöht wird, lässt sich der Durchsatz teilweise noch steigern. DXRAM wurde zusätzlich mit vier Speicherknoten und acht BG-Knoten mit unterschiedliche vielen Profilen (10.000, 100.000, 400.000 und 500.000) evaluiert (siehe Abbildung 5.17b). Der resultierende Graph bei 500.000 Benutzerprofilen umfasst mehr als 50 Millionen Knoten und 100 Millionen Kanten. Bei allen Messungen kann ein Durchsatz von mehr als 15.000 Aktionen pro Sekunde erzielt werden, bei 100.000 Profilen sogar mehr als 22.000 Aktionen pro Sekunde. Die Variation beim Durchsatz lässt sich durch die unterschiedlichen Ausprägungen der gegebenen Graphen erklären. Je nach Verteilung der Objekte auf die Speicherknoten und die Beziehungen der Objekte untereinander ergeben sich unterschiedliche Graphkonstellationen. Äquivalente Messungen für MongoDB und Cassandra konnten leider nicht durchgeführt werden, da die entsprechenden Schnittstellen Probleme mit der Menge an Daten hatten.
Zusammenfassend zeigen die Messungen eine sehr gute Performanz des Gesamtsystems für eine unterschiedliche Anzahl an Speicherknoten und Clients. Im Vergleich zu MongoDB und Cassandra ist der Durchsatz 2 bis 6-mal höher und auch für sehr große Graphen wird ein entsprechend hoher Durchsatz erreicht.

5.5 Zusammenfassung

In diesem Kapitel wurden die einzelne Komponenten, des in dieser Dissertation vorgestellten Gesamtsystems, evaluiert und mit modernen Speichersystemen verglichen.

Für die lokale Metadaten-Verwaltung wurden die CID-Tabellen (siehe Abschnitt 3.2) mit zwei unterschiedlichen Hashtabellen-Implementierungen (Java HashMap und Cuckoo-Hashtabelle) verglichen und hinsichtlich Speicherverbrauch und Ausführungszeit in Abhängigkeit vom Füllgrad bewertet. Es hat sich gezeigt, dass die CID-Tabellen den beiden Hashtabellen bei den Ausführungszeiten und besonders beim Speicherverbrauch überlegen sind. Die CID-Tabelle ist bis zu 15-mal schneller als die Java HashMap und bis zu 52-mal schneller als die Cuckoo-Hashtabelle und reduziert den Speicherverbrauch um bis zu 92%.
Die in Abschnitt 3.3 vorgestellte Speicherverwaltung wurde in zweierlei Hinsicht evaluiert. Der präsentierte Speicherallokator für sehr viele kleine Objekte wurde mit einer ganzen Reihe traditioneller und moderner Speicherallokatoren verglichen. Für die relevanten Objektgrößen liegt der durchschnittliche Speicherbedarf für die Metadaten des Allokators bei lediglich 5%, wohingegen alle anderen Speicherallokatoren mindestens 20% zusätzlichen Speicher benötigen. Auch die Evaluierung der Defragmentierung in einem Worst-Case-Szenario hat gezeigt, dass die Bereinigung und Kompaktifizierung des Speichers hervorragend funktioniert und kleinere freie Speicherbereiche wieder zu großen Speicherbereichen verschmolzen werden. Der bereinigte Speicher steht anschließend wieder für Allokationen jeglicher Größe zur Verfügung.
Ferner wurde die Kombination von Adressübersetzung und Speicherverwaltung bezüglich des Gesamtspeicherverbrauchs und des Mehrverbrauchs für die Metadaten pro Objekt untersucht und dabei auch der Durchsatz der vorgestellten Ansätze gemessen. Für Objektgrößen zwischen 16 und 64 Byte beträgt der Speicherbedarf für die Metadaten pro Objekt konstant 7 Byte, was bei gleichverteilten Objektgrößen einem Mehraufwand von 17,5% entspricht.
Abschließend wurde die lokale Metadaten-Verwaltung mit RAMCloud in einem verteilten Umfeld verglichen. In Relation zu RAMCloud konnten die vorgestellten Ansätze den Speicherverbrauch um fast 80% senken und Lesezugriffe für viele kleine Objekte bis zu 15-mal schneller ausgeführt werden.

Anschließend wurde der CID-Baum, die zentrale Datenstruktur der globale Metadaten-Verwaltung (siehe Abschnitt 4.3), evaluiert. Auch in diesem Fall bietet sich der Vergleich mit den beiden Hashtabellen-Implementierungen (Java HashMap und Cuckoo-Hashtabelle) an, da Hashtabellen für die Objektsuche besonders verbreitet sind. Betrachtet wurden neben dem Speicherverbrauch auch die durchschnittliche Ausführungszeit einer Migration und einer Get-Operation. Bei den Ausführungszeiten ist der CID-Baum etwas schlechter als die beiden Hashtabellen, wobei die Cuckoo-Hashtabelle sich bei den Get-Operationen mit steigender Anzahl

an Einträgen zunehmend verschlechtert. Die Ausführungszeiten des CID-Baumes sind dabei fast unabhängig von der Migrationsrate und der Eintragsanzahl und verlaufen nahezu konstant. Beim Speicherverbrauch schneidet der CID-Baum am Besten ab und erlaubt eine Reduzierung des Speicherbedarfs um bis zu 99% im Vergleich zu den beiden Hashtabellen. Damit stellt der CID-Baum eine sehr speichereffiziente Datenstruktur da, die sich für die gegebenen Anforderungen am besten eignet.
Darüber hinaus wurde in einer weiteren Evaluierung der Einsatz des CID-Baumes als Cache auf Clientseite betrachtet. Dabei wurden drei verschiedene Verteilungen zur Auswahl der angefragten Chunks verwendet. Auch in diesem Fall zeigen sich die Vorteile des CID-Baumes, der es ermöglicht die durchschnittliche Anfragezeit um bis zu 99,99% zu reduzieren.

Zum Schluss wurden das Gesamtsystem aus allen präsentierten Komponenten mit dem sozialen Netzwerk-Benchmark BG evaluiert und mit MongoDB und Cassandra verglichen. Dabei wurde nicht nur ein durchweg hoher Durchsatz gemessen, der über den Werten der beiden anderen Systeme liegt, sondern auch die gute Skalierbarkeit des Systems hinsichtlich Knotenanzahl, Clients und Größe des Graphen (Objektmenge) gezeigt.

Zusammenfassend lässt sich sagen, dass die vorgestellten Konzepte für die Anforderung, Milliarden sehr kleiner Objekte speichereffizient im Hauptspeicher vieler verteilter Knoten zu halten, bestens geeignet sind und eine effiziente Umsetzung ermöglichen.

Kapitel 6

Zusammenfassung

6.1 Resultat

Im Rahmen dieser Dissertation wurde ein integriertes Konzept für die lokale und globale Metadaten-Verwaltung in einem verteilten RAM-basierten Speicher vorgestellt. Der RAM-basierte Speicher ist für den Einsatz in sozialen Netzwerken oder Graphanwendungen konzipiert, wodurch sehr viele sehr kleine Objekte verwaltet werden müssen. Der Umgang mit dieser großen Anzahl an sehr kleinen Objekten ist erst in den letzten Jahren erforderlich geworden und stellt eine große Herausforderung für die Metadaten-Verwaltung dar.
Die verwalteten Objekte sind meistens zwischen 16 und 64 Byte groß und Zugriffe müssen möglichst schnell erfolgen. Die Größe der Objekte und die enorme Anzahl machen es notwendig, dass jeder Knoten bis zu einer Milliarden Objekte permanent im RAM hält. Unter diesen Gegebenheiten führt jedes zusätzlich Byte (beispielsweise für Metadaten) zu einem zusätzlichen Gigabyte verbrauchten Arbeitsspeichers. Eine besonders speichereffiziente Metadaten-Verwaltung ist daher unerlässlich.
Die Metadaten-Verwaltung gliedert sich in einen lokalen und einen globalen Bestandteil, die auch unabhängig voneinander eingesetzt werden können, als integriertes Konzept jedoch besonders effizient sind.

Eine Instanz der lokalen Metadaten-Verwaltung existiert auf jedem Knoten des Systems und kümmert sich um die Speicherverwaltung und die Adressübersetzung von globalen IDs in virtuelle Speicheradressen. Speicherverwaltung und Adressübersetzung sind eng miteinander verzahnt und nutzen Informationen der jeweils anderen Komponente für eine effiziente Arbeitsweise.
Für die Speicherverwaltung kommt ein neuartiger Speicherallokator zum Einsatz, der speziell für die speichereffiziente Verwaltung von sehr vielen sehr kleinen Objekten entwickelt wurde. Der Speicherallokator benötigt, für die beabsichtigte Anwendungsdomäne, für jedes Objekt

nur zwei Byte Metadaten, wodurch der Speicherverbrauch, im Vergleich zu traditionellen Speicherallokatoren, erheblich gesenkt werden kann. Der Verzicht auf eine Ausrichtung an Cachezeilen ermöglicht eine sehr kompakte Speicherung. Der daraus möglicherweise resultierende Geschwindigkeitsverlust ist vernachlässigbar, da die Anfragen von den Netzlaufzeiten dominiert werden. Für die Freispeicher-Verwaltung werden doppelt-verkettete Listen eingesetzt, die innerhalb des freien Speichers selbst abgelegt sind und somit keinen zusätzlichen Speicher benötigen.
Ferner wird ein neuer Defragmentierungsalgorithmus präsentiert, der den Speicher regelmäßig bereinigt, kompaktifiziert und die gegebenenfalls entstehende Fragmentierung beseitigt. Die Verkürzung der Zeiger in der Speicherverwaltung verhindert die komplette Nutzung des virtuellen Adressraums und macht eine Defragmentierung notwendig. Diese kann in festen Zeitintervallen oder bei Bedarf gestartet werden und erlaubt eine inkrementelle oder vollständige Ausführung, die selbst in einem Worst-Case-Szenario sehr gute Ergebnisse liefert. Während der Ausführung greift die Defragmentierung auf Informationen der Adressübersetzung zu, um Objekte aus fragmentierten Speicherbereichen in nicht fragmentierte Speicherbereiche zu verschieben.
Die Adressübersetzung nutzt hierarchische Tabellen (CID-Tabellen) für die Speicherung der Zuordnung einer virtuellen Speicheradresse zu einer globalen ID, wodurch eine zum Paging klassischer PC-Betriebsysteme ähnliche Struktur entsteht. Globale IDs werden in sequentieller Reihenfolge erzeugt und ermöglichen zusammen mit der konsequenten Nutzung von Offset-Zeigern ein kompakte Darstellung mit Hilfe der hierarchischen Tabellen. Die dynamische Erzeugung und Freigabe von Tabellen reduziert den Speicherverbrauch auf ein Minimum, der im Durchschnitt nur der Länge eines Offset-Zeigers (standardmäßig 5 Byte) entspricht.
Abgesehen von der Adressübersetzung sind die hierarchischen Tabellen auch für die Verwaltung und Wiederverwendung von freigewordenen globalen IDs geeignet. Mit Hilfe einer hierarchischen Suche können nicht mehr verwendete globale IDs schnell erkannt und wiederverwendet werden. Der Einsatz von Bit-Markern ermöglicht eine Beschleunigung der hierarchischen Suche, die dadurch im besten Fall in konstanter Zeit erfolgen kann. Ein CID-Cache hält freigewordene globale IDs in einem Pool zur Verfügung bereit und kann somit die Aufrufe der hierarchische Suche verringern oder sogar komplett vermeiden.

