Humboldt-Universität zu Berlin

Mathematisch-Naturwissenschaftliche Fakultät II

Institut für Informatik

Ein eingebettetes Hauptspeicher-Datenbanksystem mit Snapshot-Reads

Diplomarbeit

eingereicht von: Daniel Mauter, geb. in Ludwigslust

Betreuer / 1.Gutachter: Prof. Dr. Alexander Reinefeld

2. Gutachter: Prof. Dr. Jens-Peter Redlich

April 2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Jenseits relationaler Datenbanksysteme

Der Erfolg von relationalen *Datenbankmanagementsystemen (RDBMS)* in den letzten Jahrzehnten führte dazu, dass RDBMS fast zum Synonym von Datenmanagement und SQL zur Lingua Franca beim Datenzugriff wurde [38]. Die Hersteller der Datenbanksysteme (DBS) förderten die Verbreitung mit der steten Bereitstellung neuer Funktionalitäten. Die Kehrseite der Universalität ist die Komplexität der Software. Die Konfiguration erfordert Experten, die Leistung ist unvorhersehbar und es werden viele Hardwareressourcen verbraucht [11, 19, 12].

Stonebraker prophezeit das Ende der nach dem *"One-Size-Fits-All-Ansatz"* entworfenen DBS [41, 40, 42]. Er identifiziert mehrere Anwendungen bei denen die Entwicklung *spezialisierter DBS* zu einer Steigerung der Performanz um einen Faktor größer 10 führt. Ein Beispiel ist das *Data Warehousing*. Hier nutzen *Column Stores* [43, 8] das Wissen um typische Abfragen und organisieren die Daten spaltenorientiert und nicht zeilenorientiert, wie in gewöhnlichen RDBMS.

Die Spezialisierung auf eine Anwendung beschränkt sich nicht auf Optimierungen wie der physischen Datenorganisation im Falle von Data Warehouses. Vielmehr kann auf die Bereitstellung von Funktionalitäten, die essentiell für ein herkömmliches RDBMS sind, gänzlich verzichtet werden. Die einzige Voraussetzung ist, dass das System den Anforderungen der Anwendung gerecht wird. So kann beispielsweise auf den Einsatz von Sperrverfahren verzichtet werden, wenn kein Mehrbenutzerbetrieb gefordert wird.

Harizopoulos et al. nehmen ein existierendes DBS und entfernen schrittweise Funktionalitäten wie das Locking und das Logging [20]. Auf diesem Weg bestimmen sie den Mehraufwand, den die Bereitstellung dieser Funktionalitäten erzeugt. Nach jeder Optimierung messen sie die Anzahl der aufgerufenen CPU-Instruktionen und den Transaktionsdurchsatz bei einem Workload aus dem TPC-C Benchmark. Nach sämtlichen Optimierungen benötigt das System für den Workload nur 7% der ursprünglichen Anzahl an CPU-Instruktionen und erhöht den Transaktionsdurchsatz von 640 auf 12700 Transaktionen pro Sekunde.

Dass spezialisierte Systeme durchaus nicht nur von einer Anwendung genutzt werden können, beweisen Systeme aus der Industrie. *Dynamo* [12] von Amazon und *Bigtable* [10] von Google sind beides spezielle Lösungen für das Datenmanagement, deren Design deutlich von dem eines RDBMS abweicht. Ihnen ist gemein, dass sie innerhalb der Unternehmen von einer Vielzahl unterschiedlicher Dienste verwendet werden.

1.2 Zielstellung der Diplomarbeit

In dieser Arbeit soll der obige Trend aufgegriffen und für die Anwendung *XtreemFS* [22, 21] ein spezialisiertes Hauptspeicher-Datenbanksystem (HSDBS) entwickelt werden. XtreemFS ist ein verteiltes, objektbasiertes Dateisystem und benötigt für die Verwaltung der Metadaten ein performantes Datenbankbackend. Gegen die Verwendung eines herkömmlichen RDBMS spricht, dass XtreemFS nur einen Bruchteil dessen Funktionalität benötigt. Es genügt ein einfacheres Datenmodell, es wird keine deklarative Abfragesprache verlangt und die Transaktionsverwaltung kann stark vereinfacht werden, um nur einige Beispiele zu nennen.

In dem zu entwickelnden HSDBS werden die Daten in *Zugriffspfaden* bzw. *Indizes* gehalten. Die Zugriffspfade liegen im flüchtigen Hauptspeicher. Für die dauerhafte Speicherung von Änderungen an den Daten, muss in dieser Arbeit ein *Logging-* und *Recovery-Verfahren* konzipiert werden. Das Verfahren soll eine Kombination aus *Logging* und *Sicherungspunkten* verwenden, um eine schnelle Wiederherstellung (Recovery) der Datenbank nach einem Systemabsturz zu gewährleisten. Ein Sicherungspunkt besteht aus *Schnappschüssen* der Zugriffspfade. Im Rahmen dieser Arbeit müssen Datenstrukturen für die Zugriffspfade gefunden werden, welche das Anfertigen eines Schnappschusses ermöglichen, ohne die Transaktionsbearbeitung zu beeinträchtigen.

Der Entwurf des HSDBS soll möglichst schlank ausfallen, so dass nur die geforderten Funktionalitäten bereitgestellt werden. Das Ziel ist, auf diese Weise ein spezialisiertes DBS zu entwerfen, welches beim Einsatz in XtreemFS eine wesentlich höhere Performanz als ein universelles, nach dem One-Size-Fits-All-Ansatz entworfenes DBS.

1.3 Gliederung

In Kapitel 2 werden Grundlagen zu Datenbanksystemen beschrieben. Im darauf folgenden Kapitel 3 wird die Anwendung XtreemFS vorgestellt und es werden die Anforderungen an das in dieser Arbeit zu entwickelnde DBS spezifiziert. Den "Geschäftsbedingungen" folgt dann in Kapitel 4 der maßgeschneiderte Entwurf des Datenbanksystems SpeeDB.

Im Entwurfskapitel wird zunächst das logische Datenmodell vorgestellt. Danach werden die Architektur und die Interaktion der einzelnen Komponenten beschrieben. Im Abschnitt 4.2 wird das verwendete Logging- und Recovery-Schema vorgestellt, welches für die dauerhafte Speicherung von Änderungen an der Datenbank zuständig ist. Für die schnelle Wiederherstellung der Datenbank wird ein Algorithmus für die Rekonstruktion der Zugriffspfade

angegeben. Im Abschnitt 4.3 werden Verfahren für die Entwicklung von Zugriffspfaden beschrieben, welche die asynchrone Anfertigung von Schnappschüssen ohne den Einsatz von Sperren ermöglichen. Im letzten Abschnitt des Kapitels werden Überlegungen zu Warteschlangen in Stage-Architekturen angestellt.

Das entworfene DBS wurde prototypisch in Java implementiert. Im Kapitel 5 werden die Ergebnisse der ebenfalls vorgenommenen Evaluierung angegeben und diskutiert. Neben Vergleichen verschiedener Indizes (5.3) und Messungen des DBS (5.2) finden sich auch Beobachtungen zum Einfluss des Garbage-Collectors der Java VM(5.5).

Im Kapitel 6 werden einige themenbezogene Arbeiten anderer Autoren vorgestellt. Der Fokus wird dabei auf Logging- und Recovery-Techniken, auf Verfahren für die Rekonstruktion von Indizes und auf ausgewählte Implementierungstechniken für moderne Hardware gerichtet. Zuletzt findet sich in Kapitel 7 eine Zusammenfassung dieser Diplomarbeit sowie ein Ausblick auf die Weiterentwicklung des DBS.

Kapitel 2

Grundlagen

2.1 Datenbanksysteme

Datenbanksysteme sind eine Säule der Informatik. Seit mehreren Dekaden werden sie als ausgereifte Produkte in den unterschiedlichsten Bereichen eingesetzt und sind aus der heutigen Welt nicht mehr wegzudenken. Gleichzeitig sorgten neue Anforderungen durch Benutzer, Veränderungen der Hardware sowie das stete Wachstum der Datenfluten im "Informationszeitalter" dafür, dass ihre Entwicklung nie ganz abgeschlossen war und sie heute noch Gegenstand der Forschung sind.

Besonders stark verbreitet sind relationale Datenbankmanagementsysteme (RDBMS). Oft wird die Bezeichnung Datenbanksystem (DBS) sogar als Synonym für ein RDBMS gebraucht. Es gibt eine lange Liste von Eigenschaften und Funktionalitäten die von einem DBS gemeinhin erwartet werden. Aber auch Systeme, die deutlich von diesem allgemeinen Bild abweichen, werden als Datenbanksysteme bezeichnet. Ein gutes Beispiel hierfür ist das eingebettete Datenbanksystem *Berkeley DB*. Es bietet weder eine deklarative Abfrage wie SQL als Schnittstelle noch verwendet es eines der in Datenbanksystemen etablierten logischen Datenmodelle, wie z. B. das relationale oder objektorientierte Modell. Da der Titel dieser Diplomarbeit darauf schließen lässt, dass in ihr ein Datenbanksystem vorgestellt wird, soll zunächst der Begriff Datenbanksystem etwas abgeschwächt werden. Die folgenden abstrakten Definitionen sind angelehnt an Definitionen aus [5].

Datenbank Eine Datenbank besteht aus einer Menge benannter Datenelemente. Jedes Datenelement besitzt einen Wert. Die Menge der Werte zu einem bestimmten Zeitpunkt erfasst den Datenbankzustand.

Datenbanksystem Ein Datenbanksystem ist eine Menge von Hard- und Softwaremodulen, die Befehle für den Zugriff auf die Datenbank - sogenannte Datenbankoperationen - anbieten. Es unterstützt die nebenläufige Ausführung von Transaktionen.

Transaktionen Eine Transaktion ist eine Folge von Operationen auf den Datenelementen der Datenbank. Bei der Ausführung einer Transaktion müssen die ACID-Eigenschaften, welche im Folgenden aufgelistet werden, gewährleistet sein:

A steht für Atomarität oder auch das Alles oder Nichts Prinzip. Kommt es zu einem Abbruch einer Transaktion, so müssen alle von der Transaktion vorgenommenen Änderungen rückgängig gemacht werden.

C steht für Konsistenz. Eine Transaktion muss die Datenbank aus einem konsistenten Zustand in einen konsistenten Zustand überführen. Eine Datenbank ist in einem konsistenten Zustand, wenn sämtliche existierenden *Integritätsbedingungen* der Datenbank erfüllt sind. Ein Beispiel für eine Integritätsbedingung wäre die Angabe, dass das Gehallt eines Angestellten niedriger als das Gehalt seines Vorgesetzten sein muss.

I steht für Isolation. Bei der parallelen Ausführung mehrerer Transaktionen wird jede Transaktion so ausgeführt, als würde sie allein auf dem System operieren.

D steht für Dauerhaftigkeit. Die Änderungen einer erfolgreich abgeschlossenen Transaktion werden dauerhaft im DBS gespeichert.

Die parallele Ausführung von Transaktionen wird benötigt, um zu verhindern, dass lang andauernde Transaktionen das DBS für die Zeit ihrer Ausführung blockieren und somit den Transaktionsdurchsatz des Systems drastisch reduzieren. Dabei müssen Synchronisationsverfahren eingesetzt werden, um die Konsistenz der Datenbank im Mehrbenutzerbetrieb zu wahren. Greifen zwei zur gleichen Zeit aktive Transaktionen auf gemeinsame Daten zu, können ohne den Einsatz von Synchronisationsverfahren Anomalien auftreten. Ein Beipiel für eine Anomalie ist das *Lost Update*. Angenommen in zwei Transaktionen sollen jeweils 100 € auf ein Konto überwiesen werden. Zuerst lesen beide Transaktionen den Kontostand von 1000 € und addieren dann den Kontostand mit den 100 €. Im Anschluss schreiben sie die Summe von 1100 € auf das Konto. Bei einer korrekten, seriellen Ausführung würde das Konto nach den Transaktionen einen Betrag von 1200 € aufweisen. Die Ursache des Fehlers ist, dass die Transaktionen die zwischenzeitliche Änderung des Kontostandes nicht gesehen haben. Weitere Anomalien sind das *Dirty Read, Dirty Write, Non Repeatable Read* und das *Phantom Problem* [23]. Synchronisationsverfahren haben also die Aufgabe die Ausführung der unterschiedlichen Transaktionen so zu verschachteln, dass das Ergebnis der Ausführung korrekt ist. Ein Kriterium für eine korrekte Ausführungsfolge oder auch *Schedule* der Transaktionen ist die *Serialisierbarkeit*. Ein Schedule ist *serialisierbar* , wenn die Ausgaben aller beteiligten Transaktionen äquivalent zu einer seriellen Ausführung der Transaktionen ist. Eine serielle Ausführung der Transaktionen liegt vor, wenn die Transaktionen in einer beliebigen Reihenfolge nacheinander ausgeführt werden. Eine weitere Voraussetzung für die Äquivalenz zweier Schedules ist, dass ihre Ausführung zu einem identischen Datenbankzustand führen.[1]

[1] Diese kurze Ausführung zur Serialisierbarkeit ist dem Buch von Härder und Rahm [23, S. 412ff] entnommen. Eine umfassende, formale Abhandlung findet sich in dem Buch von Weikum und Vossen [48].

Ein serialisierbarer Schedule verhindert das Auftreten sämtlicher, oben genannter Anomalien. Die Gewährleistung, dass nur serialisierbare Schedules ausgeführt werden, vermindert den Transaktionsdurchsatz, da der notwendige Synchronisationsaufwand sehr hoch ist. Für einige Anwendungen genügt auch ein schwächeres Korrektheitskriterium, da für diese Anwendungen das Auftreten gewisser Anomalien vertretbar ist oder durch Applikationswissen ausgeschlossen werden kann. Es gibt deshalb unterschiedlich starke Konsistenzstufen, die für den Betrieb eines DBS zur Wahl stehen und eine höhere Performanz des DBS bei geringerem Schutz bieten.

2.2 Hauptspeicherdatenbanksysteme

In herkömmlichen Datenbanksystemen werden alle Daten physisch im Sekundärspeicher abgelegt. Die im Betrieb ständig notwendigen Zugriffe auf den Sekundärspeicher stellen den Flaschenhals des DBS dar. Es gilt deshalb die Anzahl der I/O-Operationen zu minimieren, wobei verschiedene Techniken zum Einsatz kommen. Es werden beispielsweise besondere Datenstrukturen wie *B-Bäume* und *Hash-Tabellen* eingesetzt, es gibt *Buffer-Manager*, die ausgefeilte Verdrängungsstrategien verfolgen, es gibt auch den Ansatz die Daten in einer geclusterten Speicherung auf der Platte an die Zugriffsmuster anzupassen,um die teueren Bewegungen des Plattenarms zu minimieren, und vieles mehr.

Mitte der 80er Jahre begann sich die Forschung mit *Hauptspeicher-Datenbanksystemen (HSDBS)* zu beschäftigen. Die technische Entwicklung ließ erkennen, dass in der Zukunft für immer mehr Anwendungen die Annahme getroffen werden konnte, dass die komplette Datenbank in den Hauptspeicher passt. Diese Annahme legte ein enormes Leistungspotential frei, stellte jedoch auch neue Anforderungen. Aufgrund der Flüchtigkeit des Hauptspeichers müssen sämtliche an den Daten vorgenommenen Änderungen auf dem Sekundärspeicher geloggt werden. Während des Datenbankbetriebes erfordert allein das Logging Zugriffe auf den Sekundärspeicher und stellt deshalb einen Flaschenhals dar. Beim Wiederherstellen(*Recovery*) des aktuellsten Datenbankzustandes nach einem Systemausfall müssen die Daten möglichst schnell in den Hauptspeicher geladen werden, um wieder einen performanten Datenbankbetrieb zu ermöglichen.

Um das Leistungspotential von HSDBS umfassend auszunutzen, mussten die Komponenten eines DBS an die neuen Bedingungen angepasst werden. Kürzere Bearbeitungszeiten der Transaktion erlaubten den Einsatz von Sperrverfahren mit einer gröberen Granularität, ein anderes physisches Datenmodell konnte verwendet werden, weil der wahlfreie Zugriff im Hauptspeicher im Gegensatz zur Festplatte schnell durchführbar ist, auf einen Buffer-Manager konnte verzichtet werden etc. . Garcia-Molina und Salem geben einen umfassenden Überblick über Anforderungen und Optimierungen in HSDBS [15].

Ein Vierteljahrhundert später gibt es heute Server mit einer Hauptspeicherkapazität von 100 Gigabyte zu einem Preis von unter 15000 € und HSDBS haben bei den namhaften Datenbanksystemherstellern in der Produktpalette Einzug gehalten. “Die Ära der HSDBS hat endgültig begonnen.” [17]

2.3 Zugriffspfade

Ein wichtiges Werkzeug in Datenbanksystemen sind Zugriffspfade bzw. Indizes. Möchte man in einer unsortierten Liste von Datenelementen nach Elementen suchen, die ein gewisses Kriterium erfüllen, müssen alle Elemente der Liste überprüftwerden. Zugriffspfade sind Datenstrukturen, die einen schlüsselbasierten Zugriff bei deutlich geringerem Aufwand bieten. Suchbäume ermöglichen eine Suche mit logarithmischem Aufwand. Bei herkömmlichen Datenbanksystemen sind Varianten des *B-Baumes* [4] sehr verbreitet. Ein Knoten eines B-Baumes lässt sich gut auf eine Speicherseite abbilden. Zudem ist der Baum ist sehr flach, wobei sämtliche Pfade von der Wurzel zu den Blättern gleich lang sind. Diese Eigenschaften sind sehr wichtig, um Datenelemente auf dem Sekundärspeicher zu indizieren, wenn nicht garantiert werden kann, dass der komplette Index im Hauptspeicher liegt.

Ein balancierter binärer Baum, wie etwa der *AVL-Baum* [1], wäre für die Indizierung von Elementen auf der Festplatte gänzlich ungeeignet, da er im Vergleich zum B-Baum viel höher ist und seine Knoten sich auch nicht so effizient auf Seiten abbilden lassen. Eine Traversierung des Baumes würde zu viele Zugriffe auf den Sekundärspeicher erfordern.

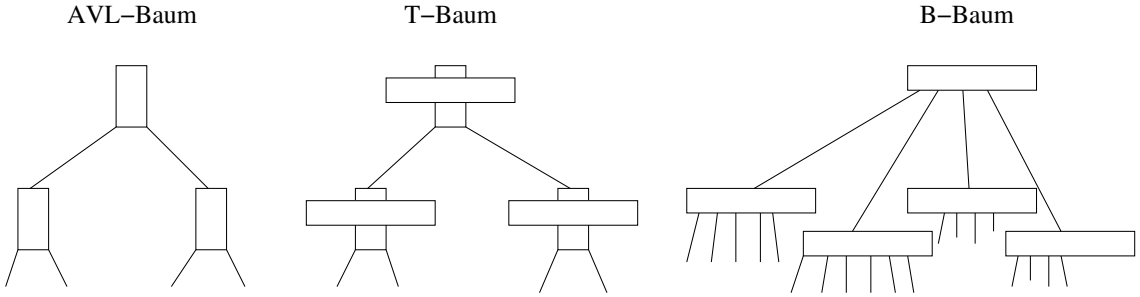


Abbildung 2.1: Struktur von AVL-,B- und T-Baum

Lehman und Carey stellten 1986 den *T-Baum* als Index für HSDBS vor [30]. Sie argumentieren, dass lange Pfade in den Indizes für Daten im Hauptspeicher unproblematisch seien, da der Zugriff auf eine Seite im Hauptspeicher sehr schnell durchführbar sei. Ein binärer Baum wie der AVL-Baum wäre jedoch ungeeignet, da er sehr viel Speicherplatz verbrauche. Der T-Baum ist eine Mischung aus dem AVL- und dem B-Baum. Er besitzt eine binäre, balancierte Struktur des AVL-Baums. Die für das Erhalten der Balancierung notwendigen Rotationen werden auch ähnlich zu den Rotationen des AVL-Baumes durchgeführt. In den Knoten nimmt er jedoch wie der B-Baum mehrere Schlüssel und zugehörige Werte auf. Dadurch wird er etwas flacher und platzsparender als der AVL-Baum. Die Autoren vergleichen mehrere Zugriffspfade beim Einsatz im Hauptspeicher, wobei sich der T-Baum für HSDBS als am besten geeignet zeigt. Siehe hierzu auch Abschnitt 6.3.

Kapitel 3

Anforderungen an das Datenbanksystem

In der Einführung dieser Diplomarbeit wurde die Entwicklung von spezialisierten Datenbanksystemen für den Einsatz in Anwendungen motiviert, die ein hochperformantes Datenbankbackend benötigen. In diesem Kapitel wird die Anwendung XtreemFS vorgestellt, für die in dieser Arbeit ein DBS entworfen wird, und es werden die Anforderungen spezifiziert, die XtreemFS an das DBS stellt.

3.1 Porträt von XtreemFS

Die Anwendung XtreemFS [22, 21] ist ein verteiltes, objektbasiertes Dateisystem für den Einsatz in *Wide Area Networks (WANs)*. Ein verteiltes Dateisystem bietet einen einheitlichen Zugriff auf Dateien, die auf verschiedenen, unter Umständen heterogenen Speicherressourcen verteilt sind. Objektbasiert heißt, dass eine Datei für den Benutzer transparent auf mehrere Objekte aufgeteilt wird, welche auf den verschiedenen Ressourcen gespeichert werden können. Für den Einsatz im WAN bietet XtreemFS eine dezentrale Struktur, Replikation von Daten und Metadaten sowie verbreitete Sicherheitsmaßnahmen wie die Nutzerauthentifizierung mittels SSL-Verbindungen.

Die Dateiinhalte oder vielmehr die aus der Datei generierten Objekte werden in sogenannten *Object Storage Devices (OSDs)* gespeichert. Mittels *Striping* kann auf eine Datei parallel zugegriffen werden und so die limitierten Bandbreiten der OSD aggregiert werden.

Die Metadaten einer Datei werden von den *Metadata and Replica Catalogs (MRCs)* verwaltet. Neben gewöhnlichen Metadaten wie den Benutzerrechten einer Datei speichern sie Informationen über die Verteilung der Objekte auf die OSDs für alle Replikate einer Datei.

Dank der POSIX-Konformität kann XtreemFS problemlos als Dateisystem auf einem Linux- oder Unixsystem eingebunden werden. Es unterstützt erweiterte Dateiattribute und *Access Control Lists (ACLs)*.

