OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

INF

FAKULTÄT FÜR INFORMATIK

Diplomarbeit

Untersuchung von Prefetching-Strategien in Objekt-relationalen Mappern

Verfasser:
Michael Lipaczewski
04.03.2011

Betreuer:
Dr.-Ing. Eike Schallehn
Otto-von-Guericke Universität Magdeburg
Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme
39106 Magdeburg, Germany

Dipl.-Ing. (FH) Philipp Noggler
Eads Deutschland GmbH
Cassidian Systems
88039 Friedrichshafen, Germany

Inhaltsverzeichnis

1 Einführung ... 9

2 Grundlagen ... 11

2.1 Datenbanken ... 11
2.2 Query ... 12
2.3 Inpedance Mismatch ... 14
2.4 Objektrelationale Mapper ... 15
2.4.1 Basisdatentypen ... 15
2.4.2 Identität ... 15
2.4.3 Repräsentation von Abhängigkeiten ... 17
2.4.4 Repräsentation von Vererbung ... 20
2.5 Persistenz ... 23
2.6 Hibernate ... 23
2.6.1 Session Konzept ... 25
2.6.2 Hibernate Cache ... 25
2.6.3 Hibernate Lazy Loading ... 26
2.6.4 Persistenz in Hibernate ... 26
2.7 Notationen ... 28
2.7.1 Objektgraph ... 28
2.7.2 Anfrage ... 28
2.7.3 Abfrage ... 28

3 Fetching-Methoden im Überblick ... 29

3.1 Anforderungen an Fetchingverfahren ... 30
3.2 Vergleichsgrundlagen ... 31
3.3 Fetchingverfahren ... 35
3.3.1 On-demand fetching ... 35
3.3.2 Prefetching ... 36
3.4 Ergebnisse des Vergleiches ... 48

4 Aufgabenstellung 51

4.1 Ausgangssituation 51
4.2 Problemstellung 52

5 Lösungsansätze 55

5.1 Lösung des Lazy Loading Problems 55
5.1.1 Eingriff in die Hibernate Konfiguration 55
5.1.2 Verwendung eines Session Proxys 56
5.1.3 Verwendung von Preload Pattern 57
5.2 Persistenzbetrachtung 58
5.3 Auswertung 58

6 Umsetzung des Prototypen 61

6.1 Implementierung der Testumgebung 61
6.2 Implementierung des on-demand Fetching 64
6.3 Implementierung des PreloadPattern 66
6.4 Implementierung des CriteriaJoiner 67

7 Performanceuntersuchung des Prototypen 69

7.1 Vergleichsgrundlagen 69
7.2 Ergebnisse 71
7.3 Auswertung 81
7.4 Aussagekraft der Ergebnisse 82
7.5 Resultate 82

8 Fazit 85

9 Literaturverzeichnis 88

10 Eigenständigkeitserklärung 91

Abbildungsverzeichnis

Abbildung 1: Aufbau einer Relation 11

Abbildung 2: Beispiel Eins-zu-Eins Beziehung 17

Abbildung 3: Eins-zu-N Beziehung 18

Abbildung 4: Beispiel N-zu-N Beziehung 19

Abbildung 5: Beispiel Objektgraph zur Vererbung 20

Abbildung 6: Beispiel-Datenbankschema für Tabelle je Klassenhierarchie 21

Abbildung 7: Beispiel -Datenbankschema für Tabelle je konkrete Klasse 21

Abbildung 8: Beispiel-Datenbankschema für Tabelle je Klasse 22

Abbildung 9: Zustandsmodell persistenter Objekte [11] 27

Abbildung 10: Fetching-Prozess 29

Abbildung 11: Datenbankschema des Beispielgraphen 31

Abbildung 12: Objektschema des Beispielgraphen 32

Abbildung 13: Objektpfad für Abfrage A 33

Abbildung 14: Objektpfad für Abfrage B 33

Abbildung 15: Objektpfad für Abfrage C 34

Abbildung 16: Funktionsweise des PrefetchGuide 41

Abbildung 17: Funktionsweise des ContextLoaders 44

Abbildung 18: Funktionsweise von AutoFetch 46

Abbildung 19: Hibernate Mapping für das Objekt Fakultät 62

Abbildung 20: Hibernate Mapping für das Objekt Vorlesung 62

Abbildung 21: Hibernate Mapping für das Objekt Person 63

Abbildung 22: Hibernate Konfigurationsdatei 63

Abbildung 23: Implementierung der preload Methode des on-demand Fetching 65

Abbildung 24: Implementierung der findAllProfs Methode des on-demand Fetching 65

Abbildung 25: Implementierung der preload Methode des PreloadPattern 66

Abbildung 26: Beispiel eines Preload Pattern des PreloadPattern 67
Abbildung 27: Implementierung der findAllProfs Methode des PreloadPattern 67
Abbildung 28: Implementierung der preload Methode des CriteriaJoiner 67
Abbildung 29: Beispiel eines Preload Pattern des CriteriaJoiner 68
Abbildung 30: Implementierung der FindAllProfs Methode des CriteriaJoiner 68
Abbildung 31: Abfragegeschwindigkeit abhängig von der Anzahl der Professoren 71
Abbildung 32: Abfragegeschwindigkeit abhängig von der Anzahl der Vorlesungen 72
Abbildung 33: Abfragegeschwindigkeit abhängig von der Anzahl der Fakultäten 73
Abbildung 34: Abfragegeschwindigkeit abhängig von der Anzahl der Vorlesungen pro Student . 74
Abbildung 35: Abfragegeschwindigkeit abhängig von der Anzahl der Studenten pro Vorlesung . 75
Abbildung 36: Aufwandsbestimmung bei einer Objekttiefe von eins 78
Abbildung 37: Aufwandsbestimmung bei einer Objekttiefe von zwei, v variabel 79
Abbildung 38: Aufwandsbestimmung bei einer Objekttiefe von zwei, s variabel 80

Tabellenverzeichnis

Tabelle 1: Ressourcenbetrachtung für on-demand Fetching 35

Tabelle 2: Ressourcenbetrachtung für Verfahren von Jürgen Kohl 38

Tabelle 3: Ressourcenbewertung für CriteriaJoiner 38

Tabelle 4: Ressourcenbetrachtung für PrefetchGuide 42

Tabelle 5: Ressourcenbetrachtung für ContextLoader 45

Tabelle 6: Vergleich Fetchingverfahren 48

Abkürzungsverzeichnis

Abkürzung	Beschreibung
C2 System	Command & Control System
DBMS	Database Management System
GUI	Graphic User Interface
HQL	Hibernate Query Language
Hsql	HyperSQL
JDBC	Java Database Connectivity
ODMG	Object Database Management Group
OODB	Object Oriented Database
OR Mapper	Object-Relational Mapper
ORDB	Object Relational Database
RDB	Relational Database
SQL	Structured Query Language
VM	Virtual Machine

1 Einführung

Objektorientierte Programmierung gehört inzwischen zum Standard in der Softwareentwicklung. Dabei werden die Datentypen immer komplexer und die Implementierungsebenen immer tiefer. Hauptursache für diesen Zuwachs ist die Verbesserung der Computersysteme, die es erlauben, auch größere Datenmengen performant im Arbeitsspeicher zu halten und zu bearbeiten. Darüber hinaus erlauben die heutigen Rechengeschwindigkeiten die Möglichkeit, mehrere Operationen parallel auszuführen. Dies ermöglichte die erfolgreiche Anwendung von objektorientierten Systemen. Darüber hinaus stellt es für Entwickler neue und einfachere Wege der Programmierung dar, wodurch die prozedurale Programmierung immer weiter zurückgeht.

Gleichzeitig gewinnen Datenbanken durch die fortschreitende Vernetzung von Computersystemen immer weiter an Bedeutungen. Dabei spielt hauptsächlich die zentrale Datenhaltung eine Rolle, wodurch sich Aspekte wie Datensicherung und Datensicherheit wesentlich besser realisieren lassen. Auch wird über Datenbanken die Mobilität und Flexibilität von vernetzten Systemen gesteigert sowie das gleichzeitige Arbeiten an mehreren geographisch verteilten Systemen ermöglicht. Aber auch die Performance der Systeme ist ein wesentlicher Bestandteil von Datenbanksystemen. So sind Suchabfragen durch die Optimierung auf genau dieses Problem wesentlich schneller als vergleichbare lokale Lösungen. Auch bei großen Datenmengen spielen Datenbanken ihre Vorteile aus. Hier wären enorme Rechenleistungen auf dem Anwendungsrechner notwendig, um ähnliche Ergebnisse zu erzielen, was nicht zuletzt wieder eine Kostenfrage darstellt.

Datenbanken bieten jedoch, abgesehen von speziell entwickelten objektorientierten Datenbanken, nicht die nötigen Fähigkeiten, um einen verlustfreien Übergang von Daten zwischen einer objektorientierten Software und einem Datenbankschema sicherzustellen. Objektorientierte Datenbanken weisen eine Vielzahl von Problemen wie beispielsweise die Performance auf, weswegen sie in der Praxis bis jetzt seltener Verwendung finden. Hauptsächlich wird jedoch Abstand von objektorientierten Datenbanken genommen, da diese weitgehend unbekannt sind und eine neue Art der Programmierung erforderlich machen. Daher wird oftmals auf bewährte Mittel zurückgegriffen.

Dagegen bestechen relationale Datenbanken durch ihre Geschwindigkeit, Leistungsfähigkeit und Einfachheit. Um jedoch ein Objekt in eine solche Datenbank speichern und laden zu können, ist es erforderlich, die entsprechenden Objekte in einfache Datensätze zu zerlegen bzw. aus diesen Datensätzen wieder Objekte zu generieren. Dazu dienen sogenannte objektrelationale Mapper.

Objektrelationale Mapper stellen die Schnittstelle zwischen objektorientierter Programmierung und relationalen Datenbanksystemen dar. Dazu werden Regeln definiert, welche den Schreib- und Lesezugriff beschreiben. Dadurch wird es der Applikation erlaubt, sich ohne spezielles Wissen den Aufbau der Datenbank auf dem Objektmodell wie auf einem lokalen Modell zu bewegen. Das Fetching, also das Abrufen der Daten aus der Datenbank, erfolgt dabei vollkommen automatisch. Jedoch ergeben sich dadurch verschiedene Probleme. Hauptsächlich beziehen sich diese Probleme dabei auf Performancefragen durch ein stückweises Laden der Objektdaten, durchgängige Datenbankanbindung sowie komplizierte Datenbankanfragen.

Als Resultat daraus wurden verschiedene Prefetching Verfahren entwickelt, die die Performance des Gesamtsystems durch das Vorausladen von Daten verbessern sollen. Dabei besteht die Möglichkeit, manuell das zu ladende Objektmodell zu bestimmen oder ein sich selbst optimierendes Verfahren zu verwenden. Eine Reihe von Verfahren sollen nun im Laufe der Arbeit sowohl erörtert als auch bewertet werden. Diese Ergebnisse werden anschließend genutzt, um für eine konkrete Problemstellung ein oder mehrere Verfahren auszuwählen und einen Prototyp zu entwickeln, der die Machbarkeit, Leistungsfähigkeit und den Performancegewinn des Verfahrens aufzeigen soll.

Ein weiterer, jedoch weit seltener betrachteter Vorteil von Prefetching Verfahren ist die Überlegung, dass bei entsprechendem Vorausladen des benötigten Objektmodells keine Datenbankverbindung mehr notwendig ist. Daraus resultierend ergeben sich Überlegungen zur Leistungssteigerung des Datenbanksystems, zur Entlastung der Netzwerkinfrastruktur sowie zu verschiedenen entwicklungstechnischen Überlegungen. Im Zuge der Entwicklung des Prototyps sollen diese Überlegungen nun konkretisiert und bewertet werden.

2 Grundlagen

2.1 Datenbanken

Datenbanken dienen der elektronischen Datenverwaltung. Hauptaufgabe dieser Systeme ist das effiziente, widerspruchsfreie und dauerhafte Speichern von Datenmengen in unterschiedlichen, bedarfsgerechten Darstellungsformen. Datenbanken selber stellen dabei nur einen logisch zusammengehörigen Datenbestand dar. Alle anderen Aspekte, welche von einer Datenbank gefordert werden, realisiert man durch ein Datenbankmanagementsystem (DBMS). Dieses System verwaltet unter anderem das Datenmodell, Speicher- und Ladevorgänge, Mehrbenutzerbetrieb, Datenschutz und –sicherheit sowie die selbstständige Optimierung der Datenbank. Als Einheit betrachtet spricht man dann von einem Datenbanksystem, welches DBMS und Datenbank zusammenfasst. Im allgemeinen Sprachgebrauch hat sich jedoch der kürzere Begriff Datenbank etabliert, obwohl dieser wissenschaftlich nicht korrekt ist.

1970 schlug Edgar F. Codd [1] erstmals einen bis heute etablierten Standard für Datenbanken vor. Grundlage dieses Konzeptes ist die Relation, welche die Beschreibung einer Tabelle darstellt. Operationen werden basierend auf diesem Relationsschema durch die relationale Algebra bestimmt und bilden damit die Grundlage für jede relationale Anfragesprache.

Im Wesentlichen besteht eine relationale Datenbank aus einer Sammlung von Tabellen, den Relationen, in welchen Datensätze in Form von Zeilen gespeichert werden. Eine Zeile wird dabei als Tupel bezeichnet. Die Spalten einer Tabelle werden als Attribute bezeichnet. Jedes Tupel in einer Relation kann damit immer nur dieselben Attribute haben.

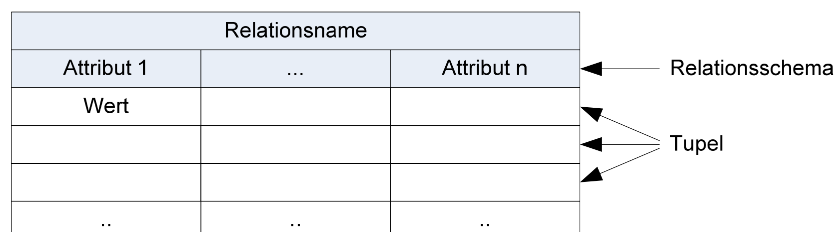


Abbildung 1: Aufbau einer Relation

Weiterhin besteht die Möglichkeit, Beziehungen zwischen diesen Relationen herzustellen und somit die Tupel über die Grenzen einer Relation hinaus zu verknüpfen. Damit ist es möglich, Abhängigkeiten zwischen zwei Einträgen zu erzeugen. Zwei Begriffe werden hierbei besonders

geprägt. Zum einen der Primärschlüssel (Primary Key), welcher zur eindeutigen Identifikation jedes Tupel verwendet wird und der nur ein einziges Mal in der Relation verwendet werden darf. Damit ist sichergestellt, dass jedes Tupel eineindeutig angesprochen werden kann. Ein Primärschlüssel kann dabei auch aus mehreren Attributen zusammengesetzt werden. In der Regel jedoch wird ein künstlicher Primärschlüssel angelegt, der unabhängig von den Attributen des Tupel ist. Der zweite Begriff ist der Fremdschlüssel (Foreign Key). Er verweist dabei auf einen Primärschlüssel eines Tupel und muss somit dieselbe Anzahl von Attributen mit denselben Typen enthalten wie der referenzierte Primärschlüssel [2][3][4].

Dieses Konzept wurde mit zunehmender Verbreitung von objektorientierter Programmierung um verschiedene Elemente zur besseren Objekthaltung erweitert. Unter anderem wurden neue Datentypen hinzugefügt, die die Umsetzung komplexer Objekte in Datenbankschemata erleichterte. Weiterhin setzte man teilweise Konzepte zur Darstellung von Vererbung sowie Mengenattributen um[5]. Diese Datenbanksysteme werden dann unter dem Begriff objektrelationale Datenbanken (ORDB) [6][7][8] geführt.

Ein anderes Konzept wurde bei objektorientierten Datenbanken (OODB) [9][10] verfolgt. Auch hier war die treibende Kraft der Vormarsch der objektorientierten Programmierung. Im Gegensatz zu ORDB verfolgt man hier allerdings die Strategie, Objekte ohne die Umwandlung in ein relationales Schema zu speichern. Damit entfallen aufwändige Konvertierungs- und Abfrageverfahren. Die Datenbank kann direkt an die Applikation angebunden werden. Des Weiteren wird das Problem der Objektidentität gelöst. So ist kein künstlicher Schlüssel zur Identifizierung eines Objektes in einer Datenbank mehr erforderlich, da der systemeigene Objektidentifikator verwendet werden kann. Jedoch hat sich diese Form der Datenbank noch nicht durchgesetzt. So gibt es bis heute wenig Schnittstellen und Tools für die Verwendung von OODB. Hauptgrund hierfür sind Performanceprobleme der Datenbankmanagementsysteme, welche sich auf eine hohe Komplexität des Speicherverfahrens zurückführen lassen. Diese Komplexität entsteht durch die Abbildung großer Vererbungs- und Assoziationsketten.

2.2 Query

Als Query wird im Datenbankkontext jede Anfrage an die Datenbank bezeichnet. Zur Formulierung der Abfragen existieren je nach Datenbanksystem verschiedene Abfragesprachen. Eine der bekanntesten dabei ist SQL (Structured Query Language) [4]. In leicht abgewandelter

Form verwendet Hibernate ebenfalls SQL, dann als HQL (Hibernate Query Language)[11] bezeichnet.

Die gängigste Abfrageform besteht bei beiden Sprachen in der Erstellung von Select-Abfragen, welche klassischerweise aus drei Teilen bestehen. Zuerst die Angabe, welche Daten (Spalten) geladen werden sollen, darauf folgend die Angabe der Tabelle, aus denen diese Daten entnommen werden und anschließend die Formulierung von Bedingungen, die die angeforderten Datensätze einschränkt.

Ein Beispiel für eine typische SQL Abfrage der Namen aller Personen aus der Tabelle Person, wobei die Person volljährig sein muss, hätte dabei die Form:

```
SELECT name FROM person WHERE age>17
```

Diese Abfrage lässt sich dabei um beliebig viele Attribute erweitern. Auch die Anzahl der Tabellen und die Anzahl der Bedingungen lassen sich erhöhen. SQL bietet darüber hinaus weitere Funktionen wie das Einschränken der Anzahl der Suchergebnisse, welche hier allerdings nicht weiter erläutert werden sollen.

Von besonderer Bedeutung für diese Arbeit ist jedoch die Möglichkeit, Tabellen miteinander zu verknüpfen. Dazu kann einer Abfrage über den Befehl JOIN mitgeteilt werden, dass eine neue virtuelle Tabelle erstellt werden soll, welche alle Spalten der angegebenen Tabellen enthält. Dabei ist die Angabe von Bedingungen notwendig, die festlegen, wie diese beiden Tabellen verknüpft werden müssen. JOIN-Statements erlauben es, komplexe Anfragen mit Bedingungen über mehrere Tabellen verteilt zu formulieren.

Dieser Sachverhalt ist wichtig, da Hibernate standardmäßig keine JOIN Befehle zur Abfrage aus Datenbanken verwendet. Wenn also Daten aus verschiedenen Datenbanken geladen werden müssen, verwendet Hibernate eine Abfrage pro Tabelle. Im weiteren Verlauf der Arbeit wird beschrieben, wie dieses Verhalten geändert werden kann und welche Auswirkungen dies auf das Ladeverhalten hat.

2.3 Inpedance Mismatch

Objektorientierte Programmierung verwendet Objekte zur Repräsentation von Daten und Modulen [12] im Gegensatz zu relationalen Datenbanken, die Tupel in Relationen [8] verwenden. Objekte bieten dabei eine Vielzahl von Möglichkeiten. Einfache Objekte bestehen lediglich aus einer Vielzahl von einfachen Datentypen. Diese lassen sich solange direkt in eine relationale Datenbank schreiben, wie sie von dieser unterstützt werden. Jedoch können Objekte auch aus komplexen Datentypen, Relationen zu weiteren Objekten oder vererbten Objekten bestehen. In diesen Fällen ist es wichtig, die Art der Relation, Vererbung oder des komplexen Datentypen zu kennen und diese möglichst soweit in einzelne Strukturen zu zerlegen, damit sie in einer Datenbank repräsentiert werden kann.

Daraus ergeben sich zwangsläufig Probleme, welche unter dem Begriff „Inpedance Mismatch“ [13] zusammengefasst werden. Hauptprobleme stellen dabei Struktur, Identität, Datenkapselung und Arbeitsweise dar [14].

Struktur beschreibt dabei die Möglichkeit der Vererbung von Objekten, die von einer generalisierten, abstrakteren Klasse abgeleitet werden. Somit sind ganze Objekthierarchien möglich, welche sich immer weiter konkretisieren und das Arbeiten mit Objekten einer übergeordneten Klasse erlauben, während die eigentlichen Objekte Repräsentationen einer konkreten Klasse sind. Eine solche Struktur ist in einer Datenbank nicht vorgesehen.

Identität betrifft die Repräsentation eines Tupel in der Datenbank. Dieser ist eindeutig durch seinen Primärschlüssel definiert. Wohingegen ein Objekt lediglich während der Laufzeit der Applikation einen Identifier besitzt. So kann bei einem erneuten Ausführen der Anwendung derselbe Identifier einem anderen Objekt zugeordnet werden. Andererseits kann aber auch dasselbe Objekt, geladen aus der Datenbank, zu verschiedenen Zeiten unterschiedliche Identifier aufweisen.

Ebenfalls unterscheiden sich die beiden Datenrepräsentationen durch die Möglichkeit der Datenkapselung. So ist es möglich, die Attribute eines Objektes durch Methoden zu schützen, wodurch nur ein indirekter Zugriff auf die Daten eines Objektes möglich ist, wohingegen Tupel einer Datenbank jederzeit geändert werden können. Viele Datenbankhersteller erweitern ihre Systeme mit dem Versuch, entsprechende Schutzmechanismen zu integrieren. Diese Mechanismen gehören jedoch nicht zum relationalen Datenbankmodell. Innerhalb von Tabellen wird mithilfe von Mengenwerten und Operationen gearbeitet. Dies erinnert eher an eine

prozedurale Programmierweise, wo das Trennen von Daten und Verhalten üblich ist. Bei der Arbeit mit Objekten ist das Verhalten mithilfe von Methoden in den Objekten enthalten, und damit gruppiert mit Daten. Eine mengenbasierte Bearbeitung von Objekten ist daher in objektorientierten Systemen nicht vorgesehen.

2.4 Objektrelationale Mapper

Die meisten Probleme des Inpedance Mismatch können durch objektrelationale Mapper gelöst, kontrolliert oder umgangen werden. Objektrelationale Mapper, kurz OR-Mapper, bilden die Schnittstelle zwischen einer objektorientierten Anwendung und einer relationalen Datenbank. Das Mapping definiert dabei, wie Objekte in der Datenbank gespeichert werden sollen. Dies ist notwendig, da Objekte und relationale Datenbanken verschiedene Strukturen wiederspiegeln.

2.4.1 Basisdatentypen

Ein grundlegendes Problem bei der Konvertierung von Objekten in eine Datenbank besteht in der begrenzten Anzahl von Datentypen des Datenbanksystems. Dabei spielen für diesen Teil der Betrachtungen nur die Basistypen eine Rolle. So unterscheidet sich beispielsweise der Zahlentyp Integer in seiner Definition zwischen Java und einem Datenbanksystem. Diese Mappings sind weitestgehend einfach zu realisieren. Wohingegen andere Datenbanktypen Längenbeschränkungen benötigen. Ein Beispiel dafür ist das Datenbankäquivalent zum einfachen String. In den meisten Datenbanksystemen existiert dieser Typ als VARCHAR und erwartet in der Regel eine Maximallänge. Dies ist wichtig, um die Speicherkapazität der Datenbank zu verringern und die Performance der Lesegeschwindigkeit zu erhöhen. Daher ist es nicht zu empfehlen, an dieser Stelle immer den größtmöglichen Wert zu verwenden.

2.4.2 Identität

Identität bezeichnet die Form, mit der sichergestellt wird, dass ein Objekt eineindeutig bestimmt werden kann. Die meisten objektorientierten Programmiersprachen verwenden zu diesem Zweck interne Objekt-Identifier, wobei diese lediglich eine Eindeutigkeit innerhalb der Laufzeit der Applikation gewährleisten. Dem gegenüber müssen Datenbanken eine eindeutige Zuordnung über diese Objekte auch über das Beenden der Applikation hinaus gewährleisten. Zu diesem Zweck wird in der Regel ein weiteres Attribut Identifier (kurz: ID) dem Objekt hinzugefügt. Damit soll sichergestellt werden, dass ein Objekt, welches aus der Datenbank

geladen wurde, nach dem Editieren auch als editiertes Objekt wieder in die Datenbank geschrieben wird, also die Datenbank aktualisiert wird, anstelle dem Ablegen eines neuen Datensatzes.

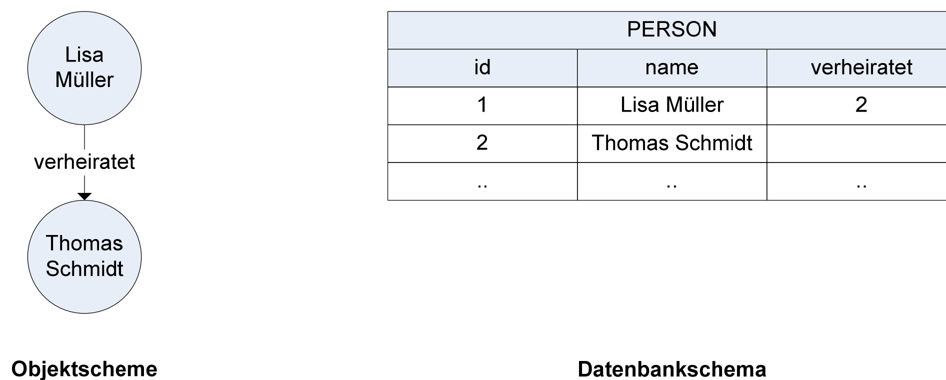
Zum Erzeugen dieser ID als Schlüssel können verschiedene Strategien verfolgt werden. Eine Möglichkeit besteht in der zufälligen Generierung des Schlüssels über einen Zufallsgenerator. Der große Vorteil dieses Prinzips liegt in der Geschwindigkeit des Verfahrens, da sich Zufallszahlen sehr schnell erzeugen lassen. Statistisch gesehen ist bei einem guten Zufallsgenerator die Möglichkeit einer doppelten ID vernachlässigbar gering. In der Praxis jedoch ist die Möglichkeit einer doppelten ID bereits ausschlaggebend für die Entscheidung, keine Zufallszahlen zu verwenden. Wesentlich gängiger ist die Verwendung von ID-Generatoren der Datenbanksysteme. Fast alle bieten verschiedene Generatoren mit verschiedenen Algorithmen an. Nachteil dieses Systems ist allerdings, dass beim Erzeugen eines Objektes ein Datenbankaufruf notwendig ist. Dies kann mitunter Performanceeinbußen zur Folge haben.