Ein entscheidender Faktor für die Metadaten-Verwaltung ist der parallele Zugriff durch mehrere gleichzeitig agierende Threads. Eine geeignete Synchronisierung erlaubt ein hohes Maß an Parallelität ohne jedoch die Integrität der Daten zu vernachlässigen. Neben traditionellen Synchronisierungsverfahren (unter anderem in Java), wie Sperren, Semaphoren und Monitoren, wurden daher auch sperrfreie Algorithmen näher betrachtet und mit den traditionellen Verfahren verglichen. Während sich die traditionellen Verfahren besonders effizient in ihrer unfairen Ausprägung verhalten, sind die vorgestellten sperrfreien Algorithmen alle fair und wesentlich

effizienter als die traditionellen Verfahren in ihrer fairen Ausprägung.
Für die CID-Tabellen bieten sich zwei Alternativen für die Synchronisierung an. Bei der ersten Alternative besitzt jede Tabelle eine separate Lese-Schreib-Sperre, was eine einfache Implementierung und Wartung ermöglicht und eine gute Parallelität bietet. Da auf der höchsten Stufe jedoch nur eine CID-Tabelle existiert und alle Zugriffe zuerst diese Tabelle durchlaufen müssen, ergibt sich an dieser Stelle eine besonders hohe Wahrscheinlichkeit einer Wettlaufsituation. Die zweite Alternative mildert diesen Umstand ab, indem auf den höheren Stufen die Granularität der Synchronisierung verfeinert wird. Auf der obersten Stufe wird eine Sperre pro Eintrag und auf der untersten Stufe pro Tabelle eingesetzt. Auf den Stufen dazwischen wird eine Sperre für einen Satz Einträge verwendet, wobei die Anzahl der zusammengefassten Einträge mit abnehmender Stufe ansteigt. Die bei beiden Alternativen verwendeten Lese-Schreib-Sperren werden mit Hilfe von CAS-Instruktionen auf einer 4-Byte Speicherstelle ausgeführt, die sich direkt in die entsprechenden CID-Tabellen integrieren lassen.
Beim CID-Cache kommt eine verzögerte Aktualisierung zum Einsatz, da es unerheblich ist, wann eine freigewordene globale ID wiederverwendet wird. Die verzögerte Aktualisierung erlaubt eine sperrfreie Ausführung, solange keine freigewordene ID existiert. Da soziale Netzwerke und Graphanwendungen überwiegend lesend auf die gespeicherten Daten zugreifen, nur sehr selten neue Daten schreiben und noch seltener Daten löschen, ist der Cache die meiste Zeit leer und die sperrfreie Ausführung möglich.
In der Speicherverwaltung werden Arenen und ein Segment-Stealing-Mechanismus benutzt, um die Synchronisierung zwischen den Threads zu minimieren. Jedem Thread wird eindeutig eine Arena und damit ein exklusiver Speicherbereich zugewiesen, in dem nur dieser Thread schreiben darf. Lesezugriffe können allerdings auch in Arenen anderer Threads durchgeführt werden. Ist der zugewiesene Speicherbereich voll, weist ein zentraler Koordinator der Arena einen neuen, aktuell nicht zugewiesenen, Speicherbereich zu. Existiert kein solcher Speicherbereich mehr oder sind alle nicht zugewiesenen Speicherbereich voll, ermöglicht der Segment-Stealing-Mechanismus den Speicherbereich eines anderen Threads zu übernehmen. Dadurch ist es jedem Thread möglich Allokationen durchzuführen, solange irgendwo im Speicher ausreichend freier Platz zur Verfügung steht. Darüber hinaus sorgt der Segment-Stealing-Mechanismus dafür, dass Arenen von nicht mehr genutzten Threads bereinigt werden.

Die globale Metadaten-Verwaltung wird mit Hilfe eines dezentralen Super-Peer-Overlays realisiert. Die Hauptaufgaben der Super-Peers sind die Objektsuche, die Überwachung der Knoten und die Koordinierung der Datenwiederherstellung bei einem Knotenausfall. Im Gegensatz zu einem zentralen Koordinator erlauben die Super-Peers eine bessere Lastverteilung der Zugriffe und des Speicherverbrauchs und ermöglichen trotzdem eine Objektsuche in $\mathcal{O}(1)$ durchzuführen, indem regelmäßig eine Liste der aktiven Super-Peers an alle Knoten des System verschickt wird. Die Super-Peers formen einen Ring und jeder Super-Peer ist für einen Teil des globalen

ID-Raums und eine Gruppe von Peers zuständig. Die ersten 16 Bit einer globalen ID beinhalten den erzeugenden Knoten und bestimmen die Position des Objektes auf dem Super-Peer-Ring. Objekte sind im Regelfall auf dem erzeugenden Knoten gespeichert, zur Auflösung von Hot-Spots werden jedoch partielle Migrationen von Objekten auf andere Knoten unterstützt.
Zur Verwaltung der Metadaten (bis zu mehrere Milliarden Einträge) wird eine modifizierte B-Baum-Struktur (CID-Baum) eingeführt, die es erlaubt mehrere IDs zu einem ID-Bereich zusammen zu fassen und zu verwalten. Ein ID-Bereich besteht aus zwei Schlüsseln (Start-ID und End-ID) und der Knoten-ID des Knotens, auf dem alle Objekte des ID-Bereichs gespeichert sind. Die sequentielle Erzeugung der globalen IDs durch den Erzeuger des Objektes führt dazu, dass im besten Fall alle Objekte eines Knotens in einem einzigen ID-Bereich zusammengefasst werden können. Der Speicherbedarf der Metadaten lässt sich so um bis zu 99,99% im Vergleich zu Hashtabellen oder Listen reduzieren. Die Verwaltungen von Metadaten zur Unterstützung von Bereichsabfragen, bei denen alle Einträge innerhalb eines Bereichs ermittelt werden, ist weit verbreitet, allerdings ist die Verwaltung von Metadaten in Form von Bereichen für Abfragen eines einzelnen Eintrags innerhalb eines Bereiches in diesem Kontext neuartig.
Der CID-Baum wird neben dem Einsatz auf den Super-Peers auch auf den Peers genutzt, um Antworten der Super-Peers zu cachen und die Last der Super-Peers zu reduzieren. Dazu beantworten die Super-Peers eine Objektsuche nicht nur mit dem Knoten, auf dem das Objekt gespeichert ist, sondern mit dem gesamten ID-Bereich, zu dem das Objekt gehört. Dadurch können gegebenenfalls auch zukünftige Anfragen aus dem Cache beantwortet werden, wovon besonders lokalitäts-basierte Anwendungen erheblich profitieren können. Für die Invalidierung der Cache-Einträge werden drei unterschiedliche Ansätze präsentiert, die für eine Beschränkung der Größe sorgen und damit eine effiziente Nutzung ermöglichen.
Neben der Objektsuche ist die Koordinierung der Datenwiederherstellung eine wichtige Aufgabe der Super-Peers. Mit Hilfe verschiedener Maßnahmen können Ausfälle von einzelnen Peers und Super-Peers maskiert werden. Darüber hinaus wird auch beschrieben, wie die globalen Metadaten bei einem Verlust rekonstruiert werden können und wie das Gesamtsystem nach einem Totalausfall wieder in den zuletzt gültigen Zustand versetzt werden kann. Für das System von entscheidender Bedeutung ist dabei die schnelle parallele Datenwiederherstellung beim Ausfall eines Speicherknotens (Peer). Die Objekte eines Knoten werden dafür in ungefähr gleich große Backupzonen unterteilt und die Objekte jeder Backupzone auf mehreren Backupknoten auf SSD repliziert. Bei einem Knotenausfall kontaktiert der zuständige Super-Peer die Backupknoten und sorgt für die parallele Wiederherstellung alle Backupzonen. In einem grundlegenden Ansatz werden die Informationen über die Backupzonen und -knoten in den CID-Baum integriert und zusammen mit den ID-Bereichen verwaltet. In einem optimierten Ansatz wird die Verzahnung von ID-Bereichen und Backupinformationen etwas gelockert und dadurch Redundanzen vermieden und die Rekonstruktion der Metadaten bei einem Knotenaus-

fall vereinfacht.
Um einen hohen Durchsatz und kurze Antwortzeiten zu ermöglichen, müssen die Super-Peers so konzipiert sein, dass es möglich ist mit mehreren Netzwerkthreads gleichzeitig zuzugreifen. Dazu verfügt jeder Knoten des CID-Baumes über eine eigene Lese-Schreib-Sperre, womit es möglich wird, dass mehrere schreibende oder lesende Threads in unterschiedlichen Zweigen des Baumes gleichzeitig ausführbar sind.

Für die lokale Metadaten-Verwaltung wurden die CID-Tabellen mit zwei Hashtabellen verglichen und bezüglich Speicherverbrauch und Zugriffszeit ausgewertet. Die CID-Tabellen haben nicht nur die beste Zugriffszeit, sondern ermöglichen dabei auch eine Reduzierung des Speicherverbrauchs um bis zu 92%. Ferner wurde die Speicherverwaltung mit gängigen Speicherallokatoren verglichen, wobei sich ebenfalls eine sehr gute Speichereffizienz zeigte, die es erlaubt den Speicherbedarf für Metadaten um mindesten 75% zu verringern. Auch die Kombination von CID-Tabellen und Speicherverwaltung schnitt im Vergleich zu anderen Systemen im verteilten Fall sehr gut ab. Der gesamte Speicherverbrauch und die Antwortzeiten der vorgestellten Konzepte waren dabei erheblich niedriger.
Ferner wurde auch der CID-Baum zusammen mit zwei Hashtabellen evaluiert. Dabei zeigte sich, dass der CID-Baum für eine geringfügig schlechtere Zugriffszeit eine Reduzierung des Speicherbedarfs um bis zu 99,99% ermöglicht und auch für sehr viele Einträge gut skaliert. Der Einsatz des CID-Baumes als Cache wurde ebenfalls untersucht und stellte sich als sehr performant heraus. Die Antwortzeiten für die Objektsuche ließen sich in den Messungen um bis zu 98% reduzieren.
Abschließend wurde das Gesamtsystem (bestehend aus allen in dieser Arbeit präsentierten Konzepten) zusammen mit MongoDB und Cassandra mit Hilfe des BG-Benchmarks evaluiert, wobei im Vergleich zu den anderen beiden Systemen deutlich bessere Ergebnisse (auch für große Datenmengen) erzielt wurden.
Zusammenfassend bescheinigen die Messergebnisse durchweg eine gute Performanz und Skalierbarkeit und eine herausragende Speichereffizienz der vorgestellten Konzepte.

6.2 Ausblick

Die im Rahmen dieser Arbeit vorgestellten Konzepte bieten ein gutes Fundament um weitergehende Forschungen durchzuführen. Nachfolgend werden einige Punkte vorgestellt, bei denen sich möglicherweise eine nähere Betrachtung und weiterführende Arbeiten lohnen.

Einsatz eines Chunk-Pools

Bei Get-Operationen in der Speicherverwaltung werden die Daten des entsprechenden Chunks eingelesen und anschließend ein neuer Chunk erzeugt und mit den Daten befüllt.