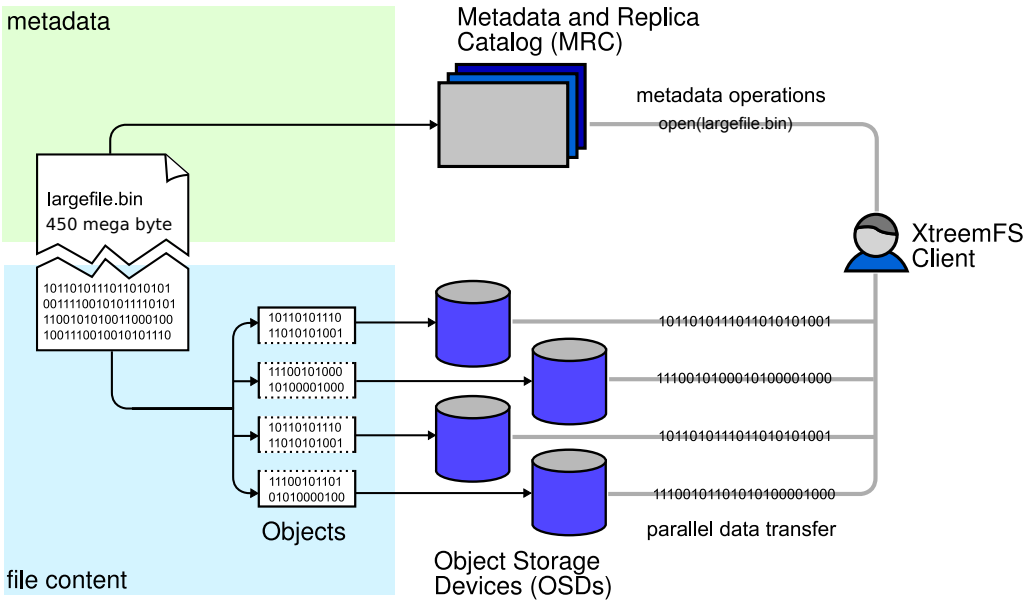


Abbildung 3.1: Beziehungen der XtreemFS-Komponenten OSD und MRC (Abb. aus [21])

In Abbildung 3.1 wird das Zusammenspiel der Komponenten OSD und MRC veranschaulicht. Ein *Client* erfragt zunächst bei einem MRC von welchen OSD er die Datei *largefile.bin* lesen kann. Die Datei liegt nicht repliziert vor, wurde aber mittels Striping auf mehrere OSD verteilt. Nachdem der MRC die Anfrage des Clients beantwortet hat, kommuniziert der Client ausschließlich mit den OSD. Der Zugriff auf die Objekte verschiedener OSD erfolgt parallel.

3.2 Anforderungen an das zu entwickelnde DBS

Als ein verteiltes Dateisystem sollte XtreemFS die folgenden Laufzeiteigenschaften besitzen:

Hohe Performanz Bei Interaktionen mit Benutzern reagiert das System sehr schnell. Die Bandbreite der Datenübertragung ist dicht am Optimum.

Hochverfügbarkeit Einzelne Ausfälle führen nicht zu einem Gesamtausfall des Systems. Teilsysteme können nach einem Neustart in kürzester Zeit den Betrieb wieder aufnehmen.

Skalierbarkeit Mit zunehmender Anzahl an Benutzern, Daten und Ressourcen bleibt das System ohne Einschränkungen benutzbar.

Bei der Entwicklung jeder Komponente des verteilten Dateisystems muss geprüft werden, inwieweit man das Design in Hinblick auf die oben genannten Eigenschaften optimieren

kann. Das vom MRC verwendete Datenbanksystem sollte demnach kurze Antwortzeiten und einen hohen Transaktionsdurchsatz aufweisen. Auch wenn die wesentlichen Maßnahmen zur Realisierung der Hochverfügbarkeit und der Skalierbarkeit auf höheren Ebenen mit Replikations- und Partitionierungsverfahren ergriffen werden, wird von dem DBS verlangt, dass es nach einem Systemausfall in kurzer Zeit den aktuellsten Datenbankzustand wiederherstellen und den Betrieb wiederaufnehmen kann.

Gemeinsam mit Entwicklern des XtreemFS-Projektes wurden weitere, nun folgende Anforderungen an das Datenbanksystem zur Metadatenverwaltung herausgearbeitet:

- Das DBS soll Zugriffspfade bereitstellen, in die Objekte in Form von Schlüssel-Werte-Paaren gelegt werden können. Die Modellierung von Relationen zwischen Objekten wird vorerst nicht gefordert.
- Es kann die Annahme getroffen werden, dass sämtliche vom Datenbanksystem zu verwaltenden Daten in den Hauptspeicher passen. Später werden Verfahren eingesetzt, die die Metadaten des Dateisystems partitionieren, wodurch sich das Volumen der Metadaten eines MRC beliebig klein halten lässt. Diese Vefahren werden jedoch in einer höheren Schicht implementiert und müssen nicht berücksichtigt werden.
- Die Anzahl der möglichen Interaktionen mit einem Dateisystem sind begrenzt. Es genügt deshalb die Bereitstellung einer Schnittstelle für die Implementierung *benutzerdefinierter Funktionen (BDF)*, welche auf den Daten operieren können. Eine deklarative Abfragesprache wie SQL wird nicht benötigt.
- Die BDF müssen transaktional ausgeführt werden. Anders als bei Transaktionen in herkömmlichen Datenbanksystemen ist es aber nicht erforderlich, eine BDF jederzeit abbrechen zu können.
- Lesende BDF sollen parallel zu anderen schreibenden und lesenden BDF ausgeführt werden können.
- Eine parallele Bearbeitung mehrerer schreibender BDF wird nicht gefordert, da das langer Bearbeitungszeiten bei der Ausführung schreibender BDF ausgeschlossen wird.
- Um aufwendige Interprozesskommunikation zu vermeiden, soll dass Datenbanksystem in die Software des MRC eingebettet werden können. Da der MRC mit der Programmiersprache Java implementiert wurde, ist folglich eine Java-Bibliothek bereitzustellen.

Mit diesen Anforderungen kann nun ein schlankes DBS entworfen werden. Der Entwurf wird im anschließenden Kapitel vorgestellt.

Kapitel 4

Entwurf des Datenbanksystems

Motiviert durch Kapitel 1 wurde ein für den Einsatz in XtreemFS spezialisiertes DBS entworfen. Die Konzeption des DBS wurde so angelegt, dass gerade die in 3.2 genannten Anforderungen erfüllt werden und kein Mehraufwand für die Bereitstellung zusätzlicher, unbenötigter Funktionalitäten entsteht. Das entstandene DBS trägt den Namen *SpeeDB* und ist ein *eingebettetes Hauptspeicher-Datenbanksystem*. In diesem Kapitel werden Eigenschaften und Architektur des Systems beschrieben. Zunächst werden die Systemkomponenten und ihr Zusammenspiel erläutert. Dafür ist auch die Vorstellung des verwendeten logischen Datenmodells und die Belegung einiger Begriffe erforderlich. Im Abschnitt 4.2 wird beschrieben wie Änderungen an der Datenbank persistent gemacht werden und nach einem Systemausfall der zuletzt aktuelle Datenbankzustand wiederhergestellt wird. Das DBS verwendet besondere Zugriffspfade, die einen parallelen Zugriff ohne den Einsatz von Sperren ermöglichen. Diese Zugriffspfade werden in Abschnitt 4.3 vorgestellt. Im letzten Abschnitt des Kapitels werden Überlegungen zu Warteschlangen in der *Stage-Architektur* angestellt.

4.1 Architektur

Das DBS wird als Bibliothek in die Clientanwendung eingebettet und läuft somit in derselben Prozessumgebung, wodurch aufwendige Interprozesskommunikation eingespart wird.

Wie der Metadaten- und Replikatskatalog (MRC) von XtreemFS wurde auch SpeeDB in einer Stage-Architektur [49] realisiert und lässt sich somit gut in den MRC einbetten. In der Stage-Architektur wird der Programmfluss einer Anwendungen in mehrere Teile dekomponiert. Die einzelnen Programmsegmente werden auf miteinander verketteten Stages abgearbeitet. Sie bietet einen besseren Schutz vor Überlastung einiger Ressourcen und skaliert deshalb besser als das *Threaded-Server-Modell*, bei dem jeder Auftrag von einem eigenen Thread abgearbeitet wird. Datenbankanfragen werden asynchron bearbeitet. Eingehende *Datenbankanfragen* werden vom Sender in eine Warteschlange vom DBS und die *Datenbankantworten* in die Warteschlange des Empfängers eingereiht.

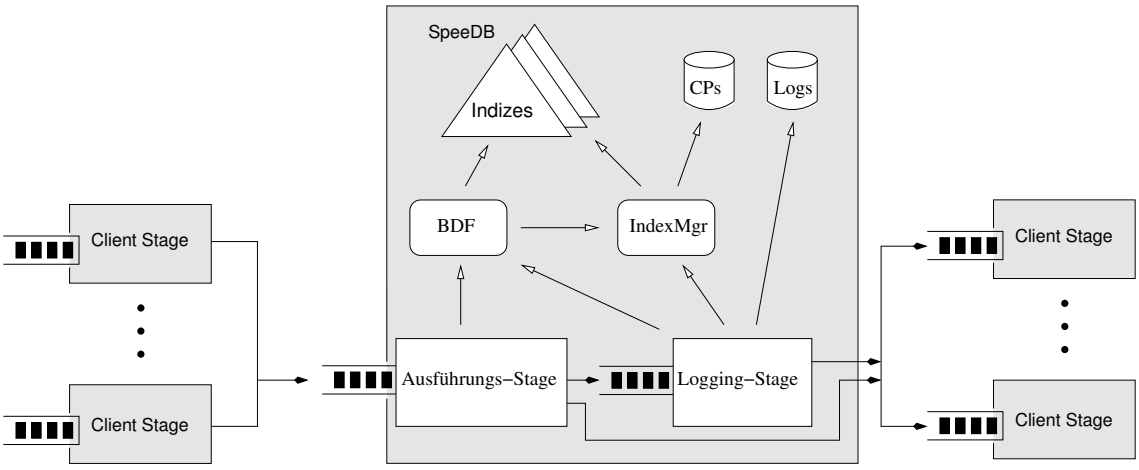


Abbildung 4.1: Stage-Architektur des DBS

Die Abbildung 4.1 skizziert die Architektur des Systems. Es ist gut zu erkennen, wie sich SpeeDB als eine logische Stage in die Architektur der Clientanwendung eingliedert. Intern verwendet das DBS zwei Stages: die Ausführungs-Stage und die Logging-Stage. Die Ausführungs-Stage ist für die Ausführung der *benutzerdefinierten Funktionen (BDF)* zuständig. Je nach verwendetem Datenbankmodus (4.1.5) führen dort ein oder mehrere Threads die BDF aus. Wird eine BDF auf der Ausführungs-Stage ausgeführt, hat sie Zugriff auf die Daten in den *Indizes*. In diesen befinden sich die in der Datenbank abgespeicherten Daten in Form von *Schlüssel-Werte-Paaren (SWP)*. Sie liegen ausschließlich im Hauptspeicher des Systems und werden vom *IndexManager* verwaltet. Die Logging-Stage ist für das Loggen der schreibenden Datenbankanfragen, die Anfertigung von *Sicherungspunkten* und der *Recovery* beim Start des DBS zuständig. Auf dieser Stage arbeiten zwei Threads: der *Group-Comitter* und der *Checkpointer*. Der Group-Comitter loggt die vorgenommenen Änderungen und verteilt die Datenbankantworten an die Empfänger. Der Checkpointer veranlasst die Anfertigung von Sicherungspunkten entsprechend einer gewählten Vereinbarung (4.2.2).Die beiden Zylinder symbolisieren die Festplatten mit den Sicherungspunkten (CPs) und den Logs.

4.1.1 Logisches Datenmodell

Eine Datenbank in SpeeDB besteht aus einer Menge von Zugriffspfaden bzw. Indizes, die Objekte in Form von Schlüssel-Werte-Paaren (SWP). Für die SWP innerhalb eines Zugriffspfades gilt, dass jedem Schlüssel maximal ein Wert zugeordnet wird und dass er mit jedem anderen Schlüssel vergleichbar ist. Es können also in einem Index nicht zwei Schlüssel-Werte-Paare mit identischem Schlüssel vorkommen. Solche Schlüssel werden auch als Primärschlüssel und Indizes mit Primärschlüsseln als Primärindizes bezeichnet.

4.1.2 Benutzerdefinierte Funktionen

Das DBS stellt keine deklarative Abfragesprache zur Datendefinition oder -manipulation bereit. Der Zugriff auf die Datenbank erfolgt über den Aufruf von benutzerdefinierten Funktionen (BDF). Diese sind vom Anwender zu implementieren und werden im Datenbankbetrieb transaktional ausgeführt, d. h. das DBS garantiert die ACID-Eigenschaften für die Ausführung der BDF.

Diese Funktionen werden vom Anwender in einer Klasse bereitgestellt. Jede Funktion ist eindeutig mit einer ganzen Zahl identifizierbar, wobei positive Zahlen für lesende und negative Zahlen für schreibende Funktionen verwendet werden. Mithilfe dieser Konvention kann das DBS zwischen lesenden und schreibenden Zugriffen unterscheiden. Die Klasse mit den BDF implementiert auch eine Methode, die während der Initialisierung von SpeeDB aufgerufen wird und in der Zugriffspfade für die Aufnahme von SWP angefordert werden.

Zur Zeit der Ausführung einer BDF kann diese mit den folgenden Methoden auf die SWP in den registrierten Zugriffspfaden zugreifen:

`put(key,value)`
: fügt ein neues SWP in den Index ein

`get(key)`
: sucht den Wert für einen gegebenen Schlüssel

`delete(key)`
: entfernt ein SWP

`update(key,value)`
: aktualisiert den Wert eines bereits existierenden Eintrags

`getRange(lowKey,highKey)`
: ermittelt alle SWP (k, v) mit $lowKey \leq k \leq highkey$

Die BDF werden als *Zweiphasentransaktionen* [42] ausgeführt. In der ersten Phase können ausschließlich lesende Zugriffe von der BDF vorgenommen werden. In Abhängigkeit des Ergebnisses aus Phase 1 wird die BDF entweder beendet oder es wird die 2. Phase gestartet, in der schreibende Operationen unwiderruflich ausgeführt werden. Ein Abbruch der BDF durch den Benutzer ist demnach nur in der 1. Phase möglich. Zweiphasentransaktionen weichen damit deutlich von herkömmlichen Transaktionen ab, die jederzeit mit einem `ABORT` abgebrochen werden können.

Durch die serielle Ausführung der BDF werden sie auch nicht vom DBS aufgrund von Sperrkonflikten abgebrochen. Die einzige Ursache für den Abbruch einer BDF in Phase 2 ist ein Applikationsfehler. In einem solchen Fall wird der Datenbankbetrieb kurzzeitig eingestellt. Die Logging-Stage wird angewiesen alle erfolgreich abgeschlossenen Änderungen zu loggen. Sämtliche sich im System befindenden Datenbankanfragen werden zurückgewiesen. Im Anschluss wird die Datenbank vom Sekundärspeicher neu geladen. Die von der

abgebrochenen BDF vorgenommenen Änderungen müssen nicht rückgängig gemacht werden, da sie nur Daten im Hauptspeicher veränderten. Es müssen somit keine Maßnahmen für das Rückgängigmachen von BDF getroffen werden.

4.1.3 Datenbankanfragen

Die Ausführung von BDF wird über Datenbankanfragen angestoßen. Die Bearbeitung von Datenbankanfragen erfolgt asynchron. Der Sender der Anfrage kann sich nach dem Absenden anderen Aufgaben widmen. Am Ende der Bearbeitung der Anfrage durch das DBS wird das Ergebnis - die Datenbankantwort - an einen in der Anfrage angegebenen Empfänger gesendet.

Eine Datenbankanfrage in SpeeDB ist ein Quadrupel $(udf, arg, out, cont)$. Dabei ist *udf* eine Zahl, die angibt, welche BDF ausgeführt werden soll, und das Objekt arg das dabei zu verwendende Argument. Mit *out* wird angegeben in welche Warteschlange die Datenbankantwort eingereiht werden soll. Der Empfänger der Antwort wird so spezifiziert. Der Parameter *cont* kann ein beliebiges Objekt sein und dient der Ablage von Kontextinformationen der Clientanwendung. Diese Kontextinformationen sind in der Datenbankantwort enthalten und ermöglichen dem Empfänger die Zuordnung zu dem Vorgang, der die Anfrage auslöste. Der Parameter *cont* ist für das DBS irrelevant und von BDF nicht lesbar. Intern wird einer Datenbankanfrage noch eine ganzzahlige Versionsnummer zugewiesen, die für Zugriffe auf *Zwei-* und *Mehrversionenindizes* benötigt wird (Siehe auch die Abschnitte 4.1.5 und 4.3).

Eine *Datenbankantwort* ist ein Tripel $(cont, res, stat)$. Der Parameter *cont* enthält das Objekt mit den Kontextinformationen aus der Datenbankanfrage. Das Ergebnis der ausgeführten BDF findet sich in Form eines beliebigen Objektes im Parameter res. Zuletzt gibt es noch einen Statuscode mit *stat*, der eine Aussage über den Bearbeitungsstatus der Datenbankanfrage trifft. Die unterschiedlichen Statuscodes werden im Folgenden aufgelistet:

`READ_FINISHED`
: Die Bearbeitung der lesenden BDF wurde erfolgreich beendet.

`READ_CAUSED_ERROR`
: Während der Bearbeitung der lesenden BDF ist ein Fehler aufgetreten. Das Ergebnis ist vermutlich nicht konsistent.

`WRITE_ABORTED`
: Die Bearbeitung der schreibenden BDF wurde vor Beginn der 2. Phase und damit vor dem ersten *Write* abgebrochen.

`WRITE_COMITTED`
: Die Bearbeitung der schreibenden BDF wurde erfolgreich beendet und ihre Änderungen dauerhaft gespeichert.

`WRITE_CAUSED_ERROR`
: Während der Bearbeitung der schreibenden BDF ist ein Applikationsfehler aufgetreten. Ein Neustart des DBS ist erforderlich.

`REJECTED`
: Im Falle eines Applikationsfehlers werden unbearbeitete Datenbankanfragen mit diesem Statuscode beendet bevor der Neustart ausgeführt wird.

4.1.4 Komponenteninteraktion

Betrieb

Die Bearbeitung einer Datenbankanfrage beginnt bei der Ausführungs-Stage. Die in einer Datenbankanfrage angegebene BDF wird von hier mit dem ebenfalls angegebenen Argument und einer ganzzahligen Versionsnummer aufgerufen. Die Versionsnummer gibt je nach verwendetem *Datenbankmodus* einen Zustand (*serieller Modus*) oder eine Datenbankversion (*Snapshot Reads*) an (Siehe 4.1.5 und 4.3). Die BDF haben, wie in 4.1.2 bereits erwähnt, mit den Operationen `put`, `get` etc. Zugriff auf die Indizes. Bei jedem Zugriff auf einen Index wird die von der Ausführungs-Stage stammende Versionsnummer mit angegeben.

Nachdem die mit einer Datenbankanfrage aufgerufene BDF ausgeführt wurde, wird die Anfrage zur Logging-Stage geleitet oder, falls es eine lesende Anfrage war, die Datenbankantwort an den Empfänger verschickt. Auf der Logging-Stage loggt der Group-Comitter die schreibenden Datenbankanfragen in Log-Dateien auf der Festplatte (Logs). Nachdem die Änderungen dauerhaft gespeichert wurden, versendet er die Datenbankantworten an die Empfänger.

Genauere Informationen zum Datenbankbetrieb in Abhängigkeit des verwendeten Datenbankmodus finden sich im Abschnitt 4.1.5.

Initialisierung

Während der Initialisierung der Datenbank wird eine Methode der BDF aufgerufen, die die benötigten Indizes beim Indexmanager beantragt. Dieser instanziert daraufhin die gewünschten und zum Datenbankmodus(4.1.5) passenden Datenstrukturen und stellt sie den BDF zur Verfügung. Daraufhin wird die Logging-Stage initialisiert. Diese bestimmt zunächst den zu ladenden Sicherungspunkt (4.2) und teilt diese Information dem Index-Manager mit. Dieser lädt dann für jeden von den BDF beantragten Index den Inhalt aus dem Sicherungspunkt (4.2.5) von der Festplatte (CP). Nachdem der Sicherungspunkt geladen wurde, wiederholt die Logging-Stage die Ausführung der im Log befindlichen BDF, die noch nicht im Sicherungspunkt enthalten sind (Siehe 4.2.4). Danach kann der normale Datenbankbetrieb aufgenommen werden.

Synchronisation beim Erstellen von Sicherungspunkten

Für die verwendete *Logging-* und *Recovery-Strategie* ist beim Erstellen *transaktionskonsistenter Sicherungspunkte* eine Synchronisation zwischen der Ausführungs-Stage und der Logging-Stage erforderlich (Siehe 4.2.3). Bei der Frage, wie die Synchronisation umzusetzen sei, war der leitende Gedanke, dass das Erstellen eines Sicherungspunktes ein vergleichsweise seltenes Ereignis ist, und dass kurze Verzögerungen bei diesem Vorgang vertretbar sind. Wichtiger ist die Performanz bei der Bearbeitung normaler Datenbankanfragen, weswegen die Kosten für die Kommunikation gering gehalten werden sollten. Die gefundene Lösung kommt ohne Sperren und gemeinsame Variablen aus.

Die Logging-Stage kann der Ausführungs-Stage eine Nachricht über einen Eintrag in deren Warteschlange mit den Datenbankanfragen zukommen lassen. Dieser Eintrag wird von dem Thread ausgewertet, der auch die schreibenden BDF ausführt. Er kann auf eine Nachricht reagierend eine Funktion in der Logging-Stage aufrufen. So wird erreicht, dass Aktionen auf der Logging-Stage durchgeführt werden können, ohne dass im Hintergrund schreibende BDF aufgerufen werden.

Es folgt ein Beispiel für die Synchronisation, welches die Inhalte aus den Abschnitten 4.3.1 und 4.2 für das Verständnis voraussetzt. Angenommen die Datenbank wird im seriellen Modus betrieben und der Checkpointer möchte einen Sicherungspunkt erstellen lassen. Er reiht die Nachricht `START_CP` in die Warteschlange der Ausführungs-Stage ein und legt seine Tätigkeit nieder. Der Thread der Ausführungs-Stage erhält wenig später die Nachricht und ruft die Funktion `start_CP()` der Logging-Stage auf. Zunächst wird für jeden Index das momentan kleinste Element gesichert. Dann wird gewartet, bis der Group-Comitter alle Datenbankanfragen, die vor Beginn des Checkpoints ausgeführt wurden, geloggt hat. Zu diesem Zeitpunkt werden weder Daten in den Zugriffspfaden noch Daten in den Logs verändert und die Log-Datei kann gewechselt. Außerdem wird der Beginn des Checkpoints in einer Datei ($lastStartedCP$) vermerkt. Jetzt ist die Vorbereitung abgeschlossen und es muss lediglich noch der Thread gestartet werden, der im Hintergrund die reservierte Version auf den Sekundärspeicher kopiert. Die Funktion `start_CP()` ist beendet und der Thread der Ausführungs-Stage kann seiner eigentlichen Arbeit - der Bearbeitung von Datenbankanfragen - wieder nachgehen.

4.1.5 Datenbankmodi

SpeeDB kann in zwei unterschiedlichen Modi betrieben werden. Sie unterscheiden sich in der Unterstützung von der nebenläufigen Ausführung lesender BDF. Im *seriellen* Modus werden alle BDF nacheinander ausgeführt. Im Modus *Snapshot-Reads* werden n lesende BDF parallel zu einer schreibenden ausgeführt. In diesem Abschnitt werden die beiden Modi genauer beschrieben.

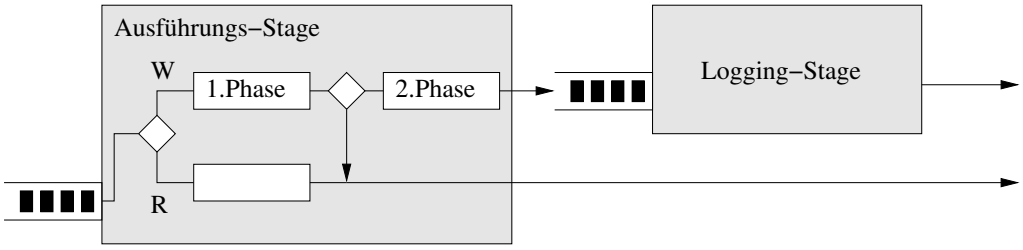


Abbildung 4.2: Anfragebearbeitung im seriellen Modus

Serieller Modus

In Abbildung 4.2 ist die Ausführungs-Stage für den seriellen Modus skizziert. Es gibt nur einen Thread, der nacheinander die Datenbankanfragen aus der eingehenden Warteschlange abarbeitet. Bei der Bearbeitung einer Datenbankanfrage wird zunächst getestet, ob eine lesende oder schreibende BDF ausgeführt werden soll. Handelt es sich um eine lesende BDF, so wird die Funktion mit dem in der Anfrage enthaltenen Argument ausgeführt. Das Ergebnis wird in die entsprechende Warteschlange des Empfängers eingereiht und die Bearbeitung der Anfrage ist abgeschlossen.