2.4.3 Repräsentation von Abhängigkeiten

Abhängigkeiten oder Relationen sind Beziehungen zwischen Objekten. Veranschaulicht soll dies durch das Beispiel eines Objekts des Typs Person werden. Dabei gehen wir von Beziehungen (z.B. Kind, Ehepartner, etc.) zwischen verschiedenen Personen aus. Relationen beschränken sich jedoch nicht auf Abhängigkeiten desselben Objekttyps. [11]

2.4.3.1 Eins-zu-Eins Beziehung

Eine Person kann, zumindest im europäischen Kulturkreis, nur einen Ehepartner haben, wobei sichergestellt ist, dass dieser Ehepartner auch nur diese Person zum Partner hat. Also hat diese Person eine Relation zu einer weiteren Person, wobei diese Relation durch die Beschränkung der Anzahl auf ein Element als Eins-zu-Eins Beziehung bezeichnet wird.



PERSON		
id	name	verheiratet
1	Lisa Müller	2
2	Thomas Schmidt	
..	..	..

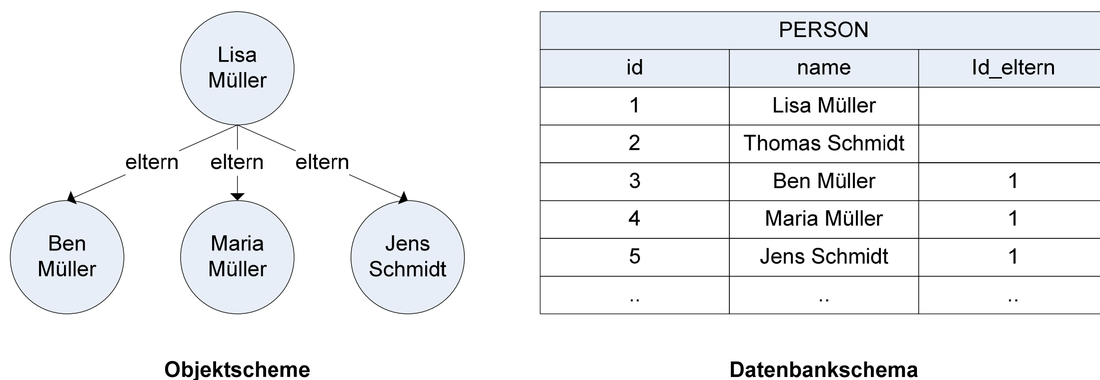
Datenbankschema

Abbildung 2: Beispiel Eins-zu-Eins Beziehung

Diese Art der Beziehung lässt sich in einer relationalen Datenbank durch Verwendung eines Fremdschlüssels realisieren. Dabei wird in der Tabelle „Person“ eine weiter Spalte „Ehepartner“ erstellt, welche den Index der anderen Person enthält.

2.4.3.2 Eins-zu-N bzw. N-zu-Eins Beziehung

Im Unterschied zur Eins-zu-Eins Beziehung besteht natürlich auch die Möglichkeit, dass ein Objekt mehrere Objekte referenziert. Dieser Fall wird dann als Eins-zu-N Beziehung bezeichnet. Ein geeignetes Beispiel hierfür ist die Beziehung einer Person mit mehr als einem Kind. Dabei gehen wir von einer Objektstruktur aus, in der die Kinder jeweils nur der Mutter, also nur einer Person, zugeordnet werden.



PERSON		
id	name	Id_eltern
1	Lisa Müller	
2	Thomas Schmidt	
3	Ben Müller	1
4	Maria Müller	1
5	Jens Schmidt	1
..	..	..

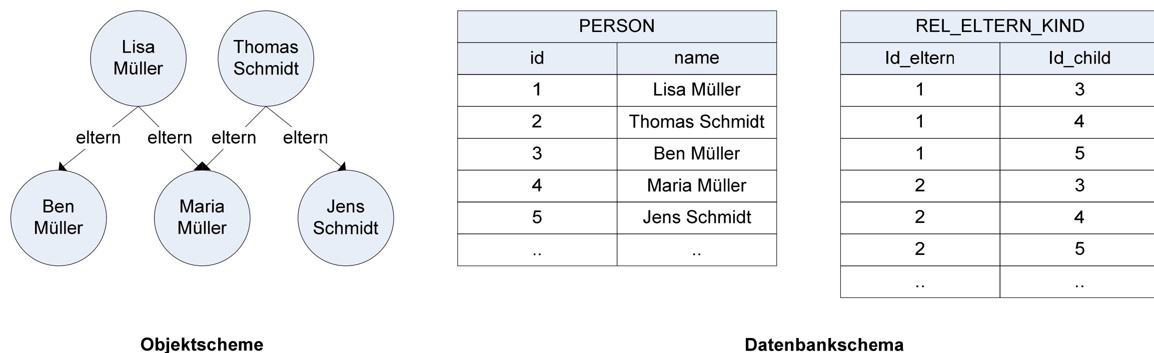
Abbildung 3: Eins-zu-N Beziehung

Ähnlich wie bei der Eins-zu-Eins Beziehung wird auch hier die Referenz direkt in der Objekttabelle hinterlegt, wobei zu beachten ist, welche Beziehungsrichtung gespeichert wird. Gibt man den Fremdschlüssel in diesem Beispiel der Mutter, muss für jedes Kind eine extra Spalte in der Tabelle erstellt werden. Das Erstellen von N-Spalten macht nicht nur die Datenbankschemen groß und unübersichtlich, sondern verringert auch die Performance des Datenbanksystems. Speichert man hingegen die entgegengesetzte Richtung, also die Mutter des Kindes als Fremdschlüssel, braucht man lediglich eine Spalte, da nur eine Mutter vorhanden sein kann.

Gleiches gilt auch für die N-zu-Eins Beziehung. Ein Beispiel dazu wären beide Elternteile, die nur ein Kind besitzen. In beiden Fällen kann der Fremdschlüssel im Objektschema gespeichert werden. Dabei muss die Richtung der Beziehung beachtet werden, damit der Schlüssel auf der N-Seite der Beziehung gespeichert wird.

2.4.3.3 N-zu-N Beziehung

Die letzte mögliche Relation ist das Kombinieren von beliebig vielen Objekten mit beliebig vielen anderen Objekten. Als Beispiel werden Relationen zu den Kindern von beiden Elternteilen zugelassen. Geht man davon aus, dass Personen geschieden sein können, ohne dass dadurch der Anspruch auf das Kind erlischt, ist dies sinnvoll. In diesem Falle haben wir eine N-zu-N Beziehung[1], in der ein Objekt auf verschiedene Objekte verweisen kann, aber auch verschiedene Objekte auf dasselbe Objekt verweisen können.



PERSON	
id	name
1	Lisa Müller
2	Thomas Schmidt
3	Ben Müller
4	Maria Müller
5	Jens Schmidt
..	..

REL_ELTERN_KIND	
Id_eltern	Id_child
1	3
1	4
1	5
2	3
2	4
2	5
..	..

Abbildung 4: Beispiel N-zu-N Beziehung

Zur Darstellung dieser Beziehung ist eine Zwischentabelle notwendig. Anderenfalls müssen wieder N-Spalten zur Sicherung der Fremdschlüssel benutzt werden, so wie bereits bei der Eins-zu-N Beziehung beschrieben wurde. Eine bessere Möglichkeit ist daher das Verwenden einer Relationstabelle. Diese enthält lediglich die Spalten für die Fremdschlüssel auf die beiden zu verknüpfenden Objekte. Jede Relation wird damit durch eine Zeile in der Relationstabelle repräsentiert. Damit ist die Anzahl der Relationen nicht beschränkt und das Datenbankschema bleibt übersichtlich.

Relationstabellen sind nicht zwangsläufig beschränkt auf N-zu-N Beziehungen und könnten auch zur Darstellung jeder anderen Relation verwendet werden. Beim Laden der Relationen ist durch dieses Modell ein zusätzlicher Datenbankaufruf notwendig. Je nach Größe und Komplexität des Datenbankschemas bedeutet das einen Mehraufwand für das DBMS. Daher sollten relationale Tabellen nur für N-zu-N Beziehungen verwendet werden.

---

[1] Der Name N-zu-N Beziehung ist nicht ganz korrekt. Genau genommen handelt es sich um eine M-zu-N Beziehung, da nicht zwangsläufig gleichviele Elemente auf beiden Seiten der Beziehung vorhanden sein müssen.

2.4.4 Repräsentation von Vererbung

Vererbung ist ein grundlegender Bestandteil von objektorientierten Systemen. Dabei erbt ein Objekt Attribute und Funktionen eines anderen, übergeordneten Objekts. Anschaulich dargestellt werden kann die Vererbung am Beispiel von Fahrzeugen. Gehen wir dabei von einem Objekt Fahrzeug aus, welchem die Attribute Bezeichnung und Kraft zugeordnet sind. Erstellt man nun zwei weitere Objekte, PKW und LKW, wobei PKW das Attribut Sitzplätze besitzt und LKW dagegen Zuladung enthalten soll. Beide Objekte benötigen aber auch alle Attribute des Objektes Fahrzeug. Es bietet sich also an, diese als Konkretisierung von Fahrzeug anzusehen. PKW und LKW erben also von der Klasse Fahrzeug.

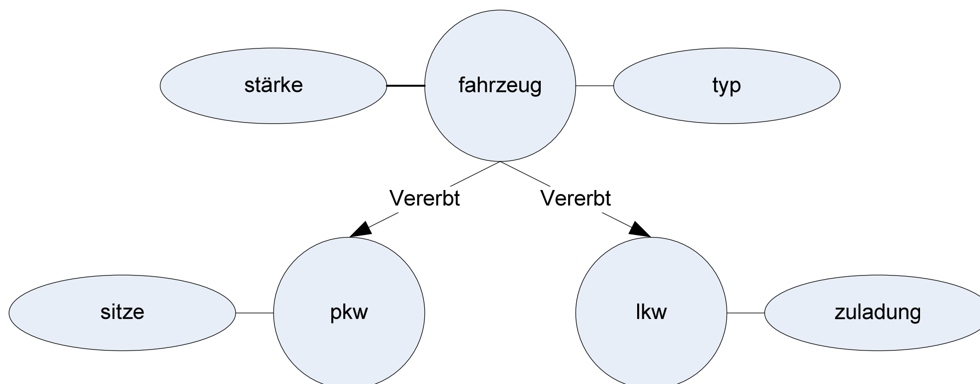


Abbildung 5: Beispiel Objektgraph zur Vererbung

Dies verhindert nicht nur die Redundanz innerhalb des Quellcodes, sonder erlaubt es auch, LKW und PKW gleich zu behandeln, solange man lediglich auf Eigenschaften von Fahrzeug zurückgreift. Dieser gleichzeitige Zugriff auf verschiedene Objekte eines übergeordneten Typs wird als polymorph bezeichnet.

Will man diese Vererbungshierarchie in einer Datenbank ablegen, ergeben sich verschiedene Anforderungen an die Redundanzfreiheit sowie die Performance der Abfrage. Im Folgenden werden daher drei mögliche Datenbankschemata am diskutierten Beispiel erläutert. [11]

2.4.4.1 Tabelle je Klassenhierarchie

Bei diesem Ansatz wird eine Tabelle erstellt, die alle Attribute aller zur Hierarchie gehörenden Objekte enthält. Weiterhin wird eine weitere Spalte benötigt, in der zusätzlich gespeichert wird, zu welcher konkreten Klasse dieses Objekt gehört.

Im Beispielfall bestände die Tabelle damit aus den Spalten Bezeichnung, Kraft, Sitzplätze, Zuladung sowie dem Klassenidentifikator, meistens als discriminator (kurz: „disc“) bezeichnet. Beide Objekttypen würden nun gleichermaßen in diese Tabelle geschrieben werden.

fahrzeug					
id	disc	typ	stärke	sitze	zuladung
1	c	Porsche	488 PS	3	
2	c	Mazda	145 PS	5	
3	t	MAN	620 PS		4,8t
4	c	Fiat	45 PS	4	
5	t	Scana	513 PS		6,2t
..	..	..	..	..	..

Abbildung 6: Beispiel-Datenbankschema für Tabelle je Klassenhierarchie

Dieser Ansatz eignet sich besonders für kleine Hierarchien mit wenigen Attributen oder wenigen nicht gemeinsam genutzten Attributen. Bei größeren Objektbäumen steigt die Anzahl der Tabellenspalten jedoch enorm an, wobei immer nur ein Bruchteil der Felder für ein Objekt benutzt wird. Dadurch erhöhen sich der Speicheraufwand und die Zugriffszeiten auf die Datenbank.

2.4.4.2 Tabelle je konkrete Klasse

Weiterhin besteht die Möglichkeit, für jede konkrete Klasse eine eigene Tabelle zu erstellen. Am Beispiel bedeutet dies, dass sowohl für PKW als auch für LKW eine Tabelle erzeugt wird. In beiden Tabellen finden sich dann die Attribute von Fahrzeug wieder. Die Objekte werden also vollkommen getrennt voneinander gespeichert.

pkw			
id	typ	stärke	sitze
1	Porsche	488 PS	3
2	Mazda	145 PS	5
4	Fiat	45 PS	4
..	..	..	..

lkw			
id	typ	stärke	zuladung
3	MAN	620 PS	4,8t
5	Scana	513 PS	6,2t
..	..	..	..

Abbildung 7: Beispiel -Datenbankschema für Tabelle je konkrete Klasse

Dieses Mapping der Vererbung ist besonders Platz sparend, da es keine überflüssigen Felder anlegt. Der größte Nachteil besteht jedoch in der Redundanz der Spalten, da die gemeinsam verwendeten Attribute in beiden Tabellen vorhanden sind.

2.4.4.3 Tabelle je Klasse

Als Folge der ersten beiden Ansätze ergibt sich das dritte Konzept. Hierbei wird nicht nur für die konkreten Klassen eine Tabelle erstellt, sondern auch für alle abstrakten Objekte. Die Tabellen der konkreten Klassen enthalten dabei lediglich ihre speziellen Attribute, während gemeinsam genutzte Felder in den Tabellen der abstrakten Klassen gespeichert werden.

fahrzeug		
id	typ	stärke
1	Porsche	488 PS
2	Mazda	145 PS
3	MAN	620 PS
4	Fiat	45 PS
5	Scana	513 PS
..	..	..

pkw	
F_id	sitze
1	3
2	5
4	4
..	..

lkw	
F_id	zuladung
3	4,8t
5	6,2t
..	..

Abbildung 8: Beispiel-Datenbankschema für Tabelle je Klasse

Dieser Ansatz vermeidet als einziger Redundanzen und hat ein normalisiertes Datenbankschema. Auch bleibt das Datenbankschema übersichtlich. Jedoch macht es die Verwendung von Fremdschlüsseln notwendig. Dadurch und durch die höhere Anzahl an notwendigen Tabellen weist es aber auch die schlechteste Abfrageperformance auf.

2.5 Persistenz

Persistenz beschreibt in der Informatik den Zustand eines Datensatzes. Dabei bezieht sich die Persistenz auf den eindeutigen und dauerhaften Zustand eines Objektes. Deutlich wird dieser Begriff bei der Erläuterung der Probleme, Persistenz zu gewährleisten. Solange ein Objekt innerhalb einer Applikation verwendet wird, ist Persistenz relativ leicht zu gewährleisten. Jedes Attribut eines Objektes kann zu jeder Zeit geändert werden, das Attribut ist aber zu jedem Zeitpunkt eindeutig definiert. Erstellt man jedoch eine Kopie dieses Objektes und führt Änderungen an diesem Objekt aus, stellt sich die Frage, welches dieser beiden Objekte nun das Richtige bzw. das Aktuellere ist. Natürlich wird normalerweise die Kopie eines Objektes als neues Objekt behandelt.

Betrachtet man nun den Fall einer Datenbank, aus der ein Objekt geladen wird, wobei eine lokale Kopie dieses Objektes erzeugt wird. Diese Kopie kann dann bearbeitet und wieder in der Datenbank gespeichert werden. Aufgrund der Tatsache, dass die Applikation weiß, dass das lokale Objekt aktueller sein muss, wird der Datensatz in der Datenbank einfach mit den neuen Werten überschrieben. Das Persistenzproblem tritt in dem Moment auf, wenn mehrere Benutzer gleichzeitig dieses Objekt bearbeiten. An dieser Stelle ergibt sich zwangsläufig die Frage, welcher Benutzer nun das Recht hat, sein aktualisiertes Objekt in die Datenbank zu schreiben. Persistenz beschreibt also die Aktualität von Datensätzen in Mehrbenutzersystemen.

2.6 Hibernate

Hibernate ist eine für Java entwickelte Open-Source Bibliothek zur Anbindung von relationalen Datenbanken an Java-Applikationen.[15] Die Hauptaufgabe ist das objekt-relationale Mapping. Darüber hinaus bietet Hibernate erweiterte Funktionalitäten wie die Anbindung an verschiedenste Datenbanken. Dabei verwendet Hibernate eine eigene Anfrage-sprache, die sogenannte Hibernate Query Language (HQL), welche je nach konfigurierter Datenbank und dem Datenbank-Dialekt die notwendigen Datenbankanfragen generiert[16]. HQL weist dabei erhebliche Ähnlichkeiten zu SQL auf und unterscheidet sich in der Syntax nur geringfügig. Damit bietet es jedoch die Möglichkeit, die Applikation vollkommen unabhängig von der Datenbank zu entwickeln [11]. Weiterhin besteht die Möglichkeit, unabhängig von einer Datenbanksprache Anfragen mithilfe von Prozeduren zu erstellen. So bietet das Hibernate Criteria Modul die Möglichkeit, Anfragen objektorientiert zu generieren. Ein Beispiel für eine

typische Criteria Anfrage, bei der der Name aller Personen aus der Tabelle Person ermittelt werden sollen, welche 18 Jahre alt sind, hätte dabei die Form:

```
Criteria crit = session.createCriteria(Person.class)
crit.add(Restrictions.eq("age", “18”))
```

Weiterhin können Filter, Cluster sowie Index- und Volltextsuchen definiert werden. Damit ist Hibernate das zurzeit umfangreichste Mapping-Werkzeug für Java[17].

Mithilfe von XML-Mapping Dateien wird die Übersetzung zwischen Datenbankschema und Java-Objekt konfiguriert. Dabei unterstützt Hibernate alle drei Arten der Vererbung (siehe Kapitel 2.4.4) sowie die Umsetzung von Relationen in Java-Collections (siehe Kapitel 2.4.3). Desweiteren automatisiert Hibernate das Cachen von Datenbankzugriffen, die Persistenzverwaltung sowie andere Performanceoptimierungsstrategien[18].

Der generelle Ablauf einer durch Hibernate unterstützen Datenbankanwendung ist dabei immer gleich aufgebaut. Nachdem die Hibernate Konfiguration initialisiert wurde, muss eine SessionFactory erstellt werden. Diese ist das zentrale Modul für Hibernate. Die SessionFactory wiederum erlaubt das Erstellen einer Session. Beim Erstellen wird diese automatisch mit der Datenbank verbunden und erhält eine Verbindung bis zum Schließen der Session aufrecht. Über diese Session können nun Objekte geladen und gespeichert werden. Dabei muss das Laden und Speichern nicht explizit vorgenommen werden, da Hibernate automatisch Änderungen an Objekten in der Datenbank tätigt sowie Objekte automatisch nachlädt, sollten diese von der Applikation angefordert werden und sich noch nicht im Hauptspeicher befinden. Erst mit dem Schließen der Session ist ein Nachladen und Speichern nicht mehr möglich. Dieses Problem ist allgemein als Lazy Loading Exception bekannt und wird im Kapitel 2.6.3 näher behandelt.

2.6.1 Session Konzept

Hibernate verwendet das Session Konzept, um Datenbankverbindungen aufzubauen. Dabei liefert die SessionFactory Instanzen einer Session, welche bei Instanziierung sofort geöffnet werden. Innerhalb dieser Session können nun Datenbankoperationen durchgeführt werden. Jede Datenbankoperation wird in Hibernate als Transaktion bezeichnet und unterstützt dabei die gängigen Transaktionsregeln nach ODMG 3.0 [5]. Es kann immer nur eine Transaktion pro Session zur gleichen Zeit geöffnet sein. Jedoch können innerhalb einer Session beliebig viele Transaktionen durchgeführt werden. Der Ablauf ist dabei immer das Erstellen der Transaktion, das Sichern oder Laden von Objekten und das Schließen oder Abbrechen der Transaktion. Das Schließen der Transaktion sorgt dafür, dass die Transaktion in die Datenbank geschrieben wird, wohingegen beim Abbrechen alle Änderungen rückgängig gemacht werden.

2.6.2 Hibernate Cache

Hibernate unterstützt verschiedene Techniken zum Optimieren von Datenbankzugriffen. Zum einen das Caching, bei dem Hibernate einen Teil der Datenbank lokal abbildet, um mehrfache Zugriffe auf die gleichen Objekte zu optimieren. Dabei versucht Hibernate zunächst, die angeforderten Objekte im Cache zu finden. Bleibt dies erfolglos, lädt Hibernate die Objekte aus der Datenbank. Andererseits dient der Cache aber auch zum Schreiben in die Datenbank, wobei erst der Cache beschrieben wird, und bei Beendigung des Schreibvorgangs diese Einträge gebündelt in die Datenbank übertragen werden.

Dadurch kann ein Objekt bereits gelesen werden, bevor es reell in die Datenbank geschrieben wurde. Dies kann mitunter für Verwirrungen sorgen. Vermieden werden kann dieses Problem, wenn am Ende jedes Schreibzyklus Hibernate veranlasst wird, den Cache mit der Datenbank zu synchronisieren. Andererseits erlaubt es dem Entwickler aber, Änderungen im Cache vorzunehmen, diese Änderungen zu begutachten und sie bei Bedarf wieder rückgängig zu machen.

Ein Cache wird für jede Session erstellt und beim Schließen der Session gelöscht. Damit ist ein Zugriff auf gecachte Objekte nur möglich, solange die dazugehörige Instanz der Session vorhanden ist.

2.6.3 Hibernate Lazy Loading

Ein weiteres Optimierungsmerkmal von Hibernate ist das sogenannte Lazy Loading. Es ermöglicht Hibernate, anstatt das komplette Objekt mit allen Attributen und referenzierten Objekten zu laden, diese Elemente zwar zu instanziieren, jedoch durch einen Proxy zu ersetzten. Greift die Applikation nun auf ein Element zu, welches durch einen Proxy repräsentiert wird, greift Hibernate auf die Datenbank zu und lädt die angeforderte Information.

Grundlage dieses Prozesses ist eine Datenbankverbindung sowie eine offene Session. Anderenfalls erzeugt Hibernate eine Lazy Loading Exception. Daher wird häufig bei Verwendung von Lazy Loading eine Session erstellt und über die gesamte Lebenszeit der Applikation aufrecht erhalten.

2.6.4 Persistenz in Hibernate

Hibernate unterstützt zudem die Persistenzverwaltung. Damit muss der Entwickler nicht selber auf den Zustand der Objekte achten. Dazu verwendet Hibernate ein Zustandsmodell, welches von der Session verwaltet wird. Ein erzeugtes Objekt erhält dabei beim Erzeugen den Zustand „transient“, was bedeutet, dass ein Objekt auf der Anwendungsseite existiert ohne eine entsprechende Repräsentation in der Datenbank. In dem Augenblick, wo es der Session zum Speichern übergeben wird, bekommt es von Hibernate eine eindeutige ID und geht damit in den Zustand der „Persistenz“ über. Das Objekt muss dabei noch nicht in die Datenbank geschrieben sein. Bis jetzt wurde es lediglich in den Hibernate Cache aufgenommen und wird mit dem nächsten Session.flush() Befehl in die Datenbank geschrieben.

Eine weitere Möglichkeit, ein transientes Objekt zu persistieren besteht darin, eine Referenz von einem persistenten Objekt auf das transiente zu erzeugen. Um den persistenten Zustand des Objektes zu gewährleisten, ist damit ein persistieren des referenzierten Objektes notwendig und wird von Hibernate automatisch durchgeführt. Beim Laden des Objektes erhält es sofort den Zustand persistent, da es zum Zeitpunkt des Ladens mit der Datenbank synchron ist.

Ein Objekt kann nur dann in den transienten Zustand zurückkehren, wenn es innerhalb einer Session gelöscht wird, da es in der Datenbank noch vorhanden ist. Das Objekt existiert dabei noch im Hauptspeicher, bis es durch Auflösung der Referenzierungen auch dort entfernt wird, wohingegen in der Datenbank das Objekt bereits entfernt wurde.