Dabei benötigen die Erzeugung des Chunks und des intern verwendeten Byte-Arrays viel Zeit. Ein Pool, indem aktuell nicht benutzte Chunk-Objekte vorgehalten werden, könnte diese Zeit verkürzen, da nur neue Chunks erzeugt werden müssen, wenn der Pool leer ist. Der Einsatz eines solchen Pools ist aber nur dann sinnvoll, wenn nicht mehr benutzte Chunks markiert und in den Pool zurück geführt werden. Die beste Lösung dafür ist eine aktive Rückführung durch die Anwendung, da nur die Anwendung bestimmen kann, wann ein Chunk nicht mehr benutzt wird. Da diese Lösung offensichtlich nicht transparent für die Anwendung erfolgen kann, wäre die Entwicklung möglicher Alternativen von großem Vorteil. Vorstellbar wäre beispielsweise eine Rückführung über bestimmte Zugriffsmuster, die Anzahl der Zugriffe oder zeitgesteuert. Dabei müsste ferner untersucht werden, welchen Einfluss eine gegebenenfalls vorhandene Garbage Collection hat. Chunk-Objekte, die von der Garbage Collection freigegeben werden, könnten statt dessen in den Pool zurückgeführt werden.
Der zu entwickelnde Chunk-Pool könnte neben der Speicherverwaltung auch in der Netzwerkschnittstelle zum Einsatz kommen um Chunks für eintreffende Chunkdaten zur Verfügung zu stellen.

Direkter Speicherzugriff über einen Chunk

Wenn mit einem lokal gespeicherten Chunk gearbeitet wird, könnte der Durchsatz und die Zugriffszeiten erheblich gesteigert werden, indem nicht auf einer Kopie der Daten innerhalb eines Chunk-Objektes, sondern direkt auf den gespeicherten Daten in der Speicherverwaltung gearbeitet wird. Dazu müsste untersucht werden inwiefern ein direkter Speicherzugriff aus einem Chunk-Objekt heraus möglich ist.
In Java beispielsweise existieren so genannte Direct-Memory-Buffer, deren zugrunde liegendes Byte-Array sich im Off-Heap direkt in der virtuellen Speicherverwaltung des Betriebssystems befindet. Allerdings lässt sich die Speicheradresse des Byte-Arrays nicht übergeben, sondern es wird immer ein neues Array angelegt, auf dessen Speicheradresse kein Einfluss genommen werden kann. In C oder C++ hingegen wäre ein direkter Speicherzugriff problemlos umzusetzen, da innerhalb der Programmiersprache die direkte Verwendung und Manipulation von Zeigern möglich ist.
Der direkte Speicherzugriff lässt sind dabei auch gut mit dem Einsatz eines Chunk-Pools kombinieren.

Unterstützung von Chunk-Gruppen

In einem sozialen Netzwerk existieren für einen Benutzer meist mehrere Objekte, auf die häufig zusammenhängend zugegriffen werden muss (beispielsweise beim Aufruf eines Benutzerprofils). Dafür kann ein zentrales Objekt verwendet werden, dass Referenzen auf alle anderen Objekte des Benutzers (Freundesliste, Profilbild, Ressourcen, Kommen-

tare, etc.) beinhaltet. Soll beispielsweise die Freundesliste angezeigt werden, wird zuerst das zentrale Objekt angefordert, die entsprechende Referenz ausgelesen und schließlich das Objekt mit der Freundesliste geholt. Die sequentielle Erzeugung von globalen IDs ermöglicht eine Optimierung dieses Ablaufes, indem die wichtigen Objekte, die für jeden Benutzer existieren (zum Beispiel die Freundesliste oder das Profilbild), zusammen mit dem zentralen Objekt erzeugt werden. Dadurch kann anhand der globalen ID des zentralen Objektes auch die globale ID der zugehörigen Objekte bestimmt werden. Wenn die globale ID des zentralen Objektes ID_i ist, dann ist beispielsweise die globale ID der Freundesliste $ID_i + 1$, die globale ID des Profilbildes $ID_i + 2$, usw. Dieses Verfahren hat offensichtlich seine Grenzen, da die entsprechenden Objekte schon beim Anlegen des zentralen Objektes erzeugt oder zumindest weitere globale IDs reserviert werden müssen.

Eine alternative Möglichkeit ist die Verwendung von dynamischen Gruppen, wobei für jeden Benutzer eine Gruppe existiert, die alle Objekte des Benutzers enthält. Wenn beispielsweise das Profil eines Benutzer angezeigt werden soll, wird die Gruppe des Benutzers angefordert und alle Objekte der Gruppe zurück geliefert. Werden neue Objekte für einen Benutzer erzeugt, werden diese zur Gruppe des Benutzers hinzugefügt und bei der nächsten Anforderung der Gruppe mitgeliefert.

Konsistenzmodelle auf Ebene der Metadaten

In dieser Arbeit wurde sich bewusst nicht mit Konsistenzmodellen beschäftigt, sondern auf eine Integration verschiedener Konsistenzmodelle oberhalb des Gesamtsystem verwiesen. So kann beispielsweise die Lock-Operation verwendet werden, um ein Konsistenzmodell auf höherer Ebene zu realisieren. Die Integration auf Ebene der Metadaten-Verwaltung kann jedoch einige Vorteile mit sich bringen und bietet sich daher für weitergehende Arbeiten an. Beispielsweise wäre es dadurch möglich verschiedene Konsistenzmodelle für unterschiedliche Objekte einzusetzen, die beim Erzeugen des Objektes oder während der Laufzeit festgelegt werden. So ist es in einem sozialen Netzwerk nicht entscheidend, dass immer die aktuellste Version einer Ressource oder eines Kommentars gelesen wird (Sequentielle Konsistenz), die Änderung des Passworts sollte allerdings sofort erfolgen (Strikte Konsistenz).

Abbildungsverzeichnis

1.1 Speicherhierarchie . . . 3

2.1 Erkennung einer Netzwerkpartitionierung . . . 14
2.2 Dienste und Komponenten von DXRAM . . . 15
2.3 Super-Peer-Overlay . . . 19
2.4 Knotenbeitritt . . . 20
2.5 Ausfall eines Peers . . . 21
2.6 Ausfall eines Super-Peers . . . 22
2.7 Adressübersetzung von CIDs . . . 25
2.8 Struktur des VMB . . . 26
2.9 Backup-Prozess . . . 28
2.10 Protokoll-Architektur . . . 30
2.11 Schreibpuffer und Primärprotokoll . . . 31
2.12 Datenwiederherstellung . . . 33

3.1 CID-Tabellen . . . 52
3.2 Layout von freiem Speicher . . . 59
3.3 Inkrementelle Defragmentierung . . . 66
3.4 Lost Update . . . 70
3.5 Arenen . . . 89
3.6 Threadverlauf . . . 92

4.1 Super-Peer-Ring . . . 102
4.2 Ausfallerkennung . . . 104
4.3 CID-Bereiche bei Chunkmigrationen . . . 108
4.4 Beispiel einer Chunkmigration in einer CID-Liste . . . 109
4.5 Suche in einem CID-Baum . . . 111
4.6 Optimierter Ansatz . . . 114
4.7 Paralleler Zugriff im CID-Baum . . . 118

5.1 Vergleich von Hash- und CID-Tabellen I . . . 133

5.2 Vergleich von Hash- und CID-Tabellen II 134
5.3 Vergleich von Hash- und CID-Tabellen III 135
5.4 Speicheroverhead verschiedener Allokatoren 139
5.5 Worst-Case-Szenario für die Defragmentierung 140
5.6 Evaluierung der Adressübersetzung und Speicherverwaltung 142
5.7 Speicherverbrauch im verteilten Fall 143
5.8 Performanz im verteilten Fall 144
5.9 Evaluierung der Datenstrukturen 146
5.10 Caching von CID-Bereichen (1% Migrationen) 149
5.11 Cachegrößen (1% Migrationen) 150
5.12 Caching von CID-Bereichen (5% Migrationen) 151
5.13 Cachegrößen (5% Migrationen) 152
5.14 Caching von CID-Bereichen (10% Migrationen) 153
5.15 Cachegrößen (10% Migrationen) 154
5.16 BG-Benchmark Aufbau 156
5.17 BG-Benchmark Evaluierung 158

Tabellenverzeichnis

3.1 Zustände der Marker Bytes . . . 57
3.2 Testsysteme . . . 77
3.3 Speicherverbrauch der Synchronisierungsobjekte . . . 78
3.4 Zeit pro Sperre . . . 79
3.5 Einsatz unterschiedlicher Synchronisierungsmöglichkeiten . . . 79
3.6 Synchronisierung in der Adressübersetzung . . . 82
3.7 Threads und Sperren . . . 91

4.1 Löschliste (ältester Eintrag) . . . 116
4.2 Löschliste (Anzahl Zugriffe) . . . 116
4.3 Löschliste (letzter Zugriff) . . . 117

5.1 Defragmentierung von freien Blöcken . . . 141

A.1 Speicherverbrauch Allokatoren (gesamt) . . . 189
A.2 Speicherverbrauch (Füllgrad 40%) . . . 190
A.3 Ausführungszeit (Füllgrad 40%) . . . 190
A.4 Speicherverbrauch (Füllgrad 75%) . . . 191
A.5 Ausführungszeit (Füllgrad 75%) . . . 191
A.6 Speicherverbrauch (Füllgrad 90%) . . . 192
A.7 Ausführungszeit (Füllgrad 90%) . . . 192
A.8 Speicherbedarf und Durchsatz der lokalen Metadaten-Verwaltung . . . 194
A.9 Speicherbedarf und Lesezugriffe (RAMCloud 80M) . . . 194
A.10 Speicherbedarf und Lesezugriffe (DXRAM 80M) . . . 195
A.11 Speicherbedarf und Lesezugriffe (DXRAM 300M) . . . 195
A.12 Speicherbedarf (1% Migrationen) . . . 196
A.13 Speicherbedarf (5% Migrationen) . . . 196
A.14 Speicherbedarf (10% Migrationen) . . . 196
A.15 Ausführungszeit Migrate-Operation (1% Migrationen) . . . 197
A.16 Ausführungszeit Migrate-Operation (5% Migrationen) . . . 197
A.17 Ausführungszeit Migrate-Operation (10% Migrationen) . . . 197

A.18 Ausführungszeit Get-Operation (1% Migrationen) 197
A.19 Ausführungszeit Get-Operation (5% Migrationen) 198
A.20 Ausführungszeit Get-Operation (10% Migrationen) 198
A.21 Caching (1% Migrationen, 1 Chunk pro Bereich) 198
A.22 Caching (1% Migrationen, 10 Chunks pro Bereich) 199
A.23 Caching (1% Migrationen, 100 Chunks pro Bereich) 199
A.24 Caching (5% Migrationen, 1 Chunk pro Bereich) 199
A.25 Caching (5% Migrationen, 10 Chunks pro Bereich) 200
A.26 Caching (5% Migrationen, 100 Chunks pro Bereich) 200
A.27 Caching (10% Migrationen, 1 Chunk pro Bereich) 200
A.28 Caching (10% Migrationen, 10 Chunks pro Bereich) 201
A.29 Caching (10% Migrationen, 100 Chunks pro Bereich) 201
A.30 DXRAM, MongoDB und Cassandra 202
A.31 Viele Profile 202

Algorithmenverzeichnis

3.1 Speicher allozieren 60
3.2 Speicher freigeben 62
3.3 Petersons Algorithmus 73
3.4 Lamports Backery Algorithmus 74
3.5 Szymanskis Algorithmus 75
3.6 Lese-Schreib-Sperre (Teil 1) 84
3.7 Lese-Schreib-Sperre (Teil 2) 85
3.8 Segmentzuweisung im Arenamanager 88
3.9 Zugriff auf die Arena 90
3.10 Exklusive CAS-Sperre 92

4.1 Suche im CID-Baum 112

Literaturverzeichnis

[1] **White House**: Big Data: Seizing Opportunities, Preserving Values [2014].