Im Falle einer schreibenden BDF wird zunächst die 1. Phase der Zweiphasentransaktion durchgeführt, in der nur lesende Zugriffe auf die Daten gestattet sind. Nur bei positivem Ausgang der 1. Phase wird die 2. Phase durchgeführt. Zwischenergebnisse aus der 1. Phase können für die Bearbeitung der BDF in der 2. Phase genutzt werden, um zu vermeiden, dass Datenzugriffe aus der ersten Phase wiederholt werden müssen. Nach Beendigung der 2. Phase wird die Datenbankanfrage zusammen mit dem Ergebnis in die Warteschlange der Logging-Stage eingereiht. Kommt es während der 1. Phase zu einem Abbruch der BDF, so wird das Zwischenergebnis - falls vorhanden - zusammen mit einem entsprechenden Statuscode (Siehe 4.1.3) unmittelbar in die Warteschlange des Empfängers eingereiht.

Der Vorteil dieses Modus ist der geringe Synchronisationsaufwand, da nur ein Thread die Datenbankanfragen bearbeitet. Parallel zu diesem Thread werden jedoch noch transaktionskonsistente Sicherungspunkte (4.2.2) im Hintergrund angelegt. Dies erfordert den Einsatz von Zweiversionen-Indizes (4.3.1), die auf Befehl den aktuellen Inhalt in einer unveränderlichen Version sichern können und Änderungen auf einer anderen Version vornehmen. Solange die veraltete Version noch für den Sicherungspunkt benötigt wird, erfolgen die Zugriffe auf die Zweiversionen-Indizes mit einer bestimmten Versionsnummer, welche bewirkt, dass die alte Version der Daten unverändert erhalten bleibt (4.3.1).

Auf den ersten Blick scheint der serielle Modus das Korrektheitskriterium der Serialisierbarkeit zu erfüllen. Alle Transaktionen werden nacheinander abgearbeitet. Es ist aber zu beachten, dass lesende BDF einen Datenbankzustand sehen, der zwar konsistent ist, aber unter Umständen noch nicht dauerhaft gespeichert wurde. Die Datenbankantworten zu lesenden Anfragen werden unmittelbar nach Beendigung der aufgerufenen BDF von der Ausführungs-Stage versendet. Schreibende Anfragen werden erst an die Logging-Stage

übergeben, bevor die Antworten versendet werden. Deshalb können kurz vor einem Systemabsturz Datenbankantworten zu lesenden Anfragen SpeeDB verlassen haben, die auf einem nicht wiederherstellbaren Zustand basieren. Diese Eigenschaft ist beim Einsatz von SpeeDB zu bedenken. Für XtreemFS wurde sie als vertretbar eingeschätzt, da nur lesende Anfragen betroffen sind. Das Problem kann aber auch beseitigt werden, indem man den Ablauf so ändert, dass eine Antwort zu einer lesenden Datenbankanfrage erst versendet wird, nachdem der gesehene Datenbankzustand persistiert wurde. Dies verringert jedoch den Transaktionsdurchsatz, denn lesende Anfragen müssen nun auf das Loggen vorangegangener schreibender Anfragen warten.

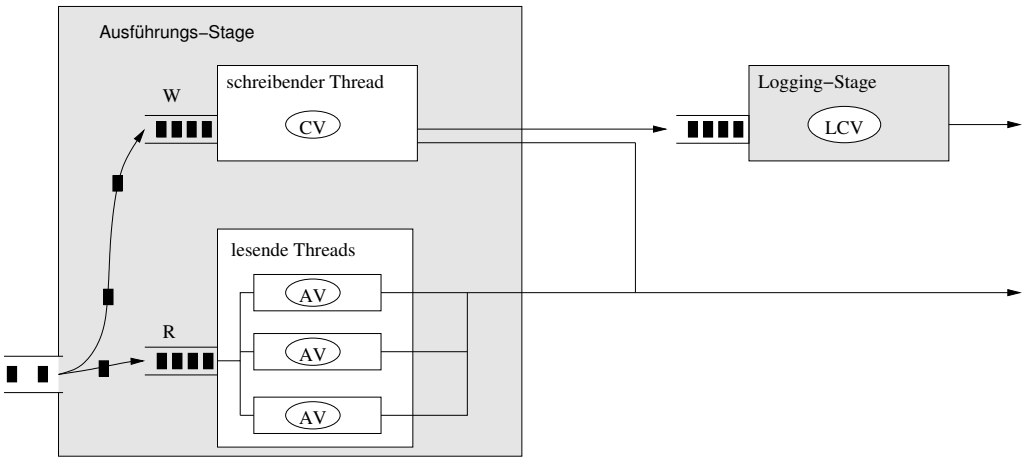


Abbildung 4.3: Anfragebearbeitung im Modus Snapshot-Reads

Snapshot-Reads Modus

Im Modus Snapshot-Reads gibt es auf der Ausführungs-Stage einen Thread für die Bearbeitung der schreibenden und mehrere Threads für die Bearbeitung der lesenden Datenbankanfragen (Abbildung 4.3). Eintreffende Datenbankanfragen werden in Abhängigkeit ihres Typs in die Warteschlange des schreibenden Threads oder in die der Gruppe mit den lesenden Threads eingereiht. Die einzelnen Threads sehen die Datenbank in unterschiedlichen Versionen. Der schreibende Thread sieht dabei immer die aktuellste Version. Die lesenden Threads hingegen sehen während der gesamten Zeit ihrer Ausführung den transaktionskonsistenten Datenbankzustand in dem die Datenbank zu Beginn der Ausführung war. Sie arbeiten also auf einen etwas veralteten, unveränderlichen Schnappschuss der Datenbank. Mit Mehrversionen-Indizes (4.3.2) ist ein paralleles Arbeiten der verschiedenen Threads auf den Daten ohne die Verwendung von Sperren möglich.

Die BDF werden im Modus Snapshot-Reads ähnlich wie im seriellen Modus ausgeführt. Die Datenbank liegt jetzt jedoch in mehreren Versionen vor. Der schreibende Thread arbeitet auf der jüngsten Version der Datenbank. Er pflegt die Variable *CurrentVersion (CV)* - also die größte Versionsnummer der Datenbank. Bevor die 2. Phase der BDF einer schreibenden

Datenbankanfrage ausgeführt wird, wird die Variable CV inkrementiert und ihr Wert der Datenbankanfrage als Versionsnummer zugewiesen. Die Zugriffe des schreibenden Threads auf die Mehrversionen-Indizes erfolgen erfolgen stets mit der Versionsnummer CV.

Auf der Logging-Stage wird die Variable *HighestComittedVersion (HCV)* gepflegt. Nach jedem *Gruppencommit* [13] wird sie auf den Wert der größten Versionsnummer aus der Menge der geloggten Datenbankanfragen gesetzt.

Auf der Ausführungs-Stage besitzt jeder Thread aus der Gruppe der lesenden Threads eine Variable *AccessedVersion (AV)*. Sie enthält die Versionsnummer des Datenbankzustandes, auf dem der Thread gerade arbeitet. Ist ein Thread nicht beschäftigt besitzt die Variable den Wert $+\infty$. *AccessedVersion* ist nach außen sichtbar und wird vom System benötigt, um bestimmen zu können, welche der existierenden Versionen noch benötigt werden und welche vom Speicher entfernt werden können.

Die Bearbeitung einer lesenden Datenbankanfrage sieht folgendermaßen aus: Zu Beginn holt sich ein Thread eine Anfrage aus der Warteschlange. Ist dies geschehen, setzt er seine Variable AV auf den aktuellen Wert von HCV. Daraufhin ruft er die angegebene BDF mit dem entsprechenden Argument auf. Existiert ein während der Ausführung gelesenes SWP in mehreren Versionen, so wird die größte Version, die kleiner oder gleich der Version AV ist, verwendet. Nachdem die BDF abgeschlossen ist, wird die Variable AV auf den Wert $+\infty$ gesetzt. Danach wird die Datenbankantwort an den Empfänger gesendet. Der Thread kann nun die nächste Anfrage aus der Warteschlange bearbeiten.

Der Vorteil des Modus Snapshot-Reads liegt in der parallelen Ausführung lesender Transaktionen. Ihre Ausführung ist von schreibenden Transaktionen unabhängig. Lesende Anfragen können ruhig länger dauern, da sie anders als im seriellen Modus keine anderen Transaktionen blockieren. So stellt auch die Anfertigung eines transaktionskonsistenten Sicherungspunktes keine Schwierigkeiten dar und kann ohne zusätzlichen Aufwand realisiert werden. Der Betrieb in diesem Modus erfüllt das Korrektheitskriterium der Serialisierbarkeit.

4.1.6 Serialisierung von Objekten

Sowohl die SWP in den Zugriffspfaden, als auch die Argumente der BDF können Instanzen beliebiger Java-Klassen sein. Das bringt den Vorteil, dass existierende Klassen nicht an SpeeDB angepasst werden müssen, wenn Instanzen von ihnen in der Datenbank abgelegt werden sollen.

Für das *logische Logging* (4.2.1) und das Erstellen von Sicherungspunkten (4.2.2) ist es notwendig SWP und Argumente der BDF in eine serialisierte Form zu bringen. Weil die Objektserialisierung von Java sehr langsam ist [47, 3], verlangt SpeeDB eine eigene Implementierung der Serialisierung und Deserialisierung der verwendeten Objekte. Dabei muss angegeben werden, wie ein Objekt in einen *Puffer* der Klasse *java.nio.ByteBuffer* geschrieben und von einem Puffer gelesen wird.

4.2 Logging und Recovery

Die Datenbank besteht aus Schlüssel-Werte-Paaren, die in Zugriffspfaden gehalten werden (4.1.1). Diese Zugriffspfade befinden sich im flüchtigen Hauptspeicher. In diesem Abschnitt wird beschrieben, wie die Änderungen an der Datenbank persistent gemacht werden und wie der jüngste transaktionskonsistente Datenbankzustand bei einem Neustart des DBS wiederhergestellt wird.

4.2.1 Logisches Logging

In SpeeDB werden Veränderungen an der Datenbank mit logischem Logging dauerhaft gespeichert. Es wird nicht geloggt, wie sich die Daten physisch ändern, sondern welche Operationen diese Änderungen verursacht haben. Der Datenbankzustand ändert sich durch die serielle Ausführung schreibender BDF. Eine schreibende Datenbankanfrage kann nur während der ersten Ausführungsphase, in welcher ausschließlich lesende Operationen gestattet sind, abgebrochen werden. Es bietet sich deshalb an, BDF als Einheit für das logische Logging zu verwenden und dabei lediglich die schreibenden BDF zu loggen, die auch die 2. Ausführungsphase passiert haben.

Wie in der Architekturskizze (Abbildung 4.1) dargestellt, werden zu loggende Datenbankanfragen von der Ausführungs-Stage an die Logging-Stage übergeben. Die Reihenfolge bleibt dabei erhalten. In der Logging-Stage werden alle zu loggenden Datenbankanfragen in die Log-Datei geschrieben. Abbildung 4.4 zeigt den Aufbau einer Log-Datei. Das Loggen einer BDF erzeugt einen Eintrag, welcher aus dem Tripel (BDF_ID, ARG_SIZE, ARG) besteht. Dabei wird in BDF_ID notiert, welche BDF aufgerufen wurde (Jede BDF ist mit einer ganzen Zahl identifizierbar 4.1.2). In ARG wird das serialisierte Argument (4.1.6), mit dem die Funktion aufgerufen wurde, und in ARG_SIZE die Länge des serialisierten Argumentes gesichert.

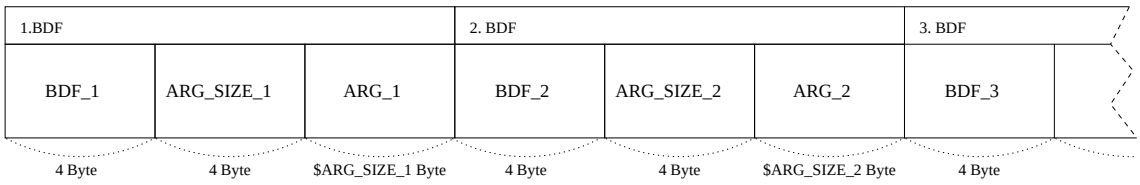


Abbildung 4.4: Aufbau einer Log-Datei

Vor dem Ende der Bearbeitung einer Datenbankanfrage und dem Versandt ihrer Datenbankantwort an den Empfänger, wird mit einem `FLUSH` erzwungen, dass die geloggten Änderungen auf den Sekundärspeicher geschrieben werden und nicht nur in einem Dateisystem-Cache im Hauptspeicher verweilen. Aus Gründen der Performanz werden dabei mehrere Datenbankanfragen gemeinsam geloggt (Gruppencommit [13]). Es werden bis zum Ablauf einer gewissen Zeit alle zu loggenden Anfragen in die Datei geschrieben. Dann wird ein `FLUSH` ausgeführt und die Bearbeitung der nun festgeschriebenen Anfragen durch den Versandt der zugehörigen Datenbankantworten an die Empfänger beendet. Der Grund für die

höhere Performanz des Gruppencommits ist, dass das Schreiben von größeren Puffern mit einer höheren Bandbreite erfolgt [15] und dass nicht bei jeder Anfrage ein aufwendiger `FLUSH` ausgeführt werden muss.

Es werden ausschließlich Informationen für die wiederholte Ausführung von BDF geloggt (*redo-only*). Das ist möglich, weil schreibende BDF seriell als Zweiphasentransaktionen ausgeführt werden und die angefertigten Sicherungspunkte transaktionskonsistent sind (Siehe unten). Ein Abbruch einer schreibenden BDF nach Beginn der 2. Phase ist auf einen Applikations- oder Systemfehler zurückzuführen und hat einen Neustart des DBS zur Folge. Da etwaige Änderungen durch die BDF nur an Daten im Hauptspeicher vorgenommen wurden, werden keine Informationen für das Rückgängigmachen (*undo*) benötigt.

4.2.2 Transaktionskonsistente Sicherungspunkte

Würde man sich auf das Loggen der Transaktionen beschränken, müssten bei der Wiederherstellung des Datenbankzustandes nach einem Systemausfall sämtliche geloggten Transaktionen erneut ausgeführt werden. Das führt ab einer gewissen Größe des Logs zu einem inakzeptablen Aufwand. Deshalb bietet SpeeDB das Erstellen von *transaktionskonsistenten Sicherungspunkten* oder auch *Checkpoints* an. Ein transaktionskonsistenter Sicherungspunkt ist eine Kopie der Datenbank - also der Zugriffspfade - in einem Zustand, der lediglich Änderungen erfolgreich abgeschlossener Transaktionen reflektiert. Bei der Recovery wird zunächst der jüngste Sicherungspunkt geladen und im Anschluss daran ein *REDO* für jede geloggte Transaktion vorgenommen, deren Änderungen noch nicht im Sicherungspunkt enthalten sind, da sie nach Beginn der Erstellung des Sicherungspunktes ausgeführt wurden.

Während der Anfertigung eines Sicherungspunktes muss der zu sichernde Datenbankzustand eingefroren werden. Um dies ohne die Blockade von schreibenden Transaktionen zu ermöglichen, werden für die Zugriffspfade Datenstrukturen, die eine nichtblockierende Anfertigung von *Schnappschüssen* unterstützen, verwendet (4.3).

Beim Erstellen eines Checkpoints wird für jeden Zugriffspfad der Datenbank ein Schnappschuss in einer eigenen Datei gesichert. Abbildung 4.5 zeigt das Format einer Datei mit dem Schnappschuss eines Zugriffspfades. In den ersten 4 Byte ist die Anzahl der SWP gesichert. Danach folgen alle SWP aus dem Zugriffspfad in aufsteigend sortierter Reihenfolge. Für jedes SWP wird ein Puffer mit der serialisierten Form des SWP BUF zusammen mit der Länge des Puffers in Byte BUF_SIZE gespeichert.

Das Erstellen von Sicherungspunkten kann von der Clientanwendung initiiert werden. Es können aber auch Vereinbarungen (*Checkpoint Policies*) für die selbständige Anfertigung von Checkpoints angegeben werden. Das DBS erstellt dann in Abhängigkeit von der Größe der Log-Datei, der Anzahl der geloggten BDF oder der abgelaufenen Zeit eigenständig Sicherungspunkte.

Bei der Anfertigung eines Sicherungspunktes wird immer die komplette Datenbank rausgeschrieben, da sonst ein aufwendiges Postprocessing beim Bau eines kompletten Sicherungspunktes aus den Änderungen und dem alten Sicherungspunkt notwendig wird. Zudem

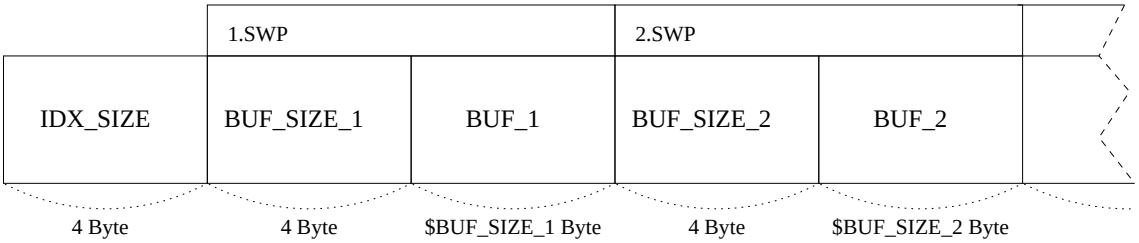


Abbildung 4.5: Aufbau einer Schnappschuss-Datei

erfordert diese Variante, dass man Veränderungen des aktuellen Datenbankzustandes im Vergleich zum alten Sicherungspunkt erkennen kann, was im seriellen Modus momentan nicht unterstützt wird.

4.2.3 Zusammenspiel von Logging und Checkpoints

Im vorigen Abschnitt wurde bereits erwähnt, dass der Aufwand für die Recovery mithilfe der Sicherungspunkte reduziert werden kann. Dabei blieben wichtige Punkte offen. So wurde nicht gesagt, wie der jüngste Checkpoint zu finden ist und ab welcher Position in der Log-Datei die geloggten BDF zu wiederholen sind. Es wird nun das verwendete *Logging- und Recovery-Schema* genauer beschrieben.

Jeder Sicherungspunkt besteht aus mehreren Schnappschuss-Dateien - eine pro Zugriffspfad. In einem *Ping-Pong-Schema* [37] werden zwei Dateigruppen $CpFiles0$ und $CpFiles1$ abwechselnd für die Erstellung eines Sicherungspunktes benutzt. Außerdem werden zwei Log-Dateien $LogFile0$ und $LogFile1$ für das Loggen der BDF verwendet.

Die Idee ist, in $LogFileX$ alle Änderungen zu loggen, die nach Beginn des in $CpFilesX$ gespeicherten Sicherungspunktes an der Datenbank vorgenommen wurden. In Abbildung 4.6 wird das Verfahren skizziert. Mit dem Beginn der Anfertigung eines Sicherungspunktes in $CpFiles1$ wird auch die Log-Datei gewechselt. Es ist dabei wichtig, dass lediglich neue, noch nicht im Sicherungspunkt reflektierte Änderungen in der neuen Datei $LogFile1$ geloggt werden. Erst nachdem die Anfertigung des Sicherungspunktes beendet wurde, kann ein neuer Sicherungspunkt erstellt werden. Geschieht dies, so wird der Sicherungspunkt in $CpFiles0$ gespeichert und nachfolgende BDF in $LogFile0$ geloggt.

Neben den Checkpoint- und Log-Dateien werden zwei weitere Dateien gepflegt. Zu Beginn der Anfertigung jedes Sicherungspunktes wird die aktuelle Ckeckpoint-Nummer (0 oder 1) in die Datei $lastStartedCP$ geschrieben. Ebenso wird nach der Fertigstellung diese Nummer in die Datei $lastFinishedCP$ geschrieben. Diese Dateien dienen dem Zweck, den jüngsten vollständigen Sicherungspunkt zu erkennen.

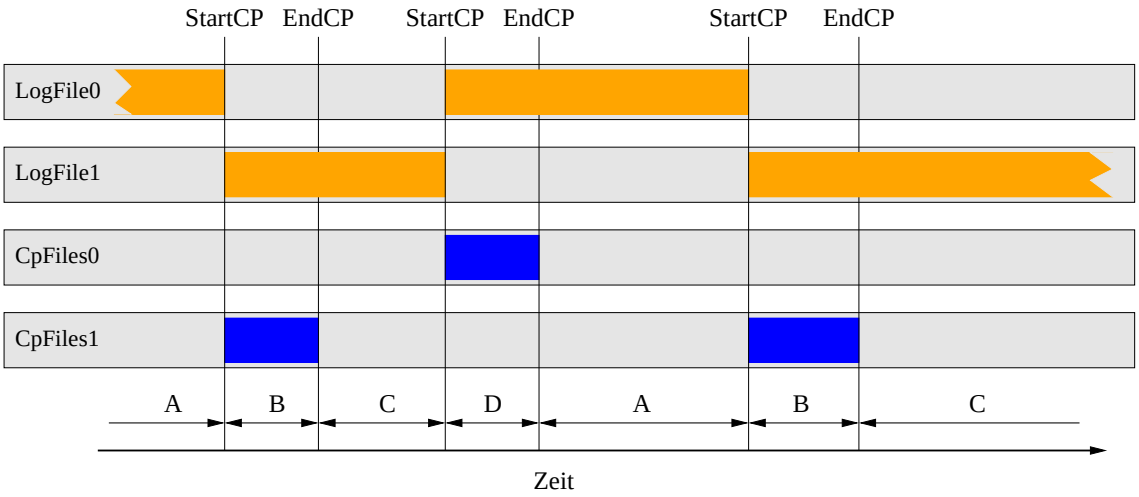


Abbildung 4.6: Logging- und Recovery Schema

4.2.4 Recovery

Bei der Recovery muss als erstes herausgefunden werden, zu welchem Zeitpunkt der Datenbankbetrieb beendet wurde, um bestimmen zu können, welche der existierenden Sicherungspunkt- und Log-Dateien für die Wiederherstellung der Datenbank zu verwenden sind. In Abbildung 4.6 erkennt man, dass zwischen vier Situationen A, B, C und D unterschieden werden muss. Der Inhalt der Dateien $lastStartedCP$ und $lastFinishedCP$ gibt Auskunft, welche der genannten Situationen vorliegt.

In Situation A enthalten beide Dateien den Wert 0, d. h. der Sicherungspunkt in $CpFiles0$ wurde vollständig gespeichert. Bei der Recovery wird zunächst dieser Sicherungspunkt geladen und im Anschluss daran die in $LogFile0$ geloggten BDF ausgeführt.