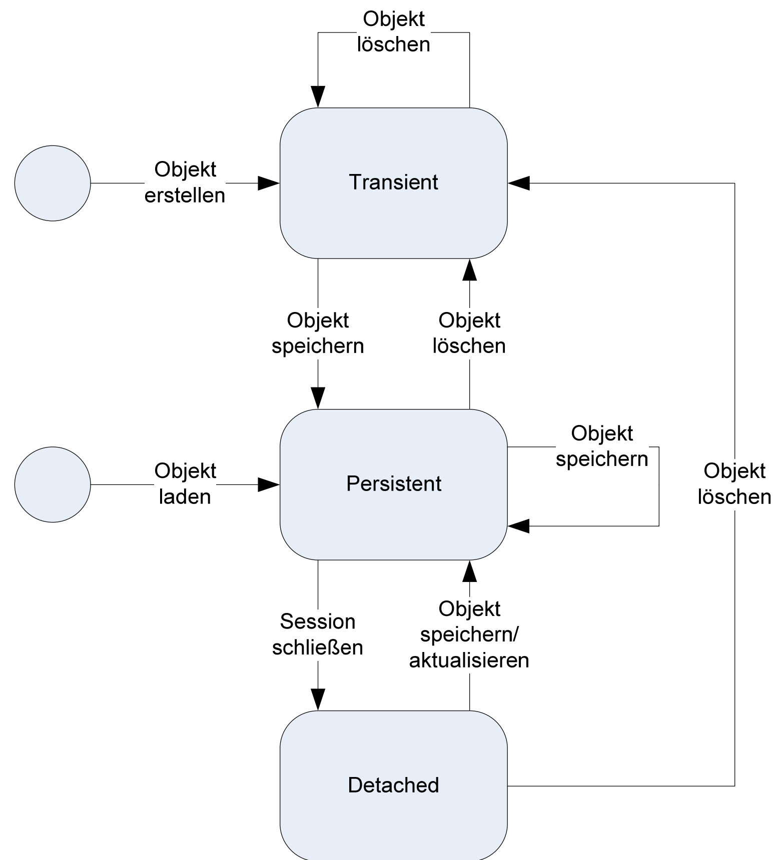


Abbildung 9: Zustandsmodell persistenter Objekte [11]

In dem Moment, in dem eine Session geschlossen wird, gehen alle mit der Session assoziierten Objekte in den Zustand „Detached“ über. In diesem Zustand kann nicht sichergestellt werden, dass Änderungen am Objekt in der Datenbank gespeichert werden bzw. Änderungen in der Datenbank an das Objekt übergeben werden. Erst durch erneutes Speichern oder Laden des Objektes wird der persistente Zustand wieder hergestellt. Dabei ist es egal, ob inzwischen eine andere Session geöffnet ist, da Hibernate Persistenz nur für Objekte gewährleistet, die durch eine Session mit der Datenbank verbunden sind. [11]

2.7 Notationen

2.7.1 Objektgraph

Objektgraphen beinhaltet Knoten für ein Objekt oder Attribut und Kanten für eine Relation zwischen einem Objekt und einem Attribut oder weiterem Objekt. Die Unterscheidung von Objekten und Attributen ist für die weiteren Ausführungen in dieser Arbeit nicht entscheidend, da der einzige Unterschied in der Fähigkeit liegt, eine weitere Relation zu beinhalten. So kann ein Objekt eine Relation zu weiteren Knoten haben, während ein Attribut ein Blatt darstellt. Zur Vereinheitlichung in der Arbeit und Vereinfachung der Beschreibung wird im Folgenden von einem Objektgraphen gesprochen.

Ein Teilgraph stellt nun eine Untermenge des Objektgraphen dar. Dabei kann ein einzelner Knoten als auch der gesamte Graph eine Teilmenge des Graphen darstellen. Daher wird im Zusammenhang mit dem Laden von Daten aus einer Datenbank vom Laden eines Teilgraphen oder -baumes gesprochen, da meistens beim Laden ein Pfad durch den Graphen und kein einzelner Knoten geladen wird.

Das root-Element stellt dabei den Ausgangsknoten innerhalb des Teilgraphen dar. Es handelt sich als um die oberste Ebene innerhalb des (Teil-)graphen, wobei das root-Element lediglich ein Element dieser Ebene ist.

2.7.2 Anfrage

Eine Anfrage oder auch Datenbankanfrage bezeichnet im Folgenden ein einziges Query, welches an die Datenbank gesendet wird. Dies ist sehr gut mit dem Verschicken eines einzigen SQL-Befehles an die Datenbank zu vergleichen, wobei sich eine Anfrage nicht zwangsläufig auf SQL beziehen muss.

2.7.3 Abfrage

Anders als bei Datenbankabfragen bezeichnet eine Abfrage im Folgenden das Besuchen von Knoten im Graphen. Dabei kann eine Abfrage aus mehr als einer Anfrage bestehen. Abfragen beziehen sich also in diesem Kontext auf Objektabfragen, wobei auch mehr als ein Knoten besucht werden kann, solange es sich um dasselbe Root-Element handelt.

3 Fetching-Methoden im Überblick

Fetching beschreibt allgemein das Verfahren, bei dem eine Applikation Daten aus einer Datenbank abruft. Fetching ist dabei eine mögliche Richtung eines Prozesses, der zwischen dem Objekt und der Datenbank stattfindet. Dabei wird das Fetching von der Applikation mit Aufruf eines oder mehrerer Objekte ausgelöst. Das Fetching stellt eine Anfrage an das Datenbanksystem, welche dann einen oder mehrere Datensätze sendet. Diese Daten werden dann mithilfe des Mappings für die Applikation in Objekte übersetzt. [2][4]

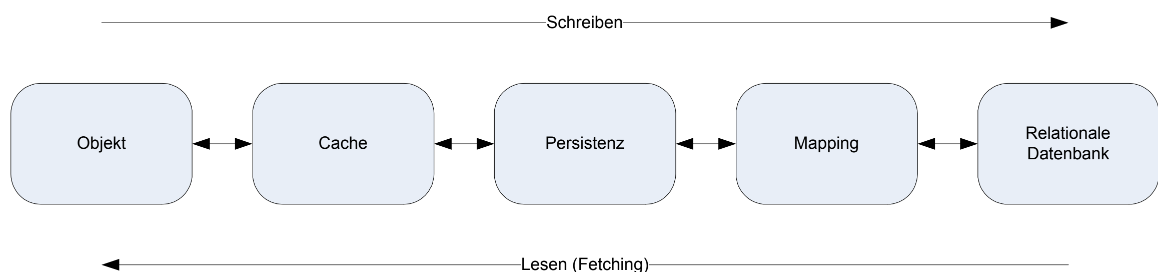


Abbildung 10: Fetching-Prozess

Der Cache und die Persistenz [11][19] sind dabei mögliche zusätzliche Schritte, die diesen Ablauf erweitern können. Der Cache repräsentiert dabei eine lokale Kopie eines Teils der Datenbank in Objektstruktur. Er dient dem Zwischenspeichern von geladenen Knoten. Findet beispielsweise ein Zugriff auf einen bereits vorher geladenen Knoten statt, wird dieser nicht erneut aus der Datenbank sondern aus dem Cache geladen.

Dadurch werden die Zugriffe auf die Datenbank verringert, was jedoch auch die Gefahr birgt, dass Änderungen in der Datenbank nicht automatisch auch im Cache stattfinden. Um den Cache und die Datenbank synchron, also persistent, zu halten, wird ein sogenannter Persistenz-Layer benötigt. Dieser sorgt nicht nur für den persistenten Zustand des Caches, sondern stellt auch sicher, dass Änderungen, die zur selben Zeit von anderen Benutzern durchgeführt wurden, überschrieben, aktualisiert oder beibehalten werden. Hierfür existieren verschiedene Strategien, wie zum Beispiel das Setzen von Markierungen, sogenannte Flaggs, die das entsprechende Objekt für andere Benutzer sperren, während es bearbeitet wird [20]. Sollte das Objekt jedoch nicht für andere Benutzer gesperrt worden sein, so dass eine Änderung in der Datenbank

möglich ist, während das Objekt bereits geladen wurde, wird diese Änderung durch den Persistenz-Layer nachvollzogen und die lokale Kopie aktualisiert [21].

3.1 Anforderungen an Fetchingverfahren

Das Fetching bietet verschiedene Möglichkeiten zum Optimieren des Prozesses. Dabei geht es hauptsächlich um zwei Aspekte. Einerseits die Verbesserung der Datenbankgeschwindigkeit und andererseits die Geschwindigkeit der Applikation. Beide Aspekte lassen sich teilweise auf die Anzahl der Datenbankabfragen zurückführen.

Um die Datenbank zu entlasten und damit die Geschwindigkeit zu erhöhen, sollten viele Einzelabfragen an die Datenbank vermieden werden. In einem Szenario, bei dem viele Applikation viele Datenbankzugriffe verursachen, erhöht sich die Antwortzeit der Datenbank bis hin zur Überlastung derselben. Um dies zu verhindern ist es notwendig, Anfragen so zu formulieren, dass möglichst viele Einzelanfragen zu wenigen zusammengefasst werden können.

Der zweite Aspekt, die Performance der Anwendung, ist selbstverständlich ebenfalls von der Geschwindigkeit der Datenbank abhängig. Jedoch unabhängig von der Länge der Wartezeit einer Datenbankabfrage verursacht jeder Zugriff auch eine Wartezeit in der Applikation. Zur Verminderung von Wartezeiten innerhalb der Applikation dient zum einen der Cache, zum anderen das Vorausladen von Teilgraphen, welche von der Applikation später gebraucht werden.

Ein Vorausladen von Teilgraphen erfordert zwangsläufig ein Regelwerk, welches angibt, welche Knoten geladen werden sollen. Dies wiederum birgt die Gefahr, dass Knoten geladen werden, die von der Applikation nicht benötigt werden, sollte die Regel zu allgemein oder ungenau formuliert sein. In diesem Falle erhöht sich das Datenvolumen, welches von der Datenbank zur Applikation übertragen werden muss, ohne dass dieses von der Applikation verwendet wird.

Ein Fetching-Verfahren kann also als optimal angesehen werden, wenn es keine überflüssigen Knoten von der Datenbank lädt und die kleinstmögliche Zahl von Datenbankabfragen zum Laden dieser Knoten verwendet.

3.2 Vergleichsgrundlagen

Zur Veranschaulichung aller Verfahren in diesem Kapitel verwenden wir ein einfaches Objektmodell, in dem möglichst unterschiedliche Relationen enthalten sind. Als Beispiel dient ein vereinfachtes Modell einer Universität.

Dabei existieren die Objekte Professor (p) und Student (s), wobei beide vom abstrakten Objekt Person abgeleitet sind, sowie Vorlesungen (v) und Fakultäten (f). Professoren und Studenten können dabei auf jeweils eine Fakultät verweisen. Weiterhin besitzen Professoren Vorlesungen. Dabei kann jeweils nur ein Professor eine Vorlesung halten. Vorlesungen beinhalten dann wiederum beliebig viele Studenten, wobei ein Student mehrere Vorlesungen besuchen kann. Mithilfe dieses Modells werden alle Arten von Relationen abgedeckt.

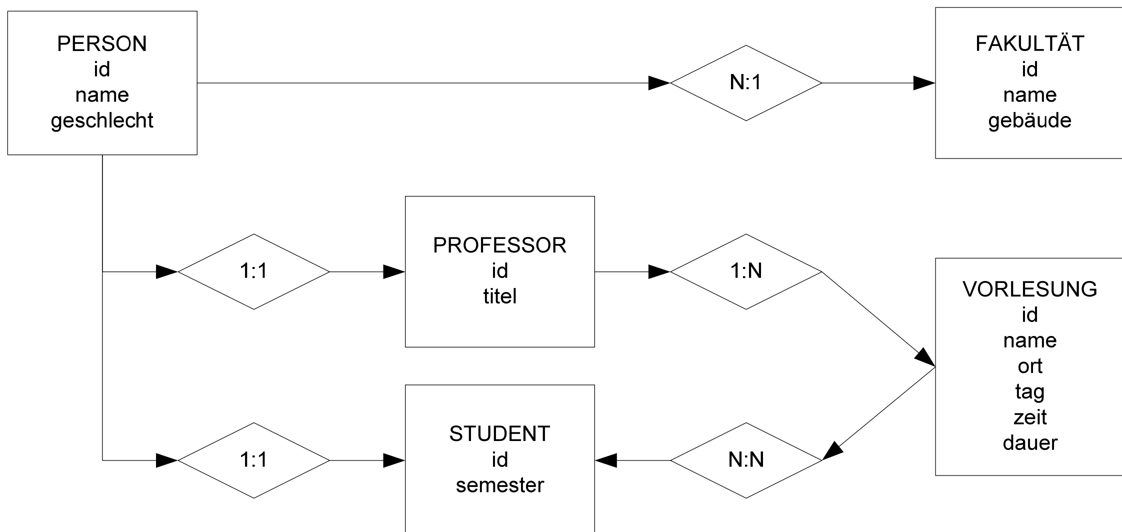


Abbildung 11: Datenbankschema des Beispielgraphen

Um die Performance der vorgestellten Ansätze zu vergleichen und die Vorzüge beziehungsweise Anwendungsgebiete zu veranschaulichen, wird im Weiteren eine Reihe von Bedingungen definiert, an denen die Algorithmen gemessen werden. Unabhängig von der Tatsache, dass diese Abfragen über ein einziges Query ermittelt werden könnten, gehen wir von einer Objektstruktur aus, in welcher gesucht wird. Dies bedeutet, dass entlang eines Objektgraphen alle Knoten besucht werden müssen, um zum gewünschten Knoten zu gelangen.

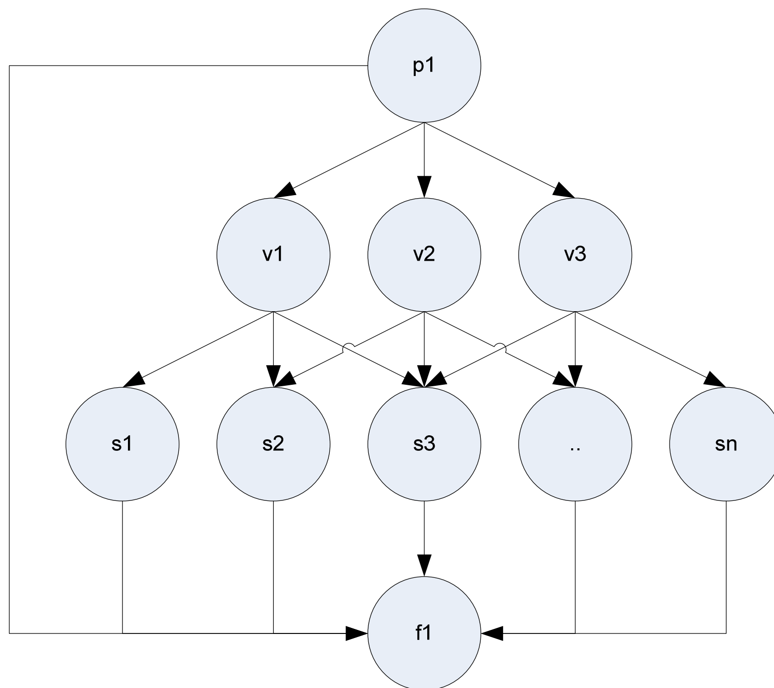


Abbildung 12: Objektschema des Beispielgraphen

Zur besseren Vergleichbarkeit der verschiedenen Verfahren wird eine feste Anzahl von Objekten in der Datenbank verwendet. Dadurch ist es möglich, quantitative Aussagen über das Verhalten des Verfahrens zu machen. Dabei gehen wir von einer Befüllung des beschriebenen Datenbankschemas mit 30 Professoren aus. Auf diese Professoren werden gleichmäßig 90 Vorlesungen verteilt, welche wiederum gleichmäßig von 1800 Studenten besucht werden. Professoren und Studenten können dabei 10 verschiedenen Fakultäten angehören.

Daraus ergeben sich entsprechende quantitative Angaben der mehrelementigen Relationen. Jeder Professor hat demnach 3 Vorlesungen mit der Einschränkung, dass ein Student insgesamt nur 3 Vorlesungen besuchen kann. Daraus ergibt sich eine Anzahl von 90 Studenten pro Vorlesung.

Um nun eine Vergleichbarkeit herzustellen, wird für jedes Verfahren eine Reihe von Abfragen durchgespielt, welche möglichst den normalen Gebrauch einer Datenbank wiederspiegeln. Eine typische Abfrage ist das Ermitteln der Namen aller Studenten, die bei einem bestimmten Professor eine Vorlesung besuchen. Diese Abfrage wird im Folgenden als Abfrage A bezeichnet.

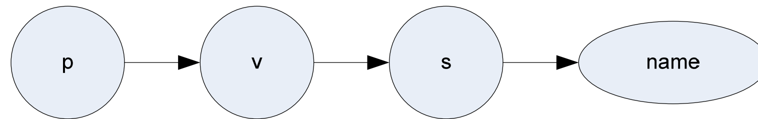


Abbildung 13: Objektpfad für Abfrage A

Abfrage A: Namen aller Studenten eines Professors

Eine weitere Abfrage mit höherer Tiefe ist das Durchsuchen der Datenbank nach Fakultäten von Studenten, die Vorlesungen eines Professors besuchen. Zu jeder Fakultät sollen sowohl das Gebäude als auch der Name geladen werden. Diese Abfrage wird im Folgenden als Abfrage B bezeichnet werden.

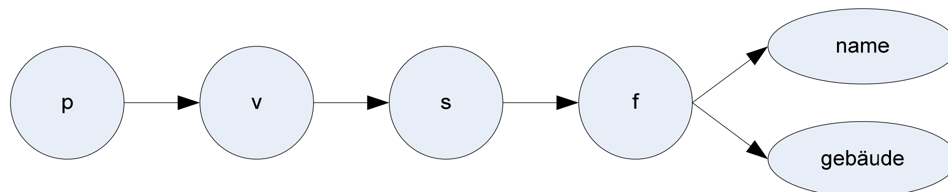


Abbildung 14: Objektpfad für Abfrage B

Abfrage B: Name und Gebäude aller Fakultäten von Studenten eines Professors

Als Letztes werden als Beispiel einer selektiven Abfrage zusätzlich zu den Bedingungen von Abfrage B nur jeweils die erste Vorlesung eines Professors sowie der erste Student einer Vorlesung gesucht. Diese Abfrage wird als Abfrage C bezeichnet.

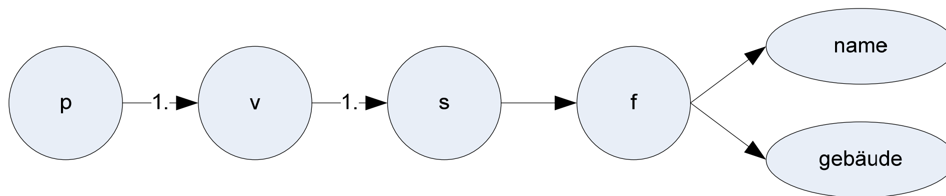


Abbildung 15: Objektpfad für Abfrage C

Abfrage C: Name und Gebäude aller Fakultäten des ersten Studenten der ersten Vorlesung eines Professors

Die Performance des Verfahrens ergibt sich durch das Zusammenspiel aus der Anzahl der Anfragen an die Datenbank für eine der Beispielanfragen, sowie der übertragenen Datenmenge. Zur Vereinfachung dieser Betrachtung beziffern wir die Dauer einer Anfrage mit einer Zeiteinheit. Dies ist die Zeit, die benötigt wird, um die Anfrage zu senden und die Antwort der Datenbank abzuwarten. Damit ist die Anzahl der Anfragen gleichzusetzen mit der Dauer der Abfrage.

Weiterhin wird das Laden jedes Knotens des zu übertragenden Teilgraphen mit einer Dateneinheit bestimmt. Daraus lassen sich Rückschlüsse auf die Netzwerkbelastung sowie die notwendige Größe des Caches schließen. Damit wird das Objekt selber ebenfalls mit einer Dateneinheit bestimmt. Da das Objekt anhand seiner ID identifiziert wird, ist das Attribut ID bereits im Laden des Objektes enthalten und muss nicht zusätzlich betrachtet werden.

3.3 Fetchingverfahren

3.3.1 On-demand fetching

On-demand Fetching ist die einfachste Möglichkeit, Daten aus einer Datenbank zu lesen. Bei diesem Verfahren werden Knoten in dem Moment abgefragt, in dem sie von der Applikation benötigt werden. Dadurch werden keine überflüssigen Daten in den Cache geschrieben. Dies verringert die Anforderungen an das Client-System, erhöht aber gleichzeitig die Anzahl der Zugriffe.

Im Spezialfall von Hibernate wird on-demand Fetching als Proxy umgesetzt. Beim Laden eines Objektes wird dieses zwar instanziiert, die Knoten werden jedoch nicht mit den notwendigen Informationen gefüllt, sondern ein Proxy referenziert auf das entsprechende Feld. Wird nun der Proxy aufgerufen, lädt Hibernate die Information aus der Datenbank.

Der charakteristische Ablauf des on-demand Fetching wird am Beispiel der Abfrage A deutlich. Hier wird jeder Knoten einzeln abgefragt. Dadurch erhält man schon bei diesem einfachen Beispiel folgende Zugriffszahlen:

	Abfrage A		Abfrage B		Abfrage C	
	Anfragen	Daten	Anfragen	Daten	Anfragen	Daten
Professoren	1	1	1	1	1	1
Vorlesungen	3	3	3	3	1	1
Studenten	180 + 180	180 + 180	180	180	1	1
Fakultäten	0		10 + 20	10 + 20	1 + 2	1+2
Summe	364	364	214	214	6	6

Tabelle 1: Ressourcenbetrachtung für on-demand Fetching

On-demand Fetching bietet das geringste Datenvolumen, da nur Knoten geladen werden, die auch gebraucht werden und die höchste Anzahl an Abfragen. Daher stellt es die Vergleichswerte für alle anderen Verfahren dar.

3.3.2 Prefetching

Beim Prefetching verwendet man gegenüber on-demand Fetching Heuristiken, die ermitteln, welche Teilbäume bei einem Datenbankabruf im Voraus geladen werden sollen. Diese Daten werden dann im Cache abgelegt, so dass bei einem späteren Aufruf eines Knotens aus diesem Teilbaum kein Datenbankzugriff mehr notwendig ist.

Vorteil dieses Verfahrens ist die Verminderung der Anzahl von Datenbankzugriffen durch das Vorwegladen von Informationen. Jedoch kann es dabei auch zu Performance-Einbußen kommen, wenn zu viele oder falsche Knoten geladen werden. Falsche Knoten sind in diesem Kontext Daten, die nicht von der Applikation gebraucht werden.

Die verschiedenen Verfahren zur Vorhersage von Knoten, die vorgeladen werden sollen, werden nach der Art ihrer Heuristik klassifiziert (nach Han et al. [22]):

3.3.2.1 Manually specified Prefetching

Bei dieser Prefetching Methode gibt der Entwickler sogenannte Preload Pattern an. Beim Laden eines Knotens wird mithilfe dieser Pattern bestimmt, welche Teilgraphen zusätzlich zu diesem Knoten gebraucht werden.

Dieser Ansatz bietet das größtmögliche Optimierungspotenzial, da der Entwickler das Pattern manuell optimieren kann, bis er mit der Performance zufrieden ist. Auch kann er auf Grund seines Wissens über die Programmzustände und Anforderungen das Pattern so weit optimieren, dass keine nicht benötigten Informationen mit geladen werden. Da bei diesem Ansatz keine Regeln für die Verwendung eines Pattern aufgestellt werden müssen, kann der Entwickler für jeden Fall ein spezielles Pattern erstellen.

Andererseits erfordert es aber auch, dass die Entwickler diese Pattern einrichten. Je nach Größe des Systems, der Datenbank und des Entwicklerteams kann die Erstellung von optimalen Preload Pattern zu einem großen Problem werden. Nämlich dann, wenn der Entwickler die Anforderungen des aufrufenden Services nicht kennt, oder der Entwickler für den Service das richtige Preload Pattern für seinen speziellen Fall nicht kennt oder findet.

Daher ist bei größeren Projekten ein gutes Pattern Management notwendig, um die Güte des Systems sicherzustellen. Dies wiederum erfordert einen gewissen Aufwand und zusätzliche Mittel, welche nicht immer zur Verfügung stehen. Daher kann es sehr schnell zu wenigen

allgemeinen Pattern kommen, welche neben dem benötigten Teilbaum auch zusätzliche, nicht benötigte Knoten enthalten.

Im optimal konfigurierten Fall bietet das manuelle Prefetching die Möglichkeit, mit denselben Datenmengen wie das on-demand Fetching auszukommen, jedoch mit den wesentlichen Einsparungen bei den Abfragen, vergleichbar mit dem page-based Prefetching (siehe Kapitel 3.3.2.3).

Verfahren nach Jürgen Kohl

Manuelles Prefetching wurde von Jürgen Kohl [23] im Java-Magazin dargestellt. Zur Vereinfachung wird das Preload Pattern von Jürgen Kohl mit PreloadPattern abgekürzt. Der Algorithmus besteht aus einer zentralen *preload* Methode, welcher man sowohl ein Objekt als auch eine Reihe von Prefetching-Vorschriften übergibt. Als Preload-Vorschrift wird die Klasse des Parent-Knoten übergeben, sowie die Angabe des Relationsnamens. Für das übergebene Objekt wird anschließend überprüft, ob eine Preload-Regel auf dieses Objekt anwendbar ist. Sollte eine Regel mit der passenden Klasse gefunden werden, wird der relationierte Knoten geladen und *preload* für diesen Knoten neu aufgerufen.

Deutlich wird dies am Beispiel der Uni. Hierbei wird als Startobjekt ein Professor übergeben. Zusätzlich dazu übergeben wird die Regel, dass, wenn ein Professor geladen wird, die Relation Fakultät mit geladen werden soll. Auch soll die Relation Vorlesungen für einen Professor geladen werden. Weiterhin gilt für die Klasse Vorlesung die Regel, dass Studenten geladen werden sollen.

Der Algorithmus ruft sich dabei rekursiv solange auf, bis keine Preload-Anweisung mehr auf die geladenen Objekte anwendbar ist. Damit es bei zyklischen oder bidirektionalen Relationen nicht zu Endlosschleifen kommt, überprüft der Algorithmus vor der Anwendung der Preload-Regeln, ob das übergebene Objekt bereits initialisiert wurde. Ist dies der Fall, werden keine Preload-Regeln auf das Objekt angewendet.

Großer Nachteil dieses Verfahrens ist die Tatsache, dass jeder Knoten besucht werden muss. Gerade bei großen Teilbäumen kann dieser Vorgang sehr lange dauern. Zusätzlich verursacht diese Methode für jeden Knoten eine neue Datenbankabfrage. Dies bedeutet, dass die Anzahl der Datenbankabfragen gleich der Anzahl der Knoten ist. Damit stellt es keine direkte Verbesserung zu on-demand Fetching dar, da auch hier die Anzahl der Datenbankabfragen

identisch zur Anzahl der Knoten ist. Der Vorteil bei diesem Verfahren gegenüber dem on-demand Fetching liegt in dem gleichzeitigen Laden aller benötigten Attribute. Damit ist beim Arbeiten mit den Daten kein Nachladen mehr notwendig. Hierbei gilt es abzuschätzen, ob ein kontinuierliches Nachladen und damit verbundene Verzögerungen einer langen einmaligen Verzögerung vorzuziehen sind oder nicht.