[2] **A. Schäfer et al.**: Big Data: Vorsprung durch Wissen - Innovationspotenzialanalyse. *Sankt Augustin: Fraunhofer-Institut für Intelligente Analyse-und Informationssysteme IAIS* [2012].

[3] **E. Shurman und J. Brutlag**: Performance Related Changes and Their User Impacts. *Velocity 2009* [2009].

[4] **Berk Atikoglu et al.**: Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12. New York, NY, USA [2012]. ISBN 978-1-4503-1097-0.

[5] **Nathan Bronson et al.**: TAO: Facebook's Distributed Data Store for the Social Graph. In *Presented as part of the 2013 USENIX Annual Technical Conference*. Berkeley, CA [2013]. ISBN 978-1-931971-01-0.

[6] **Biplob Debnath, Sudipta Sengupta und Jin Li**: FlashStore: High Throughput Persistent Key-Value Store. *Proc. VLDB Endow.*, Band 3(1-2):1414–1425 [2010]. ISSN 2150-8097.

[7] **Guanlin Lu, Young Jin Nam und D.H.-C. Du**: BloomStore: Bloom-Filter Based Memory-Efficient Key-Value Store for Indexing of Data Deduplication on Flash. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, S. 1–11 [2012]. ISSN 2160-195X.

[8] **Rajesh Nishtala et al.**: Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, Illinois [2013].

[9] **Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout und Mendel Rosenblum**: Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third*

ACM Symposium on Operating Systems Principles, SOSP '11, S. 29–41. ACM, New York, NY, USA [2011]. ISBN 978-1-4503-0977-6.

[10] **Evangelos P. Markatos**: On Caching Search Engine Query Results. *Computer Communications*, Band 24(2):137–143 [2001].

[11] **Ricardo Baeza-Yates, Aristides Gionis, Flavio P. Junqueira, Vanessa Murdock, Vassilis Plachouras und Fabrizio Silvestri**: Design Trade-Offs for Search Engine Caching. *ACM Transactions on the Web (TWEB)*, Band 2(4):20 [2008].

[12] **Berkant Barla Cambazoglu, Flavio P. Junqueira, Vassilis Plachouras, Scott Banachowski, Baoqiu Cui, Swee Lim und Bill Bridge**: A Refreshing Perspective of Search Engine Caching. In *Proceedings of the 19th International Conference on World Wide Web*, S. 181–190. ACM [2010].

[13] More Details on Today's Outage | Facebook, Sept. 2010. URL `http://www.facebook.com/note.php?note_id=431441338919`.

[14] **Jiahua He, Arun Jagatheesan, Sandeep Gupta, Jeffrey Bennett und Allan Snavely**: Dash: A Recipe for a Flash-Based Data Intensive Supercomputer. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, S. 1–11. IEEE Computer Society [2010].

[15] **Bin Shao, Haixun Wang und Yatao Li**: The Trinity Graph Engine. URL `http://research.microsoft.com/pubs/161291/trinity.pdf`. Unpublished.

[16] **John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann und Ryan Stutsman**: The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, Band 43(4):92–105 [2010]. ISSN 0163-5980.

[17] **Peter J. Braam und Michael J. Callahan**: Lustre: A SAN File System for Linux [1999].

[18] **Peter J Braam**: The Coda Distributed File System. *Linux Journal*, Band 50(6):10–20 [1998].

[19] **Alex Davies und Alessandro Orsaria**: Scale Out with GlusterFS. *Linux Journal*, Band 2013(235):1 [2013].

[20] **Athicha Muthitacharoen, Robert Morris, Thomer M. Gil und Benjie Chen**: Ivy: A Read/Write Peer-to-Peer File System. *ACM SIGOPS Operating Systems Review*, Band 36(SI):31–44 [2002].

[21] **Sanjay Ghemawat, Howard Gobioff und Shun-Tak Leung**: The Google File System. In *ACM SIGOPS Operating Systems Review*, Band 37, S. 29–43. ACM [2003].

[22] **Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes und Robert E. Gruber**: Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, Band 26(2):4:1–4:26 [2008]. ISSN 0734-2071.

[23] **Konstantin Shvachko, Hairong Kuang, Sanjay Radia und Robert Chansler**: The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, S. 1–10. IEEE [2010].

[24] HBase Homepage. URL `http://hbase.apache.org/`.

[25] **Bogdan Nicolae**: BlobSeer: Towards Efficient Data Storage Management for Large-Scale, Distributed Systems. Dissertation, Université Rennes [2010].

[26] **Bill Nowicki**: NFS: Network File System Protocol Specification. Technischer Bericht [1989].

[27] **Christopher R Hertel**: Implementing CIFS: The Common Internet File System. Prentice Hall Professional [2004].

[28] IBM DB2 Database Software. URL `http://www-01.ibm.com/software/data/db2/`.

[29] MySQL Homepage. URL `http://www.mysql.com`.

[30] H2 Database Engine. URL `http://www.h2database.com/html/main.html`.

[31] Oracle Database Homepage. URL `https://www.oracle.com/database/index.html`.

[32] ObjectDB Homepage. URL `http://www.objectdb.com`.

[33] **Paul Butterworth, Allen Otis und Jacob Stein**: The GemStone Object Database Management System. *Communications of the ACM*, Band 34(10):64–77 [1991].

[34] PostgreSQL Homepage. URL `http://www.postgresql.org`.

[35] Microsoft SQL Server Homepage. URL `http://www.microsoft.com/de-de/server-cloud/products/sql-server/default.aspx`.

[36] Neo4j Homepage. URL `http://www.neo4j.com`.

[37] OrientDB Homepage. URL `http://www.orientdb.com`.

[38] Redis Homepage. URL `http://www.redis.io`.

[39] **Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall und Werner Vogels**: Dynamo: Amazon's Highly Available Key-Value Store. *SIGOPS Oper. Syst. Rev.*, Band 41(6):205–220 [2007]. ISSN 0163-5980.

[40] **Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy und Thomas Anderson**: Scalable Consistency in Scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, S. 15–28. ACM, New York, NY, USA [2011]. ISBN 978-1-4503-0977-6.

[41] **Sudipto Das, Divyakant Agrawal und Amr El Abbadi**: G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, S. 163–174. ACM, New York, NY, USA [2010]. ISBN 978-1-4503-0036-0.

[42] **Eelco Plugge, Tim Hawkins und Peter Membrey**: The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing. Apress, Berkely, CA, USA, 1. Auflage [2010]. ISBN 1430230517, 9781430230519.

[43] **J. Chris Anderson, Jan Lehnardt und Noah Slater**: CouchDB: The Definitive Guide. O'Reilly Media, Inc. [2010].

[44] **Avinash Lakshman und Prashant Malik**: Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, Band 44(2):35–40 [2010]. ISSN 0163-5980.

[45] **Thorsten Schütt, Florian Schintke und Alexander Reinefeld**: Scalaris: Reliable Transactional P2P Key/Value Store. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, S. 41–48. ACM [2008].

[46] **Taeho Kgil und Trevor Mudge**: FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, S. 103–112. ACM [2006].

[47] **Michael Wu und Willy Zwaenepoel**: eNVy: A Non-Volatile, Main Memory Storage System. In *ACM SigPlan Notices*, Band 29, S. 86–97. ACM [1994].

[48] **Philip A. Bernstein, Colin W. Reid und Sudipto Das**: Hyder - A Transactional Record Manager for Shared Flash. In *5th Biennial Conference on Innovative Data Systems Research*, CIDR'11, S. 9–20. Asilomar, California, USA [2011].

[49] **Youngjae Kim, Aayush Gupta, Bhuvan Urgaonkar, Piotr Berman und Anand Sivasubramaniam**: HybridStore: A Cost-Efficient, High-Performance Storage System Combining SSDs and HDDs. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*, S. 227–236. IEEE [2011].

[50] Memcached Homepage. URL `http://www.memcached.org`.

[51] Hazelcast Homepage. URL `http://www.hazelcast.com`.

[52] Ehcache Homepage. URL `http://www.ehcache.org`.

[53] **Mat Johns**: Getting Started with Hazelcast. Packt Publishing Ltd [2013].

[54] **Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker und Ion Stoica**: Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, S. 10–10. USENIX Association, Berkeley, CA, USA [2010].

[55] **Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh und Christof Bornhövd**: Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, S. 731–742. ACM, New York, NY, USA [2012]. ISBN 978-1-4503-1247-9.

[56] **Hasso Plattner**: SanssouciDB: An In-Memory Database for Processing Enterprise Workloads. In *BTW*, Band 20, S. 2–21 [2011].

[57] **Alfons Kemper und Thomas Neumann**: HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *2011 IEEE 27th International Conference on Data Engineering (ICDE)*, S. 195–206. IEEE [2011].

[58] **Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang et al.**: H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, Band 1(2):1496–1499 [2008].

[59] **Florian Klein und Michael Schöttner**: DXRAM: A Persistent In-Memory Storage for Billions of Small Objects. In *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, S. 103–110 [2013].

[60] **Dong Dai, Xi Li, Chao Wang, Mingming Sun und Xuehai Zhou**: Sedna: A Memory Based Key-Value Storage System for Realtime Processing in Cloud. In *2012 IEEE International Conference on Cluster Computing Workshops (CLUSTER WORKSHOPS)*, S. 48–56 [2012].

[61] **Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro und Orion Hodson**: FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, S. 401–414. USENIX Association, Seattle, WA [2014]. ISBN 978-1-931971-09-6.

[62] **Yiming Zhang, Chuanxiong Guo, Rui Chu, Guohan Lu, Yongqiang Xiong und Haitao Wu**: RAMCube: Exploiting Network Proximity for RAM-Based Key-Value Store. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, S. 5–5. USENIX Association, Berkeley, CA, USA [2012].

[63] **Bin Shao, Haixun Wang und Yatao Li**: Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13. New York, NY, USA [2013]. ISBN 978-1-4503-2037-5.

[64] **Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum und John K. Ousterhout**: It's Time for Low Latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, S. 11. USENIX Association [2011].

[65] **Lu Wang, Yanghua Xiao, Bin Shao und Haixun Wang**: How to Partition a Billion-Node Graph. Technischer Bericht MSR-TR-2013-102 [2013].

[66] **Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang und Songwu Lu**: BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers. *ACM SIGCOMM Computer Communication Review*, Band 39(4):63–74 [2009].

[67] **Florian Klein, Kevin Beineke und Michael Schöttner**: Memory Management for Billions of Small Objects in a Distributed In-Memory Storage. In *IEEE Cluster 2014* [2014].

[68] **Florian Klein, Kevin Beineke und Michael Schöttner**: Effiziente verteilte Metadaten-Verwaltung auf Basis von ID-Bereichen in DXRAM. In *Informatiktage 2014* [2014].

[69] **Florian Klein, Kevin Beineke und Michael Schöttner**: Distributed Range-Based Meta-Data Management for an In-Memory Storage. In *4th Workshop on Big Data Management in Clouds* [2015].

[70] **Brendan O'Connor**: How Much Text Versus Metadata Is in a Tweet? URL `http://gist.github.com/brendano/1024217`.