In Situation B wurde die Anfertigung eines Sicherungspunktes in $CpFiles1$ begonnen, aber nicht abgeschlossen. Bei der Wiederherstellung der Datenbank wird deshalb der ältere aber abgeschlossene Sicherungspunkt aus $CpFiles0$ geladen. Danach werden zuerst die geloggten BDF aus $LogFile0$ und dann die aus $LogFile1$ ausgeführt. Die Situationen C und B werden analog behandelt. Im Folgenden werden noch einmal die Recoveryaktionen für die unterschiedlichen Fälle zusammengefasst:

A → `LOAD CpFiles0; REDO LogFile0;`

B → `LOAD CpFiles0; REDO LogFile0; REDO LogFile1;`

C → `LOAD CpFiles1; REDO LogFile1;`

D → `LOAD CpFiles1; REDO LogFile1; REDO LogFile0;`

Erst nach dem Abschluss der Recovery kann der normale Datenbankbetrieb wieder aufgenommen werden.

Die vorgestellte Logging- und Recovery-Strategie besitzt Eigenschaften, die eine gute Performanz erwarten lassen. Logisches, redo-only Logging minimiert das Log-Volumen. Sämtliche lesenden und schreibenden Dateizugriffe erfolgen rein sequentiell, wodurch auf magnetischen Festplatten wesentlich höhere Bandbreiten erzielt werden [36]. Es wird empfohlen das Logging und das Checkpointing auf verschiedenen Festplatten durchzuführen, sodass die Operationen sich nicht gegenseitig der Bandbreite berauben. Der sequentielle Zugriff vereinfacht auch den Einsatz von Komprimierverfahren, da variable Blockgrößen nicht zu Problemen bei der Allozierung und Fragmentierung führen [9]. Auch die Anfertigung eines Backups während des Datenbankbetriebes (*Hot-Backup*) wird durch das sequentielle Schreiben begünstigt.

4.2.5 Wiederherstellung von Zugriffspfaden

Beim Erstellen eines Schnappschusses von einem Zugriffspfad werden die enthaltenen Schlüssel-Werte-Paare in sortierter Reihenfolge auf den Sekundärspeicher geschrieben. Wird die Datenbank geladen, so stellt sich die Aufgabe während des Einlesens der Schnappschüsse die Zugriffspfade zu rekonstruieren. Intuitiv könnte man durch sukzessives Einfügen der sortierten Schlüssel-Werte-Paare den Index wiederherstellen. Die Komplexität dieser Vorgehensweise ist bei einem Index mit n Elementen $O(n \log n)$ und somit aufwendiger als ein hier vorgestelltes Verfahren, welches den Zugriffspfad mit linearem Aufwand $O(n)$ erstellt. Das Verfahren führt während der Rekonstruktion keinen Schlüsselvergleich durch. Diese Eigenschaft wird sich als besonders vorteilhaft bei aufwendigen Schlüsselvergleichen, wie im Falle von langen Zeichenketten, erweisen.

Der Algorithmus 4.1 wurde für AVL-Bäume entwickelt, kann jedoch auch für T-Bäume angepasst werden. Die Eingabe ist eine aufsteigend sortierte Liste mit Schlüssel-Werte-Paaren zusammen mit der Anzahl der Einträge in der Liste. Es genügt, wenn die Liste als Stream verfügbar ist, da beginnend mit dem ersten Element immer nur auf den Nachfolger zugegriffen wird. Das Ergebnis des Algorithmus ist ein AVL-Baum, in dem der maximale Längenunterschied zweier Pfade von der Wurzel zum Blatt 1 ist.

In Zeile 3 wird ermittelt wieviele Ebenen der größte binäre Baum hat, dessen Pfade von der Wurzel zu den Blättern gleich lang sind und dessen Knotenanzahl kleiner gleich $|L|$ ist. Die Variable $leafCount$ ist die Anzahl der Blätter, die zusätzlich an diesen perfekt balancierten Baum herangehängt werden, damit er Platz für alle Elemente aus L bietet. Dann wird die rekursiv arbeitende Funktion `buildAVL` aufgerufen, dessen Ergebnis die Wurzel des konstruierten AVL-Baumes ist.

Die Funktion baut zunächst durch rekursiven Abstieg den linken Teilbaum (12-16). Dann wird ein Knoten erzeugt mit dem Balance-Wert 0, der ein Schlüssel-Wert-Paar aus der Liste L aufnimmt (17-20). Falls die Funktion sich gerade in der Ebene der zusätzlichen Blätter befindet, wird die Anzahl der zusätzlich anzuhängenden Blätter $leafCount$ dekrementiert (21-24). Wenn es das letzte zusätzlich angehängte Blatt war, wird es in der Variablen $leaf2rebalance$ gespeichert. Dann wird der rechte Teilbaum gebaut (25-29) und im Anschluss der Knoten mit seinem linken und rechten Teilbaum verknüpft (30-33).

Nachdem der Baum konstruiert wurde, sind eventuell noch die Balance-Werte auf dem Pfad vom zuletzt angehängten Blatt $leaf2rebalance$ zur Wurzel hin zu korrigieren (7-9). Der Balance-Wert jedes Knotens ist -1, wenn das Blatt $leaf2rebalance$ sich im linken Teilbaum des Knotens befindet sonst 0 (36-43).

In Abbildung 4.7 wird ein mit diesem Verfahren konstruierter Baum gezeigt. Die Struktur entspricht einem perfekt balancierten binären Baum, an den von links zusätzliche Blätter gehängt wurden. In der Abbildung sind diese zusätzlichen Blätter grün dargestellt. Für die Knoten auf dem Pfad vom zuletzt angehängtem Blatt mit dem Schlüssel 5 zur Wurzel wurden die Balance-Werte eingezeichnet. Alle anderen Knoten besitzen den Balance-Wert 0.

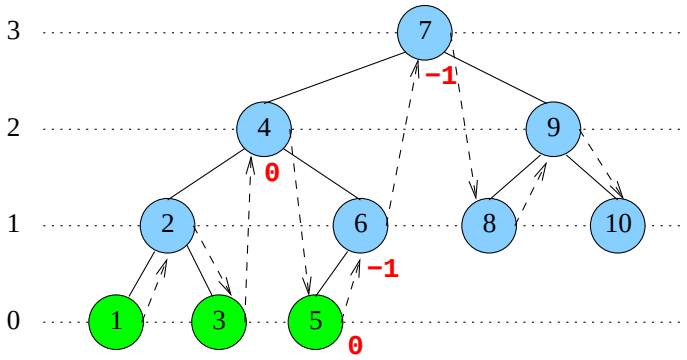


Abbildung 4.7: Rekonstruierter AVL-Baum

4.3 Zugriffspfade

In SpeeDB können verschiedene Zugriffspfade von den BDF genutzt werden. Sie unterscheiden sich in drei Kriterien: dem unterstützten Datenbankmodus (4.1.5), der Grundstruktur (AVL-,T-,B-Baum,etc.) und dem Schlüsseltyp. Die Bereitstellung von Indizes mit unterschiedlichen Schlüsseltypen ist motiviert durch schnellere Zugriffsoperationen. Indizes mit primitiven Datentypen als Schlüssel bieten eine bessere örtliche Lokalität und geringere Anzahl an Indirektionen beim Schlüsselzugriff.

Die Zugriffspfade müssen es erlauben, konsistente Schnappschüsse zu erstellen, ohne dass der normale Datenbankbetrieb eingeschränkt wird. Um dies zu ermöglichen, werden Indizes eingesetzt, die mehrere Versionen speichern können und einen parallelen Zugriff ohne den Einsatz von Sperren ermöglichen. Im seriellen Modus wird neben der aktuellen Datenbank-Version, auf die lesend und auch schreibend bei der Bearbeitung von Datenbankanfragen zugegriffen wird, eine weitere, veraltete Version für die Erstellung eines Sicherungspunktes benötigt. Operiert die Datenbank im Modus Snapshot-Reads, werden weitere Versionen für den lesenden Zugriff durch die parallele Bearbeitung von BDF benötigt. Im Abschnitt 4.3.1 wird ein eigenes Verfahren vorgestellt, welches die Entwicklung von Datenstrukturen ermöglicht, die Daten in einer aktuellen, veränderlichen und

Input : L - Eine sortierte Liste mit Schlüssel-Werte-Paaren sowie $|L|$
Output : T - Ein AVL-Baum

```
1  begin
2      leaf2rebalance ⟵ null;
3      depth ⟵ ⌊log2(|L| + 1)⌋;
4      leafCount ⟵ |L| − 2^depth + 1;
5      T.size ⟵ |L|;
6      T.root ⟵ buildAVL(depth,L);
7      if leaf2rebalance ≠ null then
8          setBalances (node2rebalance);
9      end
10 end
11 function buildAVL(level,L)
       // baue linken Teilbaum
12     if level > 1 or level = 1 and leafCount > 0 then
13         left ⟵buildAVL (level − 1,L);
14     else
15         left ⟵ null;
16     end
       // erzeuge neuen Knoten
17     keyValPair ⟵ L.pop();
18     node.key ⟵ keyValPair.key;
19     node.value ⟵ keyValPair.value;
20     node.balance ⟵ 0;
       // dekrementiere Blattzahl und merke letztes Blatt
21     if level = 0 then
22         if leafCount = 1 then leaf2rebalance ⟵ node;
23         leafCount ⟵ leafCount − 1;
24     end
       // baue rechten Teilbaum
25     if level > 1 or level = 1 and leafCount > 0 then
26         right ⟵buildAVL (level − 1, L);
27     else
28         right ⟵ null;
29     end
       // verbinde Knoten mit rechtem und linkem Teilbaum
30     if left ≠ null then left.parent ⟵ node;
31     node.left ⟵ left;
32     if right ≠ null then right.parent ⟵ node;
33     node.right ⟵ right;
34     return node;
35 end function
36 function setBalances(node)
37     while node.parent ≠ null do
38         if node.parent.left = node then
39             node.parent.balance ⟵ −1;
40         end
41         node ⟵ node.parent;
42     end
43 end function
```

Algorithmus 4.1 : Rekonstruktion eines AVL-Baumes

maximal einer älteren, eingefrorenen Version bereitstellen können. Für das Design von Datenstrukturen, die eine aktuelle, veränderliche Version und mehrere veraltete, eingefrorene Versionen bereitstellen, wurde ein Verfahren aus der Literatur ausgewählt, welches kurz im Abschnitt 4.3.2 vorgestellt wird.

Veraltete Versionen, die nicht mehr benötigt werden, müssen aus der Datenbank entfernt werden, damit der von ihnen verwendete Platz im Hauptspeicher wieder frei wird. Dieser Säuberungsprozess sollte den Datenbankbetrieb nicht blockieren und möglichst schnell durchführbar sein. Ein naiver Ansatz für einen Zugriffspfad mit mehreren Versionen wäre, bei jedem Schlüssel-Werte-Paar den Schlüssel mit der Version zu konkatenieren. Das widerspricht jedoch den eben genannten Zielen, da beim Entfernen veralteter Versionen kein anderer lesender oder schreibender Zugriff auf den Index erfolgen kann und die evtl. notwendige Restrukturierung des Baumes aufwendig ist. In einem AVL-Baum mit n Knoten wäre der Aufwand für das Entfernen von x Elementen $O(x \log_2 n)$. Die hier vorgestellten Verfahren stellen eine bessere Lösung dar.

4.3.1 Zweiversionen-Index

Es wird nun der Entwurf eines Zugriffspfades beschrieben, der zwei Versionen - eine aktuelle, veränderliche für den Datenbankbetrieb und eine veraltete, unveränderliche für das Erstellen eines Sicherungspunktes - bereitstellt. Das Verfahren macht sich dabei das Wissen über die Art des Zugriffes auf die ältere Version zu Nutze. Für die Anfertigung eines Schnappschusses wird nämlich ein kompletter *Scan* durchgeführt und es ist kein wahlfreier Zugriff erforderlich. Es genügt auf das kleinste Element der Version und auf sämtliche Nachfolger zugreifen zu können.

Es liegt die Idee zugrunde, während des normalen Datenbankbetriebes für jeden Index eine einfach verkettete Liste vom kleinsten Element des Index bis zum größten zu pflegen. Zu Beginn eines Sicherungspunktes werden die kleinsten Elemente jedes Index ermittelt und die Listen eingefroren. Für jeden Index wird dann jedes Element der Liste in sortierter Reihenfolge auf den Sekundärspeicher geschrieben. Sobald alle Indizes gesichert wurden, können die Listen wieder gepflegt werden. Änderungen an der Liste, die während der Erstellung des Sicherungspunktes unterlassen wurden, müssen nachgeholt werden, bevor ein neuer Sicherungspunkt erstellt werden kann.

Sei T ein beliebiger baumartiger Zugriffspfad und N die Menge aller Knoten von T, die Schlüssel-Werte-Paare enthalten. Bei einem AVL-Baum wäre N die Menge aller Knoten des Baumes, bei einem B+-Baum wäre N nur die Menge der Blätter des Baumes. Jeder Knoten $n \in N$ besitzt einen Zeiger $n.successor$ der auf den Nachfolger also den Knoten mit den nächstgrößeren Schlüsseln zeigt. Die Zeiger auf den Nachfolger werden für die Bildung der oben erwähnten Liste, die die im Sicherungspunkt zu sichernden Knoten enthält, benötigt. Für den Baum T wird ein Zeiger *start* eingeführt, der auf den Knoten zeigt, bei dem diese Liste beginnt. Die Methode $predecessor(n)$ liefert für einen Knoten $n \in N$ den Knoten aus N, der den nächstkleineren Schlüssel enthält. Entsprechend liefert die Methode $successor(n)$ den Knoten mit dem nächstgrößeren Schlüssel. Die Menge $dirtyNodes \subset N$

nimmt Knoten auf, bei denen der Zeiger auf den Nachfolger aufgrund der Anfertigung eines Sicherungspunktes nicht korrekt gesetzt wurde. Voraussetzung für das Erstellen eines Sicherungspunktes ist, dass $dirtyNodes = \emptyset$ gilt.

Mit diesen strukturellen Mitteln kann nun die Arbeitsweise des Verfahrens erklärt werden. Das Verfahren behandelt die folgenden Ereignisse, welche infolge der Indexoperationen *put*,*update* und *delete* auftreten können:

createNewNode - ein neuer Knoten $newNode \in N$ wird angelegt

updateNode - Schlüssel oder Daten eines Knotens $updatedNode \in N$ ändern sich

removeNode - ein Knoten $removedNode \in N$ wird entfernt

Abhängig davon, ob gerade ein Sicherungspunkt erstellt wird oder nicht, müssen die Ereignisse unterschiedlich behandelt werden. Wir betrachten zunächst die Behandlung der Ereignisse außerhalb der Anfertigung eines Sicherungspunktes:

createNewNode
$newNode.successor \longleftarrow successor(newNode)$
$predecessor(newNode).successor \longleftarrow newNode$
Der neue Knoten wird in die Liste der in einem Sicherungspunkt zu sichernden Knoten eingehängt.

updateNode
Es wird nichts unternommen, da der Knoten bereits in der Liste existiert.

removeNode
$predecessor(removedNode).successor \longleftarrow successor(removedNode)$
Der Knoten wird aus der Liste entfernt.

Wird nun ein Sicherungspunkt erstellt, so wird in dem zu sichernden transaktionskonsistenten Zustand der Knoten mit dem kleinsten Schlüssel ermittelt und im Zeiger *start* hinterlegt. Die Liste der zu sichernden Knoten wird während der Anfertigung des Sicherungspunktes eingefroren, indem auf die oben genannten Ereignisse während der Anfertigung wie folgt reagiert wird:

createNewNode
$newNode.successor \longleftarrow successor(newNode)$
$dirtyNodes.add(predecessor(newNode))$
Da die Liste mit den zu sichernden Knoten nicht verändert werden darf, wird der Zeiger auf den Nachfolger beim Vorgänger des neu eingefügten Knotens nicht gesetzt. Um das Setzen des korrekten Zeigers nach der Fertigstellung des Sicherungspunktes nachholen zu können, wird der Vorgänger des neuen Knotens in die Menge $dirtyNodes$ eingefügt.

updateNode

$newNode = copy(node2update)$
$newNode.successor \longleftarrow successor(newNode)$
$dirtyNodes.add(predecessor(newNode))$
Bei einem Update wird der zu ändernde Knoten kopiert und die Änderungen an der Kopie vorgenommen. Der Vorgänger des neuen Knotens wird wiederum der Menge $dirtyNodes$ zugeführt.

removeNode

$dirtyNodes.add(predecessor(removedNode))$
Beim Vorgänger des gelöschten Knotens wird die Korrektur des Zeigers auf den Nachfolger zu einem späteren Zeitpunkt ausgeführt, um die Liste der zu sichernden nicht zu verändern.

Nach der Fertigstellung eines Sicherungspunktes werden die Zeiger auf die Nachfolger der Knoten aus $dirtyNodes$ korrigiert ($\forall n \in L : n.successor = successor(n)$). Dieser Prozess blockiert den Datenbankbetrieb, da ein konsistentes Traversieren des Baumes nicht während schreibender Zugriffe möglich ist. Er kann aber in beliebig kleinen Schritten abgearbeitet werden, wodurch längere Unterbrechungen des Datenbankbetriebs ausgeschlossen werden können.

Für die Anfertigung eines Sicherungspunktes werden ältere, nur für den Sicherungspunkt benötigte Versionen von veränderten oder aus dem Baum entfernten Knoten im Speicher gehalten. Bei einer Implementierung des Verfahrens in einer Sprache mit manueller Speicherverwaltung (ohne Garbage-Collector) muss der von den Knoten belegte Speicher explizit wieder freigegeben werden, sobald der Sicherungspunkt komplett ist. Das lässt sich erreichen, indem man die Objekte in dem Moment, wo feststeht, dass sie nur noch für den Sicherungspunkt benötigt werden, in eine Liste einfügen und bei Fertigstellung des Sicherungspunktes den Speicher sämtlicher Objekte dieser Liste freigibt.

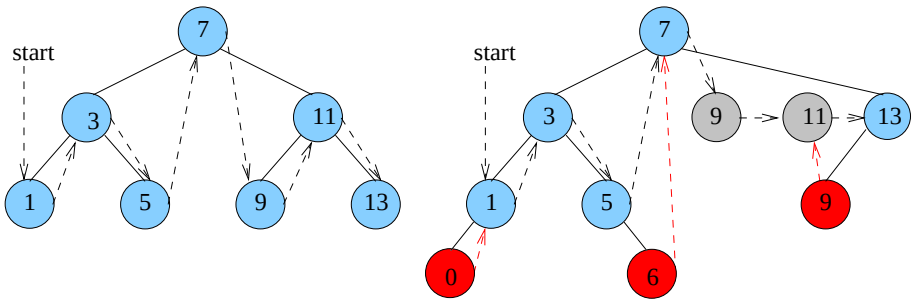


Abbildung 4.8: Binär-Baum mit 2 Versionen

In Abbildung 4.8 wird das Verfahren am Beispiel eines binären Baumes dargestellt. Die durchgehenden Linien symbolisieren die Zeiger auf den Vaterknoten, sowie auf das linke und das rechte Kind. Die gestrichelten Pfeile zeigen auf den Nachfolger in der Liste. Links

in der Abbildung sieht man einen Baum unmittelbar vor dem Beginn eines Sicherungspunktes. Rechts wird die Struktur des Baumes während des Erstellens des Sicherungspunktes gezeigt. Neu angelegte Knoten sind rot und entfernte Knoten grau dargestellt. Die schreibenden Indexoperationen *put(6)*, *put(0)*, *update(9)* und *delete(11)* führten zu den Änderungen im Zugriffspfad. Die Menge der Knoten mit inkorrekten Zeigern auf die Nachfolger ist $\{5, 7, 9(neu)\}$.

Die Synchronisation des Checkpointers mit der Ausführung der Transaktionen erfordert, dass der Transaktionsbetrieb für kurze Zeiten blockiert wird. Am Anfang werden in einem transaktionskonsistenten Zustand die kleinsten Elemente der Indizes gesichert. Am Ende müssen die Zeiger der Knoten aus der Liste *dirtyNodes* korregiert werden, was jedoch in mehreren Phasen geschehen kann. Ansonsten können die Threads völlig eigenständig auf den Daten arbeiten.

Die zusätzlichen Zeiger in der Datenstruktur erzeugen einen höheren Speicherbedarf. Im Falle eines AVL-Baumes ist es ein Zeiger pro Schlüssel-Werte-Paar im Falle eines T-Baumes mit einer Knotengröße von ns und maximaler Füllung ist es ein Zeiger je ns Schlüssel-Werte-Paare. In der Zeit, wo gerade kein Sicherungspunkt erstellt wird, können die Zeiger auf den Nachfolger genutzt werden, um Traversierungen bei Bereichsabfragen zu umgehen und somit die Anfragen zu beschleunigen.

4.3.2 Mehrversionen-Index

Ein *Mehrversionen-Index* muss neben einer aktuellen, veränderlichen Version, die bei der Ausführung von schreibenden BDF verwendet wird, mehrere ältere, unveränderliche Versionen für die Bearbeitung lesender BDF und das Erstellen von Sicherungspunkten bereitstellen. In der Veröffentlichung *"Logical and Physical Versioning in Main Memory Databases"* [35, 6] wurde ein Verfahren vorgestellt, mit dem so ein Index basierend auf baumartigen Strukturen entwickelt werden kann. Das Besondere an dem Verfahren ist, dass lesende Transaktionen ohne den Einsatz von Sperren auf die alten Versionen zugreifen können. Teile des Verfahrens wurden zur Zeit der Veröffentlichung in einer viel genutzten Softwareplattform innerhalb *Bell Labs* eingesetzt. Das komplette Verfahren wurde im *Dalí Main Memory Storage Manager* [7] implementiert.

Die Autoren unterscheiden zwischen *logischer* und *physischer Versionierung* von Datenelementen. Die *logische Versionierung* entspricht der herkömmlichen Idee des Haltens mehrerer Versionen eines Datenelementes: Wird ein Datenelement geändert, so wird zusätzlich zur alten eine neue Version angelegt. Eine ältere Version kann erst entfernt werden, wenn keine aktuelle oder zukünftige Transaktion diese Version benötigt.

Mit dem Begriff *physische Versionierung* wird eine Technik bezeichnet, mit der man baumartige Zugriffspfade verändern kann, ohne lesende Transaktionen zu blockieren. Es basiert auf der Annahme, dass das Ändern eines Zeigers oder einer Referenz innerhalb einer Datenstruktur atomar erfolgt (das ist in der Sprache Java gegeben [16]). Werden bei einer schreibenden Transaktion Änderungen an Knoten des Zugriffspfades erforderlich, so werden diese Änderungen nicht an den Knoten sondern an Kopien (physischen Versionen)

dieser vorgenommen. Genauergenommen wird bei einer schreibenden Operation ein Teil des Baumes, der sich bei Ausführung der Operation ändern würde, kopiert. Die Operation wird dann auf der Kopie ausgeführt. Am Ende wird die neue physische Version atomar in den Baum eingehängt. Die alte physische Version, bleibt solange erhalten, bis keine lesende Operation mehr auf ihr arbeitet und wird dann vom Speicher entfernt. Beim Löschen eines Datenelementes darf der Eintrag, der auf dieses Element zeigt, erst aus dem Zugriffspfad entfernt werden, wenn es keine Transaktion geben kann, die auf eine ältere Version (vor dem Löschen) zugreift.

In der Veröffentlichung geben die Autoren weitere Informationen wie und wann ältere logische und physische Versionen vom Speicher entfernt werden können. Außerdem beschreiben sie detailliert ihr Verfahren der logischen und physischen Versionierung am Beispiel des T-Baums.