	Abfrage A		Abfrage B		Abfrage C	
	Anfragen	Daten	Anfragen	Daten	Anfragen	Daten
Professoren	1	1	1	1	1	1
Vorlesungen	3	3	3	3	1	1
Studenten	180 + 180	180 + 180	180	180	1	1
Fakultäten	0		10 + 20	10 + 20	1 + 2	1+2
Summe	364	364	214	214	6	6

Tabelle 2: Ressourcenbetrachtung für Verfahren von Jürgen Kohl

CriteriaJoiner nach Benjamin Winterberg

Ein weiteres manuelles Verfahren wurde von Benjamin Winterberg unter dem Namen CriteriaJoiner[24] vorgestellt. Hierbei wird auf Grund der übergebenen Pfade durch den Objektgraphen eine Datenbankabfrage generiert. Dabei verwendet der Algorithmus zum Verknüpfen der Relationen JOIN-Statements. Anschließend werden alle Daten mit einer einzigen Abfrage aus der Datenbank geladen. Damit bietet es enorme Performance-Gewinne gegenüber dem Verfahren von Jürgen Kohl, bei dem jeder Knoten des Objektgraphen einzeln initialisiert wird.

	Abfrage A		Abfrage B		Abfrage C	
	Anfragen	Daten	Anfragen	Daten	Anfragen	Daten
Professoren	1	1	1	1	1	1
Vorlesungen	o	3	0	3	0	3
Studenten	0	180 + 180	0	180	0	180
Fakultäten	0		0	10 + 20	0	10+20
Summe	1	364	1	214	1	214

Tabelle 3: Ressourcenbewertung für CriteriaJoiner

In der Anwendung unterscheiden sich beide Verfahren durch die Art und zeitliche Reihenfolge des Aufrufes. Beim Preload Pattern von Jürgen Kohl wird der Preload-Methode ein oder mehrere Objekte übergeben, und der Algorithmus lädt dann für diese Objekte die definierten Relationen nach. Dagegen wird beim CriteriaJoiner zuerst die Abfrage ohne Objekt erstellt. Will man diese Abfrage auf bestimmte Objekte begrenzen, muss anschließend diese Abfrage um weitere Kriterien erweitert werden. Erst zum Schluss werden diese Objekte aus der Datenbank geladen. Damit vermeidet man das Laden der root-Elemente.

3.3.2.2 User-hint-based Prefetching

Das User-hint-based Prefetching bildet ein Verfahren, dass vom manuellen Prefetching abgeleitet ist. Dabei gibt nicht der Entwickler die Preload Pattern an, sondern der Benutzer. Dies kann auf unterschiedliche Arten geschehen. Zum Beispiel durch Auswahl einer Rolle oder eines Arbeitsschrittes innerhalb der Applikation oder durch konkretes Nennen der Knoten, die vom Benutzer benötigt werden.

Hierbei wird dem Benutzer nicht nur sehr viel Vertrauen gegenüber der richtigen Benutzung des Systems entgegengebracht. Es wird ihm auch eine große Verantwortung auferlegt. So muss der Benutzer Entscheidungen über Prozesse treffen, die er vielleicht gar nicht genau kennt. Daher wird in der Regel dieser Ansatz vermieden, wenn sich die Applikation nicht an eine Benutzergruppe richtet, die auf das Auswählen von Teilbäumen nicht verzichten kann. Auch widerspricht dieses Verfahren damit dem zunehmenden Trend der selbstverbessernden Datenbanksysteme.

E.E. Chang und R.H. Katz [25] verwendeten Hinweise der Benutzer zum Beginn eines Arbeitsprozesses, um das Prefetching zu beeinflussen. Dabei wurde die Gewichtung Tiefe und Breite je nach Userwahl verändert. Im Falle dieses Verfahrens handelte es sich bei der Benutzergruppe um CAD-Benutzer. Aufgrund der Komplexität dieser Programme kann von einem geschulten Personal ausgegangen werden, welches in der Lage sein könnte, die benötigten Hinweise zur Verbesserung der Performance zu liefern.

Jedoch ist diese Situation nicht der Normalfall, so dass ein Performancegewinn nicht in jedem Fall als gegeben angesehen werden kann. Daher ist auch keine exakte Performanceanalyse des Verfahrens möglich.

3.3.2.3 Page-based Prefetching

Beim Page-based Prefetching wird bei Abfrage eines Knotens ein Cluster von Knoten gleichen Typs mitgeladen. Da Cluster jedoch hauptsächlich in OODB's verwendet werden, findet dieses Verfahren nur selten Eingang in ORDB's. Eine Umsetzung für relationale Datenbanken besteht durch das Laden aller verknüpften Knoten. Je nach Größe der Datenbank muss eine große Menge an Daten übertragen werden. Bei Anwendungen, die fast immer alle Knoten benötigen, kann dies jedoch ein sinnvoller Ansatz sein. In allen anderen Fällen führt es zu einer Mehrbelastung des Netzwerkes und des Caches.

Da das Verfahren für OODB's erstellt wurde, kann keine spezielle Implementierung erläutert und in die Ressourcenbetrachtung aufgenommen werden. Lediglich der eben vergleichbare Ansatz für ORDB's könnte für die Betrachtung herangezogen werden, wobei die Performance des Verfahrens unter dem der eigentlichen Page-Based Verfahren liegt. Eine Betrachtung mit dem angesprochenen Vergleichsverfahren würde also keine aussagekräftigen Ergebnisse liefern [26][27].

3.3.2.4 Object-level/page level Access pattern-based Prefetching

Bei dem Ansatz des Access pattern-based Prefetching geht man davon aus, dass Datenbankzugriffe für einen Knoten beispielhaft für alle weiteren Knoten des gleichen Typs sind. Ausgehend von diesem Ansatz kann also für jeden Knoten als root-Knoten ein Pattern aufgrund der Zugriffsgewohnheiten erstellt werden. Dieser Ansatz ist besonders geeignet, wenn man von vielen Knoten immer denselben Teilgraphen benötigt.

PrefetchGuide von Wook-Shin Han at. Al.

Ein Vertreter dieses Ansatzes ist der PrefetchGuide von Wook-Shin Han et. Al. [22][28] Hierbei werden während einer Abfrage die besuchten Knoten gespeichert. Dieses gespeicherte Abfrageprotokoll wird als access-log set bezeichnet. Wird dann ein weiterer Knoten aufgerufen, überprüft der PrefetchGuide, ob bereits ein Knoten dieses Typs geladen wurde. Ist dies der Fall, werden die Elemente vorausgeladen, welche beim letzen Aufruf eines Knoten dieses Typs verwendet wurden. Dabei unterscheidet der Algorithmus zwischen zwei Arten von Access-Log Sets. Iterative Pattern für einfache Graphen und rekursive Pattern für Graphen mit bidirektionalen oder zyklischen Strukturen.

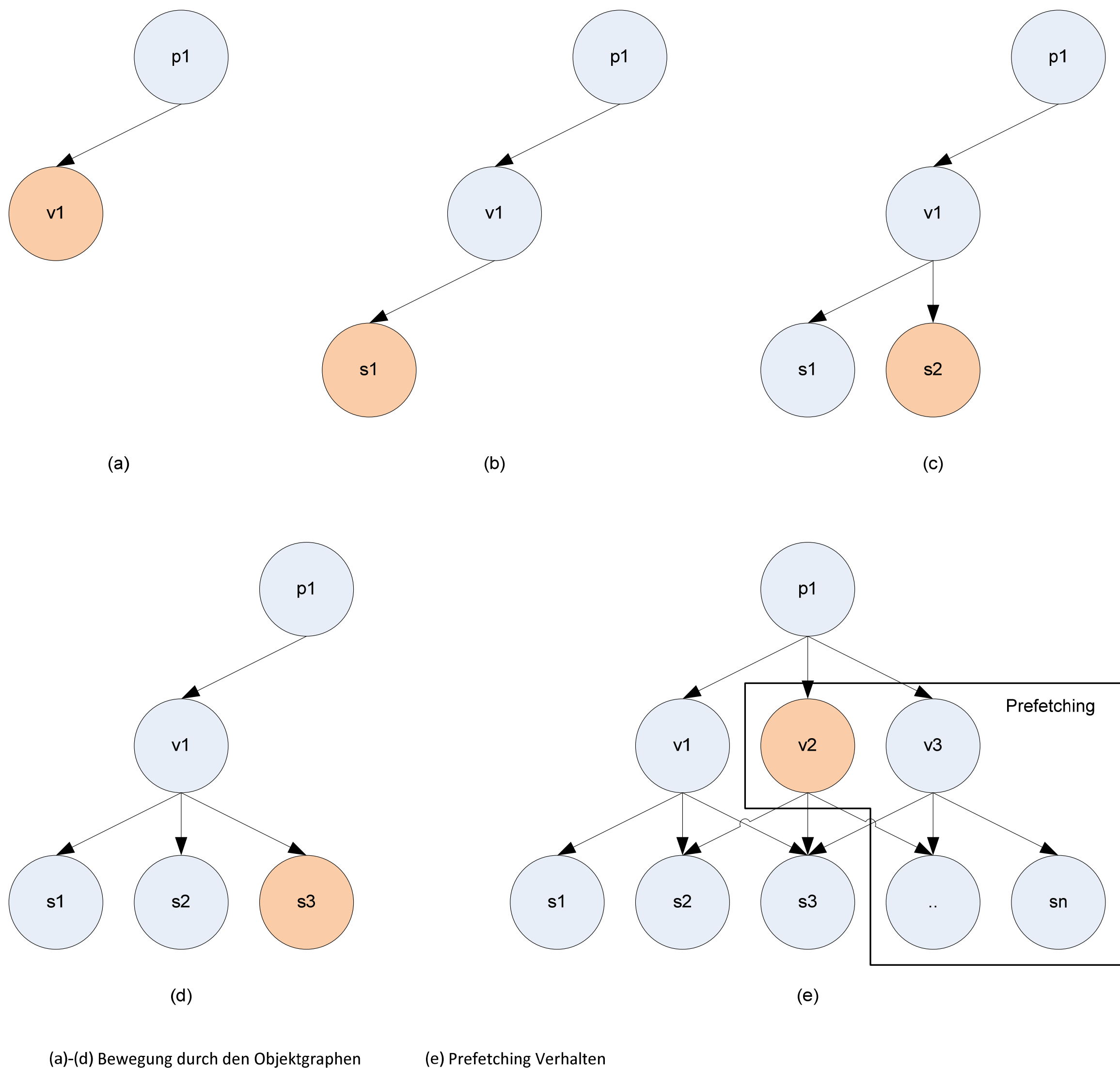


Abbildung 16: Funktionsweise des PrefetchGuide

Zum Verdeutlichen wird eine Beispielabfrage verwendet, in der Professor p1 als root-Element verwendet wird. Ausgehend von diesem Objekt werden nun alle Vorlesungen (v1 – v3) mit allen Studenten (s1 – sn) besucht. Dabei wird zunächst die erste Vorlesung besucht und anschließend jeder Student der Vorlesung. Wird nun die nächste Vorlesung geladen, erkennt der PrefetchGuide das Muster, welches durch den Aufruf der ersten Vorlesung erstellt wurde und lädt automatisch für alle weiteren Vorlesungen die referenzierten Studenten mit.

Mit diesem Verfahren lassen sich laut Aussage der Entwickler 3,5-mal schnellere Zugriffszeiten als beim on-demand Fetching erzielen. Dieser Ansatz ist besonders gut geeignet für Anwendungen, die viele Datensätze mit denselben Anforderungen benötigen. Bei einer

Anwendung, die für jedes Objekt desselben Datentypen verschiedene Operationen ausführt, verliert man den Prefetch-Vorteil.

	Abfrage A		Abfrage B		Abfrage C	
	Anfragen	Daten	Anfragen	Daten	Anfragen	Daten
Professoren	1	1	1	1	1	1
Vorlesungen	2	3	2	3	1	1
Studenten	90 + 90	180 + 180	90	180	1	1
Fakultäten	0		10 + 20	10 + 20	1	1+2
Summe	183	364	123	214	4	6

Tabelle 4: Ressourcenbetrachtung für PrefetchGuide

Estimating Prophet nach Mark Palmer et. Al.

Ein weiteres Verfahren wird als Teil von Fido, einem Cache mit Vorhersagestrategie, unter dem Namen Estimating Prophet [29] vorgestellt. Dabei wird ein spezielles Datenmodel zur Speicherung aller Query sowie zusätzlicher Informationen verwendet. Dieser repräsentiert einen k-dimensionalen Raum, in denen die Query aufgrund von k-Eigenschaften gespeichert werden. Dies entspricht der Beschreibung von assoziativen Speichern, wie sie von Potter [30] oder Kanerva [31] beschrieben werden.

Jedes Query wird dabei in diesem Raum gespeichert. Zusätzlich wird zu jedem Query die Häufigkeit des Auftretens sowie die Güte dieses Pattern gespeichert. Dieses Datenobjekt wird als Unit Pattern beschrieben. Nach jedem durchgeführten Query wird ein Training durchgeführt. Dabei wird auf der Grundlage des letzten Query die Häufigkeit aller Unit Pattern aktualisiert bzw. ein neues Pattern angelegt, sollte dieses noch nicht existieren. Dabei wird das aufgetretene Pattern bestärkt und alle anderen abgeschwächt, bis sie einen Grenzwert unterschreiten und gelöscht werden. Somit wird sichergestellt, dass nur Pattern mit regelmäßigem Auftreten gewertet werden. Weiterhin wird die Richtigkeit der letzten Vorhersage bestimmt und die zugrunde liegenden Pattern entsprechend neu bewertet. Anschließend wird überprüft, ob die aktuelle Größe des Datencontainers die maximal zulässige Grenze überschritten hat. Ist dies der Fall, werden solange Pattern entsprechend ihrer niedrigsten Wertung gelöscht, bis die Größe unter dem definierten Schwellwert liegt.

Beim eigentlichen Preload wird in diesem Raum um die aktuelle Query ein k-dimensionaler Kreis mit einem festen Radius gezogen. Dieses Verfahren hat zwei große Vorteile. Einerseits können damit kleinere Abweichungen innerhalb der Query ausgeglichen werden, andererseits kann damit auch eine gezielte Veränderung in eine bestimmte Richtung nach verfolgt und vorhergesagt werden. Die Summe der Query innerhalb des Kreises bildet dann das Preload Pattern für die Abfrage.

Da es sich hierbei um ein lernendes Verfahren handelt, ist keine exakte Ressourcenbetrachtung möglich. Auch ist die Verwendung des theoretisch optimierten Zustandes als Anhaltspunkt ungeeignet, da dieser Zustand in der Regel nicht erreicht werden kann und die Auswertung verfälschen würde. Daher ist lediglich eine allgemeine Betrachtung des Verfahrens möglich.

3.3.2.5 Context-based Prefetching

Context based Prefetching ist das Verfahren, welches am häufigsten eingesetzt wird. Dabei werden alle Objekte geladen, die eine Gemeinsamkeit haben, auch wenn diese unterschiedlichste Ausmaße annehmen können. Angefangen von der Tatsache, dass beide Objekte in derselben Tabelle gespeichert werden bis hin zu einer zusätzlichen Datenstruktur mit Informationen über den Context des Objekts.

ContextLoader nach Philip A. Bernstein et. Al.

Mit dem ContextLoader [32] stellt Bernstein einen Ansatz vor, welcher sich auf die Zugehörigkeit zu einer Collection bezieht. Dabei wird beim Abrufen einer Collection diese als Context erkannt und gespeichert. Wird nun ein Teilbaum der Collection abgefragt, werden alle Teilbäume der im Context gespeicherten Knoten mitgeladen. Da dies in einer Abfrage geschehen kann, ist der zusätzliche Datenbankaufwand sehr gering.

Zum Verdeutlichen wird eine Beispielabfrage verwendet, in der Professor p1 als root-Element verwendet wird. Ausgehend von diesem Objekt wird nun die Collection Vorlesungen (v1 - v3) geladen. Wird nun eine Vorlesung besucht, werden alle anderen Vorlesungen der Collection mitgeladen. Gleiches passiert beim Laden der Studenten (s1 - sn). Dabei spielt es keine Rolle, welches Objekt der Collection geladen wird. Es muss also nicht zwangsläufig das erste Element sein. Weiterhin ist zu beachten, dass auch Studenten geladen werden, welche nicht zur besuchten Vorlesung gehören. Bei Verwendung von Vorlesung v1 als root-Element, würde

lediglich Studenten s1 – s3 durch das Prefetching erfasst werden. Da sich der aktuelle Kontext jedoch auf die Collection Vorlesungen bezieht, werden auch die Studenten aller anderen Vorlesungen geladen.

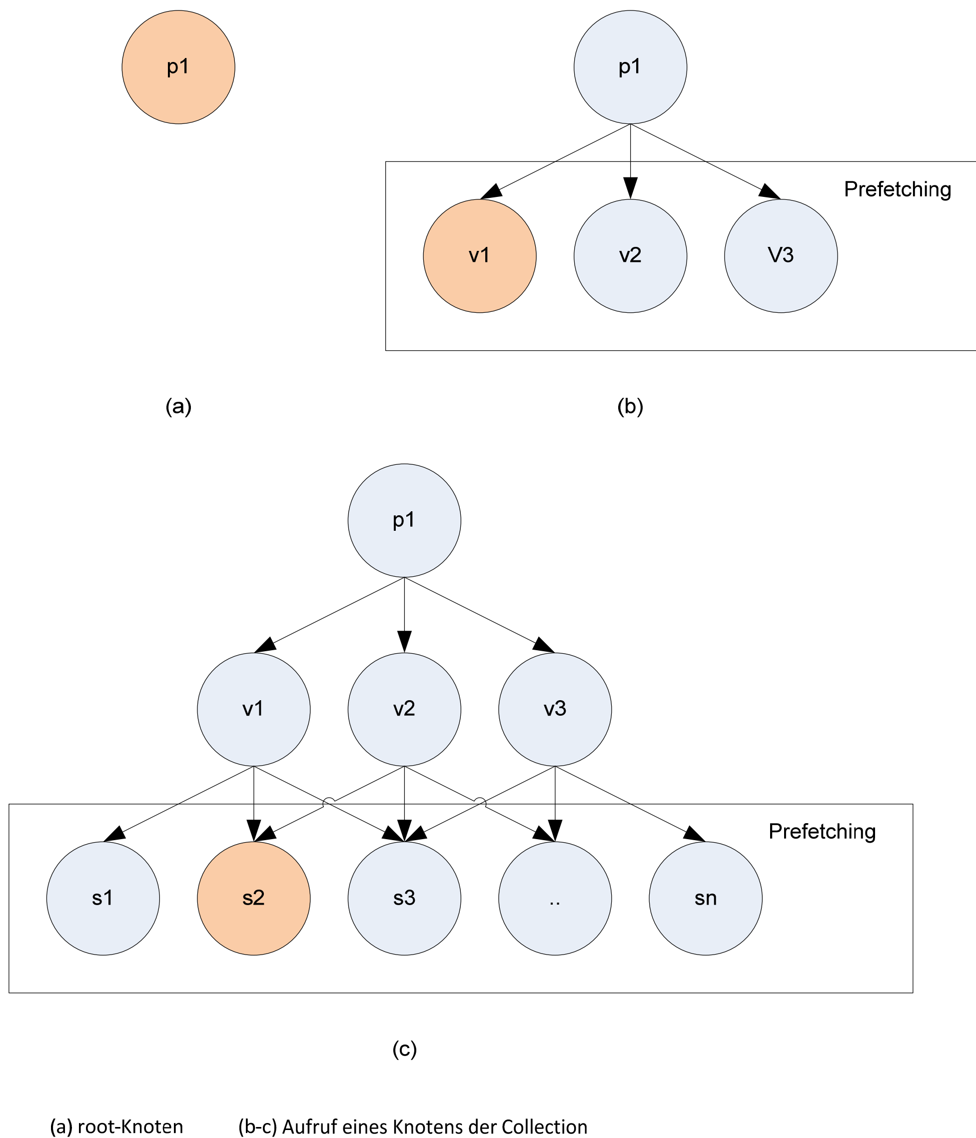


Abbildung 17: Funktionsweise des ContextLoaders

Dieser Ansatz eignet sich besonders gut für eine Breitensuche in Graphen. Häufig werden immer dieselben Informationen von Knoten in einer Collection benötigt. In diesem Fall besitzt das Verfahren eine besonders gute Abfragezeit. Wenn jedoch lediglich entlang eines Graphen eine Tiefensuche durchgeführt wird, kann es zu einer Überladung des Caches und damit Verschlechterung der Performance führen. Aufgrund dessen ergeben sich bei unterschiedlichen Abfrageszenarien unterschiedliche Performancegewinne oder Verluste. Richtig angewendet

bietet das Verfahren laut dem Autor bis zu 70 Prozent Geschwindigkeitsgewinn gegenüber on-demand Fetching.

	Abfrage A		Abfrage B		Abfrage C	
	Anfragen	Daten	Anfragen	Daten	Anfragen	Daten
Professoren	1	1	1	1	1	1
Vorlesungen	1	3	1	3	1	3
Studenten	1+1	180 + 180	1	180	1	180
Fakultäten	0	0	1 + 2	10 + 20	1+2	10+20
Summe	4	364	6	214	6	214

Tabelle 5: Ressourcenbetrachtung für ContextLoader

3.3.2.6 Path-based Prefetching

Path-based Prefetching beschreibt einen Ansatz, bei dem durch Analyse der vergangenen Abfragen und speziell der Abfragepfade Schlüsse auf das weitere Verhalten des Benutzers geschlossen werden. Dazu müssen die Abfragen gespeichert und ausgewertet werden. Hierzu ist eine geeignete Datenstruktur unablässig. Dadurch ergibt sich ein erhöhter Speicheraufwand. Dieser wird umso größer, je länger die Laufzeit der Applikation ist. Dafür besitzt das Verfahren die Möglichkeit, auf das Verhalten der Applikation seit dessen Start zurückzugreifen. Bei immer wiederkehrenden Aktionsmustern lässt sich darüber eine genaue Vorhersage definieren.

AutoFetch nach Ali Ibrahim et. Al.

AutoFetch von Ali Ibrahim & William R. Cook [33] als Vertreter des Path-based Prefetching ist ein Verfahren, bei dem alle Query zur statistischen Vorhersage von Preload Pattern herangezogen werden. Dabei ist die erste Verwendung nicht optimiert. Mit jedem Query erhöht sich über die Zeit die richtige Vorhersage der benötigten Objekte. Die Analyse wird dabei in Abhängigkeit zum gegenwärtigen root-Knoten durchgeführt. Es gibt also für jeden Knoten eine separate Analyse und damit Vorhersage.

Wird also ein Objekt des Typs Professor geladen, erkennt AutoFetch Professor als root-Knoten. Alle von diesem Objekt ausgehenden Knoten werden dann in einem Abfrageprofil gespeichert bzw. gezählt.

Ruft man also von Professor eine Vorlesung auf und von dort einen Studenten, werden diese Informationen gespeichert. Beim nächsten Aufruf eines Professors werden dann nur eine Vorlesung und die Fakultät des Professors geladen. Daraufhin erhält man ein Abfrageprofil, was angibt, dass in zwei von zwei Fällen eine Vorlesung geladen wurde, und nur in einem von zwei Fällen eine Fakultät bzw. ein Student. Der dritte Aufruf enthält den gesamten Baum von einem Professor bis zur Fakultät eines Studenten.

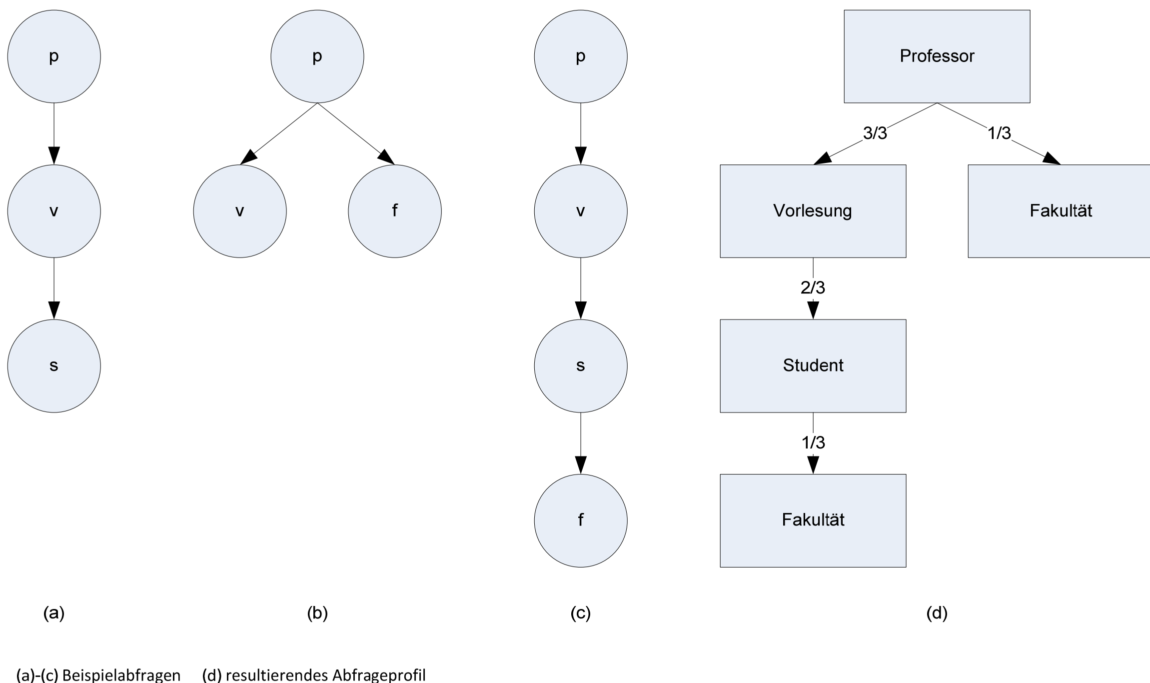


Abbildung 18: Funktionsweise von AutoFetch

Je mehr Abfragen gemacht werden, desto zuverlässiger wird die Aussage über das Ladeverhalten der Applikation. Wenn jedoch mehrere verschiedene Abfragen dasselbe root-Objekt benutzen, ist es unwahrscheinlich, dass eine gute Vorhersage für jeden Fall getroffen werden kann. Daher ist es notwendig, diese Abfragen zu klassifizieren. Neben der Möglichkeit, gleiche Query Strings oder Query Kriterien zu verwenden, besteht die Möglichkeit, diese aufgrund des Programmstatus zu definieren. Da das Ermitteln eines Programmstatus (Variablen, Bibliotheken, etc.) sehr aufwendig sein kann, wurde entschieden, den call stack, also die Aufrufhierarchie zu verwenden. In Kombination mit der Zeilennummer des Query ergeben sich so die besten Ergebnisse.