[71] **Mike Krieger**: Storing Hundreds of Millions of Simple Key-Value Pairs in Redis. URL `http://instagram-engineering.tumblr.com/post/12202313862/storing-hundreds-of-millions-of-simple-key-value`.

[72] **Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao und Jianzhong Li**: Efficient Subgraph Matching on Billion Node Graphs. *Proc. VLDB Endow.*, Band 5(9):788–799 [2012]. ISSN 2150-8097.

[73] **K.P. Birman, D.A. Freedman, Qi Huang und P. Dowell**: Overcoming CAP with Consistent Soft-State Replication. *Computer*, Band 45(2):50–58 [2012]. ISSN 0018-9162.

[74] **Simon S.Y. Shim**: Guest Editor's Introduction: The CAP Theorem's Growing Impact. *Computer*, Band 45(2):21–22 [2012]. ISSN 0018-9162.

[75] **Florian Klein, Kevin Beineke und Michael Schöttner**: Asynchronous Logging and Fast Recovery for a Large-Scale Distributed In-Memory Storage. In *INFORMATIK 2014* [2014].

[76] **Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek und Hari Balakrishnan**: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, S. 149–160. ACM, New York, NY, USA [2001]. ISBN 1-58113-411-8.

[77] **Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee und Young Ik Eom**: SFS: Random Write Considered Harmful in Solid State Drives. In *FAST*, S. 12 [2012].

[78] **Feng Chen, David A. Koufaty und Xiaodong Zhang**: Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *ACM SIGMETRICS Performance Evaluation Review*, Band 37, S. 181–192. ACM [2009].

[79] **Mendel Rosenblum und John K Ousterhout**: The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, Band 10(1):26–52 [1992].

[80] **Stephen M. Rumble et al.**: Log-Structured Memory for DRAM-Based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. Santa Clara, CA [2014]. ISBN ISBN 978-1-931971-08-9.

[81] **Stephen M. Rumble**: Memory and Object Management in RAMCloud. Dissertation, Stanford University [2014].

[82] **Liang He, Bin Shao, Yatao Li und Enhong Chen**: Distributed Real-Time Knowledge Graph Serving. In *2015 International Conference on Big Data and Smart Computing (BigComp)*, S. 262–265. IEEE [2015].

[83] **Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao und Zhongyuan Wang**: A Distributed Graph Engine for Web Scale RDF Data. *Proceedings of the VLDB Endowment*, Band 6(4):265–276 [2013].

[84] **Brent Callaghan, Theresa Lingutla-Raj, Alex Chiu, Peter Staubach und Omer Asad**: NFS over RDMA. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O Convergence: Experience, Lessons, Implications*, S. 196–208. ACM [2003].

[85] Distributed Data Management with VMware vFabric GemFire. Technischer Bericht, VMware, Inc., Palo Alto, CA, USA [2011]. URL `http://www.vmware.com/files/pdf/VMware-vFabric-GemFire-Distributed-Data-Management-TWP.pdf`.

[86] **Inc. GemStone Systems**: GemFire Enterprise Architectural Overview [2006]. URL `http://www.gemstone.com/pdf/GemFire_Architecture.pdf`.

[87] vFabric GemFire 6.6 Documentation. URL `http://pubs.vmware.com/vfabric5/index.jsp?topic=/com.vmware.vfabric.gemfire.6.6/about_gemfire.html`.

[88] Memcached. URL `https://en.wikipedia.org/wiki/Memcached`.

[89] **Avinash Lakshman**: Cassandra – A Structured Storage System on a P2P Network. URL `https://www.facebook.com/notes/facebook-engineering/cassandra-a-structured-storage-system-on-a-p2p-network/24413138919`.

[90] **Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch und Christos Karamanolis**: Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *ACM SIGOPS Operating Systems Review*, Band 41, S. 159–174. ACM [2007].

[91] **Xingbo Wu, Yuehai Xu, Zili Shao und Song Jiang**: LSM-Trie: An LSM-treebased Ultra-Large Key-Value Store for Small Data. In *USENIX Annual Technical Conference* [2015].

[92] **Richard Von Mises**: Über Aufteilungs-und Besetzungswahrscheinlichkeiten [1939].

[93] **Martin Raab und Angelika Steger**: “Balls into Bins”—A Simple and Tight Analysis. In *Randomization and Approximation Techniques in Computer Science*, S. 159–170. Springer [1998].

[94] **Mehedi Masud, Gopal Chandra Das, Anisur Rahman und Arunashis Ghose**: A Hashing Technique Using Separate Binary Tree. *Data Science Journal*, Band 5:143–161 [2006].

[95] **Nikolas Askitis**: Fast and Compact Hash Tables for Integer Keys. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC '09. Darlinghurst, Australia, Australia [2009]. ISBN 978-1-920682-72-9.

[96] **Mahima Singh und Deepak Garg**: Choosing Best Hashing Strategies and Hash Functions. In *Advance Computing Conference, 2009. IACC 2009. IEEE International*, S. 50–55. IEEE [2009].

[97] **Rasmus Pagh und Flemming Friche Rodler**: Cuckoo Hashing. *Journal of Algorithms*, Band 51(2):122–144 [2004].

[98] **Dimitris Fotakis, Rasmus Pagh, Peter Sanders und Paul Spirakis**: Space Efficient Hash Tables with Worst Case Constant Access Time. In *STACS 2003*, Band 2607 von *Lecture Notes in Computer Science* [2003]. ISBN 978-3-540-00623-7.

[99] **Andrea W. Richa, M. Mitzenmacher und R. Sitaraman**: The Power of Two Random Choices: A Survey of Techniques and Results. *Combinatorial Optimization*, Band 9:255–304 [2001].

[100] **Pedro Celis, P.-A. Larson und J. Ian Munro**: Robin Hood Hashing. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*, S. 281–288. IEEE [1985].

[101] **Maurice Herlihy, Nir Shavit und Moran Tzafrir**: Hopscotch Hashing. In *Distributed Computing*, S. 350–364. Springer [2008].

[102] **Aaro M. Tenenbaum**: Data Structures Using C. Pearson Education India [1990].

[103] **Martin Faust, David Schwalb, Jens Krueger und Hasso Plattner**: Fast Lookups for In-Memory Column Stores: Group-Key Indices, Lookup and Maintenance. In *ADMS'12: Proceedings of the 3rd International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures at VLDB'12* [2012].

[104] **Tapio Lahdenmaki und Mike Leach**: Relational Database Index Design and the Optimizers. John Wiley & Sons [2005].

[105] **Hongjun Lu, Yuet Yeung Ng und Zengping Tian**: T-Tree or B-Tree: Main Memory Database Index Structure Revisited. In *Database Conference, 2000. ADC 2000. Proceedings. 11th Australasian*, S. 65–73 [2000].

[106] **Kong-Rim Choi und Kyung-Chang Kim**: T*-Tree: A Main Memory Database Index Structure for Real Time Applications. In *Proceedings of Third International Workshop on Real-Time Computing Systems and Applications*, S. 81–88. IEEE [1996].

[107] B-Tree: The Database Problem. URL https://en.wikipedia.org/wiki/B-tree#The_database_problem.

[108] **Yinan Li, Bingsheng He, Qiong Luo und Ke Yi**: Tree Indexing on Flash Disks. In *IEEE 25th International Conference on Data Engineering. ICDE'09.*, S. 1303–1306. IEEE [2009].

[109] **René B. Jørgensen, Martin V .and Rasmussen, Simonas Šaltenis und Carsten Schjønning**: FB-Tree: A B+-tree for Flash-Based SSDs. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, S. 34–42. ACM [2011].

[110] **GAPJOO NA, Bongki Moon und Sang-Won Lee**: IPL B-Tree for Flash Memory Database Systems. *Journal of information science and engineering*, Band 27:111–127 [2011].

[111] **David Detlefs, Al Dosser und Benjamin Zorn**: Memory Allocation Costs in Large C and C++ Programs. *Software: Practice and Experience*, Band 24(6):527–542 [1994]. ISSN 1097-024X.

[112] **Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe und Paul R. Wilson**: Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGOPS Oper. Syst. Rev.*, Band 34(5):117–128 [2000]. ISSN 0163-5980.

[113] **Sanjay Ghemawat und Paul Menage**: TCMalloc : Thread-Caching Malloc. URL http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[114] **Jason Evans**: A Scalable Concurrent malloc(3) Implementation for FreeBSD [2006].

[115] **Andrew S. Tanenbaum**: Modern Operating Systems [2009].

[116] **Ralph Hülsenbusch**: Multicore-Techniken im Vergleich - Krieg der Kerne [2006].

[117] **Albert Y. Zomaya und Young Choon Lee**: Energy Efficient Distributed Computing Systems. Wiley [2012]. ISBN 978-0470908754.

[118] **Andrew S. Tanenbaum und Herbert Bos**: Modern Operating Systems. Prentice Hall [2014]. ISBN 013359162X.

[119] Intel® Xeon® Processor E5-2699 v3. URL http://ark.intel.com/products/81061/Intel-Xeon-Processor-E5-2699-v3-45M-Cache-2_30-GHz.

[120] **Herb Sutter**: The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software [2005].

[121] **Edsger W Dijkstra**: Solution of a Problem in Concurrent Programming Control. In *Pioneers and Their Contributions to Software Engineering*, S. 289–294. Springer [2001].

[122] **Peter Mandl**: Grundkurs Betriebssysteme: Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation. Springer-Verlag [2013].

[123] **Robert H.B. Netzer und Barton P. Miller**: What Are Race Conditions?: Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, Band 1(1):74–88 [1992].

[124] **Leslie Lamport**: The Mutual Exclusion Problem: PartII — Statement and Solutions. *Journal of the ACM (JACM)*, Band 33(2):327–348 [1986].

[125] **James H Anderson**: Lamport on Mutual Exclusion: 27 Years of Planting Seeds. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, S. 3–12. ACM [2001].

[126] **Maurice Herlihy und Nir Shavit**: The Art of Multiprocessor Programming. In *PODC*, Band 6, S. 1–2 [2006].

[127] **Boleslaw K. Szymanski**: A Simple Solution to Lamport's Concurrent Programming Problem with Linear Wait. In *Proceedings of the 2nd International Conference on Supercomputing*, S. 621–626. ACM [1988].

[128] **Boleslaw K. Szymanski**: Mutual Exclusion Revisited. In *Proceedings of the 5th Jerusalem Conference on Information Technology*, S. 110–117. IEEE [1990].

[129] **Carsten Schroeder**: Sperrfreie Algorithmen für Mehrere Threads. Bachelor thesis, Heinrich-Heine-Universität Düsseldorf, Germany [2015].

[130] **Per Brinch Hansen, Edsger W Dijkstra und CAR Hoare**: The Origins of Concurrent Programming: From Semaphores to Remote Procedure Calls. Springer-Verlag New York, Inc. [2002].

[131] **Edsger W. Dijkstra**: Hierarchical Ordering of Sequential Processes. *Acta informatica*, Band 1(2):115–138 [1971].

[132] **Gary L. Peterson**: Myths About the Mutual Exclusion Problem. *Information Processing Letters*, Band 12(3):115–116 [1981].

[133] **Leslie Lamport**: A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, Band 17(8):453–455 [1974].

[134] **Nancy A Lynch**: Distributed Algorithms. Morgan Kaufmann [1996].

[135] **Doug Lea**: The java.util.concurrent Synchronizer Framework. *Science of Computer Programming*, Band 58(3):293–309 [2005].