4.4 Warteschlangen der Stages

Bei der Stage-Architektur besitzt jede Stage eine Warteschlange, in die zu bearbeitende Aufträge eingereiht werden. Mit der Länge der Warteschlange wächst auch die Zeit, die ein Auftrag benötigt, um diese Stage zu passieren. Angenommen man sendet 3 Millionen Aufträge mit einer Geschwindigkeit von 300 Aufträgen pro ms an eine Stage, die nur 100 Aufträge pro ms bearbeiten kann. Nach 10 s ist der letzte Auftrag abgesandt. Zum gleichen Zeitpunkt sind von der Stage aber nur 1 Million Aufträge bearbeitet worden. Das bedeutet, dass eine Warteschlange mit einer Länge von 2 Millionen Aufträgen entsteht, die erst nach weiteren 20 s abgearbeitet ist. Die maximale Bearbeitungszeit unter den 3 Millionen Aufträgen beträgt also mindestens 20 s.

Um solch hohe Bearbeitungszeiten zu verhindern, kann man den Warteschlangen eine maximale Kapazität zuweisen. Will eine Stage einen Auftrag in die volle Warteschlange einer anderen Stage einfügen, so kann die einfügende Stage blockieren und so einen Rückstau (*backpressure*) erzeugen [49]. Dieser Rückstau stellt zum einen ein Signal dar, denn eine vordere Stage kann auf ihn reagieren, indem sie neue Requests ablehnt. Zum anderen wirkt er als *Scheduler*, da blockierte Stages Ressourcen freigeben, die von überlasteten Stages verwendet werden können.

In SpeeDB ist die Logging-Stage der Flaschenhals. Da sie die letzte Stage ist, wurde den Warteschlangen der vorderen Stages eine maximale Kapazität zugewiesen. Bei hoher Last wird ein Rückstau erzeugt, der von der Clientanwendung behandelt werden kann. In einer ersten Implementierung stellte sich heraus, dass nach der Einführung einer Kapazität zwar die mittlere Bearbeitungszeit einer Anfrage abnahm, der Transaktionsdurchsatz jedoch ebenfalls. Der Grund dafür war, dass die blockierte Stage ihre Tätigkeit wieder aufnahm, sobald mindestens ein Auftrag in die vorher gefüllte Warteschlange eingereiht werden konnte. Nachdem dieser und evtl. wenige andere eingereiht wurden, war die Warteschlange wieder voll und die Tätigkeit musste aufs Neue niedergelegt werden. Dieser ständige Wechsel hat einen höheren Kommunikationsaufwand beim Threadscheduling zur

Folge. Gleichzeitig wird die zugeteilte Rechenzeit von einem Thread nicht vollständig ausgeschöpft, wodurch sich die Anzahl der *Kontextwechsel* (*Contextswitches*) erhöht.

Für die Stages in SpeeDB wurde eine blockierende Warteschlange erdacht, die neben der Kapazität c einen Schwellwert s mit $0 \leq s < c$ als Parameter annimmt. Blockierte, wartende Stages werden erst benachrichtigt, wenn sich der Füllstand der Warteschlange auf s gesenkt hat und somit $c - s$ freie Plätze verfügbar sind. Wird ein wartender Thread aufgeweckt, kann er garantiert $c - s$ Aufträge bearbeiten und deren Ergebnisse in die Warteschlange einreihen. Untersuchungen im Kapitel 5 belegen, dass diese Variante der Warteschlange die Anzahl der Kontextwechsel erheblich reduziert.

Kapitel 5

Evaluierung

Das im vorangegangenem Kapitel entworfene HSDBS wurde in der Diplomarbeit prototypisch implementiert. In diesem Kapitel wird zunächst über den aktuellen Stand der Implementierung berichtet. Dann werden die Ergebnisse der Messungen verschiedener Indizes vorgestellt. Die Performanz des entwickelten DBS wird in Abschnitt 5.4 analysiert. Fokussiert werden dabei der Transaktionsdurchsatz und die mittlere Antwortzeit einer Transaktion, die Geschwindigkeiten von Logging- und Recoveryaktionen sowie die Bedeutung unterschiedlicher Warteschlangeneinstellungen. Sämtliche Messungen wurden auf einem Referenzsystem durchgeführt, welches in Abschnitt 5.2 beschrieben wird. Am Ende des Kapitels wird im Abschnitt 5.5 auf Eigenschaften des *Java-Garbage-Collectors* eingegangen, die sich entscheidend auf die Eigenschaften des DBS auswirkten.

5.1 Stand der Implementierung

Im Rahmen der Diplomarbeit wurden zunächst der AVL-Baum und der T-Baum in unterschiedlichen Varianten mit dem Ziel implementiert, die Zugriffsgeschwindigkeiten der Datenstrukturen zu vergleichen.

Im Anschluss wurde die Entwicklung des im vorigen Kapitel beschriebenen DBS SpeeDB begonnen. Der aktuelle Stand ist ein funktionierender Prototyp, der momentan allerdings nur im seriellen Modus(4.1.5) betrieben werden kann. Für den Modus Snapshot-Reads fehlt noch die Implementierung eines Mehrversionen-Index (4.3.2). Als Zweiversionen-Index für den seriellen Modus wurde ein Zweiversionen-AVL-Baum nach dem im Abschnitt 4.3.1 beschriebenen Verfahren implementiert. Die Logging- und Recovery-Strategie wurde wie beschrieben implementiert. Die Anfertigung von Sicherungspunkten kann jedoch bis jetzt nur manuell angestoßen werden, da die Angabe von Vereinbarungen für das Erstellen von Sicherungspunkten (siehe 4.2.2) noch nicht unterstützt wird.

Der implementierte Prototyp fertigt beim Austausch von Objekten mit der Clientanwendung keine Kopien dieser Objekte an, wodurch die Clientanwendung direkten Zugriff auf

die Daten der Anwendung hat. Das birgt ein hohes Potential für die Entstehung von Inkonsistenzen in sich, steigert aber auch die Leistung, da keine Objektkopien erzeugt werden müssen [15]. In der Zukunft wird das DBS per Default eine saubere Trennung zwischen den Daten der Clientanwendung und der Datenbank vornehmen und nur über eine Option diesen performanteren Betrieb zur Wahl stellen.

5.2 Referenzsystem

Alle Messungen wurden auf demselben System durchgeführt. Es besitzt eine Quadcore-CPU des Typs Intel Xeon E5335 und 4GB Hauptspeicher. Das Betriebssystem ist ein 64Bit Linux mit dem Kernel 2.26.22.1. Als virtuelle Maschine wurde die 32Bit-Version des Java SE Runtime Environment 6 Update 10 von Sun verwendet. Gegen die Verwendung der 64Bit-Version sprach der höhere Arbeitsspeicherbedarf dieser in Kombination mit dem vergleichsweise kleinen Arbeitsspeicher des Systems.

Als Sekundärspeicher stehen identische SAS-Platten zur Verfügung. Eine von ihnen wurde für die Speicherung der Logdateien und die andere für die Sicherungspunktdateien verwendet, um beiden Vorgängen die maximale Bandbreite bei Zugriffen auf die Festplatte zur Verfügung zu stellen. Infolge unterschiedlicher Dateisysteme auf den verwendeten Partitionen ergaben sich unterschiedliche Bandbreiten. Die Partition mit den Logdateien war mit dem Dateisystem *ext3* formatiert und es konnte mit 78 MByte/s geschrieben und 119 MByte/s gelesen werden. Die Partition mit den Sicherungspunktdateien war mit dem Dateisystem *ReiserFS* formatiert und erlaubte einen schreibenden Zugriff mit 75 MByte/s und einen lesenden mit 96 MByte/s.

5.3 Zugriffspfade

In diesem Abschnitt werden die Zugriffsgeschwindigkeiten unterschiedlicher Zugriffspfade bzw. Indizes ermittelt. Weil Zugriffspfade einen elementaren Bestandteil des entwickelten DBS darstellen und bei jeder Transaktion verwendet werden, ist es wichtig zu wissen, wie schnell die Zugriffe bei unterschiedlichen Indizes erfolgen.

Es wurden die Zugriffsgeschwindigkeiten der selbst implementierten Datenstrukturen AVL-Baum und T-Baum sowie des *Rot-Schwarz-Baumes (RB-Baum)* aus der Java-API ermittelt. Die beiden erstgenannten Strukturen wurden jeweils in einer generischen und einer nicht generischen Variante bereitgestellt. Die generische Variante erlaubt die Verwendung von Schlüsseln, die Instanzen einer beliebigen Java-Klasse sind, welche die Schnittstelle *Comparable* implementiert. In einem Knoten der Datenstruktur wird als Schlüssel ein Pointer zum entsprechendem Objekt gespeichert. Dahingegen können in die nicht generische Variante nur Schlüssel-Werte-Paare abgelegt werden, die einen Schlüssel eines bestimmten primitiven Datentyps besitzen (z.B. *int* oder *long*). Bei der Aufnahme eines neuen Schlüssels wird eine Kopie von ihm angefertigt und direkt in der Datenstruktur gespeichert. Die Verwendung primitiver Datentypen verringert die Anzahl an notwendigen

Indirektionen und erlaubt insbesondere beim T-Baum eine bessere örtliche Lokalität der Daten, da Schlüssel eines Knotens aufeinanderfolgend im Speicher alloziert werden.

Bei jeder Messung wurden s ganze Zahlen (Integer) in unsortierter Reihenfolge in einen zu Beginn leeren Index eingefügt. Die für das Aufbauen des Index mit Indexgröße s benötigte Zeit t wurde gemessen und aus ihr die Geschwindigkeit der Einfügeoperationen nach der Formel $v = \frac{s}{t}$ berechnet. Im Anschluss wurde jede zuvor eingefügte Zahl in einer erneut randomisierten Reihenfolge einmal im Index gesucht und so die Geschwindigkeit der Leseoperationen ermittelt. Es wurden jeweils die generischen Strukturen AVL-,T- und RB-Baum sowie die nicht generischen Strukturen AVL-Baum_PT und T-Baum_PT vermessen.

5.3.1 Geschwindigkeit der Zugriffsoperationen

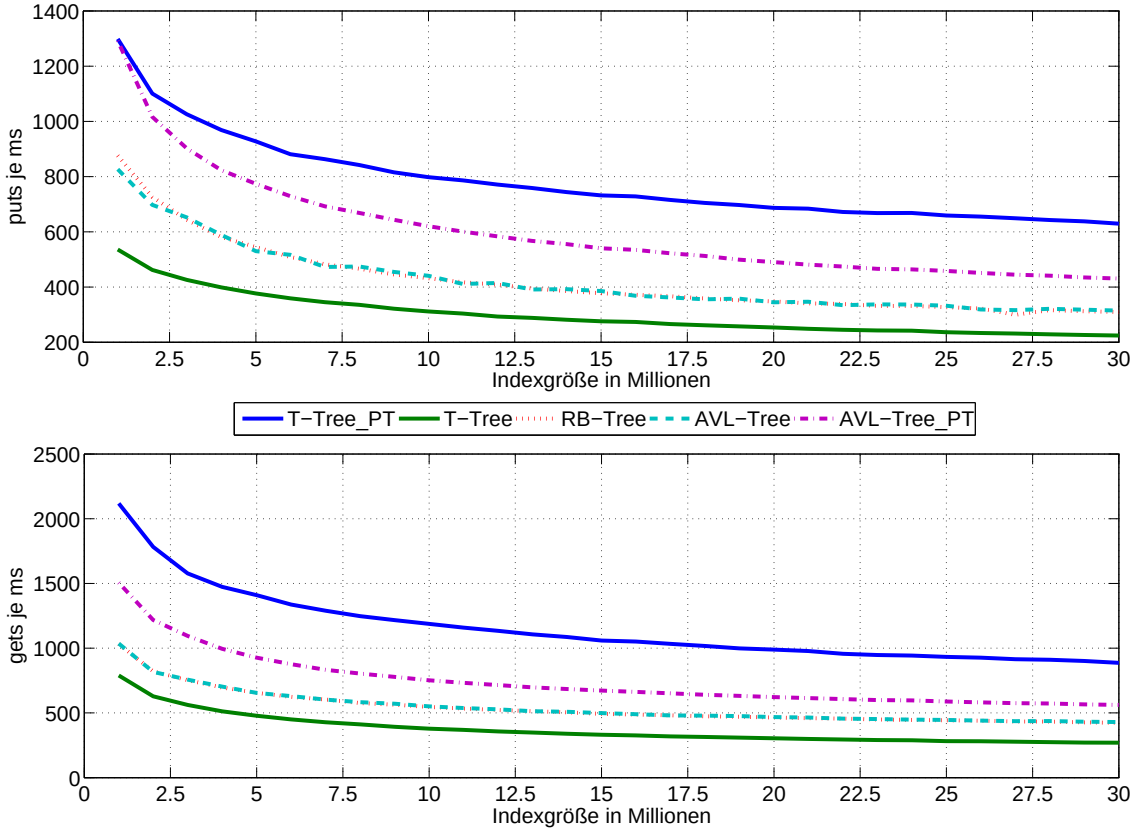


Abbildung 5.1: Indexperformanz bei variierender Indexgröße

Im ersten Test wurden die Lese- und Einfügegeschwindigkeiten der oben genannten Datenstrukturen für variierende Indexgrößen s ermittelt. Die Knotengröße der T-Bäume war dabei $ns = 50$. Die Indexgröße s nahm Werte von einer bis zu 30 Millionen an. Abbildung 5.1 zeigt das Ergebnis dieser Messung. Im oberen Teil sind die Geschwindigkeiten

der Einfüge- im unteren die der Leseoperationen dargestellt. Bei allen Strukturen nehmen die Geschwindigkeiten der Zugriffsoperationen logarithmisch mit wachsender Indexgröße ab. Wie erwartet (siehe oben) zeigen die nicht-generischen Varianten bessere Werte als die generischen. Die beste Zugriffsgeschwindigkeit zeigte der nicht-generische T-Baum. Dies entspricht ebenfalls der oben begründeten Erwartung.

Dass der generische T-Baum die schlechtesten Werte zeigt, überrascht, da er flacher ist, als der AVL-Baum und der Knoten mit dem gesuchten Element schneller gefunden wird. Ein Grund für dieses Ergebnis könnte sein, dass der Knoten eines T-Baumes deutlich größer ist, als der eines AVL-Baumes, und dies zu einer schlechteren örtlichen Lokalität führt. Die Überlegung dabei ist folgende: Angenommen bei der Traversierung des Baumes wurde für das Lesen eines Knotens eine Cache-Line gelesen. Die Auswertung des Knotens ergibt, dass die Suche im linken Teilbaum fortzuführen ist. Die Wahrscheinlichkeit, dass der nächste Knoten in derselben Cache-Line liegt, ist umso größer, je mehr Knoten in eine Cache-Line passen bzw. je kleiner ein Knoten ist. Um diese Vermutung zu überprüfen, müsste man die Cache-Misses bei der Traversierung beider Bäume messen. Ein Tool, was dies leistet ist das Programm *OProfile*.

Der implementierte generische AVL-Baum zeigte die gleichen Zugriffsgeschwindigkeiten wie der Rot-Schwarz-Baum aus der Java-API, was auf die nahezu identischen Knotenstrukturen beider Bäume zurückzuführen ist. Eine Messung im Profiler von Netbeans ergab, dass beide Strukturen denselben Platz im Hauptspeicher belegen.

Im zweiten Test wurde überprüft, wie sich die Geschwindigkeiten von Einfüge- und Leseoperationen der T-Bäume in Abhängigkeit von der Knotengröße ns verändern. Die Indexgröße war dabei konstant $s = 10^7$. Die Knotengröße ns variierte von 1 bis 200. In Abbildung 5.2 ist das Ergebnis der Messung dargestellt. Beide T-Baum-Varianten zeigen ähnliche Kurvenverläufe beim Einfügen und Lesen, wobei die nicht-generische Variante einen deutlich schnelleren Zugriff bietet. Beim Einfügen fangen beide T-Bäume auf einem sehr niedrigen Niveau an, steigern sich dann bis zu einer gewissen Knotengröße und fallen daraufhin linear ab. Die Leistungssteigerung zu Beginn ist vermutlich darauf zurückzuführen, dass die Arrays, die die Schlüssel bzw. die Referenzen auf die Schlüssel enthalten, zu klein sind, um eine Cache-Line auszufüllen, und so die örtliche Lokalität geringer ausfällt. Auch diese Vermutung ist durch ein Zählen der Cache-Misses mit *OProfile* zu überprüfen. Gute Werte beim Einfügen stellen sich ab einer Knotengröße von 10 ein. Der lineare Abfall der Einfügegeschwindigkeit ab einer gewissen Knotengröße ist darauf zurückzuführen, dass beim Einfügen eines Schlüssels oder einer Referenz in einen Knoten andere Elemente des Arrays verschoben werden. Das ist notwendig, weil die Elemente im Array sortiert gehalten werden. Der Aufwand eines Shifts wächst linear mit der Knotengröße.

Die Kurven der Lesegeschwindigkeiten steigen ebenfalls zu Beginn an und bleiben dann konstant. Ab einer Knotengröße um die 40 ist es für das Lesen irrelevant, ob der Baum flacher und die Knoten größer oder der Baum höher und die Knoten kleiner sind. Sowohl die Traversierung des Baumes als auch die binäre Suche nähern sich logarithmisch dem gesuchten Schlüssel. Zudem führt auch die binäre Suche im Array zu Cache-Misses, wenn

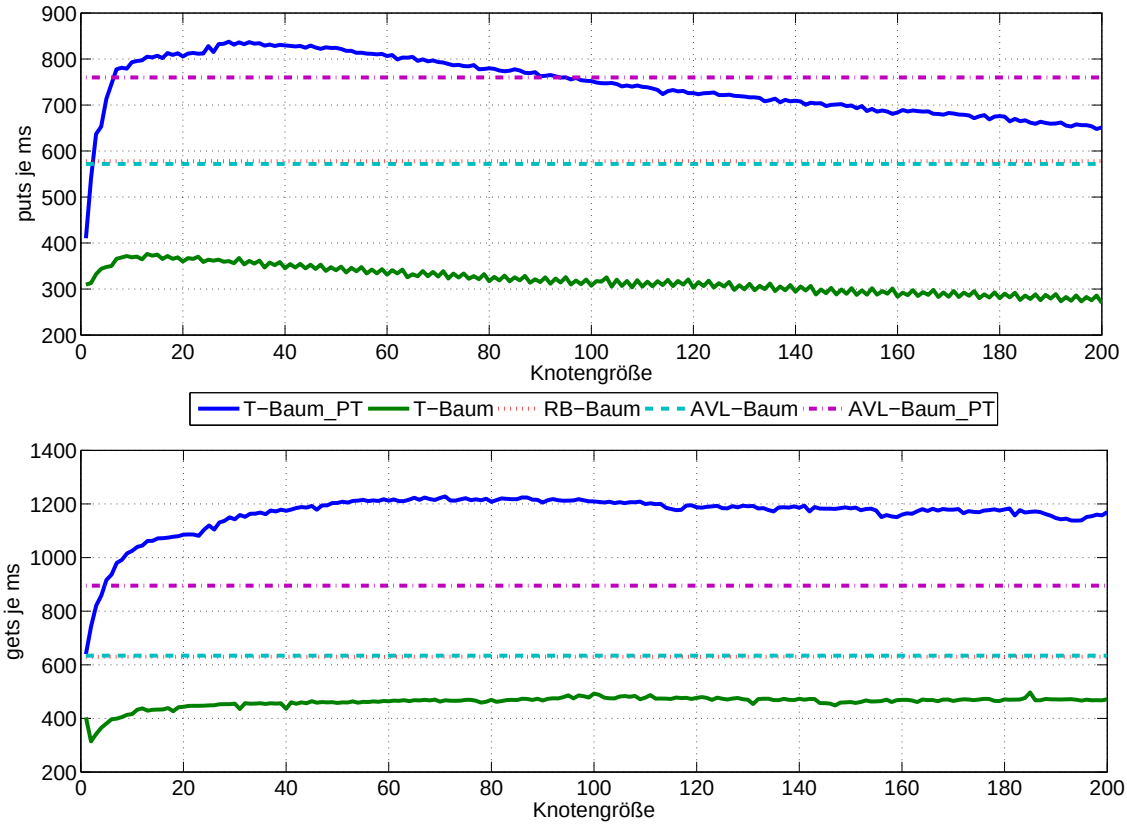


Abbildung 5.2: Indexperformanz bei variierender Knotengröße

das Array sich über viele Cache-Lines erstreckt, weshalb keine bessere Performanz bei größeren Knoten zu erwarten ist.

Die Ergebnisse der Messungen mit variierender Knotengröße zeigen, dass die Knotengröße entscheidend für die Geschwindigkeiten der Zugriffsoperationen ist. Sinnvolle Werte liegen im Bereich 10 bis 80. Kleinere Größen aus dem Bereich steigern die Einfügegeschwindigkeit, größere begünstigen Leseoperationen und erhöhen die Speichereffizienz der Datenstruktur.

5.3.2 Indexrekonstruktion

In Abschnitt 4.2.5 wird ein Verfahren für die Konstruktion eines Indexes aus einer sortierten Liste von Schlüsseln beschrieben. Um die Überlegenheit dieses Verfahrens gegenüber der naiven Methode, bei der der Baum sukzessive aufgebaut wird, zu zeigen, wurde der folgende Test durchgeführt.

In einer Datenbank wurde ein Index mit 15 Millionen Schlüssel-Werte-Paaren aufgebaut. Die Schlüssel waren vom Typ *long* und die Werte *null*. Dann wurde ein Sicherungspunkt angefertigt und das DBS beendet. Das Laden des Sicherungspunktes beim Neustart des DBS wurde nun jeweils einmal mit beiden Verfahren durchgeführt. Die Ladezeit betrug bei Verwendung des vorgestellten Verfahrens $5,6\ s$ und bei Verwendung des naiven Verfahrens $10,4\ s$. Somit ist das vorgestellte Verfahren deutlich besser als das naive.

Die Bedeutung des verwendeten Verfahrens verringert sich bei Verwendung realer Werte ($>$*null*), da dann ein größerer Anteil der Zeit für das Lesen des Sicherungspunktes vom Sekundärspeicher und der Deserialisierung der Werte verwendet wird. Die Bedeutung steigt jedoch mit der Komplexität der Schlüsselvergleiche, da das naive Verfahren $n \log n$ und das vorgestellte keine Vergleichsoperationen vornimmt (Es nutzt die gegebene Sortierung).

5.4 Messungen des Datenbanksystems SpeeDB

- Wie performant ist das System im Datenbankbetrieb?
- Mit welchen Geschwindigkeiten werden Logging- und Recoveryaktionen durchgeführt?
- Wie wirken sich unterschiedliche Einstellungen der Warteschlangen auf die Eigenschaften des DBS aus?

In den folgenden Unterabschnitten werden Untersuchungen des DBS SpeeDB und deren Ergebnisse vorgestellt.

5.4.1 Transaktionsdurchsatz und Antwortzeit

Hier soll untersucht werden, wie schnell das System die Datenbankanfragen bzw. die benutzerdefinierten Funktionen ausführt. Die entscheidenden Größen bei der Bewertung der

Performanz eines DBS sind der *Transaktionsdurchsatz* und die *Antwortzeit*. Der Transaktionsdurchsatz gibt an, wie viele Transaktionen ein System pro Zeiteinheit abarbeiten kann. Systeme mit einem geringeren Transaktionsdurchsatz erfordern eher eine Lastverteilung auf mehrere Systeme und erhöhen dadurch die Hardwarekosten. Die Antwortzeit gibt an, wieviel Zeit für die Bearbeitung einer Anfrage benötigt wird. Viele Anwendungen verlangen eine Garantie, dass die Antwortzeiten unterhalb einer gewissen Grenze liegen, weswegen klare Angaben über die Antwortzeit eines DBS unabdingbar sind.