Der größte Nachteil dieses Verfahrens ist die nicht optimierte Startsituation. Gerade bei Applikationen mit kurzer Lebenszeit verringert sich der Nutzen des AutoFetch enorm. Je länger

die Applikation läuft, desto größer ist der Nutzen. Die zugrunde liegende Dokumentation dieses Verfahrens spricht hier von Reduzierung der Queryanzahl um 99,8% bei gleichzeitigem Performancegewinn von bis zu 99,7% gegenüber einem nicht optimierten System. Diese Werte entsprechen dem Zustand der manuell optimierten Abfragen.

Da es sich hierbei, ebenso wie beim Estimating Prophet um ein lernendes Verfahren handelt, ist keine exakte Ressourcenbetrachtung sondern lediglich eine allgemeine Betrachtung des Verfahrens möglich.

3.4 Ergebnisse des Vergleiches

Die Güte eines Prefetching Verfahrens ermittelt sich aus der Anzahl der Datenbankzugriffe im Vergleich zum on-demand Fetching und dem Verhältnis aus geladenen Daten und tatsächlich benötigten Daten.

	Abfrage A		Abfrage B		Abfrage C	
	Anfragen	Daten	Anfragen	Daten	Anfragen	Daten
on-demand Fetching	364	364	214	214	6	6
PreloadPattern	364	364	214	214	6	6
CriteriaJoiner	1	364	1	214	1	214
PrefetchGuide	183	364	123	214	4	6
ContextLoader	4	364	6	214	6	214

Tabelle 6: Vergleich Fetchingverfahren

Tabelle 6 fasst die beispielhaften Aufwandsbetrachtungen der einzelnen Verfahren zusammen. Deutlich zu erkennen ist die gleiche Performance des on-demand Fetching gegenüber dem PreloadPattern. Ebenso wie das Eager Loading benötigt es für jede Information eine einzelne Select-Anfrage. Allerdings weist es gegenüber anderen Verfahren ein besseres Verhalten bei selektiven Anfragen auf, da es nur die benötigten Daten lädt. Der CriteriaJoiner als zweites manuelles Verfahren zeigt dagegen eine schlechte selektive Performance auf, da es generell über Join Statements große Menge von Daten mit einer Anfrage abruft. Daher ist die Performance bei allgemeinen Anfragen am besten. Ähnlich schneidet der ContextLoader ab, da er ebenfalls mithilfe von Verallgemeinerungen versucht, große Datenmengen mit wenigen Abfragen zu laden. Dagegen beeindruckt der PrefetchGuide mit einer sehr guten Performance bei selektiven Abfragen und kann bei allgemeinen Abfragen die Zahl der notwendigen Anfragen fast halbieren. Damit bietet es im Schnitt die beste Performance für beide Anwendungsbereiche.

Bei den selbst lernenden Verfahren ist ein Vergleich schwieriger. Für die Performance des Systems mithilfe von AutoFetch oder dem Estimating Prophet sind vor allem Systemkonfiguration, Benutzerverhalten sowie Datenbankstruktur maßgeblich. Die endgültige Entscheidung über die Verwendung eines dieser Systeme kann erst aufgrund von Tests erfolgen, die an dem Zielsystem durchgeführt werden müssen. Beide Verfahren klingen viel versprechend und scheinen ein hohes Optimierungspotential zu besitzen, das weit über den nicht lernenden Verfahren liegt. Im Allgemeinen ist das AutoFetch das einfacher zu verstehende Verfahren, wird

jedoch eingeschränkt durch die Unterscheidung jeder Abfrage. Dies erlaubt zwar eine wesentlich bessere Differenzierung und damit auch Optimierung, jedoch wird jede neue Abfrage unoptimiert begonnen. Dagegen kann der Estimating Prophet mithilfe seines globalen Modells auch bei unbekannten Situationen Aussagen über das mögliche spätere Verhalten treffen. Dies geht jedoch wiederum auf Kosten der Fehlerrate, so dass die Wahrscheinlichkeit für eine falsche Vorhersage in Situationen, die nicht dem Durchschnitt entsprechen, steigt. Zudem ist dieses Verfahren vom Aufwand der Implementierung sowie der Vorarbeit zum Einsatz des Verfahrens enorm hoch.

Neben der reinen Datenbankabfrage gibt es zudem noch einen weiteren Grund, Prefetching dem on-demand Fetching vorzuziehen. So bedarf es zum on-demand Fetching einer ständigen Datenbankverbindung, die gewährleistet, dass benötigte Daten aus der Datenbank gelesen werden können. Gängige DBMS können nur eine gewisse Anzahl von Verbindungen bereitstellen, und sichern dazu auch jeder Verbindung eine gewisse Bandbreite. Dabei bestimmt sich die Bandbreite aus der Anzahl der offenen Verbindungen. Eine hohe Zahl von Verbindungen führt also zu Bandbreiteneinschränkungen aller Applikationen. Daher kann Prefetching verwendet werden, um Datenbankverbindungen zwischenzeitlich zu schließen. Dabei kommt es darauf an, dass genügend Daten vorgeladen werden, um ein ständiges Verbinden zur Datenbank zu vermeiden. Für diese Anwendungsfälle ist die Betrachtung der Datenmenge zweitrangig. Auch die Anzahl der Anfragen stellt nur ein geringes Kriterium dar. Vielmehr geht es um Zuverlässigkeit und Sicherheit.

Abschließend muss jedoch erwähnt werden, dass es sich bei den hier erläuterten Verfahren lediglich um eine beispielhafte Auswahl der Verfahren handelt. Aufgrund der Komplexität und Vielfältigkeit der Möglichkeiten kann keine Garantie auf Vollständigkeit übernommen werden. Dabei ist zu beachten, dass sich das Problem des Vorausladens nicht nur auf Datenbanken beschränkt. So findet ein Großteil der Arbeit im Bereich Speicherverwaltung sowie Caching statt. Auch hier wird versucht, mithilfe von Heuristiken Aussagen über das zukünftige Verhalten des Benutzers bzw. des Systems zu treffen. Die Erprobung der Überführung dieser Verfahren auf das Datenbankproblem sowie der damit verbundene Aufwand sind jedoch nicht Teil dieser Arbeit. Daher kann nicht ausgeschlossen werden, dass ein Verfahren mit besserer Performance als die hier vorgestellten existiert.

4 Aufgabenstellung

Der Diplomarbeit liegt die Anforderung zugrunde, aus den Erkenntnissen der verschiedenen Preload Pattern einen Prototyp zu entwerfen, der den Anforderungen der Applikation gerecht wird.

Bei der Anwendung handelt es sich um ein C2-System. Dabei gehen wir von mehreren unabhängigen geographischen Sektoren aus, welche jeweils über ein Serversystem mit allen nötigen Serverapplikationen verfügen. Die dazugehörigen Datenbankserver synchronisieren sich dabei untereinander mithilfe einer Datenbankreplikation. Zu jedem Sektor gehört ein Sektor Hauptquartier, mehrere regionale Hauptquartiere, wobei das Sektor Hauptquartier ebenfalls die Rolle eines regionalen Hauptquartiers übernimmt, sowie mehrere verteilte Stationen pro regionalem Hauptquartier, welche wiederum über mehrere Fahrzeuge verfügen. Alle Gebäude und Fahrzeuge verfügen über ein eigenes Funknetzwerk und/oder Kabelverbindungen und haben damit Zugriff auf die Serverarchitektur und die Datenbank.

4.1 Ausgangssituation

Das System wird in Java implementiert. Die applikationsseitigen Datenbankzugriffe werden durch das Hibernate Framework gewährleistet. Dabei sind alle Mappings auf Lazy Loading konfiguriert worden. Die dadurch notwendigen Sessions werden durch eine SessionFactory bereitgestellt. Dabei wird eine Session bei erstmaligem Aufruf eines Services geöffnet und erst beim Beenden des Services geschlossen.

Dieser Ansatz birgt jedoch gewisse Risiken. Sollte die Anwendung nicht ordnungsgemäß geschlossen werden, was beispielsweise durch einen Absturz des Hostsystems oder Instabilitäten innerhalb von Java passieren könnte, bleiben Sessions und damit verbundene Datenbankverbindungen geöffnet. Andererseits kommt es durch vorzeitiges Schließen der Sessions zu sogenannten Lazy Loading Exceptions, wie bereits in Kapitel 2.6.3 erläutert.

Lange Sessions führen jedoch auch dazu, dass der Zustand eines Systems im Quellcode nicht genau bestimmt werden kann. Es kann damit nicht zu jeder Zeit festgestellt werden, ob eine Session geöffnet ist oder nicht. Dadurch wird das Schreiben und Debuggen der Software schwierig.

Zur Entwicklung wird die agile Softwareentwicklungsmethode[34] Scrum[35] verwendet, welche als einen wichtigen Punkt das Programmieren jedes Entwicklers an jedem Codesegment enthält. Demzufolge erhöht sich die Anforderung an die Lesbarkeit des Quellcodes. Leider ist dies mit langen Sessions nur bedingt möglich, da meistens nur schwer ersichtlich ist, wie die Sessions gehandhabt werden.

Ein weiterer Nachteil durch lange Sessions ist die Performance des Datenbankservers. Zu jeder offenen Session gehört eine Datenbankverbindung und jeder Datenbankverbindung muss durch das DBMS eine anteilige Bandbreite bereitgestellt werden. Somit verringert sich die Bandbreite für andere Datenbankverbindungen.

4.2 Problemstellung

Zur Lösung der Probleme wurde entschieden, das System auf kurze Sessions umzustellen. Das bedeutet, dass nach dem Datenbankzugriff die Session geschlossen wird. Dadurch kann auf viele Aspekte der SessionFactory verzichtet werden, so dass diese lediglich für die Erstellung von Sessions eingesetzt wird. Auch wird der Code transparenter, da das Öffnen und Schließen zeit- und damit codenah zum eigentlichen Datenbankzugriff geschieht. Der größte Vorteil ergibt sich jedoch in der Performance des Datenbankservers. Diesem steht nun durch das Schließen gegenwärtig nicht benutzter Datenbankverbindungen mehr Bandbreite für die übrigen Verbindungen zur Verfügung.

Gleichzeitig zur Umstellung auf kurze Sessions, die das Bearbeiten der Services erforderlich machen, sollen auch Fassaden eingerichtet werden, die eine Schnittstelle der Services gegenüber der Businesslogik[2] darstellen. Innerhalb dieser Fassaden soll dann auch das Session Management realisiert werden.

Jedoch ergeben sich durch kurze Sessions Probleme in Bezug auf Lazy Loading. Wie bereits erwähnt, sind offene Sessions eine Notwendigkeit für das Funktionieren von Lazy Loading. Werden Sessions nun gleich nach einer Datenbankabfrage geschlossen, können Daten nicht mehr nachgeladen werden. Häufige Exceptions wären die Folge. Daher muss ein Weg gefunden werden, dieses Problem zu beheben.

[2] Die Logikschicht bezeichnet die mittlere Schicht im gängigen drei-Schichten Modell der Softwarearchitektur. Schwerpunkt dieses Modells ist die Aufteilung einzelner Softwareaspekte in konzeptionelle Schichten. Dabei hat jede Schicht nur Zugriff auf „tiefere" Schichten. [44]

Die Aufgabe besteht also, neben dem Erstellen der Service-Fassaden, in der Umstellung der Services auf ein Datenbankkonzept mit kurzen Sessions bei gleichzeitiger Vermeidung von Lazy Loading Problemen. Hierbei stellt Lazy Loading das zentrale Problem dar, weswegen im Folgenden verschiedene Lösungsstrategien zu diesem Problem diskutiert und bewertet werden.

5 Lösungsansätze

5.1 Lösung des Lazy Loading Problems

Die Grundidee zur Lösung von Lazy Loading Exceptions besteht in der Vermeidung von Lazy Loading an Stellen, wo es nicht erwünscht ist. Ein solches Verfahren, bei dem alle Informationen direkt von der Datenbank gelesen werden, nennt sich Eager Loading.

Eager Loading bildet damit den Gegensatz zu Lazy Loading. Im Detail bedeutet die Lösung, dass bei Aufruf eines Knotens der gesamte Teilgraph ab diesem Knoten geladen wird. Dadurch ist jedoch der Ladeaufwand sehr hoch, wobei nicht vorhersehbar ist, welche Teile des geladenen Graphen am Ende von der Applikation verwendet werden. Gerade bei bidirektionalen Graphen kann es im schlechtesten Fall zu einem Laden des kompletten Datenbestandes kommen, ohne dass dies notwendig gewesen wäre. Dadurch ergeben sich mitunter erhöhte Ladezeiten, da mehr Daten in kürzerer Zeit als beim Lazy Loading übertragen werden müssen.

5.1.1 Eingriff in die Hibernate Konfiguration

Der naheliegendste Ansatz zur Vermeidung von Lazy Loading besteht in der Deaktivierung der Lazy Loading Funktion in Hibernate. Dies lässt sich entweder über einen globalen Parameter, über die Anpassung der Mapping Dateien oder durch die Verwendung entsprechender Syntax bei der Formulierung der Anfrage steuern.

Die Setzung des globalen Parameters bietet die einfachste und schnellste Möglichkeit, Einfluss auf das Ladeverhalten von Lazy Loading zu nehmen. Hierdurch ergeben sich aber genau die vorher angesprochenen Probleme bezüglich der Datenmenge, die geladen wird. Der Entwickler hat keine Möglichkeit, dieses zu beeinflussen.

Beim Anpassen der Mapping Dateien von Hibernate besteht die Möglichkeit, durch Verwendung eines „Join“-Statements das Ladeverhalten zu beeinflussen. Die Grundeinstellung „Select“ erlaubt es dabei Hibernate, jeden Knoten durch eine separate Select Abfrage zu laden. Bei Verwendung von „Join“ kann Hibernate gezwungen werden, alle Knoten mithilfe eines einzigen Query zu beziehen. Dadurch beinhaltet die Antwort der Datenbank bereits alle Knoten des Objektes.

Der Vorteil gegenüber dem globalen Eager Loading Parameter besteht darin, dass für jeden Knoten einzeln bestimmt werden kann, ob er mit geladen werden soll. Jedoch ist dies nur knotenabhängig definierbar. In der Praxis benötigen unterschiedliche Services, die denselben Knoten aufrufen, jedoch nicht immer dieselben Teilbäume dieses Knotens. Dies ist jedoch über die Parameter des Mappings nicht steuerbar.

Die letzte Möglichkeit, Hibernate zum Eager Loading zu bewegen, besteht im Editieren der Query. Dabei wird wie beim Manipulieren der Mapping Dateien ein „Join“ Statement verwendet. Die Auswirkungen auf die Funktionsweise sind ebenfalls identisch, nur dass das Query nicht an die Knotenstruktur gebunden ist. So kann für jeden Service anwendungsfallspezifisch ein Query erstellt werden, was einen gewissen Mehraufwand bedeutet, jedoch die Möglichkeit zur Optimierung von Ladeprozessen bietet.

Einschränkungen gibt es jedoch bei der Verwendung des „Join“ Befehls in Query. Zur Vermeidung von Endlosschleifen durch bidirektionale Graphen wurde die Verwendung des „Join“ Statements auf maximal einen beschränkt. Damit ist das Eager Loading auf eine Ebene beschränkt, was die Nutzbarkeit dieses Ansatzes erheblich einschränkt. Ähnliches gilt auch für das Editieren der Mapping Dateien. Hierbei erlaubt Hibernate zwar mehr „Join“-s, begrenzt aber dennoch die Anzahl.

An dieser Stelle errichtet Hibernate für alle tiefer liegenden Knoten abermals Proxys, die ein Nachladen gewährleisten. Ein Eager Loading mit dem Ansatz der Veränderung der Hibernate-Konfiguration ist damit nur bis zu einer bestimmten Tiefe möglich.

Bei der hier vorliegenden Anwendung ist es aber zwingend erforderlich, dass sie unabhängig von der Tiefe des Graphen funktioniert. Eine Begrenzung der maximalen Tiefe würde bei Verwendung kurzer Sessions und größerer Tiefen zu Fehlern führen.

5.1.2 Verwendung eines Session Proxys

Alternativ zur Umgehung des Lazy Loadings besteht die Möglichkeit, das Session Management so zu konfigurieren, dass bei Notwendigkeit einer Session diese erstellt und danach geschlossen wird. Der offensichtliche Vorteil besteht darin, dass die bisherige Lazy Loading Struktur nicht verändert werden muss.

Der Ansatz überschreibt die Hibernate Funktion *initialize()*, welche zum Laden eines Knotens verwendet wird. Hier wird eine mögliche Lazy Loading Exception abgefangen und die Session geöffnet. Anschließend wird erneut versucht, den Knoten zu laden. Der Proxy könnte in die bereits vorhandene Session Factory integriert werden. Damit wäre gewährleistet, dass weiterhin jeder Service seine eigene Session verwendet, was die Parallelisierung der Services vereinfacht. Andererseits muss damit weiterhin eine Sessioninstanz gehalten werden. Ein Abschaffen der SessionFactory ist damit also nicht möglich.

Auch ist damit der aktuelle Zustand der Session unerheblich. Der Entwickler muss der Session also keine Beachtung mehr schenken, da die Session sich im Bedarfsfall einfach aufbaut. Andererseits besteht dadurch aber keine Möglichkeit mehr, auf den Zustand der Session Einfluss zu nehmen. Ebenfalls ein Problem könnte ein ständiges Öffnen und Schließen der Datenbankverbindung sein, wenn durch Lazy Loading weitere Knoten nachgeladen werden. Die Auswirklungen auf die Performance des Datenbankservers sind dabei nicht abzuschätzen.

5.1.3 Verwendung von Preload Pattern

Preload Pattern geben dem Entwickler die Möglichkeit, das Laden von Objekten mit der Möglichkeit zur Angabe von zusätzlichen Informationen zu versehen. Dabei kann angegeben werden, welche Teile des Objektgraphen zusammen mit dem angeforderten Objekt mitgeladen werden sollen.

Hibernate besitzt auch ein eigenes Preload-Verfahren. Dabei setzt es auf Context-based Preloading ähnlich dem Ansatz von Berstein et. Al. [32]. Wird für ein Objekt in einer geladenen Collection ein referenziertes Objekt besucht, werden für alle Objekte dieser Collection diese Relation nachgeladen. Allerdings muss dieses Ladeverhalten explizit für jede Collection in der Mapping Datei von Hibernate angegeben werden. Dazu dient der Befehl fetch="subselect". Ähnlich wie bei der Verwendung von „Join" in den Mapping Dateien ist es dann aber nicht mehr möglich, auf das Ladeverhalten Einfluss zu nehmen.

Möchte man zur Laufzeit des Programms entscheiden, welche Teile des Objektgraphen geladen werden sollen, muss eine eigene Preload-Methode implementiert werden. Dieser kann eine Liste mit Objekten übergeben werden, welche geladen werden sollen. Dabei besteht die Möglichkeit, diese Pattern manuell zu erstellen oder automatisiert erstellen zu lassen.

Die Vor- und Nachteile der einzelnen Verfahren wurden im Kapitel 3.3 erläutert. Da die Angabe des Pattern zur Laufzeit des Programms geschieht, kann für jeden Aufruf ein anderes Pattern verwendet werden. Dadurch kann das Preload-Verhalten sehr detailliert gesteuert werden.

Diese Methode bietet neben der speziellen Anwendbarkeit noch andere Vorteile. Durch den direkten Einfluss auf das Ladeverhalten wird das Laden von Objekten verständlicher und nachvollziehbarer. Wo man vorher auf die Methoden von Hibernate hoffen musste, wobei die genauen Prozesse nicht ersichtlich waren, kann bei diesem Prinzip abgelesen werden, welche Teile des Graphen geladen werden. Tritt dann in der Anwendung ein Fehler wegen fehlender Ressourcen auf, kann dieser wesentlich schneller erkannt und behoben werden.

Allerdings stellt das Implementieren einen Mehraufwand abhängig von der Größe des Systems dar. Wohingegen das Editieren von Mapping Dateien oder der Hibernate Konfigurationsdatei schnell implementiert werden kann, muss für die Verwendung von Preload Pattern eine Reihe von Methoden zum richtigen Laden der Objekte entwickelt werden. Bei entsprechender Größe des Objektgraphen ist dies eine umfangreiche Aufgabe.

5.2 Persistenzbetrachtung

Ein weiterer Aspekt, der bei der Verwendung von kurzen Sessions beachtet werden muss, ist die Persistenz. Diese wird normalerweise von Hibernate selber verwaltet (siehe Kapitel 2.6.4). Jedoch benötigt Hibernate zum Überprüfen des Zustandes eines Objektes eine bestehende Datenbankverbindung. Dies ist bei dem geplanten System aber nicht vorgesehen. An dieser Stelle wird die Persistenz über einen zusätzlichen Service auf Datenbankseite geregelt. Beim Starten der Applikation meldet sich diese bei dem Service an und erhält fortan Notifikationen über Änderungen in der Datenbank. Diese lösen dann einen Nachladeprozess aus, welcher auch über das Preload Pattern erfolgt. Damit ist sichergestellt, dass Änderungen in der Datenbank zur Laufzeit der Applikation aktualisiert werden.

5.3 Auswertung

Eine Anforderung an die Preload-Eigenschaften ist das Optimieren des Ladevorgangs. In erster Linie steht hierbei das Vorausladen aller benötigten Daten, so dass keine ständige Datenbankverbindung mehr notwendig ist. Andererseits ist es aber unnötig, alle Daten aus der

Datenbank zu laden. Es ist also wichtig sicherzustellen, dass nur der notwendige Teil geladen wird.

Die Methode, Hibernate über einen globalen Parameter zum Eager Loading zu zwingen, bietet dabei zu wenige Konfigurationsmöglichkeiten. Daher wurde entschieden, dieses Verfahren nicht zu wählen.

Ähnliches gilt auch für die Angabe von Eager Loading Eigenschaften in den Mapping Dateien von Hibernate. Hier existiert zwar die Möglichkeit, das Verhalten von Hibernate selektiver zu steuern. Jedoch benutzen mehrere Module mit unterschiedlichen Anforderungen an das Datenmodell dieselben Mapping Dateien. Damit entfallen auch diese Hibernate-internen Ansätze.

Anpassbarkeit auf Modulebene erlauben demnach die Veränderung der Hibernate-Query sowie das Verwenden von Preload Pattern. Zudem existiert noch die Möglichkeit, das System auf dem jetzigen Stand zu lassen, und das Lazy Loading durch ein automatisiertes Session-Konzept (SessionProxy) zu erweitern.

Der SessionProxy wurde jedoch verworfen, da durch diese Maßnahme der Zustand des Systems nicht mehr eindeutig definiert ist. Aufgrund der Tatsache, dass Sessions an beliebiger Stelle geöffnet werden können, ergeben sich ungeahnte Folgeprobleme. Daher wurde auch dieses Konzept zur Lösung des Problems verworfen.

Die Entscheidung, ob nun Query verändert werden sollten, um Preload-Eigenschaften zu erzielen, oder Preload Pattern angewendet werden, war dagegen wesentlich schwieriger. Beide bieten die Möglichkeit, modulspezifisch Teilbäume zu selektieren. Auch der mögliche Detailgrad der Selektion ist annähernd identisch. Eine mögliche Beschränkung bei der Query-Manipulation ist die Begrenzung der möglichen Joins. Dies kann jedoch durch die Verknüpfung mehrerer Query ausgeglichen werden. Bei komplexen Teilgraphen ist dadurch jedoch eine Vielzahl von Query notwendig. Dies wiederum beeinträchtigt die Lesbarkeit des Codes. Andererseits bringt das Implementieren von Preload Pattern einen erhöhten Aufwand mit sich. Dazu gehört der Umbau aller Module sowie das Erstellen der Preload Pattern für jedes Modul.

Ausschlaggebend war schließlich die Tatsache, dass Preload Pattern eine bereits erprobte und ausgereifte Technik bieten, und zudem durch automatisierte Ansätze erweitert werden können. Dagegen ist die Funktionalität von Hibernate bezüglich der Join-Statements nicht

ausreichend dokumentiert. Auch würde sich bei externen Bibliotheken die Fehlersuche schwieriger gestalten. Sucht man beispielsweise nach der Ursache einer aufgetretenen Lazy Loading Exception, ist die Überprüfung eines Preload Pattern einfacher als das Nachverfolgen der Ladevorgänge von Hibernate.

Durch die Verwendung von Preload Pattern ergibt sich eine elegante Möglichkeit, Dateien transparent, einheitlich und selektiv auszuwählen. Zudem erlaubt es den Einsatz von automatisierten Preload Techniken.

Nachdem die Entscheidung zugunsten von Preload Pattern getroffen wurde, bestand die Frage nach dem Verfahren der verschiedenen Preload Pattern. Hauptaugenmerk dabei lag auf der Zuverlässigkeit der Algorithmen. Da die Session nach dem Laden geschlossen werden sollte, ist kein Nachladen von fehlenden Daten möglich. Daher ist es wichtig, ein Verfahren zu wählen, was im schlimmsten Fall zu viele Daten lädt, aber niemals zu wenige.

Ebenfalls entscheidend ist der Implementierungsaufwand. Je nach gewähltem Verfahren können eine zusätzliche Datenstruktur, der Zugriff auf alle Datenbankprozesse sowie möglicherweise die Verwendung umfangreicher Systemressourcen benötigt werden. Da für die Aufgabenstellung weniger die Performance des Systems als vielmehr die Stabilität der Anwendung im Vordergrund stand, wurde beschlossen, ein manuelles Preloading umzusetzen. Es bietet die Möglichkeit, sicherzustellen, dass alle benötigten Daten geladen werden. Erfordert aber auch das manuelle Erstellen und Optimieren der Preload Pattern. Ein weiterer Aspekt ist die psychologische Komponente der Entwickler. Unabhängig von statistischen oder experimentellen Ergebnissen ist das Überlassen von wichtigen Funktionen an ein automatisches System meist nicht erwünscht. Dabei spielen der Verlust der Kontrolle sowie die Undurchsichtigkeit des konkreten Prozesses eine entscheidende Rolle.