[136] **Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes und Doug Lea**: Java Concurrency in Practice. Addison-Wesley [2006].

[137] **Scott Oaks und Henry Wong**: Java Threads. O'Reilly Media, Inc. [2004].

[138] **Bill Joy, Guy L Steele Jr, James Gosling und Gilad Bracha**: The Java Language Specification [1998].

[139] **Thomas J. Bergin und Richard G. Gibson**: History of Programming Languages II. ACM Press New York [1996].

[140] **Charles Antony Richard Hoare**: Monitors: An Operating System Structuring Concept. *Communications of the ACM*, Band 17(10):549–557 [1974].

[141] **Douglas Lea**: Concurrent Programming in Java: Design Principles and Patterns. Addison-Wesley Professional [2000].

[142] **Mendel Rosenblum und John K Ousterhout**: The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, Band 10(1):26–52 [1992].

[143] Wolfram Gloger's malloc Homepage. URL `http://www.malloc.de/en/`.

[144] **Hans-J. Boehm**: BoehmGC. URL `http://www.hboehm.info/gc/`.

[145] **Doug Lea und Wolfram Gloger**: A Memory Allocator [1996].

[146] **Sebastian Kniesburges, Andreas Koutsopoulos und Christian Scheideler**: Re-Chord: A Self-stabilizing Chord Overlay Network. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, S. 235–244. ACM, New York, NY, USA [2011]. ISBN 978-1-4503-0743-7.

[147] **Sean Christopher Rhea, Dennis Geels, Timothy Roscoe und John Kubiatowicz**: Handling Churn in a DHT. Computer Science Division, University of California [2003].

[148] **T.M. Shafaat, A. Ghodsi und S. Haridi**: Handling Network Partitions and Mergers in Structured Overlay Networks. In *Seventh IEEE International Conference on Peer-to-Peer Computing*, S. 132–139 [2007].

[149] **Jeffrey Dean**: Underneath the Covers at Google: Current Systems and Future Directions. *Google I/O*, Band 8 [2008].

[150] **Sakti P. Ghosh und Michael E. Senko**: File Organization: On the Selection of Random Access Index Points for Sequential Files. *Journal of the ACM (JACM)*, Band 16(4):569–579 [1969].

[151] **R. Bayer und E.M. McCreight**: Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, Band 1(3):173–189 [1972]. ISSN 0001-5903.

[152] **Rudolf Bayer**: Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta informatica*, Band 1(4):290–306 [1972].

[153] **Kim S. Larsen und Rolf Fagerberg**: Efficient Rebalancing of B-Trees with Relaxed Balance. *International Journal of Foundations of Computer Science*, Band 7(02):169–186 [1996].

[154] **Ryan Scott Stutsman**: Durability and Crash Recovery in Distributed In-Memory Storage Systems. Dissertation, Stanford University [2013].

[155] **Giorgos Margaritis und Stergios V Anastasiadis**: Efficient Range-Based Storage Management for Scalable Datastores. *IEEE Transactions on Parallel and Distributed Systems*, Band 25(11):2851–2866 [2014].

[156] **Bogdan Nicolae, Gabriel Antoniu und Luc Bougé**: Enabling High Data Throughput in Desktop Grids Through Decentralized Data and Metadata Management: The BlobSeer Approach. In *Euro-Par 2009 Parallel Processing*, S. 404–416. Springer [2009].

[157] **Bogdan Nicolae, Gabriel Antoniu und Luc Bougé**: BlobSeer: Efficient Data Management for Data-Intensive Applications Distributed at Large-Scale. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, S. 1–4. IEEE [2010].

[158] **Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise und Alexandra Carpen-Amarie**: BlobSeer: Next-Generation Data Management for Large Scale Infrastructures. *Journal of Parallel and Distributed Computing*, Band 71(2):169–184 [2011].

[159] **Changxi Zheng, Guobin Shen, Shipeng Li und Scott Shenker**: Distributed Segment Tree: Support of Range Query and Cover Query over DHT. In *IPTPS* [2006].

[160] **Mark De Berg, Marc Van Kreveld, Mark Overmars und Otfried Cheong Schwarzkopf**: Computational Geometry. Springer [2000].

[161] **Marcos K. Aguilera, Wojciech Golab und Mehul A. Shah**: A Practical Scalable Distributed B-Tree. *Proc. VLDB Endow.*, Band 1(1):598–609 [2008]. ISSN 2150-8097.

[162] **Benjamin Sowell, Wojciech Golab und Mehul A. Shah**: Minuet: A Scalable Distributed Multiversion B-Tree. *Proceedings of the VLDB Endowment*, Band 5(9):884–895 [2012].

[163] **Ashwin R. Bharambe, Mukesh Agrawal und Srinivasan Seshan**: Mercury: Supporting Scalable Multi-Attribute Range Queries. In *ACM SIGCOMM Computer Communication Review*, Band 34, S. 353–366. ACM [2004].

[164] **Artur Andrzejak und Zhichen Xu**: Scalable, Efficient Range Queries for Grid Information Services. In *Proceedings of Second International Conference on Peer-to-Peer Computing*, S. 33–40. IEEE [2002].

[165] **Praveen Yalagandula und J. Browne**: Solving Range Queries in a Distributed System. *Dept. of Computer Sciences, University of Texas at Austin* [2004].

[166] **Adina Crainiceanu, Prakash Linga, Johannes Gehrke und Jayavel Shanmugasundaram**: Querying Peer-to-Peer Networks Using P-Trees. In *Proceedings of the 7th International Workshop on the Web and Databases*, S. 25–30. ACM [2004].

[167] **Yatin Chawathe, Sriram Ramabhadran, Sylvia Ratnasamy, Anthony LaMarca, Scott Shenker und Joseph Hellerstein**: A Case Study in Building Layered DHT Applications. *ACM SIGCOMM Computer Communication Review*, Band 35(4):97–108 [2005].

[168] **James Aspnes und Gauri Shah**: Skip Graphs. *ACM Transactions on Algorithms (TALG)*, Band 3(4):37 [2007].

[169] **Jürg Nievergelt, Hans Hinterberger und Kenneth C Sevcik**: The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems (TODS)*, Band 9(1):38–71 [1984].

[170] Understanding Memory Management. URL `https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html`.

[171] Java (JVM) Memory Model and Garbage Collection Monitoring Tuning. URL `http://www.journaldev.com/2856/java-jvm-memory-model-and-garbage-collection-monitoring-tuning`.

[172] **Jeremy Manson**: The Java Memory Model. Dissertation, College Park, MD, USA [2004].

[173] Wikipedia: C Dynamic Memory Allocation. URL `http://en.wikipedia.org/w/index.php?title=C_dynamic_memory_allocation&oldid=594269868`.

[174] The GNU C Library. URL http://www.gnu.org/software/libc/manual/html_mono/libc.html#Memory.

[175] **Chuck Lever und David Boreham**: Malloc() Performance in a Multithreaded Linux Environment [2000].

[176] **Jipan Yang**: GLibc Malloc Internal (1): Arena, Bin, Chunk and Sub Heap [2014]. URL https://jipanyang.wordpress.com/2014/06/09/glibc-malloc-internal-arena-bin-chunk-and-sub-heap-1/.

[177] Understanding glibc Malloc [2015]. URL https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/.

[178] Scalable Memory Allocation Using jemalloc. URL http://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919.

[179] **James Golick**: How tcmalloc Works. URL http://jamesgolick.com/2013/5/19/how-tcmalloc-works.html.

[180] **Sumita Barahmand und Shahram Ghandeharizadeh**: BG: A Benchmark to Evaluate Interactive Social Networking Actions. In *CIDR*. Citeseer [2013].

[181] **Kristina Chodorow**: Scaling MongoDB. O'Reilly Media, Inc. [2011].

[182] **Sumita Barahmand, Shahram Ghandeharizadeh und Jason Yap**: A Comparison of Two Physical Data Designs for Interactive Social Networking Actions. In *Proceedings of the 22nd ACM International Conference on Conference on Information & Knowledge Management*, CIKM '13, S. 949–958. ACM, New York, NY, USA [2013]. ISBN 978-1-4503-2263-8.

Anhang A

Messergebnisse

Nachfolgend werden die in Kapitel 5 verwendeten Messergebnisse tabellarisch aufgeführt.

A.1 Lokale Metadaten-Verwaltung

Die Messergebnisse beziehen sich auf die Evaluierungen in Abschnitt 5.2.

A.1.1 Speicherverwaltung

Objektgröße	**DXRAM** *(in GB)*	**Java** *(in GB)*	**glibc** *(in GB)*	**hoard** *(in GB)*	**jemalloc** *(in GB)*	**tcmalloc** *(in GB)*	**boehmgc** *(in GB)*
16 B	11,250	19,981	20,004	10,039	10,231	10,067	10,011
20 B	11,000	19,981	16,004	16,056	16,364	16,091	16,011
24 B	10,833	16,668	13,337	13,383	13,367	13,416	13,344
28 B	10,714	17,174	17,147	11,474	11,692	11,503	17,153
32 B	10,625	14,997	15,004	10,042	10,231	10,067	15,010
36 B	10,556	15,529	13,337	17,838	13,637	13,429	13,344
40 B	10,500	13,992	12,004	16,056	12,278	12,087	12,011
44 B	10,455	14,589	14,549	14,598	11,161	10,986	14,556
48 B	10,417	13,362	13,337	13,383	10,231	10,077	13,344
52 B	10,385	13,874	12,312	12,355	12,590	12,383	12,318
56 B	10,357	12,830	11,433	11,474	11,692	11,503	11,439
60 B	10,333	13,354	13,337	10,710	10,914	10,733	13,344
64 B	10,313	12,539	12,504	10,042	10,231	10,067	12,510
gleichverteilt	10,500	14,876	13,882	13,192	12,110	11,915	13,888

Tabelle A.1: Speicherverbrauch Allokatoren (gesamt).

A.1.2 CID-Tabellen

Füllgrad 40%

Anzahl Objekte	CID-Tabellen *(in MB)*	Java HashMap *(in MB)*	Cuckoo Hashing *(in MB)*
419.430	2,1	26,4	16,0
838.860	4,1	52,8	32,0
1.677.721	8,1	105,6	64,0
3.355.443	16,1	211,2	128,0
6.710.886	32,1	422,4	256,0
13.421.773	64,1	844,8	512,0
26.843.546	128,1	1.689,6	1.024,0
53.687.092	256,2	3.379,2	2.048,0
107.374.184	512,4	6.758,4	4.096,0
214.748.368	1.024,0	-	8.192,0
429.496.736	2.048,0	-	16.384,0

Tabelle A.2: Speicherverbrauch (Füllgrad 40%).

Anzahl Objekte	CID-Tabellen			Java HashMap			Cuckoo Hashing		
	Min. *(in ms)*	Max. *(in ms)*	Ø *(in ms)*	Min. *(in ms)*	Max. *(in ms)*	Ø *(in ms)*	Min. *(in ms)*	Max. *(in ms)*	Ø *(in ms)*
419.430	5	13	5	6	28	6	27	125	47
838.860	9	19	10	12	48	14	55	321	107
1.677.721	19	31	20	24	74	26	138	636	233
3.355.443	39	54	40	49	152	53	315	1.078	454
6.710.886	87	103	90	99	845	124	695	3.227	1.026
13.421.773	159	176	160	195	1.301	516	1.380	5.359	2.069
26.843.546	318	343	320	397	2.913	1.191	3.140	6.505	4.589
53.687.092	638	657	644	1.944	3.338	2.725	6.018	17.526	9.016
107.374.184	1.281	1.312	1.285	3.671	7.911	5.425	17.788	39.642	23.316
214.748.368	2.567	2.599	2.583	-	-	-	36.725	78.053	48.319
429.496.736	5.136	5.162	5.148	-	-	-	90.540	255.696	139.729

Tabelle A.3: Ausführungszeit (Füllgrad 40%).