Um den Transaktionsdurchsatz und die Antwortzeit möglichst realistisch zu bestimmen, wurde eine kleine Menge von BDF für den Einsatz als Backend des Metadaten- und Replikatskatalog (MRC) von XtreemFS implementiert. In der Datenbank gibt es einen Index, der Instanzen der Java-Klasse *FileEntity* aufnehmen kann. Eine solche Instanz enthält die Metadaten einer Datei. Die Klasse *FileEntity* enthält zwei Felder des Typs *String* und acht des Typs *Long*. Jede Instanz besitzt einen eindeutigen Schlüssel vom Typ *Long*. Unter den implementierten BDF finden sich die Funktionen *putFile(FileEntity file)* und *getFile(Long fileID)* für die Aufnahme neuer bzw. die Abfrage existierender Dateimetadaten.

Mit diesen BDF wurde nun der folgende Test durchgeführt. Zunächst wurde die Datenbank initial mit 5 Millionen Instanzen der Klasse *FileEntity* gefüllt. Danach wurden 5 Millionen Anfragen an das DBS gesendet. Die Zeit zwischen dem Absenden der ersten Anfrage und dem Erhalt der letzten Antwort wurde in der Clientanwendung gestoppt und für die Berechnung des Transaktionsdurchsatzes ($= 5 * 10^6 / gestoppteZeit$) verwendet. Davon unabhängig wurde noch für jede Anfrage die Antwortzeit gemessen. Die Antwortzeit bezeichnet die Zeit zwischen dem Erhalt der Anfrage und dem Versand der Antwort. Dieser Versuch wurde für verschiedene Mixturen aus lesenden (*getFile()*) und schreibenden Anfragen (*putFile()*) ausgeführt. Der Zugriff auf die Daten erfolgte immer in zufälliger, unsortierter Reihenfolge, um ein möglichst generisches Worst-Case-Szenario zu schaffen.

In Abbildung 5.3 werden für die unterschiedlichen Anfragemixturen der gemessene Transaktionsdurchsatz und die mittleren Antwortzeiten von lesenden und schreibenden Anfragen dargestellt. Werden ausschließlich lesende Anfragen gesendet, bearbeitet das System 328 Abfragen in einer Millisekunde. Mit einem wachsenden Anteil an schreibenden Anfragen sinkt der Durchsatz linear auf 231 Anfragen je Millisekunde. Das erklärt sich durch den höheren Aufwand schreibender Anfragen, da bei ihnen Zugriffspfade restrukturiert werden und durch das notwendige Logging ein Zugriff auf den Sekundärspeicher erforderlich wird. Mit dem Bearbeitungsaufwand wächst auch die mittlere Antwortzeit. Schreibende Anfragen dauern immer länger als Lesende, weil sie einen längeren Weg im DBS zurücklegen. Ist die Bearbeitung einer lesenden Anfrage bei der Ausführungs-Stage beendet, muss die schreibende Anfrage in die Warteschlange der Logging-Stage eingereiht werden und im Anschluss auf das *Gruppencommit* warten.

Die Standardabweichung der mittleren Antwortzeiten war nahezu konstant. Sie betrug 180 ms bei den schreibenden und 90 ms bei den lesenden Anfragen. Diese hohen Abweichungen werden durch *Garbage-Collections* der virtuellen Maschine verursacht (siehe Abschnitt 5.5). Infolge der Garbage-Collections gab es im Extremfall sowohl lesende als auch schreibende Anfragen mit einer Antwortzeit von einer Sekunde. Von den lesenden Anfragen

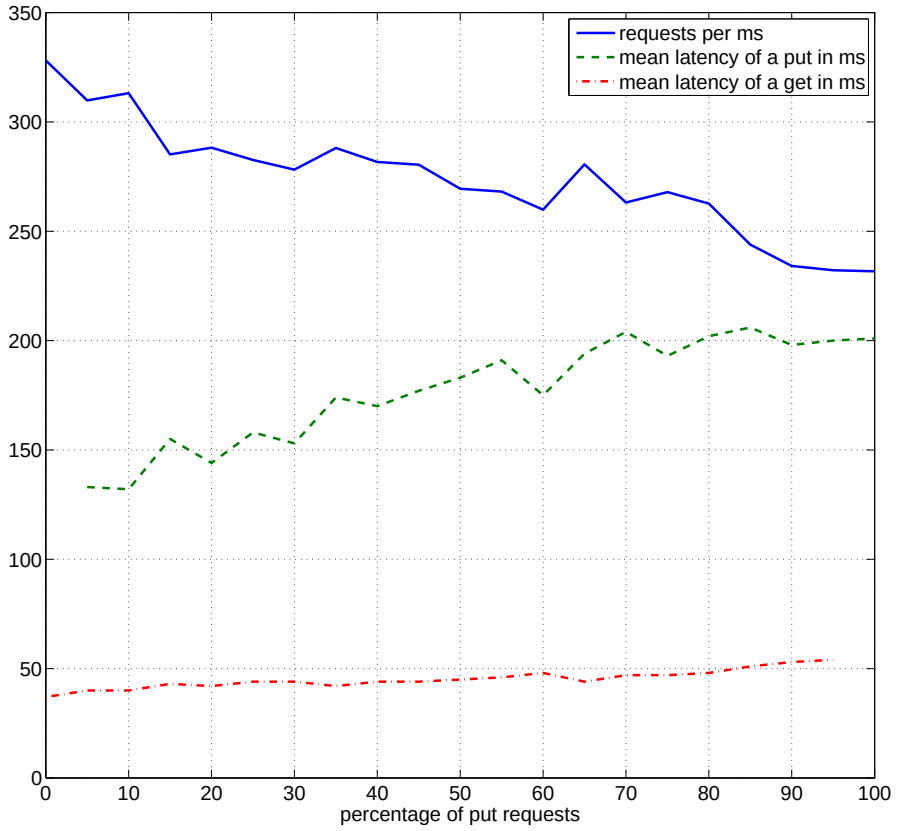


Abbildung 5.3: Transaktionsdurchsatz und mittlere Antwortzeiten

benötigten weniger als 2 % mehr als 100 ms und von den schreibenden benötigten ca. 6% mehr als 250 ms für die Ausführung.

Versuch bei geringer Last Diese Ergebnisse wurden bei maximaler Last gemessen. Bei geringerer Last stellen sich erwartungsgemäß niedrigere Antwortzeiten ein, was im folgenden Versuch beobachtet wurde. Ähnlich zum obigen Versuch wurde zunächst eine Datenbank mit 5 Millionen Einträgen angelegt. Dann wurden 2 Minuten lang Anfragen (25% davon schreibend) an das System gesendet. Um die Last zu reduzieren wurde die Clientanwendung nach 100 abgesendeten Anfragen für 1 ms blockiert. In den zwei Minuten wurden so 1,4 Millionen Anfragen gesendet und bearbeitet, was einem Durchsatz von 11 Anfragen pro ms entspricht[1]. Bei dieser Last sank die mittlere Antwortzeit schreibender Anfragen von 160 ms auf 70 ms. Die Abweichung der Antwortzeit sank von 180 ms auf 28 ms, was sich dadurch erklären lässt, dass beim Auftritt einer Garbage-Collection weniger Anfragen im System sind und somit weniger Anfragen mit einer hohen Antwortzeit auftreten. Nur 0,18% der schreibenden Anfragen dauerten länger als 250 ms. Bei den lesenden Anfragen sank die mittlere Antwortzeit von 44 ms auf unter 1 ms. Die Abweichung von der mittleren Antwortzeit betrug lediglich 3 ms und nur 27 der insgesamt 1025809 lesenden Anfragen dauerten länger als 100 ms.

5.4.2 Logging und Recovery

Wie lange dauert das Erstellen und das Laden eines Sicherungspunktes? Wie schnell werden Änderungen an der Datenbank geloggt und wieviel Zeit beansprucht das *REDO* geloggter Transaktionen beim Start des DBS? Für die Beantwortung dieser Fragen wurde der folgende Versuch durchgeführt.

Zunächst wurde eine BDF implementiert, die Instanzen einer Klasse *DummyEntity* in einen Index der Datenbank ablegen kann. Jede Instanz der Klasse *DummyEntity* besitzt einen eindeutigen Schlüssel vom Typ *Long*. Neben dem Schlüssel besitzt jede Instanz als Wert noch einen beliebig langen *ByteBuffer*. Da der Speicher des Referenzsystems relativ klein ist, wird der Wert der Instanz beim Schreiben und Lesen des Schlüssel-Werte-Paares auf und vom Sekundärspeicher nur simuliert. Wird z. B. eine Instanz auf die Platte geschrieben, wird zunächst der Schlüssel in 8 Bytes und die Länge des Buffers in den nächsten 8 Bytes gesichert. Dann werden entsprechend der Länge des zu simulierenden Buffers beliebige Bytes geschrieben. So kann die I/O-Last erhöht werden, ohne den verfügbaren Hauptspeicher aufzubrauchen. Dabei ist jedoch zu bedenken, dass der Aufwand für die Serialisierung und Deserialisierung von Objekten reduziert wird. Bei der Verwendung richtiger Objekte werden Logging- und Recoveryaktionen etwas mehr Zeit in Anspruch nehmen.

Mit der implementierten BDF wurden Datenbanken mit unterschiedlich vielen und großen Einträgen erzeugt. Dabei wurde dieser feste Ablauf befolgt:

[1] Die Anfragen wurden blockweise abgesendet, da die kleinste Zeiteinheit beim Threadscheduling in Java 1 ms ist und mehrere Anfragen sich aus dem Grunde nicht gleichmäßig auf 1 ms verteilen lassen.

1. Lege eine neue Datenbank an.
2. Sende eine bestimmte Anzahl an Anfragen an das System. Jede der Anfragen legt eine Instanz der Klasse *DummyEntity* mit einer bestimmten Objektgröße in der Datenbank ab (bei unsortierter Reihenfolge) und messe die Zeit.
3. Starte das System neu und messe die Zeit, die für das *REDO* der geloggten Transaktionen benötigt wird.
4. Lege einen Sicherungspunkt an und stoppe die dafür verwendete Zeit.
5. Starte das System nochmals neu und messe die Zeit, die für das Laden des Sicherungspunktes verwendet wird.

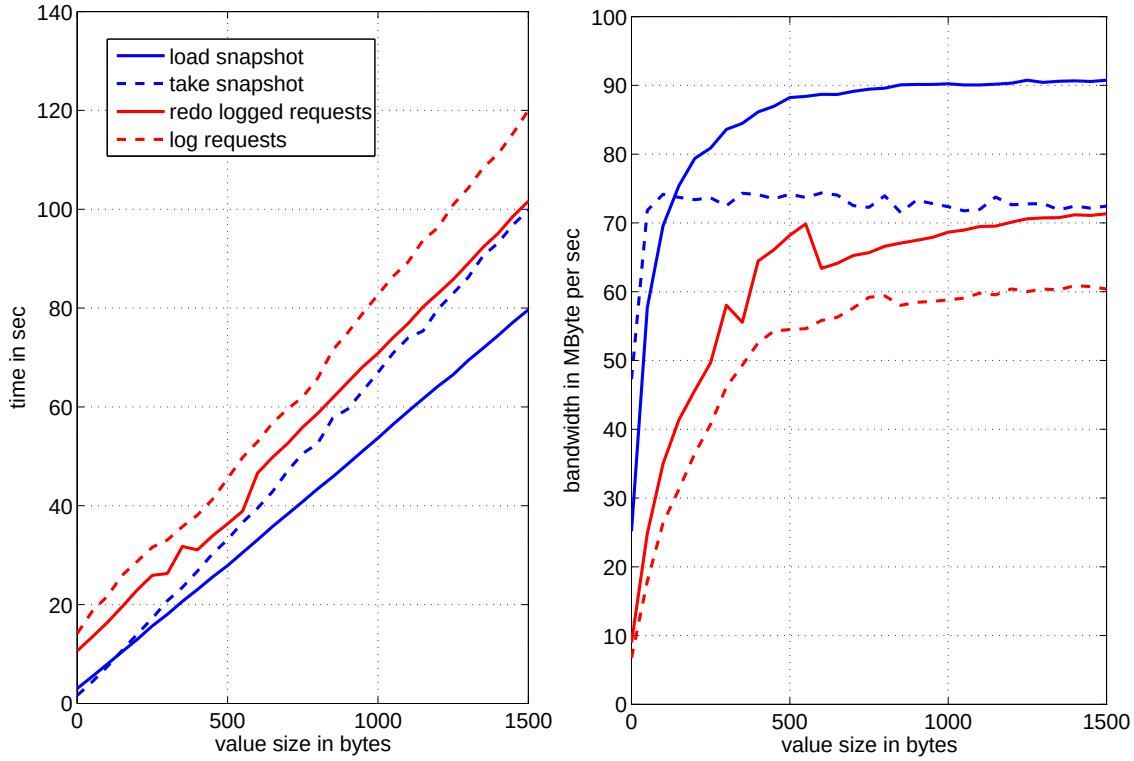


Abbildung 5.4: Logging und Recovery bei variierender Objektgröße

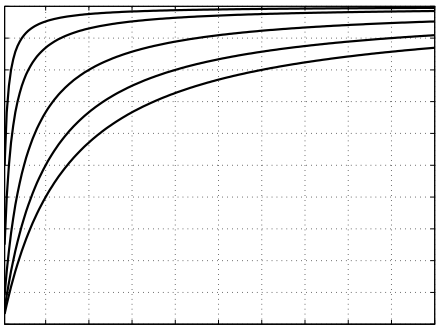
Dieser Ablauf beinhaltet also alle für die Bewertung der Logging- und Recoveryeigenschaften interessanten Funktionen: Erst werden alle Daten geloggt, dann vom Log gelesen, daraufhin in einem Sicherungspunkt gespeichert und zuletzt wird dieser geladen. Vor dem Neustart des DBS wurden jeweils die Caches des Betriebssystemkerns mit dem folgenden Befehl geleert: `echo 1 | sudo tee -a /proc/sys/vm/drop_caches > /dev/null` .

So wurde verhindert, dass beispielsweise die Logdatei beim Start des DBS nicht aus dem Hauptspeicher sondern von der Festplatte gelesen wurde.

In Abbildung 5.4 sieht man das Ergebnis einer Messung, bei welcher der obige Ablauf mit jeweils 5 Millionen Objekte variierender Größe durchgeführt wurde. Im linken Teil der Abbildung ist die für die jeweilige Aktion benötigte Gesamtzeit in Sekunden dargestellt. Im rechten Teil sieht man die aus der Gesamtzeit und der Größe der verwendeten Datei resultierende Bandbreite in MByte/s. Der Blick auf die Gesamtzeit zeigt die zu erwartende lineare Abhängigkeit von der Objektgröße. Man erkennt auch, dass die lesenden Zugriffe im Allgemeinen schneller erfolgen, als die schreibenden und Aktionen mit dem Log langsamer sind als Aktionen mit einem Sicherungspunkt.

Die Kurven der Bandbreiten ähneln sich im Verlauf. Sie sind monoton wachsend, das Wachstum nimmt aber mit wachsender Objektgröße ab. Dieser Kurvenverlauf soll im Folgenden erklärt werden. Die Bandbreite B ist der Quotient aus der transportierten Datenmenge M und der verstrichenden Zeit t. Nehmen wir an, die Datenmenge ist das Produkt aus einer Konstanten c_1 und der Objektgröße s. Die im Versuch gemessene Zeit ist nur zu einem Teil von s abhängig. Ein weiterer Teil c_3 wird als konstant angenommen, so dass $t = c_2 \cdot s + c_3$ ist. Damit ergibt sich für die Bandbreite die Funktion:

$$B = \frac{c_1 \cdot s}{c_2 \cdot s + c_3}$$



In der rechten Abbildung sieht man Kurven dieser Funktion für unterschiedliche Werte der Konstante c_3. Je größer der Zeitaufwand c_3 ist, desto langsamer steigt die Kurve an. Die Ähnlichkeit zu den Kurven der Bandbreiten in Abbildung 5.4 ist gut erkennbar. Es bleibt also zu klären, wie dieser von der Objektgröße s unabhängige konstante Zeitaufwand entsteht.

Die Kurve der Bandbreite beim Erstellen eines Sicherungspunktes lässt darauf schließen, dass der von s unabhängige Aufwand c_3 sehr gering ist. Sie steigt sehr steil an und erreicht bereits bei Objektgröße $s = 50$ einen Wert von 72 MByte/s. Für größere Objekte bleibt die Bandbreite dann konstant hoch und befindet sich somit nah an der maximalen Bandbreite von 75 MByte/s.

Beim Laden eines Sicherungspunktes, beim Loggen von Transaktionen und beim REDO von geloggten Transaktionen ist der von der Objektgröße s unabhängige konstante Zeitaufwand c_3 größer. Die Ursache dieses Aufwandes ist, dass während der Ausführung dieser drei genannten Operationen zusätzliche, kurzlebige Objekte entstehen infolge derer der Garbage-Collector sogenannte *Minor-Collections* durchführt (siehe 5.5). Die Anzahl der zusätzlichen Objekte und damit auch der Aufwand durch den Garbage-Collector ist abhängig von der Objektanzahl. Beim Loggen der Transaktionen werden durch die parallele Anfragebearbeitung besonders viele kurzlebige Objekte angelegt. Bei der wiederholten Ausführung geloggter Transaktionen stellt auch die Pflege des Zugriffspfades einen

von der Objektanzahl abhängigen und damit konstanten Aufwand dar, da im Gegensatz zur normalen Anfragebearbeitung das Lesen, Deserialisieren und Ausführen der geloggten Anfrage momentan von nur einem Thread vorgenommen wird. Beim Fortsetzen der Implementierung ist zu prüfen, wie das Anlegen kurzlebiger Objekte vermieden und das REDO parallelisiert[2] werden kann.

Der Zugriff auf den Sekundärspeicher erfolgt während des Ladens eines Sicherungspunktes ab einer Objektgröße von 500 Byte mit einer Geschwindigkeit von 88 MByte/s. Das sind 92% der maximalen Bandbreite von 96 MByte/s. Die Bandbreite bei den Zugriffen auf die Logdatei ist für Objekte, die kleiner als 500 Byte sind, zu niedrig. Für größere Objekte erfolgt zumindest das Schreiben des Logs mit 60 MByte/s und nähert sich langsam der maximalen Geschwindigkeit von 78 MByte/s. Besonders niedrig ist die erreichte Bandbreite beim REDO - also beim Lesen des Logs. Zwar werden Objekte mit einer Größe um 500 Bytes mit 60-70 MByte/s gelesen, der hohe Aufwand durch die Deserialisierung und der erneuten Ausführung der BDF (inkl. sequentiellem Indexaufbau) führen jedoch zu einer Bandbreitenkurve, die sich nur sehr langsam der maximalen Bandbreite von 119 MByte/s nähert.

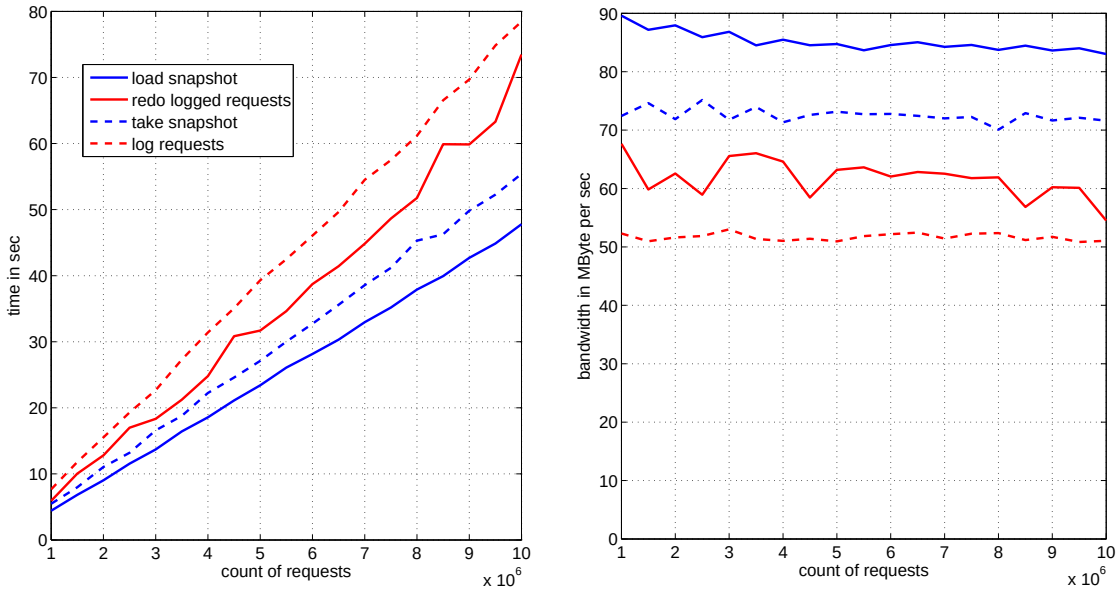


Abbildung 5.5: Logging und Recovery bei variierender Objektanzahl

In einer weiteren Messung wurden der obige Ablauf für eine variierende Anzahl an Objekten mit einer festen Größe von 400 Byte durchgeführt. Das Ergebnis ist in der Abbildung 5.5 dargestellt. Im linken Teil der Abbildung erkennt man die lineare Abhängigkeit der jeweils benötigten Gesamtzeit von der Objektanzahl. Die rechts dargestellten Bandbreiten sind relativ konstant. Allein die Bandbreiten der lesenden Zugriffe scheinen mit wachsender Objektanzahl leicht abzunehmen. Besonders deutlich ist dies beim REDO der geloggten

[2]Ein Thread könnte die BDF serialisieren und ein anderer sie ausführen.

Transaktionen der Fall. Hier kann die Abnahme aber damit erklärt werden, dass mit der Objektanzahl auch der Aufwand der Indexpflege (siehe oben) wächst.

Einfluss von Schnappschüssen auf den Durchsatz Auch wenn das Erstellen von Sicherungspunkten nichtblockierend erfolgt, ließ sich ein Rückgang des Transaktionsdurchsatzes während der Anfertigung eines Schnappschusses beobachten. Der Verursacher ist wiederum der Garbage-Collector. Für das Schreiben der Objekte in die Checkpoint-Datei, werden sie serialisiert. Dabei entstehen zusätzliche, kurzlebige Objekte, welche zu einer größeren Aktivität des Garbage-Collectors führt. Infolge der zusätzlichen Collections fällt der Transaktionsdurchsatz geringer aus (siehe dazu auch Abschnitt 5.5). Im ersten Versuch aus Abschnitt 5.4.1 betrug der Transaktionsdurchsatz 280 Anfragen pro Sekunde (bei einem Anteil von 25% schreibender Anfragen). Legt man parallel zu den Anfragen einen Sicherungspunkt an, sinkt der Durchsatz auf 230 Anfragen pro ms. Das entspricht einem Leistungsabfall von 18%.

Bei geringerer Last wird der Transaktionsdurchsatz nicht so stark beeinflusst, da weniger Anfragen von den zusätzlichen Collections betroffen sind (siehe Abschnitt 5.4.1).