Eine Automatisierung von Preload Pattern ist dennoch zu einem späteren Zeitpunkt denkbar. Das manuelle Prefetching beruht auf der Übergabe von Prefetching Parametern. Dabei werden diese Parameter manuell erstellt und übergeben. Hier wäre der Einsatz von Automatisierungen denkbar, bei denen als Ergebnis, ein Preload Pattern, an die Preload Klasse übergeben wird. Damit wäre ein mögliches Erweitern des Systems gegeben, wobei bei dieser Maßnahme dann die Performance des Systems im Vordergrund stehen würde.

6 Umsetzung des Prototypen

Um zu entscheiden, welches manuelle Preloading Verfahren vorzuziehen ist, wurde entschieden, sowohl das Verfahren von Jürgen Kohl als auch den CriteriaJoiner zu implementieren. Zusätzlich muss ein weiteres Verfahren implementiert werden, welches den Ausgangszustand des aktuellen Systems wiederspiegelt. Dabei handelt es sich um ein on-demand Fetching.

Alle Verfahren werden so implementiert, dass jedes innerhalb einer Fassade eingebettet ist. Diese Fassaden zeichnen sich durch eine standardisierte Schnittstelle zwischen den individuellen Verfahren und den Anforderungen des Services aus. Für den Prototyp wurde das Datenbankmodell aus Kapitel 3.2 verwendet. Da dieses Modell das root-Objekt Professor nahelegt, wurde die Fassade ebenfalls als Professor-Fassade definiert. Demnach gibt es neben der allgemein gehaltenen *preload* Funktion, welche das eigentliche Verfahren aufruft eine *FindAllProfs* Methode, über die die Fassade für die Tests angesprochen werden kann. Weiterhin existieren Methoden wie *FindProfByID* und weitere selektive Methoden, die jedoch jeweils nur eine Abwandlung der *FindProfByID* Methode darstellen.

6.1 Implementierung der Testumgebung

Für die Testumgebung wurde ein HsqlDB Server [36] verwendet. Dabei handelt es sich um eine kleine und leicht zu konfigurierende relationale SQL Datenbank. Diese wird häufig für Testzwecke verwendet, da sie verschiedene Modi bietet, welche unter anderem das lokale Ausführen innerhalb der Entwicklungsumgebung erlaubt. Für die Testumgebung wurde der Server für den Stand-Alone Server Mode auf dem Testrechner konfiguriert und über die Eingabekonsole gestartet und gestoppt.

Die Anbindung der Testumgebung an die Datenbank erfolgt über Hibernate, wobei Hibernate seinerseits einen JDBC Treiber [37] für die Kommunikation mit der Datenbank verwendet. Weiterhin benötigt Hibernate Apache log4j [38] für die Ausgabe von Informationen und Fehlermeldungen.

Mithilfe der Mapping Dateien aus Abbildung 19 bis Abbildung 21 wurde die Datenbankstruktur aus Kapitel 313.2 nachgebildet. Abbildung 19 stellt dabei das Beispiel für die

einfachste Art eines Mappings dar, da das Objekt Fakultät lediglich eine ID sowie zwei Attribute enthält.

```
<hibernate-mapping package="com.eads.PreloadTestProject">
 <class dynamic-update="true" lazy="true" name="Faculty"
table="FACULTY">
  <id name="id">
   <generator class="sequence"/>
  </id>
  <property generated="never" lazy="true" name="name"/>
  <property generated="never" lazy="true" name="building"/>
 </class>
</hibernate-mapping>
```

Abbildung 19: Hibernate Mapping für das Objekt Fakultät

Abbildung 20 enthält die Definition eines Sets mit einer N:N Beziehung. Dabei muss die Klasse des abhängigen Objektes definiert werden.

```
<hibernate-mapping package="com.eads.PreloadTestProject">
 <class dynamic-update="true" lazy="true" name="Lecture"
table="LECTURE">
  <id name="id" type="long">
   <generator class="sequence"/>
  </id>
  <property generated="never" lazy="true" name="name"/>
  <property generated="never" lazy="true" name="day"/>
  <property generated="never" lazy="true" name="time"/>
  <property generated="never" lazy="true" name="duration"/>
  <property generated="never" lazy="true" name="location"/>
  <set lazy="true" name="students" sort="unsorted"
table="REL_LECTURE_STUDENT">
   <key column="lecture_id"/>
   <many-to-many class="com.eads.PreloadTestProject.Student"
    column="student_id" unique="false"/>
  </set>
 </class>
</hibernate-mapping>
```

Abbildung 20: Hibernate Mapping für das Objekt Vorlesung

Aufwendiger ist das Mapping des Objektes Person, siehe Abbildung 21, da es sowohl abstrakte als auch konkrete Klassen enthält. Bei der Konfiguration dieser Vererbung bietet Hibernate die in Kapitel 2.4.4 vorgestellten Möglichkeiten. Für die Testumgebung wurde die Konfiguration für ein normalisiertes Datenbankschema verwendet. Zudem enthält es eine Vielzahl von unterschiedlichen Relationen.

```
<hibernate-mapping package="com.eads.PreloadTestProject">
 <class dynamic-update="true" lazy="true" name="Person" table="PERSON">
  <!-- defining abstract class Person -->
  <id name="id">
   <generator class="sequence"/>
  </id>
  <property generated="never" lazy="true" name="name"/>
  <property generated="never" lazy="true" name="age"/>
  <property generated="never" lazy="true" name="gender"/>
  <many-to-one class="com.eads.PreloadTestProject.Faculty"
      column="faculty" lazy="proxy" name="faculty"/>

  <!-- defining class Professor -->
  <joined-subclass dynamic-update="true"
      extends="com.eads.PreloadTestProject.Person" lazy="true"
      name="com.eads.PreloadTestProject.Professor" table="PROFESSOR">
   <key column="id"/>
   <property generated="never" lazy="true" name="institute"/>
   <property generated="never" lazy="true" name="degree"/>
   <set lazy="true" name="lectures" sort="unsorted" table="LECTURE">
    <key column="professor_id"/>
    <one-to-many class="com.eads.PreloadTestProject.Lecture"/>
   </set>
  </joined-subclass>
  <!-- defining class Student -->

  <joined-subclass dynamic-update="true"
      extends="com.eads.PreloadTestProject.Person" lazy="true"
      name="com.eads.PreloadTestProject.Student" table="STUDENT">
   <key column="id"/>
   <property generated="never" lazy="true" name="semester"/>
  </joined-subclass>
 </class>
</hibernate-mapping>
```

Abbildung 21: Hibernate Mapping für das Objekt Person

```
<hibernate-configuration>
 <session-factory>
  <property name="hibernate.connection.driver_class">
      org.hsqldb.jdbcDriver</property>
  <property name="hibernate.connection.password"/>
  <property name="hibernate.connection.username">sa</property>
  <property name="hibernate.dialect">
      org.hibernate.dialect.HSQLDialect</property>
  <property name="connection.url">
      jdbc:hsqldb:hsql://localhost:9001/uni</property>
  <!--  <property name="hibernate.hbm2ddl.auto">update</property> -->
  <property name="show_sql">false</property>
  <property name="format_sql">false</property>
  <mapping
      resource="com/eads/PreloadTestProject/mapping/Lecture.hbm.xml"/>
  <mapping
      resource="com/eads/PreloadTestProject/mapping/Faculty.hbm.xml"/>
  <mapping
      resource="com/eads/PreloadTestProject/mapping/Person.hbm.xml"/>
 </session-factory>
</hibernate-configuration>
```

Abbildung 22: Hibernate Konfigurationsdatei

Alle Mapping Dateien müssen in der globalen Hibernate Konfigurationsdatei hinterlegt werden, damit Hibernate diese beim Initialisieren der Konfiguration einlesen und verarbeiten kann. Neben den Mapping Dateien enthält die Konfiguration ebenfalls die Informationen der Datenbankverbindung. Weiterhin ist es über eine Vielzahl von Parametern möglich, zusätzliche Funktionalitäten von Hibernate zu aktivieren bzw. zu deaktivieren. In der Konfiguration aus Abbildung 22 wurde das automatische Aktualisieren des Datenbankschemas aufgrund der Mappings aktiviert. Hibernate unterstützt zudem auch die Möglichkeit, Objektklassen auf der Grundlage von Mapping Dateien zu erstellen, sowie Mapping Dateien aufgrund von Klassen oder Relationsschemas zu generieren.

6.2 Implementierung des on-demand Fetching

Bei der Benutzung des on-demand Fetching wird von einer normalen Systembenutzung ausgegangen. Demzufolge hat man verteilt über eine Session Zugriffsprozesse auf Objekte, welche gerade vom System benötigt werden. Für den Prototypen und die damit verbundene Performanceanalyse sind jedoch nur die reinen Zugriffszeiten ausschlaggebend. Daher wurde als Vereinfachung lediglich das rekursive Abarbeiten des gesamten Objektbaumes implementiert.

Diese Rekursion findet in der in Abbildung 23 dargestellten *preload* Methode statt. Dabei muss der Methode eine Liste von root-Objekten übergeben werden. Anschließend wird das Objekt je nach Typ auf weitere Collections untersucht und die Methode mit der neuen Collection aufgerufen. Als Abbruchkriterium wird der Methode dabei die maximale Rekursionstiefe übergeben.

Diese Art der Implementierung ist für ein laufendes System ungeeignet, da es explizit den Typ des Objektes ermitteln muss. Da es sich hierbei jedoch um einen Prototypen handelt, reicht dieses Modell um es mit den Preload-Verfahren vergleichen zu können.

```
protected void preload(Object entity, int depth) {
      if (entity == null) {
            return;
      }

      if (entity instanceof Collection) {
            for (Object resultEntity : (Collection) entity) {
                  preload(resultEntity, depth);
            }
      }

      if (depth > 0) {
            if (entity instanceof Professor) {
                  Professor item = (Professor) entity;
                  preload(item.getLectures(),depth-1);
            }
            else if (entity instanceof Lecture) {
                  Lecture item = (Lecture) entity;
                  preload(item.getStudents(),depth-1);
            }
            else if (entity instanceof Student) {
                  Student item = (Student) entity;
                  item.getFaculty();
            }
      }
}
```

Abbildung 23: Implementierung der preload Methode des on-demand Fetching

Ebenso handelt es sich bei der *FindAllProfs* Methode in Abbildung 24 um eine vereinfachte Version eines endgültigen Systems. So entspricht das Sessionhandling innerhalb einer Methode nicht dem Standard. Daneben werden innerhalb der Methode lediglich die root-Elemente für die *preload* Methode mithilfe einer Hibernate Criteria geladen und anschließend die Ergebnisliste an die *preload* Methode übergeben.

```
public List<Professor> findAllProfs(int depth) {
      Session session = factory.openSession();
      Criteria crit = session.createCriteria(Professor.class);

      List<Professor> result = crit.list();
      preload(result,depth);

      if (session != null && session.isOpen())
            session.close();
      return result;
}
```

Abbildung 24: Implementierung der findAllProfs Methode des on-demand Fetching

6.3 Implementierung des PreloadPattern

Das Preload Pattern nach Jürgen Kohl, kurz PreloadPattern, weist ähnliche Strukturen wie das on-demand Fetching auf. Auch hier wird mit Hilfe von Rekursion der Objektbaum durchlaufen. Einer der Unterschiede in der *preload* Methode aus Abbildung 25 ist die Verwendung des Befehls Hibernate.initialize(Collection c). Mit Hilfe dieser Funktion kann Hibernate angeregt werden, alle Elemente der Collection zu laden. Damit ist es möglich, mehrere Elemente mit einer Datenbankanfrage zu laden, wohingegen des on-demand Fetching jedes Objekt über eine einzelne Anfrage lädt.

```
protected void preload(Object entity, Preload[] preloads) {
      if (entity == null) {
            return;
      }

      Hibernate.initialize(entity);

      if ((preloads == null) || (preloads.length == 0)) {
            return;
      }

      if (entity instanceof Collection) {
            for (Object resultEntity : (Collection) entity) {
                  preload(resultEntity, preloads);
            }
      } else {
            for (Preload preload : preloads) {
                  if (preload.getModelClass().isInstance(entity)) {
                        Object getterResult =
                              invokeGetter(entity, preload);
                        preload(getterResult, preloads);
                  }
            }
      }
}
```

Abbildung 25: Implementierung der preload Methode des PreloadPattern

Des Weiteren erfolgt der rekursive Aufruf bei diesem Verfahren nicht wie beim on-demand Fetching angerichtet oder durch das explizite Formulieren von Bedingungen innerhalb der Funktion, sondern wird durch eine Reihe von Preload Pattern gesteuert (siehe Abbildung 26). Ein Pattern besteht dabei aus einem Tupel aus Objektklasse und einen zu dieser Objektklasse gehörendem Attribut. Dabei wird überprüft, ob das gegenwärtige root-Objekt der *preload* Methode in einem Preload Pattern enthalten ist. Ist dies der Fall, wird die dazugehörige Collection als neues root-Objekt der *preload* Methode übernommen.

```
Preload[] PROFESSOR_LECTURE_STUDENT_FACULTY = {
      new Preload(Professor.class, "lectures"),
      new Preload(Lecture.class, "students"),
      new Preload(Student.class, "faculty")
}
```

Abbildung 26: Beispiel eines Preload Pattern des PreloadPattern

Dagegen stellt sich die *findAllProfs* Methode in Abbildung 27 genauso wie die Methode beim on-demand Fetching dar. Auch hier wurde zur Vereinfachung das Sessionmanagement innerhalb der Methode geschrieben. Ebenfalls wird wie beim bereits vorgestellten on-demand-Fetching die initialen root-Elemente geladen und der *preload* Methode übergeben.

```
public List<Professor> findAllProfs(Preload[] preloads) {
      Session session = factory.openSession();
      Criteria crit = session.createCriteria(Professor.class);

      List<Professor> result = crit.list();
      preload(result, preloads);

      if (session != null && session.isOpen())
            session.close();
      return result;
}
```

Abbildung 27: Implementierung der findAllProfs Methode des PreloadPattern

6.4 Implementierung des CriteriaJoiner

Im Gegensatz zu den anderen beiden Verfahren wird bei der in Abbildung 28 dargestellten *preload* Methode keine initialen Elemente benötigt. Diese Methode erstellt lediglich ein zusammengesetztes Criteria aufgrund der eingegebenen Preload Pattern.

```
protected DetachedCriteria preload(CriteriaPath[] paths) {
      CriteriaJoiner joiner = CriteriaJoiner.forClass(Professor.class);
      if (paths != null) {
            for (CriteriaPath path : paths) {
                  joiner.addPath(path);
            }
      }
      return joiner.getDetachedCriteria();
}
```

Abbildung 28: Implementierung der preload Methode des CriteriaJoiner

Ein Pattern ist dabei ein Pfad durch den Objektgraphen, wie er in Abbildung 29 dargestellt wird. Aus diesem wird ein sogenanntes Detached Criteria erstellt, welches sich von einem normalen Criteria dadurch unterscheidet, dass es nicht an eine Session gebunden ist und daher ohne diese generiert werden kann.

```
CriteriaPath[] PROFESSOR_LECTURE_STUDENT_FACULTY = {
      new CriteriaPath("lectures/students/faculty")
}
```

Abbildung 29: Beispiel eines Preload Pattern des CriteriaJoiner

Um das Criteria ausführen zu können und damit eine Datenbankanfrage auszulösen, muss das Detached Criteria an eine Session gebunden werden. Dies geschieht in der *findAllProfs* Methode, welche in Abbildung 30 dargestellt ist. Um eine selektive Anfrage zu generieren, muss im Gegensatz zu den anderen beiden Verfahren, das allgemeine Criteria erstellt und anschließend um Restriktionen erweitert werden. Bei den anderen Verfahren genügt es dagegen, lediglich weniger root-Objekte zu übergeben. Sollte eine solche Möglichkeit notwendig werden, muss jedes Objekt als Restriktion auf dieses Objekt zu dem Criteria hinzugefügt werden.

Allerdings bietet dieses Verfahren die Option, alle angeforderten Daten mit einer Anfrage aus der Datenbank zu laden. Dies geschieht mit dem Befehl crit.list(), welcher eine Liste von root-Objekten der Anfrage zurückliefert.

```
public List<Professor> findAllProfs(CriteriaPath[] paths) {
      Session session = factory.openSession();

      Criteria crit = preload(paths).getExecutableCriteria(session);
      List<Professor> result = crit.list();

      if (session != null && session.isOpen())
            session.close();
      return result;
}
```

Abbildung 30: Implementierung der FindAllProfs Methode des CriteriaJoiner

7 Performanceuntersuchung des Prototypen

Im Folgenden werden die unterschiedlichen Implementierungen der Preload-Verfahren auf Ihre Leistungsfähigkeit hin untersucht. Dabei geht es vorrangig um die Aussage, welches Verfahren auch bei großen Datenmengen schnell und stabil bleibt.

7.1 Vergleichsgrundlagen

Zusätzlich zur Implementierung der Fetching-Verfahren wurde ein automatisiertes Befüllen der Datenbank mit generierten Werten geschaffen. Als Testdatenbank dient das Modell, welches bereits im Kapitel 3.2 vorgestellt wurde. Das Befüllen der Datenbank ist dabei mit verschiedenen Parametern steuerbar:

$$p_{abs} - Anzahl\ der\ Professoren$$

$$v_{rel} - Anzahl\ der\ Vorlesungen\ pro\ Professor$$

$$s_{rel} - Anzahl\ der\ Studenten\ pro\ Vorlesung$$

$$n_{rel} - Anzahl\ der\ Vorlesungen\ pro\ Student$$

$$f_{abs} - Anzahl\ der\ Fakultäten$$

Absolute Zahlen stellen dabei die Gesamtzahl von Objekten dieses Typs innerhalb der Objektstruktur dar, während relative Zahlen die Anzahl der Objekte darstellen, welche zu einem übergeordneten Objekt gehören. Aus den Steuerparametern ergeben sich folgende absolute Zahlen:

$$V_{abs} = p_{abs} \cdot v_{rel}$$

$$S_{abs} = \frac{p_{abs} \cdot v_{rel} \cdot s_{rel}}{n_{rel}}$$

Die Verteilung der Fakultäten auf die Personen erfolgt dabei über einen Zufallsgenerator. Alle anderen Abhängigkeiten werden gleichmäßig auf die vorhandenen Objekte verteilt.

Mithilfe dieses Modells werden im Folgenden eine Reihe von Tests zum Vergleich der unterschiedlichen Verfahren durchgeführt. Untersucht werden soll dabei die Abhängigkeit der

Geschwindigkeit des Verfahrens von der Tiefe des Objektgraphen. Weiterhin interessant ist die Abhängigkeit der Objektanzahl von der Geschwindigkeit. Zu diesem Zweck werden unterschiedliche Abfrageszenarien untersucht, welche dann unter Verwendung unterschiedlich großer Objektbäume getestet werden.

Betrachtet werden für die Befüllung der Datenbank ausgewählte Parameter. Zur Untersuchung der Performance sind lediglich die Anzahl der Relationen sowie die Anzahl der Objekte wichtig. Da Professoren an dieser Stelle als Einstiegspunkt für die Abfragen dienen, kann damit nicht die Geschwindigkeit des Preload-Verfahrens bestimmt werden. Lediglich eine Aussage über die Verwendbarkeit bei großen Datenmengen erscheint an dieser Stelle sinnvoll.

$$p = 10,20, \dots ,200$$

Ebenfalls gering ist der Einfluss der Fakultäten. Da es sich hierbei um eine n:1 Beziehung handelt, können maximal $p_{abs} + p_{abs} \cdot s_{abs}$ Fakultäten verwendet werden. Damit lässt sich kein wesentlich größerer Datenaufwand generieren. Lediglich für die Relationen bzw. die Objekttiefe ist dieser Wert interessant, wobei die Anzahl für die Relationsbetrachtung egal ist.

$$f = 10,20, \dots ,200$$

Hingegen ist der Einfluss der Vorlesungen sowohl maßgeblich für die Datenbankgröße als auch die Relationen, da sich über die Anzahl der Vorlesungen ebenfalls die Anzahl der Studenten beeinflussen lässt. In Kombination mit der relativen Anzahl der Studenten lassen sich somit viele Objekte erzeugen. Wohingegen die Größe *n* ausschlaggebend ist für die Relationsanzahl, wobei beachtet werden muss, dass *n* indirekt proportional zur Studentenanzahl steht.

$$v = 10,20, \dots ,200$$

$$n = 1,2, \dots ,20$$

$$s = 10,20, \dots ,200$$

Der Parameter d bezeichnet zusätzlich die Objekttiefe, welche für die jeweilige Untersuchung gewählt wurde. Dabei ist der kleinstmögliche Wert für d null, und beschreibt lediglich das Laden des root-Objektes Professor. Aufgrund der Objektstruktur des Testsystems ist damit eine maximale Tiefe von drei möglich, und umfasst in diesem Zustand alle Objekte von Professor bis Fakultät.

7.2 Ergebnisse

Im Folgenden werden die Ergebnisse des Tests als statistischer Mittelwert aus 100 Durchläufen dargestellt.

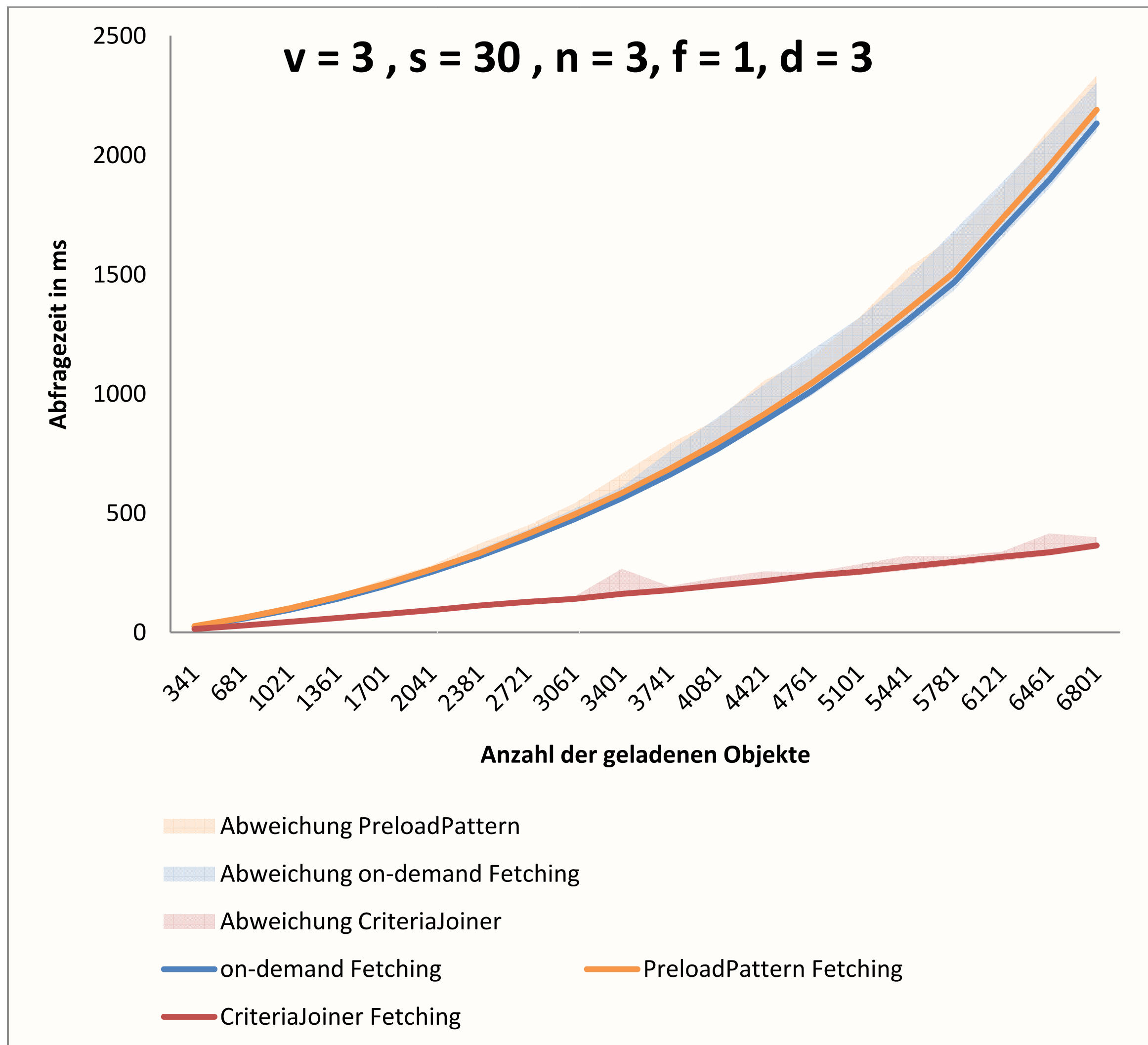


Abbildung 31: Abfragegeschwindigkeit abhängig von der Anzahl der Professoren

Erwartungsgemäß zeigt sich aus Abbildung 31 beim CriteriaJoiner bei steigender Objektgröße des Gesamtbaumes ein linearer Anstieg der Abfragezeit. Da immer nur eine Anfrage an die Datenbank formuliert wird, ist dies leicht nachvollziehbar. Die steigende Zeit lässt sich demzufolge auf das Erstellen der Java-Objekte bzw. die längere Verarbeitungsdauer der Datenbank zurückführen.

Dagegen zeigt sich sowohl beim PreloadPattern als auch beim on-demand Fetching ein quadratischer Anstieg der Abfragezeit. Dies ist zurückzuführen auf den rekursiven Aufbau beider

Verfahren. Dadurch ergibt sich bei steigender Objektzahl ein umfangreicherer Baum, welcher wesentlich mehr Rekursionen benötigt.