Füllgrad 75%

Anzahl Objekte	CID-Tabellen (in MB)	Java HashMap (in MB)	Cuckoo Hashing (in MB)
786.432	3,8	46,0	16,0
1.572.864	7,6	92,0	32,0
3.145.728	15,1	184,0	64,0
6.291.456	30,1	368,0	128,0
12.582.912	60,1	736,0	256,0
25.165.824	120,1	1.472,0	512,0
50.331.648	240,2	2.944,0	1.024,0
100.663.296	480,4	5.888,0	2.048,0
201.326.592	960,6	11.766,0	4.096,0
402.653.184	1.921,2	-	8.192,0
805.306.368	3.842,4	-	16.384,0

Tabelle A.4: Speicherverbrauch (Füllgrad 75%).

Anzahl Objekte	CID-Tabellen			Java HashMap			Cuckoo Hashing		
	Min. (in ms)	Max. (in ms)	Ø (in ms)	Min. (in ms)	Max. (in ms)	Ø (in ms)	Min. (in ms)	Max. (in ms)	Ø (in ms)
786.432	9	19	9	11	40	12	78	2.618	131
1.572.864	8	29	18	22	68	25	215	337	245
3.145.728	37	50	37	45	123	47	419	801	506
6.291.456	74	93	74	91	230	97	922	1.496	1.060
12.582.912	150	167	152	179	447	252	1.989	3.876	2.396
25.165.824	300	318	302	362	1.429	715	4.241	9.832	5.004
50.331.648	597	617	600	775	1.821	907	11.042	33.111	15.179
100.663.296	1.194	1.214	1.202	3.166	32.749	7.359	24.174	31.688	25.819
201.326.592	2.414	2.432	2.418	9.741	48.106	26.000	59.420	99.771	70.077
402.653.184	4.814	4.831	4.819	-	-	-	120.253	226.586	158.091
805.306.368	9.615	9.743	9.632	-	-	-	265.802	359.253	315.563

Tabelle A.5: Ausführungszeit (Füllgrad 75%).

Füllgrad 90%

Anzahl Objekte	CID-Tabellen (in MB)	Java HashMap (in MB)	Cuckoo Hashing (in MB)
943.718	4,6	54,4	16,0
1.887.436	9,1	108,8	32,0
3.774.873	18,1	217,6	64,0
7.549.747	36,1	435,2	128,0
15.099.494	72,1	870,4	256,0
30.198.988	144,1	1.740,8	512,0
60.397.976	288,2	3.481,6	1.024,0
120.795.952	576,4	6.963,2	2.048,0
241.591.904	1.152,8	13.926,4	4.096,0
483.183.808	2.305,6	-	-
966.367.616	4.611,2	-	-

Tabelle A.6: Speicherverbrauch (Füllgrad 90%).

Anzahl Objekte	CID-Tabellen			Java HashMap			Cuckoo Hashing		
	Min. (in ms)	Max. (in ms)	Ø (in ms)	Min. (in ms)	Max. (in ms)	Ø (in ms)	Min. (in ms)	Max. (in ms)	Ø (in ms)
943.718	11	20	11	13	46	15	236	739	298
1.887.436	23	34	27	28	82	39	529	604	561
3.774.873	47	60	49	56	148	70	1.074	1.424	1.169
7.549.747	90	108	92	110	281	133	2.260	4.394	2.611
15.099.494	180	198	182	220	534	261	4.020	6.030	4.565
30.198.988	361	379	362	436	1.037	511	8.940	11.699	10.047
60.397.976	717	737	719	864	2.028	1.009	21.327	35.131	24.691
120.795.952	1.446	1.480	1.453	3.942	8.376	6.485	54.217	64.022	58.478
241.591.904	2.887	2.903	2.892	10.697	72.362	42.640	149.807	153.549	151.678
483.183.808	5.755	5.817	5.787	-	-	-	-	-	-
966.367.616	11.518	11.549	11.539	-	-	-	-	-	-

Tabelle A.7: Ausführungszeit (Füllgrad 90%).

A.1.3 CID-Tabellen und Speicherverwaltung

Objekt-größe	Speicher-bedarf	Metadaten pro Objekt	Put		Get	
			Operationen pro Sekunde	Daten pro Sekunde	Operationen pro Sekunde	Daten pro Sekunde
	(in GB)	*(in Byte)*		*(in MB)*		*(in MB)*
16	5,80	7,003236044198275	14.904.503	227,425	13.452.551	205,270
17	6,05	7,003236047923560	14.845.882	240,688	13.438.008	217,863
18	6,30	7,003236051648850	14.773.216	253,599	13.344.876	229,080
19	6,55	7,003236055374140	14.665.045	265,728	13.214.455	239,443
20	6,80	7,003236059099430	14.491.783	276,409	13.086.986	249,614
21	7,05	7,003236062824720	14.539.275	291,180	13.037.116	261,096
22	7,30	7,003236066550010	14.344.240	300,954	12.988.809	272,516
23	7,55	7,003236070275300	14.285.166	313,338	12.868.594	282,266
24	7,80	7,003236074000597	14.204.655	325,119	12.794.493	292,843
25	8,05	7,003236077725880	14.114.309	336,511	12.685.954	302,457
26	8,30	7,003236081451170	14.023.711	347,725	12.633.430	313,253
27	8,55	7,003236085176460	13.878.205	357,353	12.523.166	322,462
28	8,80	7,003236088901750	13.765.430	367,577	12.474.381	333,102
29	9,05	7,003236092627040	13.792.911	381,464	12.397.611	342,875
30	9,30	7,003236096352330	13.774.696	394,097	12.322.314	352,544
31	9,55	7,003236100077620	13.539.934	400,293	12.215.985	361,152
32	9,80	7,003236122429371	13.440.734	410,179	12.154.261	370,919
33	10,05	7,003236126154660	13.428.343	422,607	12.077.750	380,102
34	10,30	7,003236129879950	13.325.628	432,083	11.967.176	388,035
35	10,55	7,003236133605240	13.191.079	440,300	11.930.844	398,235
36	10,80	7,003236137330530	13.151.467	451,520	11.836.878	406,387
37	11,05	7,003236141055820	13.054.855	460,653	11.733.383	414,024
38	11,30	7,003236144781110	13.017.300	471,742	11.765.070	426,362
39	11,55	7,003236148506400	12.888.409	479,362	11.594.537	431,239
40	11,80	7,003236152231693	12.800.663	488,307	11.534.908	440,022
41	12,05	7,003236155956980	12.744.690	498,326	11.480.827	448,908
42	12,30	7,003236159682270	12.724.442	509,669	11.320.939	453,453
43	12,55	7,003236163407560	12.568.548	515,411	11.331.867	464,697
44	12,80	7,003236167132850	12.486.706	523,963	11.238.143	471,571
45	13,05	7,003236170858140	12.427.123	533,314	11.201.809	480,730
46	13,30	7,003236174583430	12.417.327	544,736	11.114.071	487,563
47	13,55	7,003236178308720	12.262.774	549,651	11.087.859	496,988
48	13,80	7,003236182034016	12.194.792	558,233	10.994.007	503,266
49	14,05	7,003236185759300	12.154.574	567,984	10.919.196	510,254
50	14,30	7,003236189484590	12.085.909	576,301	10.894.301	519,481

Tabelle A.8 – *Fortsetzung auf der nächsten Seite*

Tabelle A.8 – *Fortsetzung von der vorherigen Seite*

Objekt-größe	Speicher-bedarf (in GB)	Metadaten pro Objekt (in Byte)	Put: Operationen pro Sekunde	Put: Daten pro Sekunde (in MB)	Get: Operationen pro Sekunde	Get: Daten pro Sekunde (in MB)
51	14,55	7,003236193209880	12.026.518	584,938	10.734.040	522,076
52	14,80	7,003236196935170	11.957.903	593,005	10.741.438	532,679
53	15,05	7,003236200660460	11.896.551	601,308	10.673.447	539,487
54	15,30	7,003236204385750	11.831.618	609,310	10.638.192	547,850
55	15,55	7,003236208111040	11.769.238	617,321	10.568.716	554,351
56	15,80	7,003236211836338	11.733.005	626,610	10.502.358	560,886
57	16,05	7,003236215561620	11.607.848	630,996	10.486.216	570,025
58	16,30	7,003236219286910	11.566.721	639,791	10.412.918	575,971
59	16,55	7,003236223012200	11.475.486	645,689	10.395.283	584,909
60	16,80	7,003236226737490	11.394.427	651,994	10.288.123	588,691
61	17,05	7,003236230462780	11.347.261	660,117	10.220.691	594,580
62	17,30	7,003236234188070	11.329.457	669,886	10.215.782	604,037
63	17,55	7,003236237913360	11.240.663	675,356	10.126.371	608,407
64	17,80	7,010346006602049	11.219.184	684,765	10.109.658	617,045

Tabelle A.8: Speicherbedarf und Durchsatz der lokalen Metadaten-Verwaltung.

A.1.4 Vergleich mit RAMCloud

RAMCloud mit 80 Millionen Objekten pro Knoten

Anzahl Knoten	Speicher-bedarf (in GB)	Metadaten (in GB)	Zufällige Lesezugriffe 10.000 (in ms)	Zufällige Lesezugriffe 100.000 (in ms)	Zufällige Lesezugriffe 1.000.000 (in ms)	Sequentielle Lesezugriffe 10.000 (in ms)	Sequentielle Lesezugriffe 100.000 (in ms)	Sequentielle Lesezugriffe 1.000.000 (in ms)
1	15,4	12,4	787	8.467	85.233	320	3.090	29.881
2	30,7	24,7	908	8.449	83.663	319	3.051	29.848
4	61,4	49,4	927	8.609	86.581	315	3.012	29.820
8	122,9	98,9	963	9.224	92.300	298	2.891	29.023
16	245,8	197,8	1.487	14.722	145.434	413	4.018	39.937

Tabelle A.9: Speicherbedarf und Lesezugriffe (RAMCloud 80M).

DXRAM mit 80 Millionen Objekten pro Knoten

Anzahl Knoten	Speicher-bedarf (in GB)	Metadaten (in GB)	Zufällige Lesezugriffe			Sequentielle Lesezugriffe		
			10.000 (in ms)	100.000 (in ms)	1.000.000 (in ms)	10.000 (in ms)	100.000 (in ms)	1.000.000 (in ms)
1	3,6	0,6	86	274	2.430	34	130	2.073
2	7,1	1,1	95	301	2.058	37	129	2.057
4	14,2	2,2	114	345	2.119	36	165	1.843
8	28,3	4,3	126	424	2.151	37	142	1.559
16	56,6	8,6	225	683	2.644	58	190	2.356

Tabelle A.10: Speicherbedarf und Lesezugriffe (DXRAM 80M).