5.4.3 Warteschlangen

In Abschnitt 4.4 wurde eine blockierende Warteschlange vorgestellt, die wartende Threads erst aufweckt, wenn der Füllstand einen gewissen Schwellwert erreicht. Um den Einfluss der Warteschlangenparameter auf das Gesamtsystem zu erkennen, wurde ein Test mit variierenden Kapazitäten und Schwellwerten durchgeführt. Die Parameter betrafen die Warteschlange der Ausführungs-Stage und die der Logging-Stage. Bei dem Test wurden 10 Millionen schreibende Anfragen an eine initial leere Datenbank gesendet, wobei jede Anfrage wie im vorangegangenem Versuch eine Instanz der Klasse *DummyEntity* in der Datenbank speichert. Um den I/O-Operationen ein Gewicht zu verleihen und so die Logging-Stage zum Flaschenhals zu machen, wurde der als Wert zu schreibende *ByteBuffer* auf die Länge 200 gesetzt. Außerdem wurde der Garbage-Collector durch die Wahl einer sehr großen *Young-Generation* deaktiviert (siehe 5.5). Es wurde bei jedem Lauf die für die Bearbeitung der 10 Millionen Anfragen benötigte Gesamtzeit und die mittlere Antwortzeit einer Anfrage gemessen. Gleichzeitig wurde mit dem Linux-Befehl *time* die Anzahl der freiwilligen Kontextwechsel und die mittlere beanspruchte CPU-Leistung ermittelt.

In den verschiedenen Testläufen nahm die Kapazität c Werte zwischen 1000 und 20000 (in Tausenderschritten) an. Für jede Kapazität c wurden Schwellwerte zwischen 999 und $c - 1$ (ebenfalls in Tausenderschritten) gewählt. In Abschnitt 4.4 wurde gesagt, dass es ungünstig ist, wenn ein wartender Thread aufgeweckt wird, sobald ein Platz frei ist. Dieser Fall liegt bei einer Kapazität c in Kombination mit einem Schwellwert $s = c - 1$ vor. In Abbildung 5.6 sieht man im oberen Teil den aus der Gesamtzeit errechneten Transaktionsdurchsatz und im unteren die mittlere CPU-Auslastung. Ein Blick auf den Transaktionsdurchsatz liefert die Erkenntnis, dass der größte Durchsatz bei hohen Kapazitäten (16000

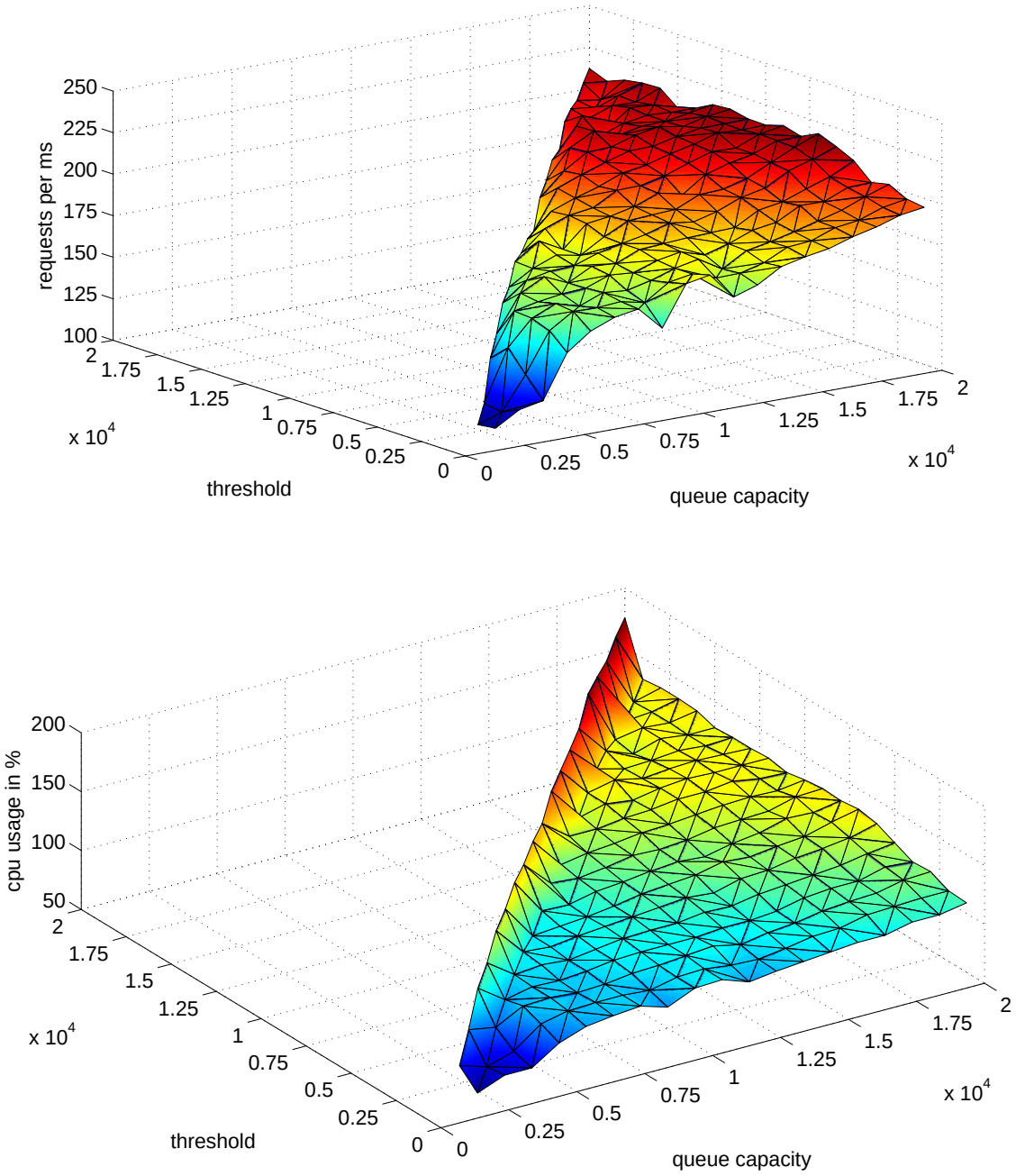


Abbildung 5.6: Durchsatz und CPU-Nutzung bei var. Warteschlangeneinstellungen

bis 20000) mit einem mittleren Schwellwert (8000 bis 15000) erreicht wird. Mit der Länge der Warteschlangen nimmt offensichtlich auch der Transaktionsdurchsatz ab.

Die dargestellte CPU-Auslastung bestätigt die Vermutung, dass es ungünstig ist wartende Threads aufzuwecken, sobald nur ein Platz in der Schlange frei ist. Es ist zu erkennen, dass die CPU-Auslastung auf der Kante mit dem ungünstigen Schwellwert ($s = c - 1$) deutlich ansteigt. Betrachten wir beispielsweise die Läufe mit einer Kapazität von 20000. Die CPU-Auslastung steigt drastisch von 132% bei einem Schwellwert von 18999 auf 175% beim ungünstigen Schwellwert von 19999 an. Der Ausgabe des Befehls *time* war zu entnehmen, dass beim Schwellwert von 18999 die Anzahl der freiwilligen Kontextwechsel 15000 betrug. Beim Schwellwert 19999 traten 2400000 Kontextwechsel auf. Als direkte Folge der höheren Anzahl an Kontextwechsel stieg die Zeit, die der Prozess für Operationen des Betriebssystemkerns verwendete von 21 s auf 35 s. Ein größerer Aufwand im Thread-Scheduling spiegelt sich demnach sehr deutlich in der CPU-Last wieder.

Betrachtet man den Transaktionsdurchsatz bei den ungünstigen Schwellwerten, fällt auf, dass dieser trotz der höheren CPU-Auslastung leicht abfällt. Die Abnahme des Durchsatzes fällt zwar relativ gering aus, es ist jedoch davon auszugehen, dass die Performanz stärker einbricht, sobald weniger CPU-Ressourcen - sei es durch zusätzlicher Last anderer Anwendungen oder anderer Hardwarevoraussetzungen - zur Verfügung stehen. Ein deutlicher Anstieg der Antwortzeiten ist in dem Fall ebenfalls zu erwarten.

Des Weiteren ist aus der Kurve der CPU-Auslastung abzulesen, dass die Auslastung sowohl mit der Kapazität als auch mit dem Schwellwert der Warteschlange linear steigt. Bis jetzt konnte noch keine Erklärung für diese Abhängigkeiten nachgewiesen werden.

Die Abbildung 5.7 zeigt die gemessenen mittleren Antwortzeiten einer Anfrage bei den unterschiedlichen Einstellungen der Warteschlangen. Generell kann man sagen, dass die Anfragen eine kürzere Zeit beanspruchen, wenn kürzere Warteschlangen verwendet werden. Das ist verständlich, denn je kürzer die Warteschlangen sind, desto weniger Zeit wartet eine Anfrage auf ihre Bearbeitung.

Mit der Wahl der Parameter ist einem offensichtlich ein Mittel für die Beeinflussung wichtiger Eigenschaften des Datenbanksystems in die Hand gelegt worden. Benötigt eine Anwendung sehr kurze Antwortzeiten, ist eine kleine Kapazität der Warteschlangen zu wählen. Der Preis für die schnellere Bearbeitung einer Anfrage ist aber ein geringerer Durchsatz. Je größer die mittlere Antwortzeit sein darf, desto größer sollte die maximale Kapazität der Warteschlangen gewählt werden um einen höheren Transaktionsdurchsatz zu erzielen. Ganz klar zu vermeiden ist die Wahl eines zu hohen Schwellwerts, da diese keinerlei Vorteile, unter gewissen Umständen jedoch erhebliche Nachteile einbringt.

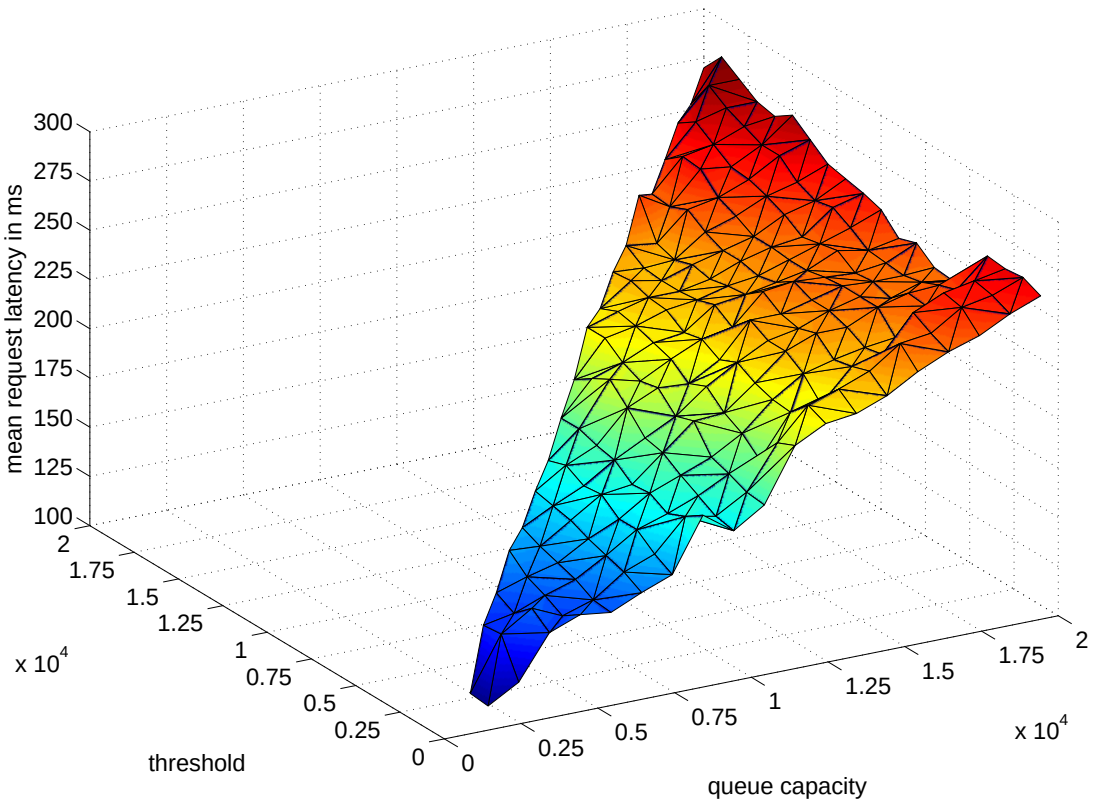


Abbildung 5.7: Durchschnittliche Antwortzeit bei var. Warteschlangeneinstellungen

5.5 Garbage-Collection

In den beschriebenen Messungen wurde des Öfteren der Garbage-Collector von Java erwähnt. Meist war dies der Fall, wenn auffällige Messergebnisse zu erklären waren. In diesem Abschnitt wird die Funktionsweise des Garbage-Collectors genauer beschrieben und ihre Auswirkungen auf den Betrieb von SpeeDB erläutert.

In Java identifiziert der Garbage-Collector nicht mehr benötigte Objekte und gibt deren Speicherplatz frei. Da die meisten Objekte nur eine kurze Lebensdauer haben, verwaltet er gewöhnliche Objekte in zwei Speicherbereichen: einen für die jungen Objekte (*Young-Generation*) und einen für die alten (*Tenured-Generation*) [44]. Für jedes neue Objekt wird zunächst Speicher aus dem erstgenannten Bereich alloziert. Wenn dieser Bereich vollläuft, wird eine sogenannte *Minor-Collection* durchgeführt. Dabei werden sämtliche Objekte aus dem Bereich der Young-Generation überprüft. Wird ein Objekt nicht mehr benötigt, kann der belegte Platz freigegeben werden. Falls dies nicht der Fall ist, wird das Alter des Objektes betrachtet. Wenn es alt genug ist, wird es in den Bereich der Tenured-Generation verschoben, andernfalls bleibt es in der Young-Generation. Jeder Minor-Collection steht aber nur ein begrenzter Raum - der sogenannte *Survivor-Space* - für das Halten von jungen Objekten in der Young-Generation zur Verfügung. Sobald dieser aufgebraucht ist, werden lebendige Objekte unabhängig vom Alter in den Bereich der Tenured-Generation verschoben [45]. So kann es passieren, dass sehr junge Objekte den Bereich der alten auffüllen.

Wenn der Bereich der Tenured-Generation voll läuft, wird eine *Major-Collection* ausgeführt, bei der für alle existierenden Objekte (auch aus der Young-Generation) geprüft wird, ob sie noch gebraucht werden und gegebenenfalls Speicher freigegeben. Eine Major-Collection ist demzufolge aufwendiger, als eine Minor-Collection. Während der Ausführung einer Collection ist die Anwendung blockiert.

SpeeDB wurde so implementiert, dass jede Datenbankanfrage durch ein sogenanntes *Request-Objekt* repräsentiert wurde. Zu Beginn jeder Anfrage wurde ein neues Objekt angelegt und am Ende weggeworfen. Es zeigte sich, dass es dem Garbage-Collector von Java bei manchen Testläufen nicht gelang die kurzlebigen Request-Objekte in der Young-Generation zu belassen. Sie füllten den Bereich der Tenured-Generation auf, was zu Major-Collections führte. Diese wiederum hatten hohe Antwortzeiten von Datenbankanfragen, die zur Zeit der Major-Collection im System waren, zur Folge.

In Abbildung 5.8 wird das Problem veranschaulicht. Sie zeigt den zeitlichen Verlauf der Speicherbelegung und das Auftreten von Datenbankanfragen, deren Bearbeitung länger als eine Sekunde dauerte. In dem Test, der zu diesem Verlauf führte, wurden 10 Millionen schreibende und 50 Millionen lesende Anfragen an das DBS gesendet. Die Datenbank war zu Beginn des Tests leer. Die schreibenden Anfragen fügten dann je ein neues Objekt pro Anfrage hinzu. Jeder schreibenden Anfrage folgten fünf lesende Anfragen. Zu Beginn ist nur wenig Speicher belegt. Im Verlauf steigt der belegte Platz jedoch an. Die kleinen Zacken in der Linie sind auf durchgeführte Minor-Collections zurückzuführen. Jede von ihnen setzt etwas Speicher frei. Ihre Ausführung dauert ca. eine Sekunde. Da die Anwendung während einer Collection blockiert ist, treten auch Datenbankanfragen mit

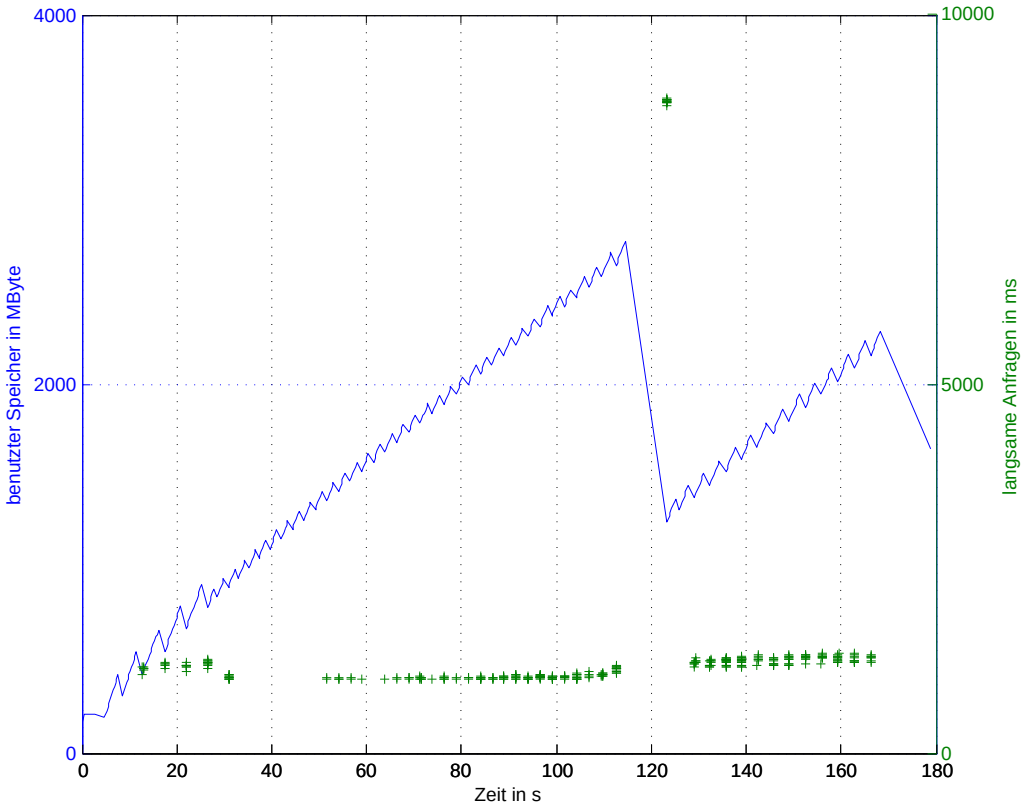


Abbildung 5.8: Hohe Latenzen bei Garbage-Collections

langen Antwortzeiten (länger als 1 s) auf. Dem Garbage-Collector gelingt es jedoch nicht alle kurzlebigen Objekte also die Request-Objekte im Bereich der Young-Generation zu lassen. Deshalb füllt sich nach und nach der Bereich der Tenured-Generation und etwa bei Sekunde 116 wird eine Major-Collection durchgeführt. Das Gros der kurzlebigen Objekte im Tenured-Bereich kann gelöscht werden, da sie nicht mehr benötigt werden. So werden 1,4 GB dealloziert. Die Durchführung der Major-Collection dauert 8 s, wodurch Requests mit einer sehr langen Antwortzeit entstehen. Am Ende des Tests wird eine Major-Collection vom Test angestoßen. Man erkennt, dass lediglich 1,7 GB für die Datenbank benötigt werden.

Bei mehreren durchgeführten Tests hat sich die Arbeitsweise des Garbage-Collectors deutlich verbessert, wenn man die Option `-XX:SurvivorRatio=1` verwendet hat. Sie vergrößert den Survivor-Space und damit die Anzahl der lebendigen Objekte, die bei einer Minor-Collection im Bereich der Young-Generation bleiben können. Dadurch verringert sich die Wahrscheinlichkeit, dass kurzlebige Objekte in den Tenured-Bereich gelangen. Bei den durchgeführten Messungen dieses Kapitels wurde auf eine korrekte Arbeitsweise des Garbage-Collectors geachtet, sodass keine extrem langen Antwortzeiten entstanden. Für die Messungen bei variierenden Warteschlangeneinstellungen aus Abschnitt 5.4.3 wurde der Garbage-Collector mit der Wahl einer sehr großen Young-Generation deaktiviert.

Die virtuelle Maschine ermöglicht auch die Wahl eines Garbage-Collectors, welcher parallel zur Anwendung läuft ohne diese lange zu blockieren (*concurrent low pause collector*). Tests mit diesem Collector führten zu einem geringeren Transaktionsdurchsatz, wobei das Problem der sehr hohen Antwortzeiten fortbestand.

Objekt-Pools Für eine Datenbank ist das Verhalten des Garbage-Collectors sicher als problematisch zu bewerten. Bei einem größeren Heap wächst der Aufwand für eine Major-Collection und damit die Zeit in der die Anwendung blockiert ist. Um das Problem der Garbage-Collection zu umgehen, soll in einer zukünftigen Version von SpeeDB die Implementierung so geändert werden, dass kurzlebige Objekte wiederverwendet werden. Sie werden in *Object-Pools* verwaltet. Es wird z. B. einen Objekt-Pool für Request-Objekte geben. Bei einer neuen Anfrage an das DBS wird ein Request-Objekt aus dem Pool geholt. Am Ende der Anfrage wird das Objekt nicht wie in der jetzigen Implementierung weggeworfen sondern an den Pool zurückgegeben, wo es recycelt wird und für zukünftige Anfragen verwendet werden kann. Diese Änderung des Codes sorgt für eine enorme Entlastung des Garbage-Collectors.

Zudem ist bei der Verwendung von Objekt-Pools für Request-Objekte zu erwarten, dass mit einer begrenzten Anzahl an zulässigen Request-Objekten ein ausreichender Überlastungsschutz des DBS gegeben ist. Auf die Angabe von maximalen Kapazitäten bei Warteschlangen (siehe Abschnitt 4.4) könnte dann verzichtet werden, wodurch der Synchronisationsaufwand verringert würde.

Die fehlende Kontrolle über den Garbage-Collector zur Laufzeit wurde bereits in einer früheren Arbeit als problematisch bewertet [39]. Die Autoren lösten das Problem ebenfalls mit Objekt-Pools.

Kapitel 6

Themenbezogene Arbeiten

6.1 Logging und Recovery in HSDBS

Aufgrund der Flüchtigkeit des Mediums Hauptspeicher waren Logging- und Recoverytechniken für HSDBS über mehrere Jahre Gegenstand der Forschung. Da in einem HSDBS ausschließlich Logging- und Recoverytechniken einen Zugriff auf den Sekundärspeicher erfordern, stellen sie einen Flaschenhals des Systems dar und müssen mit großer Sorgfalt konzipiert werden. Es gibt mehrere Arbeiten, die Anforderungen an die Techniken herausstellen [15, 14], existierende Techniken vergleichen [37, 26] und eigene Ansätze vorstellen [18, 50, 24, 14, 37, 26, 31, 28].

Beim Logging gibt es zwei Varianten: das *logische* und das *physische Logging*. Das physische Logging notiert die Änderungen der Daten auf Byte-Ebene. Werden Speicherseiten infolge einer Operation geändert, so wird mit dem physischen Logging die Änderung der Speicherseiten protokolliert. Beim logischen Logging hingegen wird der Aufruf der Operation zusammen mit dem verwendeten Argument geloggt. Da mit logischem Logging die Semantik einer Operation erfasst wird, kann das Volumen der Logdatei erheblich reduziert werden [24]. In SpeeDB wird deshalb mit logischem Logging der Aufruf der atomar und seriell ausgeführten BDF geloggt. Die Daten werden von SpeeDB auch nicht in Seiten verwaltet. Wollte man physisches Logging verwenden, könnte man die Daten wie in Siren [32] in logischen Seiten verwalten.