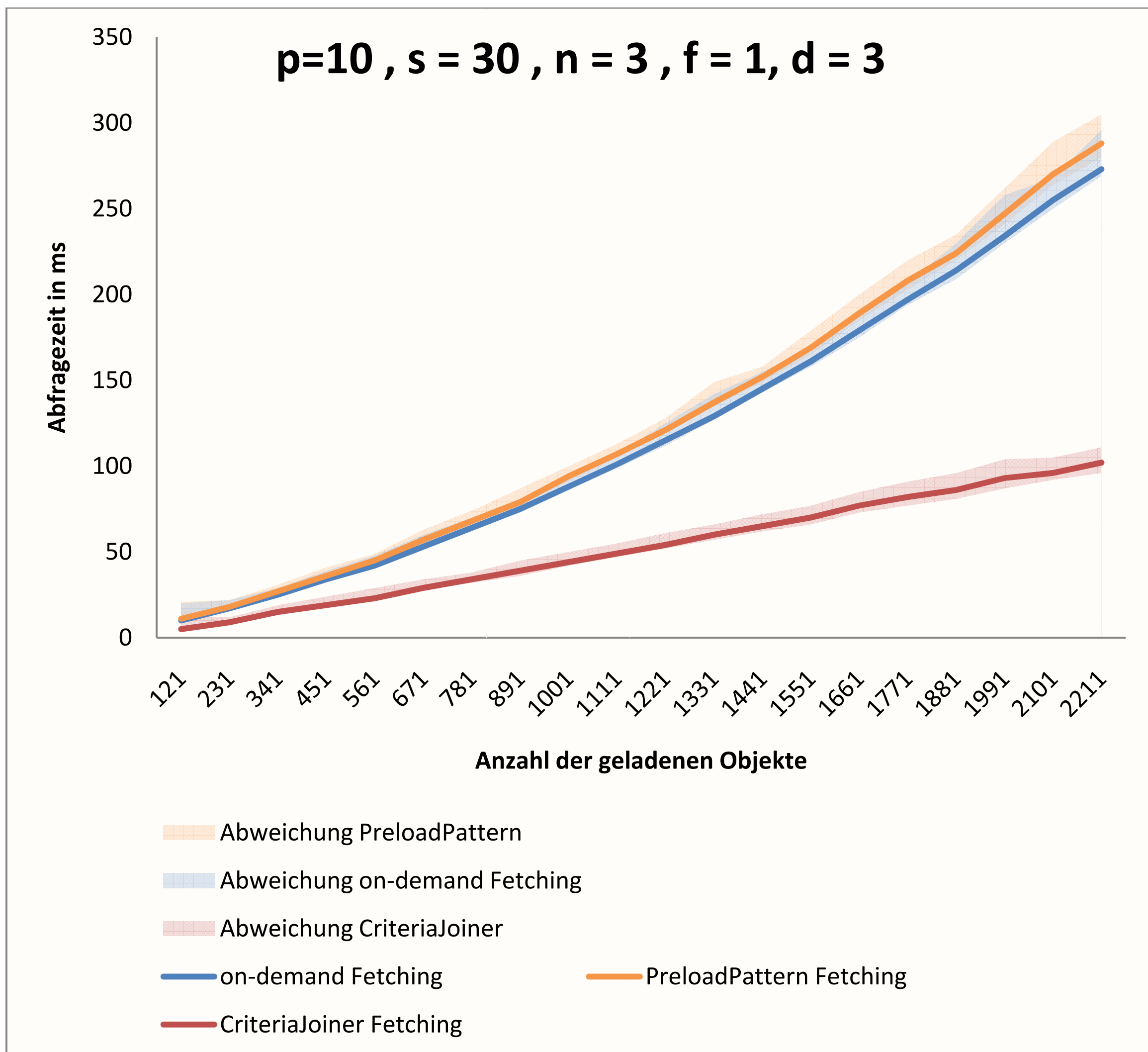


Abbildung 32: Abfragegeschwindigkeit abhängig von der Anzahl der Vorlesungen

Abbildung 32 belegt die Ergebnisse aus der Untersuchung der Abhängigkeit des Gesamtbaumes auf das Ladeverhalten. Durch den geringeren Einfluss der ersten Collection gegenüber der gesamten Objektanzahl ist der Anstieg jedoch geringer.

Ebenso zeigen sich die Ergebnisse in der Abbildung 33. Hierbei steigt lediglich die Anzahl der Fakultäten, also die letzte Relation, wobei in jedem Fall jeder Student eine Fakultät hat. Lediglich die mögliche Verteilung der Fakultäten ändert sich mit zunehmender Anzahl. Damit ist die Abfrage beim CriteriaJoiner immer gleich, was auch eine Auswertung der von Hibernate durchgeführten Abfragen ergeben hat, welches den konstanten Aufwand wiederspiegelt. Der

lineare Anstieg der anderen beiden Verfahren erklärt sich wiederum durch die zusätzlich notwendigen Datenbankabfragen für jedes zusätzliche Initialisieren eines Objektes.

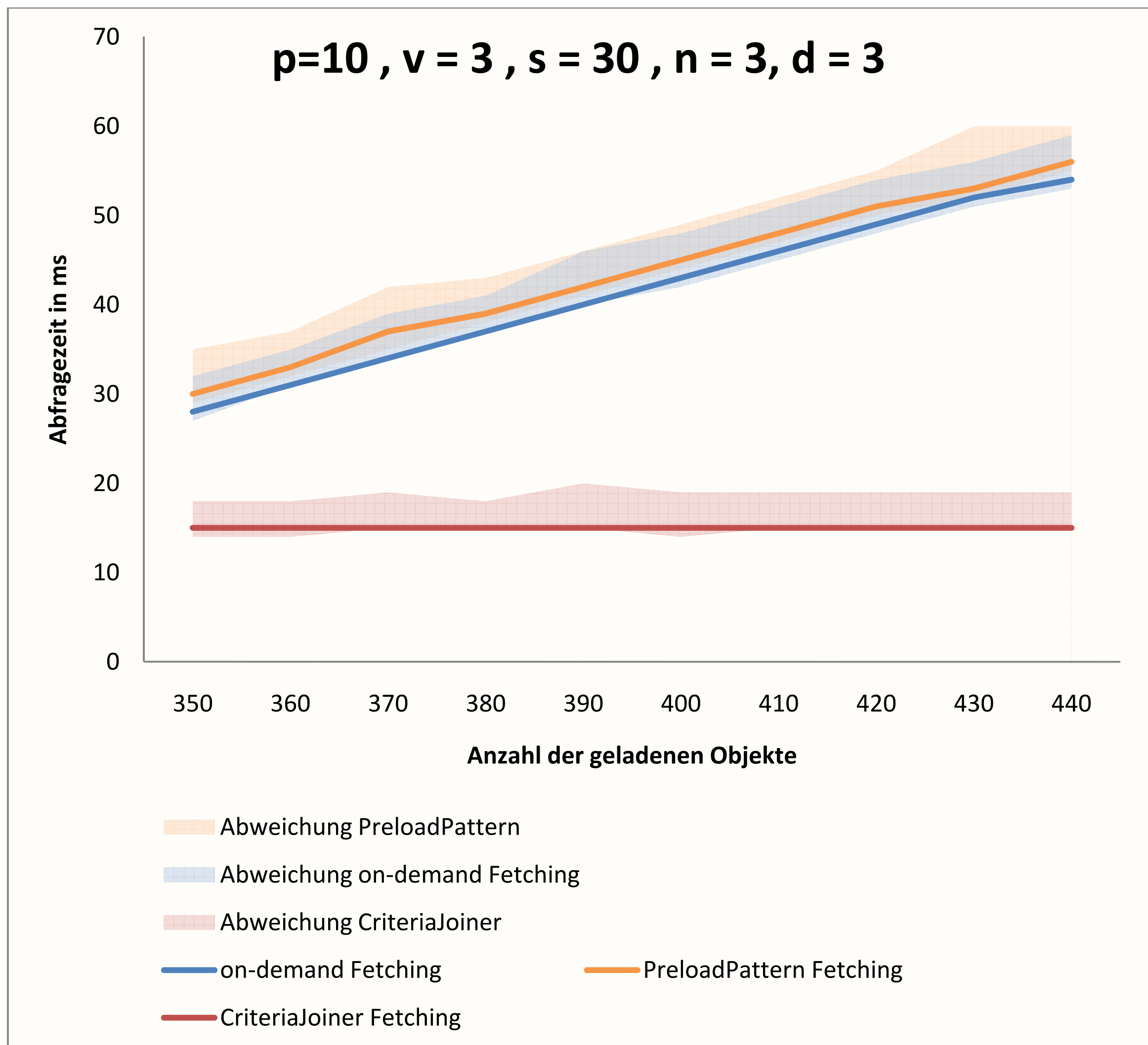


Abbildung 33: Abfragegeschwindigkeit abhängig von der Anzahl der Fakultäten

Aus Abbildung 34 ist abzulesen, dass alle Verfahren unabhängig von der Anzahl der möglichen Vorlesungen eines Studenten sind. Dabei handelt es sich um die Rückrichtung der N:N Beziehung zwischen Vorlesungen und Studenten. Auch hier lässt sich lediglich wieder eine Abhängigkeit zwischen der Objektanzahl und der Laufzeit feststellen.

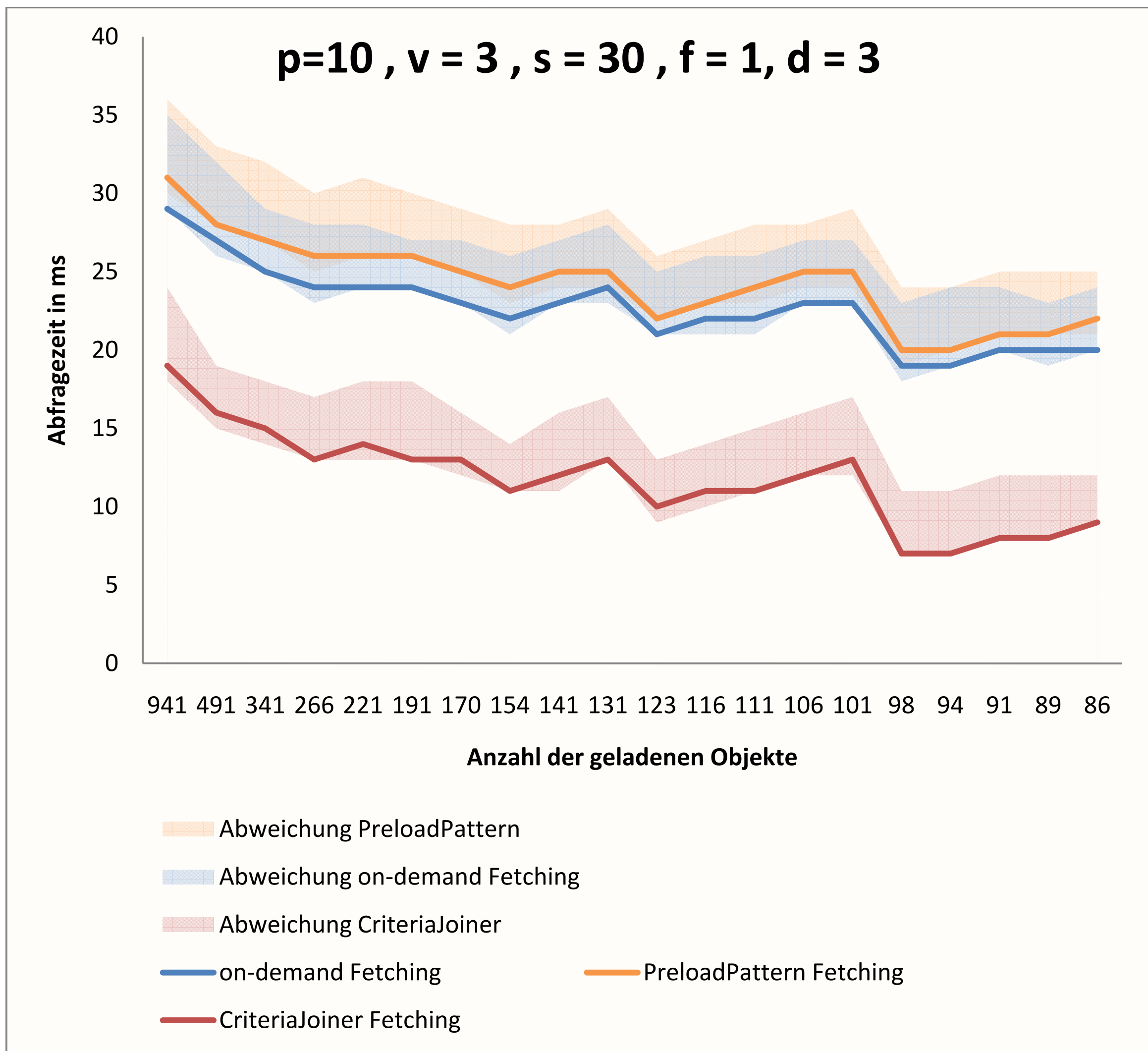


Abbildung 34: Abfragegeschwindigkeit abhängig von der Anzahl der Vorlesungen pro Student

Ein interessanter Aspekt wird in Abbildung 35 deutlich. Hier steigt der Aufwand des CriteriaJoiner schneller als der des on-demand Fetching oder des. Dies geschieht in Abhängigkeit der Studenten einer Vorlesung. Durch den linearen Anstieg der Objektzahl und der vermuteten Invarianz gegenüber der Relationskomplexität ergibt sich die Frage, in welcher Situation die Performance des CriteriaJoiner unter die des PreloadPattern.

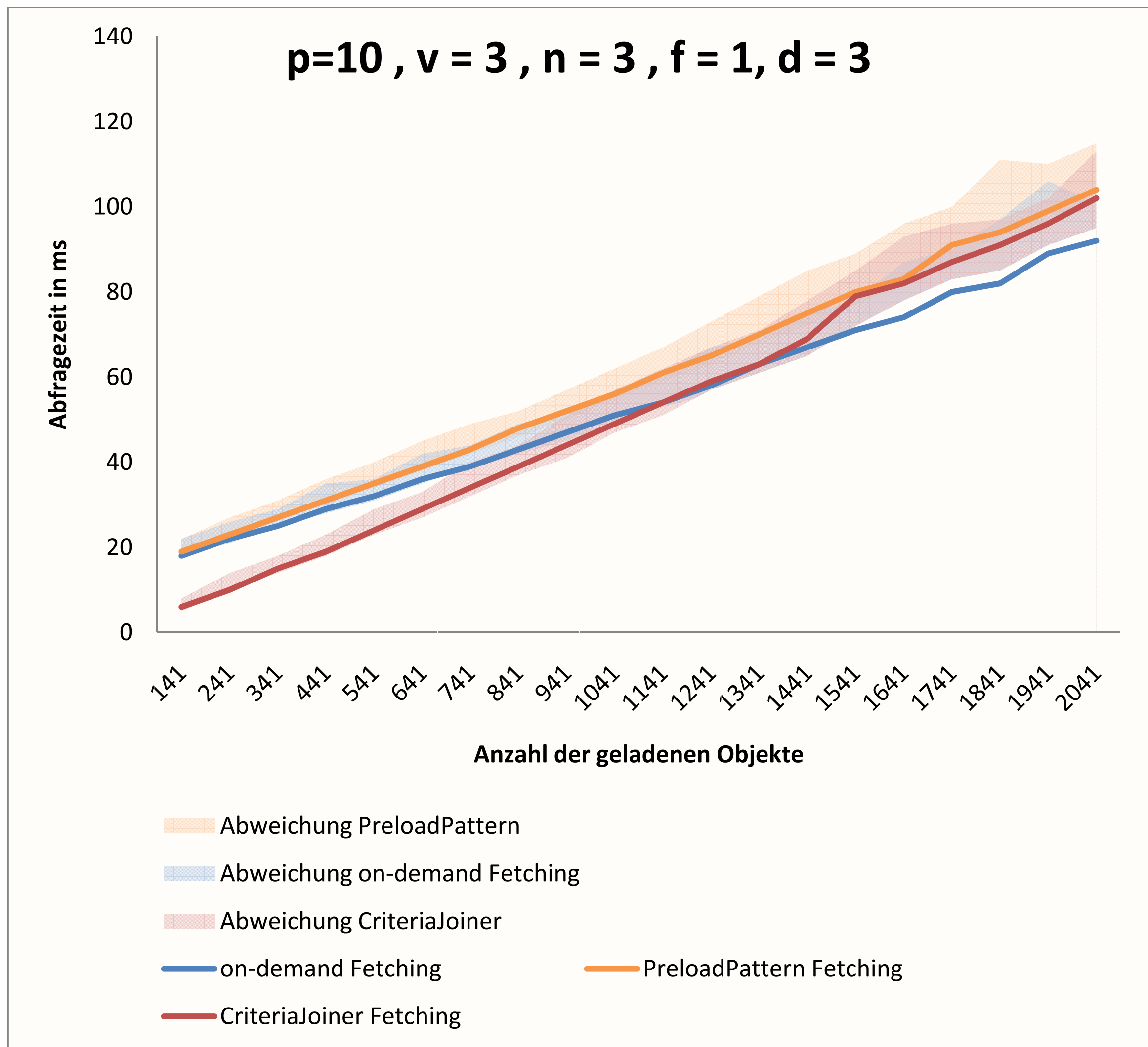


Abbildung 35: Abfragegeschwindigkeit abhängig von der Anzahl der Studenten pro Vorlesung

Aufschluss über diesen Sachverhalt bringt die Aufwandsbetrachtung. Als Grundvoraussetzung für eine solche Betrachtung wird von einem konstanten Zeitaufwand für einen Aufwandsschritt ausgegangen. Weiterhin müssen alle Parameter größer als null gewählt werden. Nunmehr werden weiterhin lediglich die beiden Implementierungen der Preload-Verfahren untersucht. Bei der rekursiven Programmierung des Preload Pattern von Jürgen Kohl kann eine Aufwandsbetrachtung über die Bestimmung der Anzahl von rekursiven Funktionsaufrufen vorgenommen werden.

$$n_{rec} = p_{abs} \cdot v_{rel} + p_{abs} \cdot v_{rel} \cdot s_{rel}$$

Hierbei ist zu beachten, dass das Preload-Verfahren als Eingabeparameter eine Menge von root Objekten erwartet. Daher werden für das hier verwendete Testszenario die Professoren mit

einer separaten Abfrage geladen, bevor das Preload begonnen wird. Damit geht das Laden der Professoren mit einem konstanten Wert in die Aufwandsbetrachtung ein und ist damit vernachlässigbar.

Bei dem Verfahren des CriteriaJoiner hingegen findet der eigentliche Aufwand innerhalb der Datenbank statt. Dabei wird für die Abfrage eine Ergebnisstabelle erstellt, welche die Datenbankrelationen in eine einzelne Tabelle auflöst. Die Anzahl der entstehenden Zeilen innerhalb der Ergebnistabelle wird als Aufwandsabschätzung betrachtet. Auch hier wird ein konstanter Aufwand je Zeile angenommen.

$$n_{cj} = p_{abs} \cdot v_{rel} \cdot s_{rel}$$

Um nun den Aufwand der unterschiedlichen Verfahren vergleichen zu können, werden zusätzlich die Zeiten für das Abarbeiten eines rekursiven Aufrufs bzw. das Erstellen einer Tabellenspalte benötigt. Mit Hilfe dieser beiden Parameter kann dann der Schnittpunkt des Aufwandes der Verfahren bestimmt werden.

$$t_{cj} \cdot p_{abs} \cdot v_{rel} \cdot s_{rel} = t_{rec} \cdot (p_{abs} \cdot v_{rel} + p_{abs} \cdot v_{rel} \cdot s_{rel})$$

$$t_{cj} \cdot p_{abs} \cdot v_{rel} \cdot s_{rel} = t_{rec} \cdot p_{abs} \cdot v_{rel}(1 + s_{rel})$$

$$t_{cj} \cdot s_{rel} = t_{rec} \cdot (1 + s_{rel})$$

Diese vereinfachte Darstellung des Aufwandes beschreibt die Änderung des Verhaltens bei der dritten Objekttiefe. Der Aufwand ist dabei lediglich von der Größe der Collections der dritten Ebene abhängig. Unter der Annahme, dass die Zeit für eine einfache Select-Abfrage kleiner als die Zeit zum Erstellen einer Ergebniszeile ist, kann ein Schnittpunkt im Aufwand bestimmt werden. Um zu ermitteln, wie sich das Verhalten durch unterschiedliche Objekttiefen ändert, kann die Gleichung verallgemeinert werden.

$$t_{cj} \cdot k_0 \cdot k_1 \cdot \ldots \cdot k_n = t_{rec} \cdot \sum_{i=1}^{n} k_0 \cdot k_1 \cdot \ldots \cdot k_i \; ; \; n > 0$$

$$k_i - \mathit{Anzahl\ der\ relativen\ Relationen\ in\ der\ Objekttiefe\ i}$$

$$n - \mathit{Objekttiefe}$$

Um diese Formel zu verifizieren, wurden verschiedene Testparameter erstellt. Um genügend große differenzierbare Resultate zu erzielen, wurde die Anzahl der Professoren auf zehn gesetzt. Weiterhin werden außer dem laufenden Parameter alle anderen auf eins gesetzt. Da die Formel größtenteils Multiplikatoren enthält, können damit die nicht betrachteten Variablen leicht aus der Gleichung entfernt werden. Dadurch kann die Formel jeweils vereinfacht und die theoretischen Ergebnisse mit den experimentell ermittelten Ergebnissen verglichen werden.

$$t_{cj} \cdot k_0 \cdot k_1 = t_{rec} \cdot k_0 \cdot k_1$$

Demnach muss bei einer Objekttiefe von eins, wobei die Anzahl der Vorlesungen, also die Variable k_1, verändert wird, ein fast gleicher Aufwand der beiden Verfahren nachgewiesen werden können. Lediglich durch die unterschiedlichen Zeiten für einen Schritt kommt es zu einem unterschiedlich starken Anstieg. Dieses Verhalten kann durch Abbildung 36 belegt werden.

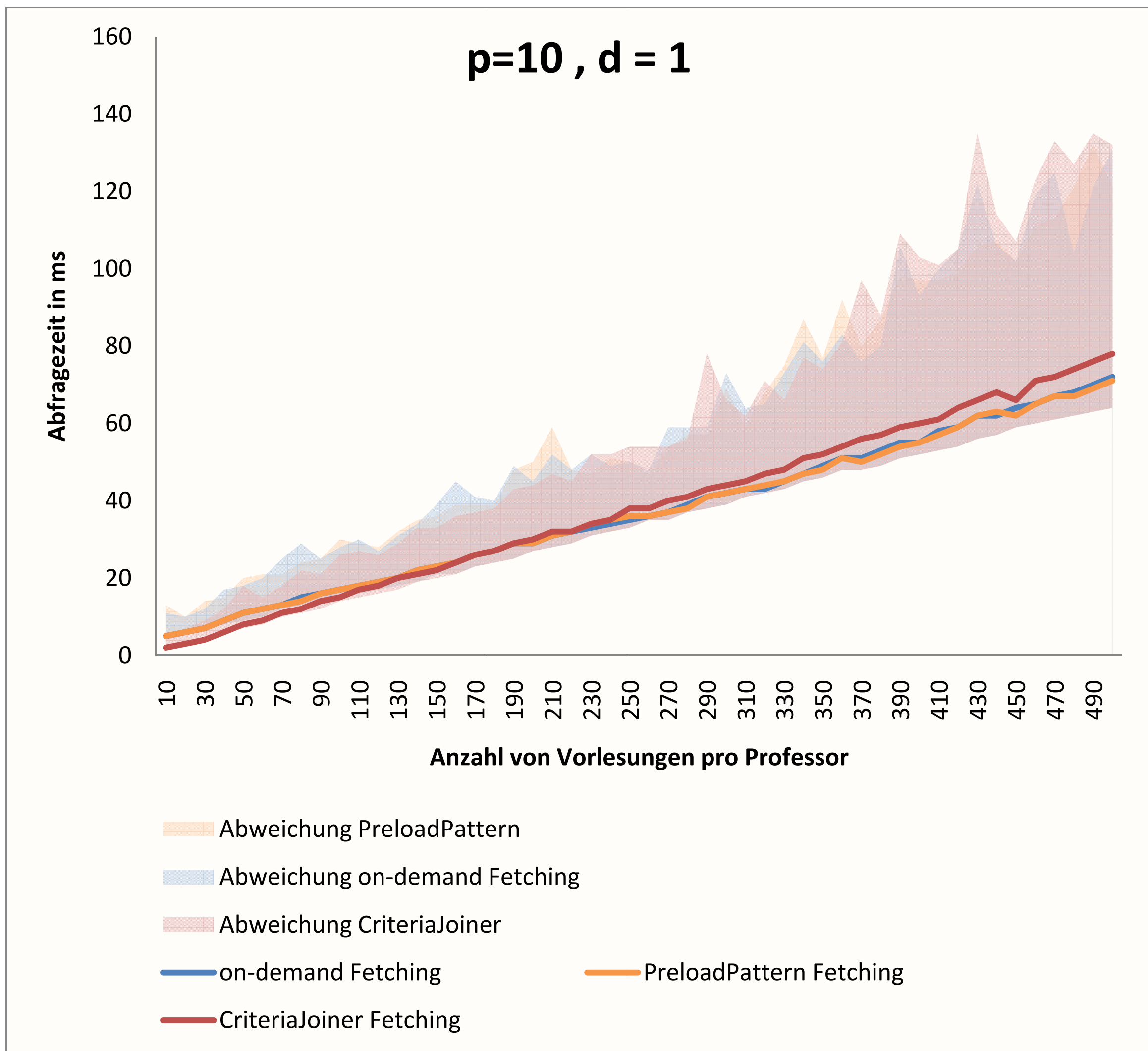


Abbildung 36: Aufwandsbestimmung bei einer Objekttiefe von eins

Für eine Objekttiefe von zwei, bei der ebenfalls die Größe k_1 variabel ist, ergibt sich demnach ein mehr als doppelter Anstieg des Aufwandes für das PreloadPattern als für das CriteriaJoiner Verfahren. Dies spiegeln auch die Ergebnisse der Tests wieder, welche in Abbildung 37 abgelesen werden können.