DXRAM mit 300 Millionen Objekten pro Knoten

Anzahl Knoten	Speicher-bedarf (in GB)	Metadaten (in GB)	Zufällige Lesezugriffe			Sequentielle Lesezugriffe		
			10.000 (in ms)	100.000 (in ms)	1.000.000 (in ms)	10.000 (in ms)	100.000 (in ms)	1.000.000 (in ms)
1	13,2	2,0	86	269	2.559	46	141	1.577
2	26,5	4,1	79	277	2.064	35	137	1.873
4	52,7	8,0	115	355	2.141	38	136	1.847
8	105,5	16,1	127	444	2.170	38	137	1.640
16	211,0	32,2	217	684	2.643	58	200	2.266

Tabelle A.11: Speicherbedarf und Lesezugriffe (DXRAM 300M).

A.2 Globale Metadaten-Verwaltung

Die Messergebnisse beziehen sich auf die Evaluierungen in Abschnitt 5.3.

A.2.1 CID-Baum

Speicherbedarf

Anzahl Objekte	CID-Baum *(in Byte)*	CID-Liste *(in Byte)*	Java HashMap *(in Byte)*	Cuckoo HT *(in Byte)*
1.000.000	555.0111	718.030	86.285.650	33.556.168
10.000.000	5.547.160	7.296.904	900.657.560	268.437.192
100.000.000	55.403.775	74.302.751	8.805.300.636	2.147.485.384
1.000.000.000	554.291.890	-	-	-

Tabelle A.12: Speicherbedarf (1% Migrationen).

Anzahl Objekte	CID-Baum *(in Byte)*	CID-Liste *(in Byte)*	Java HashMap *(in Byte)*	Cuckoo HT *(in Byte)*
1.000.000	2.671.410	3.471.275	86.285.888	33.556.168
10.000.000	26.686.162	35.316.217	900.657.801	268.437.192
100.000.000	266.996.370	359.875.879	8.805.300.879	2.147.485.384
1.000.000.000	2.670.010.148	-	-	-

Tabelle A.13: Speicherbedarf (5% Migrationen).

Anzahl Objekte	CID-Baum *(in Byte)*	CID-Liste *(in Byte)*	Java HashMap *(in Byte)*	Cuckoo HT *(in Byte)*
1.000.000	5.098.549	6.762.164	86.286.177	33.556.168
10.000.000	50.913.644	65.307.083	900.658.090	268.437.192
100.000.000	509.312.526	663.123.873	8.805.301.150	2.147.485.384
1.000.000.000	5.093.551.945	-	-	-

Tabelle A.14: Speicherbedarf (10% Migrationen).

Ausführungszeit Migrate-Operation

Anzahl Objekte	CID-Baum *(in ns)*	CID-Liste *(in ns)*	Java HashMap *(in ns)*	Cuckoo HT *(in ns)*
1.000.000	748	2.286	260	154
10.000.000	621	23.664	281	202
100.000.000	839	269.071	332	295
1.000.000.000	1.125	-	-	-

Tabelle A.15: Ausführungszeit Migrate-Operation (1% Migrationen).

Anzahl Objekte	CID-Baum *(in ns)*	CID-Liste *(in ns)*	Java HashMap *(in ns)*	Cuckoo HT *(in ns)*
1.000.000	636	10.381	237	143
10.000.000	763	118.354	275	202
100.000.000	1.027	1.967.943	788	294
1.000.000.000	1.352	-	-	-

Tabelle A.16: Ausführungszeit Migrate-Operation (5% Migrationen).

Anzahl Objekte	CID-Baum *(in ns)*	CID-Liste *(in ns)*	Java HashMap *(in ns)*	Cuckoo HT *(in ns)*
1.000.000	671	19.958	228	143
10.000.000	844	226.287	274	204
100.000.000	1.127	3.715.332	869	303
1.000.000.000	1.457	-	-	-

Tabelle A.17: Ausführungszeit Migrate-Operation (10% Migrationen).

Ausführungszeit Get-Operation

Anzahl Objekte	CID-Baum *(in ns)*	CID-Liste *(in ns)*	Java HashMap *(in ns)*	Cuckoo HT *(in ns)*
1.000.000	97	56	56	66
10.000.000	86	66	94	112
100.000.000	94	77	62	151
1.000.000.000	102	-	-	-

Tabelle A.18: Ausführungszeit Get-Operation (1% Migrationen).

Anzahl Objekte	CID-Baum *(in ns)*	CID-Liste *(in ns)*	Java HashMap *(in ns)*	Cuckoo HT *(in ns)*
1.000.000	105	60	56	66
10.000.000	96	68	96	111
100.000.000	105	76	61	145
1.000.000.000	116	-	-	-

Tabelle A.19: Ausführungszeit Get-Operation (5% Migrationen).

Anzahl Objekte	CID-Baum *(in ns)*	CID-Liste *(in ns)*	Java HashMap *(in ns)*	Cuckoo HT *(in ns)*
1.000.000	110	67	56	70
10.000.000	103	75	92	118
100.000.000	113	81	61	177
1.000.000.000	125	-	-	-

Tabelle A.20: Ausführungszeit Get-Operation (10% Migrationen).

A.2.2 Caching

1% Migrationen

Verteilung		Zeit pro Operation: Ohne Caching *(in µs)*	Zeit pro Operation: Mit Caching *(in µs)*	Cachegröße *(in Byte)*
Gleichverteilung		140.408	2.449	56.854.680
Normalverteilung	0,2	140.216	518	6.903.296
	0,5	140.256	867	16.479.464
	1,0	140.311	1.360	31.673.384
Zipf-Verteilung	1,0	140.186	1.875	56.813.912
	1,5	140.059	317	11.636.696
	2,0	140.057	107	230.672

Tabelle A.21: Caching (1% Migrationen, 1 Chunk pro Bereich).

Verteilung		Zeit pro Operation Ohne Caching *(in µs)*	Mit Caching *(in µs)*	Cachegröße *(in Byte)*
Gleichverteilung		140.233	541	5.935.056
Normalverteilung	0,2	140.164	285	799.464
	0,5	140.177	330	1.910.000
	1,0	140.190	391	3.687.720
Zipf-Verteilung	1,0	140.121	386	5.926.040
	1,5	140.051	167	3.374.904
	2,0	140.052	90	75.832

Tabelle A.22: Caching (1% Migrationen, 10 Chunks pro Bereich).

Verteilung		Zeit pro Operation Ohne Caching *(in µs)*	Mit Caching *(in µs)*	Cachegröße *(in Byte)*
Gleichverteilung		140.173	271	599.152
Normalverteilung	0,2	140.118	190	90.336
	0,5	140.129	213	212.248
	1,0	140.140	261	415.696
Zipf-Verteilung	1,0	140.090	174	596.408
	1,5	140.046	113	539.176
	2,0	140.046	93	22.912

Tabelle A.23: Caching (1% Migrationen, 100 Chunks pro Bereich).

5% Migrationen

Verteilung		Zeit pro Operation Ohne Caching *(in µs)*	Mit Caching *(in µs)*	Cachegröße *(in Byte)*
Gleichverteilung		140.557	9.225	259.184.304
Normalverteilung	0,2	140.311	1.309	29.240.632
	0,5	140.385	2.615	69.244.624
	1,0	140.442	4.489	132.214.720
Zipf-Verteilung	1,0	140.236	7.506	256.187.856
	1,5	140.071	542	22.651.896
	2,0	140.058	114	494.880

Tabelle A.24: Caching (5% Migrationen, 1 Chunk pro Bereich).

Verteilung		Zeit pro Operation Ohne Caching (in μs)	Mit Caching (in μs)	Cachegröße (in Byte)
Gleichverteilung		140.362	1.463	28.861.568
Normalverteilung	0,2	140.197	394	3.605.008
	0,5	140.210	610	8.714.336
	1,0	140.257	908	16.678.600
Zipf-Verteilung	1,0	140.159	1.202	28.890.968
	1,5	140.060	274	8.436.800
	2,0	140.053	103	164.032

Tabelle A.25: Caching (5% Migrationen, 10 Chunks pro Bereich).

Verteilung		Zeit pro Operation Ohne Caching (in μs)	Mit Caching (in μs)	Cachegröße (in Byte)
Gleichverteilung		140.197	384	2.903.328
Normalverteilung	0,2	140.138	262	401.192
	0,5	140.155	288	959.400
	1,0	140.165	337	1.866.096
Zipf-Verteilung	1,0	140.112	271	2.897.840
	1,5	140.054	149	2.071.896
	2,0	140.050	101	50.744

Tabelle A.26: Caching (5% Migrationen, 100 Chunks pro Bereich).

10% Migrationen

Verteilung		Zeit pro Operation Ohne Caching (in μs)	Mit Caching (in μs)	Cachegröße (in Byte)
Gleichverteilung		140.641	16.219	475.708.248
Normalverteilung	0,2	140.379	2.047	52.182.824
	0,5	140.439	4.239	122.915.696
	1,0	140.508	7.555	233.770.160
Zipf-Verteilung	1,0	140.273	13.262	466.708.320
	1,5	140.070	673	29.066.584
	2,0	140.070	114	684.216

Tabelle A.27: Caching (10% Migrationen, 1 Chunk pro Bereich).

Verteilung		Zeit pro Operation Ohne Caching *(in µs)*	Mit Caching *(in µs)*	Cachegröße *(in Byte)*
Gleichverteilung		140.416	2.447	55.986.792
Normalverteilung	0,2	140.213	554	6.825.288
	0,5	140.260	893	16.325.408
	1,0	140.303	1.421	31.473.856
Zipf-Verteilung	1,0	140.182	2.146	56.053.040
	1,5	140.063	347	11.684.128
	2,0	140.058	97	229.888

Tabelle A.28: Caching (10% Migrationen, 10 Chunks pro Bereich).

Verteilung		Zeit pro Operation Ohne Caching *(in µs)*	Mit Caching *(in µs)*	Cachegröße *(in Byte)*
Gleichverteilung		140.238	528	5.641.840
Normalverteilung	0,2	140.160	282	761.440
	0,5	140.171	333	1.831.600
	1,0	140.184	383	3.553.656
Zipf-Verteilung	1,0	140.122	374	5.640.272
	1,5	140.051	178	3.352.168
	2,0	140.052	104	71.128

Tabelle A.29: Caching (10% Migrationen, 100 Chunks pro Bereich).

A.3 Gesamtsystem

Die Messergebnisse beziehen sich auf die Evaluierungen in Abschnitt 5.4.

A.3.1 BG-Benchmark

Anzahl Knoten	**DXRAM** *(in Operationen / Sekunde)*	**MongoDB** *(in Operationen / Sekunde)*	**Cassandra** *(in Operationen / Sekunde)*
1	6.072	3.633	1.893
2	11.999	3.657	2.851
4	19.069	3.633	3.376
8	12.016	3.398	3.765

Tabelle A.30: DXRAM, MongoDB und Cassandra.

Anzahl Profile	**DXRAM** *(in Operationen / Sekunde)*
10.000	20.077
100.000	22.154
400.000	14.913
500.000	17.058

Tabelle A.31: Viele Profile.

Eidesstattliche Erklärung

An Eidesstatt erkläre ich, dass ich diese Dissertation selbstständig und ohne unzulässige fremde Hilfe unter der Beachtung der „Grundsätze zur Sicherheit guter wissenschaftlicher Praxis an der Heinrich-Heine-Universität“ angefertigt habe und dass ich diese in der jetzigen oder in einer ähnlichen Form noch keiner anderen Fakultät eingereicht habe.

Monheim am Rhein, 25. September 2015 Florian Klein