Zusätzlich zum Logging werden immer Sicherungspunkte oder auch Checkpoints angefertigt. Ein Sicherungspunkt fängt einen Datenbankzustand ein. Bei der Recovery nach einem Systemausfall wird erst der Sicherungspunkt geladen und dann geloggte Änderungen, die noch nicht im Sicherungspunkt reflektiert sind, eingespielt. Danach ist der aktuellste, konsistente Datenbankzustand rekonstruiert und das DBS kann den Betrieb wieder aufnehmen. Ohne Sicherungspunkte, würde die Recovery zuviel Zeit in Anspruch nehmen.

Sicherungspunkte werden nach der Konsistenz des eingefangenen Datenbankzustandes klassifiziert: Es gibt *transaktionskonsistente*, *aktionskonsistente* und *unscharfe (fuzzy) Sicherungspunkte*. Ein transaktionskonsistenter Sicherungspunkt darf nur Datenbankzustände

sichern, die keine Änderungen nicht comitteter Transaktionen enthalten. Die Datenbank befindet sich in einem transaktionskonsistenten Zustand, wenn keine schreibenden Transaktionen aktiv sind. Ein aktionskonsistenter Sicherungspunkt darf nur Datenbankzustände sichern, in denen sämtliche Aktionen bzw. Operationen der Transaktionen abgeschlossen sind. Unscharfe Sicherungspunkte bzw. fuzzy Checkpoints können hingegen einen beliebigen Datenbankzustand einfangen.

Fuzzy Checkpoints sind leicht zu Erstellen, ohne dabei in Konflikt mit parallelen Transaktionen zu kommen [37]. Jedoch ist die Kombination von logischem Logging mit fuzzy Checkpoints sehr ungünstig, da sich ein konsistenter Datenbankzustand aus einem fuzzy Checkpoint nur sehr schwer ohne physische Logs rekonstruieren lässt [50]. In [50] wird deshalb ein hybrides Logging betrieben: während der Anfertigung eines fuzzy Checkpoints wird physisch und ansonsten logisch geloggt. In SpeeDB sollte aber ausschließlich logisches Logging praktiziert werden. Da mit den BDF keine Aktionen sondern Transaktionen geloggt werden, musste nach einem Verfahren gesucht werden, dass transaktionskonsistente Sicherungspunkte in Kombination mit logischem Logging verwendet. Die transaktionskonsistenten Sicherungspunkte sollten dabei erstellt werden können, ohne den normalen Datenbankbetrieb zu blockieren.

Mit der *Copy-On-Update-Methode* [13, 14] können transaktionskonsistente Checkpoints nicht blockierend erstellt werden. Bei der Methode wird zu Beginn eines Sicherungspunktes der aktuelle Datenbankzustand eingefroren. Transaktionen, die Daten ändern wollen, kopieren zunächst die Daten und führen die Änderungen auf der Kopie durch. Durch das Erstellen der Kopien entsteht ein potentiell doppelter Speicherbedarf [32] der Datenbank. Wann die Methode zu einem doppelten Speicherbedarf führt, hängt jedoch von der Granularität der Kopien ab. In [32] gehen die Autoren von Kopien auf Seitenebene aus, d. h. es wird eine Kopie von einer Seite erstellt, sobald ein Bit dieser geändert werden soll. Angenommen eine Seite enthält 10000 Tupel. Bei Gleichverteilung der zu ändernden Tupel auf die Seiten erfordert bereits die Änderung von $\frac{1}{10000}$ aller Tupel der Datenbank die Kopie sämtlicher Seiten und damit den doppelten Speicherbedarf.

In [32] werden Tupel deshalb nicht in physische Seiten des Hauptspeichers abgelegt, sondern in logischen Seiten verwaltet. Bei der Änderung eines Tupels wird dann nicht die komplette Seite kopiert sondern nur das Tupel (*tuple shadowing*). Eine logische Seite besteht aus einer verketteten Liste von Tupeln, die leicht geändert werden kann. Zur Zeit eines Checkpoints bleibt der Inhalt einer logischen Seite unverändert. Die von Transaktionen vorgenommenen Änderungen werden als *pending operations* (unerledigte Operationen) in der Liste vermerkt und sind für andere Transaktionen sichtbar. Wird beispielsweise ein Tupel geändert, wird eine Kopie des Tupels erzeugt, die Änderung an der Kopie vorgenommen. Im Anschluss wird das Einfügen der neuen Version in die logische Seite als unerledigte Operation für diese Seite vermerkt. So wird erreicht, dass bei der Änderung von einem Tupel nur dieses und nicht eine komplette Seite kopiert wird. Durch die feinere Granularität wird bei der gleichen Anzahl an geänderter Tupel weniger Platz verbraucht als bei Kopien auf Seitenebene. Werden alle Tupel geändert, so führt demzufolge auch das Kopieren auf Tupelebene zum doppelten Platzbedarf.

In dem in dieser Diplomarbeit entwickelten DBS kommt ebenfalls die Copy-On-Update-Methode zum Einsatz und es wird dabei auch auf Tupel- bzw. Objektebene gearbeitet (Siehe Abschnitt 4.3). Transaktionskonsistente Sicherungspunkte können so mit ähnlich geringem Bedarf an zusätzlichen Hauptspeicher erstellt werden, ohne Transaktionen zu blockieren.

6.2 Rekonstruktion von Zugriffspfaden

Die schnelle Rekonstruktion von Indizes beim Laden einer Datenbank in den Hauptspeicher begünstigt die Recoverygeschwindigkeit des Systems. In dieser Diplomarbeit wurde ein Verfahren mit linearem Aufwand für die Rekonstruktion der Primärindizes angegeben. Das vorgestellte Verfahren setzt allerdings voraus, dass die Elemente eines Indizes im Sicherungspunkt sortiert abgelegt werden. Im Allgemeinen kommen in DBS aber auch Sekundärindizes zum Einsatz, wodurch ein Datenelement in mehreren Indizes auftreten kann und eine Sortierung der Datenelemente im Sicherungspunkt nicht für alle Indizes gegeben ist. In [25, 29] werden Verfahren für die Rekonstruktion von B+-Bäumen aus einer unsortierten Menge an Datenelementen beschrieben. Ideen des schnellsten Verfahrens *Max-PL* [29] lassen sich auch für die Rekonstruktion von T-Bäumen verwenden.

6.3 Techniken für moderne Hardware

In den letzten Jahrzehnten wuchs die Geschwindigkeit der Prozessoren wesentlich schneller als die des Hauptspeichers, was dazu führte, dass Hauptspeicherzugriffe heute für datenintensive Anwendungen zunehmend der neue Flaschenhals sind. In der Speicherhierarchie klafft ein immer größer werdendes Loch zwischen dem *Cache* und dem Hauptspeicher. Findet sich eine angeforderte Cache-Line nicht im Cache, liegt ein *Cache-Miss* vor und sie muss aus dem Hauptspeicher geholt werden. Das Lesen der Cache-Line ist mit einer großen Verzögerung verbunden, in der die CPU warten muss. Die Entwicklung performanter Programme erfordert deshalb Techniken, die die zeitliche und örtliche Lokalität von Daten und Instruktionen erhöhen, damit die Anzahl der Cache-Misses reduziert werden [2].

Datenstrukturen Lehmans T-Baum [30] wurde als Datenstruktur für HSDBS entworfen. Allerdings waren Hauptspeicherzugriffe zu der Zeit vernachlässigbar günstig, so dass die Lokalität der Daten nicht berücksichtigt wurde. Messungen, bei denen der T-Baum mit anderen Datenstrukturen verglichen wurde [30], würden auf moderner Hardware zu einem anderen Ergebnis führen und den T-Baum als ungeeignete Struktur einstufen [34, 33]. Der B-Baum hingegen ist flacher als der T-Baum. Zudem zeigt er eine bessere zeitliche Lokalität, weil durchschnittlich die Hälfte der Schlüssel eines Knotens gelesen werden. Beim T-Baum werden dagegen oft nur der kleinste und der größte Schlüssel der Knoten gelesen.

Sogenannte *cache conscious* Datenstrukturen berücksichtigen die Größe einer *Cache-Line* beim Versuch Daten mit einer möglichst geringen *Cache-Miss-Rate* bereitzustellen. Chilimbi et al. nennen Techniken für den Entwurf solcher Strukturen [46]. Die daraus entnommene Technik des *Structure-Splittings* führte auch bei dem in dieser Arbeit implementierten T-Baum zu höheren Zugriffsgeschwindigkeiten. Rao und Ross entwarfen Varianten von B+-Bäumen die *cache conscious* sind [34].

Systemarchitekturen Die Stage-Architektur wie in SEDA [49] ermöglicht eine höhere Lokalität der Instruktionen, da ein Thread einer Stage immer nur einen Programmteil abarbeitet. Voraussetzung dafür ist aber, dass die Threads unterschiedlicher Stages auch auf verschiedenen CPUs ausgeführt werden. Ist das nicht der Fall, kann die Lokalität durch häufige Kontextwechsel zerstört werden. In *Cohort* werden deshalb die Aufträge der Stages von verfügbaren Prozessoren hordenweise ausgeführt [27]. So wird erreicht, dass auf einem Prozessor derselbe Code für eine gewisse Zeit ausgeführt wird. Inspiriert von diesen Ansätzen motiviert Harizopoulos den Entwurf eines DBS in einer Stage-Architektur [19].

Kapitel 7

Zusammenfassung und Ausblick

In diesem Kapitel werden eine Zusammenfassung der Diplomarbeit sowie Ideen für die weitere Entwicklung des in dieser Arbeit entworfenen DBS SpeeDB gegeben.

7.1 Zusammenfassung

In dieser Diplomarbeit wurde ein aktueller Trend aufgegriffen und für die Metadatenverwaltung des verteilten, objektbasierten Dateisystems XtreemFS ein Datenbanksystem maßgeschneidert. Die Motivation dabei war, mit diesem spezialisierten System eine höhere Performanz als mit herkömmlichen, nach dem *One-Size-Fits-All*-Ansatz gestrickten Datenbanksystemen zu erzielen. Es wurde zunächst zusammen mit den XtreemFS-Entwicklern eine Anforderungsspezifikation durchgeführt. Daraufhin wurde ein DBS entworfen, das gerade den gestellten Anforderungen gerecht wird und keinen Mehraufwand für die Bereitstellung nicht benötigter Funktionen betreibt.

Bei dem entworfenen System handelt es sich um ein eingebettetes Hauptspeicher-Datenbanksystem. Es besitzt ein einfaches Datenmodell, bei dem lediglich *Schlüssel-Werte-Paare* in Zugriffspfaden abgelegt werden. Der Zugriff auf die Datenbank erfolgt über den Aufruf von *benutzerdefinierten Funktionen (BDF)*. Diese werden als *Zweiphasentransaktionen* ausgeführt, wobei ein Abbruch nur während der 1. Phase (der Lesephase) nicht aber während der 2. Phase (der Schreibphase) gestattet ist.

Schreibende benutzerdefinierte Funktionen wurden in der Anforderungsspezifikation als schnell durchführbar angenommen. Lesenden Funktionen soll hingegen für längere Zeit der Zugriff auf einen konsistenten Datenbankzustand gewährt werden. Unter diesen Voraussetzungen wurde der Entwurf so gestaltet, dass das System zwei Betriebsmodi anbietet: den seriellen Modus und den Modus *Snapshot-Reads*.

Im seriellen Modus werden sämtliche Funktionen nacheinander ausgeführt. Im Modus Snapshot-Reads werden beliebig viele lesende Funktionen parallel zu den nacheinander auszuführenden schreibenden Funktionen bearbeitet. Dabei garantiert das DBS, dass jede

Operation der lesenden Funktion die Datenbank in dem Zustand sieht, in dem sie zu Beginn der Funktion war. Etwaige Änderungen durch schreibende Funktionen bleiben für die lesende Funktion somit unsichtbar.

Die Eigenschaften des Systems erlauben auch die Konzeption eines besonderen *Logging-* und *Recovery-Schemas*. Es werden transaktionskonsistente Sicherungspunkte im Zusammenspiel mit logischem Logging verwendet. Während der Anfertigung der Sicherungspunkte wird die Transaktionsbearbeitung nicht blockiert, da die Transaktionen für Änderungen an den Daten eine neue Version im Speicher anlegen (*copy on update*). Logisches Logging reduziert das Volumen der Logdatei, weil sämtliche bei der Ausführung einer BDF vorgenommenen Änderungen in einem Logeintrag zusammengefasst werden. Gleichzeitig werden in dem vorgestellten Schema nur Informationen für die wiederholte Ausführung (*Redo*) einer benutzerdefinierten Funktion geloggt. Ein *Undo-Log* wird nicht benötigt, weil die BDF als Zweiphasentransaktionen durchgeführt werden und ein Abbruch deshalb nur bei einem Applikationsfehler auftreten kann. Änderungen einer Transaktion müssen bei einem Applikationsfehler nicht rückgängig gemacht werden können, da sie lediglich Daten im Hauptspeicher betreffen, die im Fehlerfall verworfen werden.

Sämtliche Zugriffe auf den Sekundärspeicher erfolgen sequentiell. Für das schnelle Laden eines Sicherungspunktes während der Recovery wurde ein Algorithmus angegeben, der die Zugriffspfade mit linearem Aufwand rekonstruiert.

Für die Konservierung älterer Datenbankversionen im Modus *Snapshot-Reads* wurde ein Verfahren aus der Literatur ausgewählt, mit dem Datenstrukturen kreiert werden können, die mehrere Versionen der Daten ohne den Einsatz von Sperren bereithalten. Für das Erstellen von Sicherungspunkten wird auch im seriellen Betriebsmodus die Konservierung einer älteren Version benötigt. In der Arbeit wurde ein eigenes Verfahren entwickelt, was dies leistet.

Das eingebettete DBS wurde in der *Stage-Architektur* realisiert, die besser skaliert als das *Threaded-Server-Modell* und eine höhere zeitliche Lokalität bei Instruktionen und Daten zulässt. In der Arbeit wurden auch Überlegungen zu den Warteschlangen der Stages angestellt, die zu einer nachweislich besseren Performanz führten.

Der Entwurf des DBS wurde prototypisch in Java implementiert. Im Anschluss an die Implementierung wurde eine ausführliche Evaluierung durchgeführt. In einem Versuch bei dem Dateimetadaten aus der Datenbank gelesen wurden, zeigte das System einen maximalen Transaktionsdurchsatz von mehr als 300 Anfragen pro ms. Die mittlere Antwortzeit lag dabei unter 50 ms. Bei ausschließlich schreibenden Anfragen sank der Durchsatz auf 230 Anfragen pro ms, wobei die mittlere Antwortzeit 200 ms betrug. Es wurden auch die Geschwindigkeiten von Logging- und Recovery-Aktionen ermittelt. Im Rahmen der Evaluierung wurde ebenfalls eine genauere Betrachtung des *Garbage-Collectors* der virtuellen Maschine von Java erforderlich, um mehrere auffällige Werte zu verstehen.

7.2 nächste Schritte und Richtungen

Nach der Bereitstellung einiger noch fehlender Funktionaltitäten können Code-Optimierungen vorgenommen werden. Es ist zu prüfen, wie sich die Leistung bei der Verwendung von Objekt-Pools (Siehe Abschnitt 5.5) ändert und ob sie als Überlastungsschutz verwendet werden können, was die Angabe von maximalen Kapazitäten für die Warteschlangen der Stages redundant machte. Interessant ist auch, wie sich die Performanz der Warteschlangen ändert, wenn die Warteschlangen in Arrays organisiert werden. Benachbarte Einträge würden auch aufeinanderfolgend im Speicher alloziert, was eine bessere örtliche Lokalität zur Folge hätte. Ferner könnten Komprimierverfahren ausgewählt werden, um das Volumen der Logs und Checkpoints zu verringern. Die Tatsache, dass sämtliche Zugriffe auf diese Dateien sequentiell erfolgen, ist sehr vorteilhaft für die Bewältigung dieser Aufgabe. Auch die Verwendung von B-Bäumen anstelle von T- und AVL-Bäumen würde das System zusätzlich beschleunigen (Siehe Abschnitt 6.3).

Nach dem Optimierung des entwickelten Systems SpeeDB, ist ein Vergleich zu ähnlichen Systemen beim Einsatz im XtreemFS-Szenario interessant. Kontrahenten wären *Berkeley DB JE*, *h2*, *db4o*, *hsqldb*, *derby* und andere.

Als umfangreichere Weiterentwicklung wäre die Verfeinerung des logischen Datenmodells sinnvoll. Das Datenmodell von *Berkeley DB* könnte hier als Vorbild dienen. Die Motivation dabei ist, dass das DBS Informationen über das Schema der Daten einer Anwendung für ein bessere physische Datenorganisation verwenden kann. So könnte Redundanz vermindert und Traversierungen von Zugriffspfaden eingespart werden. Die Aufgabe des für viele Anwendungen zu einfachen Schlüssel-Werte-Ansatzes und die Unterstützung eines komplexeren Modells, würde den Einsatz des DBS auch in einem breiteren Anwendungsspektrum erlauben.

Eine weitere interessante Richtung, ist die Entwicklung zum verteilten Datenbanksystem. Mit Partitionierungs- und Replikationsverfahren würde die Skalierbarkeit und die Hochverfügbarkeit auf der Ebene des DBS und nicht der Clientanwendung realisiert werden.

Literaturverzeichnis

[1] G.M. Adel'son-Vel'skii and E.M. Landis. An algorithm for the organization of information. In *Dokl. Akad. Nauk USSR*, 1962.

[2] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. Dbmss on a modern processor: Where does time go? In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[3] Baris Aktemur, Joel Jones, Samuel N. Kamin, and Lars Clausen. Optimizing Marshalling by Run-Time Program Generation. In *GPCE*, pages 221–236, 2005.

[4] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, February 1972.

[5] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[6] P. Bohannon, D. Leinbaugh, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databases. Technical Report 113880-951031-12, AT&T Bell Laboratories, 1995.

[7] Philip Bohannon, Daniel Lieuwen, Rajeev Rastogi, Avi Silberschatz, S. Seshadri, and S. Sudarshan. The Architecture of the Dalí Main-Memory Storage Manager. *Multimedia Tools and Applications*, 4(2):115–151, March 1997.

[8] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 225–237, Asilomar, CA, USA, January 2005.

[9] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. *SIGPLAN Not.*, 27(9):2–9, 1992.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th*

USENIX Symposium on Operating Systems Design and Implementation, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[11] Surajit Chaudhuri and Gerhard Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *In VLDB*, pages 1–10, 2000.

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[13] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 1–8, New York, NY, USA, 1984. ACM.

[14] Margaret H. Eich. Main memory database recovery. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 1226–1232, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[15] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[16] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[17] Jim Gray. The Next Database Revolution. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 1–4, New York, NY, USA, 2004. ACM Press.

[18] R. B. Hagmann. A Crash Recovery Scheme for a Memory-Resident Database System. *IEEE Trans. Comput.*, 35(9):839–843, 1986.

[19] Stavros Harizopoulos. A Case for Staged Database Systems. In *In Proceedings of 1st Conference on Innovative Data Systems Research*, 2003.

[20] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 981–992, New York, NY, USA, 2008. ACM.

[21] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. The XtreemFS architecture—a case for object-based file systems in Grids. *Concurr. Comput. : Pract. Exper.*, 20(17):2049–2060, 2008.

[22] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. XtreemFS - a case for object-based storage in Grid data management, 2007.

[23] Theo Härder and Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer, 2 edition, 2001.

[24] H. Jagadish, A. Silberschatz, and S. Sudarshan. Recovering from Main-Memory Lapses. In *Proceedings of the 19th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Dublin*, 1993.

[25] Sang-Wook Kim and Hee-Sun Won. Batch-construction of B+-trees. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 231–235, New York, NY, USA, 2001. ACM.

[26] V. Kumar and A. Burger. Performance Measurement of Main Memory Database Recovery Algorithms Based on Update-in-Place and Shadow Approaches. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):567–571, 1992.

[27] James R. Larus and Michael Parkes. Using Cohort Scheduling to Enhance Server Performance. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 182–187, New York, NY, USA, 2001. ACM.

[28] Dongho Lee and Haengrae Cho. Checkpointing schemes for fast restart in main memory database systems. In *In 1997 IEEE Pacific Rim Conf. on Communications, Computers, and Signal Processing, Victory, BC*, pages 663–668, 1997.

[29] Ig-Hoon Lee, Junho Shim, and Sang-Goo Lee. Fast Rebuilding B+-Trees for Index Recovery. *IEICE - Trans. Inf. Syst.*, E89-D(7):2223–2233, 2006.

[30] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[31] Eliezer Levy and Avi Silberschatz. Incremental Recovery in Main Memory Database Systems. Technical report, Austin, TX, USA, 1992.

[32] Antti-Pekka Liedes and Antoni Wolski. SIREN: A Memory-Conserving, Snapshot-Consistent Checkpoint Algorithm for in-Memory Databases. *22nd International Conference on Data Engineering (ICDE'06)*, 2006.

[33] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal*, 9(3):231–246, 2000.

[34] Jun Rao and Kenneth A. Ross. Making b+- trees cache conscious in main memory. *SIGMOD Rec.*, 29(2):475–486, 2000.

[35] Rajeev Rastogi, S. Seshadri, Philip Bohannon, Dennis W. Leinbaugh, Abraham Silberschatz, and S. Sudarshan. Logical and Physical Versioning in Main Memory Databases. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 86–95, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[36] Mendel Rosenblum. *The Design and Implementation of a Log-Structured File System*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.

[37] K. Salem and H. Garcia-Molina. System M: A Transaction Processing Testbed for Memory Resident Data. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):161–172, 1990.

[38] Margo Seltzer. Beyond relational databases. *Commun. ACM*, 51(7):52–58, 2008.

[39] Mehul A. Shah, Michael J. Franklin, Samuel Madden, and Joseph M. Hellerstein. Java support for data-intensive systems: experiences building the telegraph dataflow system. *SIGMOD Rec.*, 30(4):103–114, 2001.

[40] Michael Stonebraker, Chuck Bear, Ugur Cetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stanley B. Zdonik. One Size Fits All? Part 2: Benchmarking Studies. In *CIDR*, pages 173–184, 2007.

[41] Michael Stonebraker and Ugur Cetintemel. Öne Size Fits All̈: An Idea Whose Time Has Come and Gone. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[42] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*, pages 1150–1160, 2007.

[43] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented dbms. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[44] Inc. Sun Microsystems. Tuning Garbage Collection with the 5.0 Java™Virtual Machine. `http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html#1.1.%20Types%20of%20Collectors|outline`, 2003 (accessed January 20, 2009).

[45] Inc. Sun Microsystems. Frequently Asked Questions about Garbage Collection in the Hotspot™Java™Virtual Machine. `http://java.sun.com/docs/hotspot/gc1.4.2/faq.html`, February 6, 2003 (accessed January 20, 2009).

[46] Sc Io Us, Trishul M. Chilimbi, and Mark D. Hill. Making pointer-based data structures cache conscious. *IEEE Computer*, 33:2000, 2000.

[47] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesińska, Rutger F. H. Hofman, Ceriel J. H. Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a flexible and efficient Java-based Grid programming environment: Research Articles. *Concurr. Comput. : Pract. Exper.*, 17(7-8):1079–1107, 2005.

[48] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems – Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann Publishers, San Mateo, CA, USA, 2001.

[49] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, volume 35, pages 230–243, New York, NY, USA, December 2001. ACM Press.

[50] Seung-Kyoon Woo, Myoung-Ho Kim, and Yoon-Joon Lee. An effective recovery under fuzzy checkpointing in main memory databases. *Information & Software Technology*, 42(3):185–196, 2000.