$$t_{cj} \cdot k_0 \cdot k_1 = 2 \cdot t_{rec} \cdot k_0 \cdot k_1$$

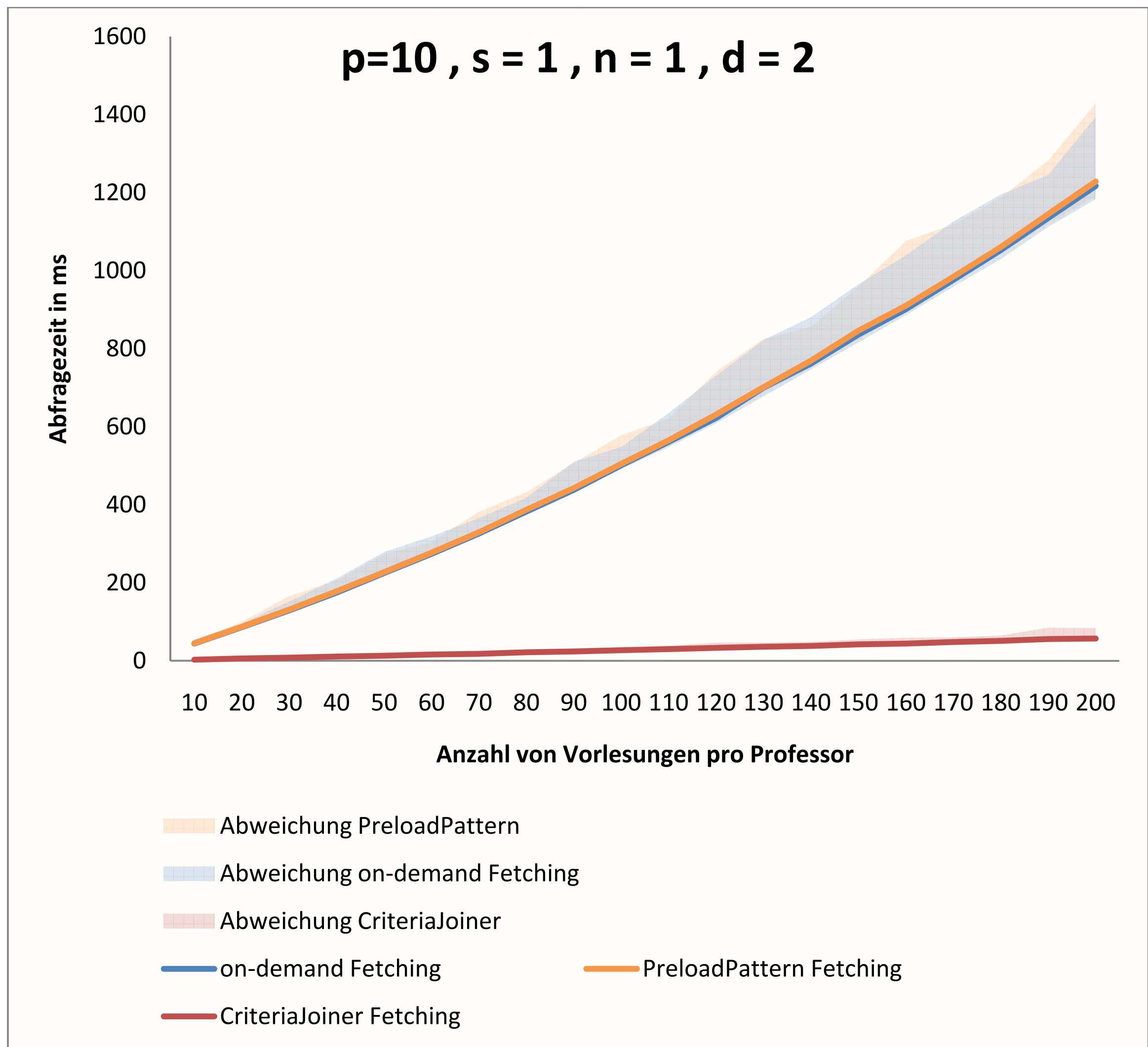


Abbildung 37: Aufwandsbestimmung bei einer Objekttiefe von zwei, v variabel

Weiterhin kann bei einer Objekttiefe von zwei auch der Parameter k_2, also die Anzahl der Studenten pro Vorlesung, variabel durchlaufen werden. Wie bereits erwähnt, muss es bei dieser Kombination der Parameter zu einer Kreuzung des Aufwandes kommen.

$$t_{cj} \cdot k_0 \cdot k_1 \cdot k_2 = t_{rec} \cdot k_0 \cdot k_1 (1 + k_2)$$

$$t_{cj} \cdot k_2 = t_{rec} \cdot (1 + k_2)$$

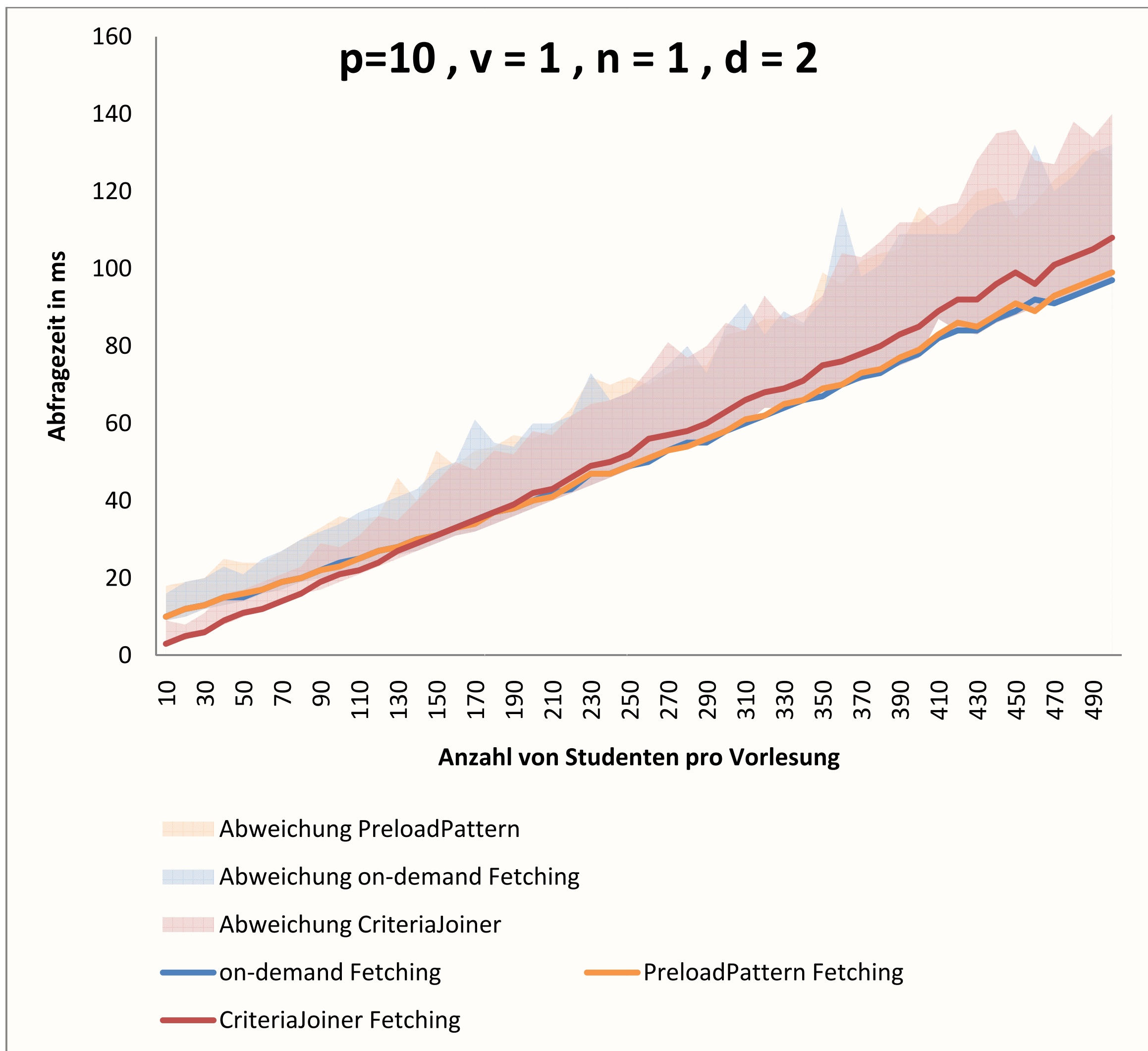


Abbildung 38: Aufwandsbestimmung bei einer Objekttiefe von zwei, s variabel

Dieses Ergebnis wird auch in den Testergebnissen in Abbildung 35 wiedergespiegelt, da lediglich eine geringfügige Veränderung an den Parametern vorgenommen wurde. Mithilfe dieser Gleichung lässt sich nun eine allgemeine Aussage über das Verhalten des CriteriaJoiners erstellen. Lediglich bei einem starken Anstieg der Collectiongröße der letzten besuchten Objektebene verhält sich der CriteriaJoiner nicht schneller als das Verfahren von Jürgen Kohl. In allen anderen Fällen, inklusive dem gleichmäßigen Anstieg aller Parameter, weißt der CriteriaJoiner eine bessere Performance nach.

7.3 Auswertung

Erkennbar ist der deutliche Geschwindigkeitsgewinn bei der Verwendung des CriteriaJoiner. Im schlechtesten Fall erreicht das Verfahren einen Zeitaufwand, der annähernd dem des PreloadPattern entspricht. Dies erklärt sich wie erwartet durch die Verwendung einer einzelnen Anfrage an die Datenbank.

Im Normalfall ist ein linearer Anstieg der Zeitersparnis bei Verwendung des CriteriaJoiners gegenüber den anderen Verfahren bei steigender Objektanzahl zu erkennen. Bei höheren Objekttiefen und damit entsprechenden Relationen kann ein quadratischer Anstieg der Zeitersparnis beobachtet werden. Lediglich beim Anstieg der Größe der tiefsten möglichen Collectiongröße kommt es zu einem Verhalten, welches dem Preload Pattern von Jürgen Kohl entspricht.

Dagegen zeigt sich zwischen dem on-demand Fetching und dem PreloadPattern kaum ein Unterschied in der Zeit. Auch dieses Ergebnis war zu erwarten, da es ähnlich wie beim on-demand-Fetching jeden Knoten erst bei Aufruf nachlädt. Die Tatsache der schlechteren Performance des PreloadPattern ist lediglich in der Komplexität des Algorithmus begründet. Um ein on-demand Fetching nachzubilden, wurde lediglich ein rekursiver Algorithmus geschaffen, der jede Collection bis zu einer bestimmten Tiefe öffnet, um so ein Initialisieren des Objektes durch Hibernate zu erzwingen. Das Preload Pattern von Jürgen Kohl ist ähnlich aufgebaut, verwendet aber noch zusätzliche Abfragen und Sicherheitsaspekte, die das Verfahren verlangsamen. Die Auswertung der Anfragen an die Datenbank haben jedoch gezeigt, dass das PreloadPattern bereits ganze Collections mit einem Statement initialisiert, wodurch die Anzahl der Select-Statements im Vergleich zum on-demand Fetching geringer ist.

Neben der reinen Zeitbetrachtung müssen auch anderer Parameter berücksichtigt werden. So ist die Verwendung des CriteriaJoiner leicht eingeschränkt durch die Verwendung der Criteria Komponente von Hibernate. Da bei dem existierenden System bereits Filter und Query erstellt wurden, müssen diese bei Verwendung des CriteriaJoiner angepasst werden, da diese nicht in das Criteria Konzept integriert werden können. Dies bedeutet einen entsprechenden Mehraufwand. Dagegen kann das Konzept von Jürgen Kohl sehr schnell in die laufende Entwicklung integriert werden, da es die Verwendung des alten Konzeptes unterstützt. Lediglich müssen nach der Anfrage per Filter oder Query die erhaltenen Objekte an die *preload* Methode übergeben werden.

Normalerweise wäre ein Performancegewinn dieser Größenordnung jederzeit einem einfach zu implementierendem Verfahren vorzuziehen. An dieser Stelle handelt es sich aber um ein kommerzielles System, dessen Entwicklungskosten und –zeit eng bemessen sind und daher eine Implementierungsdifferenz von geschätzten zwei Wochen ausschlaggebend für die Verwendung eines leistungsschwächeren Systems sein kann. Daneben steht selbstverständlich die Überlegung, eine Umsetzung des besseren Konzeptes als mögliche Verbesserung am Projektende oder bei einer neuen Version des Produktes zu realisieren.

7.4 Aussagekraft der Ergebnisse

Da eine Zeitmessung ohne Einfluss von äußeren Parametern nicht möglich war, wurde eine statistische Analyse der Verfahren durchgeführt. Als äußere Parameter können Systemauslastung sowie interne Betriebssystemprozesse benannt werden. Diese führten in den Tests zu Abweichungen in den Messergebnissen. Daher wurde jeder Messwert durch 100 Durchgänge ermittelt, aus denen dann jeweils der minimale, maximale sowie der durchschnittliche Wert bestimmt wurde. Diese Messung wurde zehn Mal durchgeführt, wobei die endgültigen Parameter wieder die jeweiligen Mittelwerte dieser Durchläufe sind. Dabei wurde zur Eliminierung von statistischen Ausreißern jeweils der größte und kleinste Wert nicht berücksichtigt. Alle Tests wurden auf demselben Rechner durchgeführt, um Unterschiede in der Performanz der Hardware auszuschließen. Auch wurden die Tests zeitnah und ohne Unterbrechung ausgeführt, um möglichst unterschiedliche Auslastungen durch fremde Einflüsse auszuschließen.

Des Weiteren handelt es sich bei den Implementierungen nicht zwangsläufig um eine Lösung mit der besten Performance. Daher sind diese Ergebnisse zwar maßgeblich und bezogen auf die Größe der Differenz demzufolge auch aussagekräftig, jedoch kann eine geringfügige Verbesserung durch Optimierung des Quellcodes nicht ausgeschlossen werden.

7.5 Resultate

Letztendlich wurde aufgrund der Testergebnisse der CriteriaJoiner übernommen. Dieser erwies sich als überaus leistungsfähig, wenn auch in der Handhabung schwieriger als die anderen Verfahren. Größter Vorteil liegt jedoch in dem linearen Verhalten auch bei größeren Objektmengen. Dagegen spielt der Einfluss der Relationsanzahl und -tiefe eine geringere Rolle, da die verwendete Datenbank für die Zielanwendung keine tiefen Objektgraphen aufweist.

Ein kleines Problem ergab sich bei der Umsetzung des CriteriaJoiner in der Zielapplikation. Dabei wurde festgestellt, dass eine Einschränkung auf eine Menge von Objekten durch Restriktionen, wie sie in Kapitel 6.4 vorgestellt wurden, durch die verwendete Oracle Datenbank auf 1000 Objekte begrenzt ist. Daher war es notwendig, bei höheren Objektanzahlen diese Anfrage in mehrere aufzuteilen. Die Performance des CriteriaJoiner wird dabei nur unerheblich verschlechtert.

Da das Hauptaugenmerk für diesen Anwendungsfall auf dem Vorausladen lag, wurden keine Performanceuntersuchungen für das System unternommen. Jedoch konnte das Lazy Loading Problem vollständig gelöst werden. Das Verarbeiten des Algorithmus in Servicefassaden erlaubte zudem bei anfänglichen Lazy Loading Exception eine genaue Lokalisierung und schnelle Behebung der fehlerhaften Preload Pattern.

Im Gegensatz zum Fall des Lazy Loading, in dem zu wenige Informationen geladen und damit ein Fehler ausgegeben wurde, fällt das Laden von zu vielen Daten in der laufenden Applikation nicht auf. Damit kann nicht gewährleistet werden, dass das Verfahren optimal konfiguriert ist. Dieses Problem kann jedoch nicht oder nur schwer manuell gelöst werden.

Eine Möglichkeit bestände in der schrittweisen Erstellung der Preload Pattern, indem man ausgehend von einem leeren Pattern, für jede Exception ein Pattern hinzufügt, bis die Applikation keine entsprechenden Fehlermeldungen mehr ausgibt. Dieser Zustand kann dann als optimal angesehen werden. Mehrere Gründe sprechen jedoch gegen dieses Verfahren. Hauptsächlich ist die Umsetzung bei einem großen Projekt schwierig und zeitaufwendig, zumal bei jeder neuen Version der Software dieser Prozess wiederholt werden müsste. Zum anderen erfordert es umfangreiche Tests, bei der jede mögliche Situation simuliert wird um sicherzustellen, dass kein Preload Pattern übersehen wurde. Eine Gewährleistung der Stabilität dieser Art ist selbst mit automatisierten Testverfahren unmöglich.

Eine weitere Möglichkeit, welche eher praktikabel erscheint, besteht in der Definition der Preload Pattern durch den GUI Architekten. Dieser entscheidet bereits bei der Gestaltung der Benutzeroberfläche über die verwendeten Objekte und Attribute indem er die dargestellten Information definiert. Allerdings erfordert dieses Vorgehen entsprechende Kenntnisse des GUI Entwicklers über die Vorgänge der Datenhaltungsschicht, insbesondere über das Mapping von Hibernate.

Wünschenswert wäre daher die zukünftige Anwendung eines automatisierten Verfahrens, welches selbstständig die Pattern aktualisiert. Dadurch wäre das System wartungsärmer gegenüber Softwareaktualisierungen. Zusätzlich bietet das Verfahren die Möglichkeit eines Performancegewinns aufgrund der automatischen Anpassung der Applikation auf den Benutzer.

8 Fazit

Diese Arbeit hat gezeigt, dass es eine Vielzahl von Preload-Verfahren mit unterschiedlichsten Ansätzen gibt. Dabei wurde festgestellt, dass jedes Preload-Verfahren unterschiedliche Stärken und Schwächen aufweist. Daher kann kein allgemeingültiges Urteil über diese Verfahren gefällt werden. Die Einsatzfähigkeit eines Verfahrens muss für jeden Anwendungsfall spezifisch bestimmt werden. Daher kann diese Arbeit nicht die Frage des richtigen Preload-Verfahrens lösen. Lediglich eine Einführung in das Themengebiet sowie eine Entscheidungshilfe kann mit den hier dargestellten Fakten gegeben werden.

Bevor allerdings Preload-Verfahren zum Einsatz kommen, sollten die in dieser Arbeit erläuterten Alternativen in Betracht gezogen werden. So bietet das Lazy Loading mitunter enorme Performancegewinne, ohne dabei einen umfangreichen Implementierungsaufwand in Kauf zu nehmen. Erst bei größeren Systemen, wobei sich die Größe des Systems an dieser Stelle durch den Aufwand gegenüber der Datenbank bestimmt, lohnt sich ein Preload-Verfahren. Ebenfalls erscheint ein Preloading sinnvoll, wenn ein Session-Handling zu groß oder unübersichtlich wird oder ein ständiger Datenbankzugriff nicht gewährleistet werden kann, so wie es bei dem Anwendungsfall dieser Arbeit gegeben war.

Auch dann ist die Implementierung eines eigenen Preload-Verfahrens noch nicht zwingend notwendig. Oftmals reicht schon die Verwendung der Hibernate eigenen Mittel, um ein einfaches und doch effizientes Preload zu realisieren. Auch an dieser Stelle hängt es stark davon ab, welche Daten und in welchen Mengen diese Daten geladen werden.

Für diese beiden Spezialfälle lohnt sich die Implementierung eines manuellen Preload-Verfahrens. Diese sind sehr schnell umsetzbar und leicht zu verstehen. Damit kann auch die Abgabe an Kontrolle, die der Entwickler im Gegensatz zu automatisierten Ansätzen im großen Maße verliert, begrenzt werden. Dies erleichtert die Umstellung des Systems. Jedoch ist der Performancegewinn gegenüber dem on-demand Fetching nicht immer gewährleistet, wie die Tests in dieser Arbeit gezeigt haben. Dieser Sachverhalt ist jedoch bei der Entscheidung für den Einsatz von Preload-Verfahren bei diesen Fällen zweitrangig gewesen.

Für alle anderen Fälle steht bei der Einführung von Preload-Verfahren jedoch die Performance im Vordergrund. Ein solcher Performancegewinn ist am stärksten bei den sich selbst optimierenden Verfahren, da sie ohne ständigen Wartungsaufwand auch bei

unterschiedlichen Anwendungsfällen eine Verbesserung vorweisen können. Ebenfalls sind große Systeme so unübersichtlich, dass manuelle Verbesserungen in vielen Fällen nicht so erfolgreich sind, wie es automatisierte Verfahren sein könnten.

Weiterhin sollte beachtet werden, dass das spätere Einführen eines Preload-Verfahrens zwangsläufig Umbaumaßnahmen im System notwendig machen. Je nach Art und Aufbau des Systems ist damit ein nicht unbeträchtlicher Aufwand verbunden. Daher lohnt es sich bereits vor dem Projektstart in der Planungsphase die Verwendung eines Preload-Verfahrens in Betracht zu ziehen. Sollte sich später zeigen, dass das gewählte Verfahren nicht den gewünschten Effekt erzielt, kann es leicht ausgetauscht werden. Dabei ist der Austausch wesentlich einfacher als das spätere Hinzufügen eines Verfahrens.

9 Literaturverzeichnis

[1] Edgar F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communikations of the ACM*, pp. 377-387, Juni 1970.

[2] C. J. Date, *An Introduction to Database Systems*, 8th ed.: Pearson Addison, 2003.

[3] Alfons Kemper und André Eickler, *Datenbanksysteme. Eine Einführung*, 5th ed. München: Oldenbourg Wissenschaftsverlag, 2004.

[4] Gunter Saake und Kai-Uwe Sattler, *Datenbanken & Java. JDBC, SQLJ, ODMG und JDO*, 2nd ed.: Dpunkt Verlag, 2003.

[5] R. G. Cattell et al., *The Object Data Standard: ODMG 3.0*, 1st ed., Morgan Kaufmann, Ed., 2000.

[6] Can Türker und Gunter Saake, *Objektrelationale Datenbanken. Ein Lehrbuch*, 1st ed.: Dpunkt Verlag, 2005.

[7] Gunter Saake, Ingo Schmitt und Can Türker, *Objektdatenbanken - Konzepte, Sprachen, Architekturen*, 1st ed. Bonn, Deutschland: International Thomson Publishing, 1997.

[8] Edgar F. Codd, *The relational model for database management: version 2*, Addison-Wesley Longman Publishing Co, Ed., 1990.

[9] Andreas Heuer, *Objektorientierte Datenbanken. Konzepte, Modelle, Standards und Systeme*, 2nd ed. München: Addison-Wesley, 1997.

[10] Rainer Unland, *Objektorientierte Datenbanken. Konzepte und Modelle*. Bonn, Deutschland: Thomson Publishing, 1995.

[11] Robert F. Beeger, Arno Haase und Stefan Roock, *Hibernate. Persistenz in Java-Systemen mit Hibernate 3*, 1st ed.: Dpunkt Verlag, 2007.

[12] Deborah J. Armstrong, "The quarks of object-oriented development," *Commun*, no. 49, pp. 123-128, Februar 2006.

[13] Christopher Ireland, David Bowers, Michael Newton und Kevin Waugh, "A Classification of Objekt-Relational Impendance Mismatch," in *First International Conference on Advances in Databases*, 2009, pp. 36-43.

[14] Craig Russell, "Bridging the object-relational divide," *Queue. 6*, no. 3, pp. 18-28, Juli 2008.

[15] Christian Bauer und Gavin King, *Java-Persistence mit Hibernate*.: Hanser Verlag, 2007.

[16] de.Wikipedia.org. (2011, Februar) Wikipedia.org. [Online]. http://de.wikipedia.org/wiki/Hibernate_(Framework)

[17] Richard Oates, Thomas Langer, Stefan Wille, Torsten Lueckow und Gerald Bachlmayr, *Spring & Hibernate. Eine praxisbezogene Einführung*.: Hanser Verlag, 2006.

[18] Sebastian Hennebrüder, *Hibernate, Das Praxisbuch für Entwickler*.: Galileo Press, 2007.

[19] Heiko Kirsche, *Persistenz in Objekt-orientierten Programmiersprachen*. Berlin: Logos, 1997.

[20] Andreas Holubek, Rudolf Jansen, Robert Munsky und Eberhard Wolff, *Java-Persistenzstrategien*.: Entwickler.Press, 2004.

[21] Michael Plöd, *JPA 2.0: Die Java Persistence API - Grundlagen und Praxiswissen*.: Dpunkt Verlag, 2011.

[22] Wook-Shin Han, Kyu-Young Whang und Yang-Sae Moon, "A Formal Framework for Prefetching Based on the Type-Level Access Pattern in Object-Relational DBMSs," *IEEE Transactions on Knowledge and Data Engineering*, no. 17, pp. 1436-1448, Oktober 2005.

[23] Jürgen Kohl, "Faules Vorausladen," *Java Magazin*, no. 4/2008, pp. 14-19, März 2008.

[24] Benjamin Winterberg. (2009, September) bwinterberg.blogspot.com. [Online]. http://bwinterberg.blogspot.com/2009/09/hibernate-preload-pattern.html

[25] E.E. Chang und R.H. Katz, "Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented DBMS," in *Proceedings of International Conference on Management of Data*, Portland, Oregon, May 1989, pp. 348-357.

[26] Won Kim, "Architecture of the ORION next-generation database system," in *IEEE Transactions on Knowledge and Database Engineering 2*, 1990.

[27] C. Lamp, "The ObjectStore System," in *Communications of the ACM 34*, 1991, pp. 50-63.

[28] Wook-Shin Han, Yang-Sae Moon und Kyu-Young Whang, "PrefetchGuide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational DBMSs," *Information Sciences: an International Journal*, no. 152, pp. 47-61, Juni 2003.

[29] Mark Palmer und Stanley B. Zdonik, "Fido: a chache that learns to fetch," in *Proceedings of 17th International Conference on Very Large Data Bases*, Barcelona, Spanien, 1991, pp. 255-264.

[30] T. Potter, "Storing and Retrieving Data in a Parallel Distributed Memory System," State Univerity of New York, Birmingham, PhD Thesis 1987.

[31] P. Kanerva, "Sparse Distributed Memory," *MIT Press*, 1988.

[32] Philip A. Bernstein, Shankar Pal und David Shutt, "Context-based prefetch for implementing objects on relations," in *Proceedings of 25th International Conference on Very Large Data Bases*, Edinburgh, Scotland, September 1999, pp. 327-338.

[33] William R. Cook Ali Ibrahim, "Automatic Prefetching by Traversal Profiling in Object Persistence Architectures," in *ECOOP 2006 - Object-Oriented Programming*, Nantes, Frankreich, Juli 2006, pp. 50-73.

[34] Henning Wolf und Wolf-Gideon Bleek, *Agile Softwareentwicklung*, 2nd ed.: Dpunkt Verlag, 2010.

[35] Roman Pichler, *Scrum - Agiles Projektmanagement erfolgreich einsetzen*, 1st ed.: Dpunkt Verlag, 2007.

[36] hsql Development Group. (2011, Februar) HyperSQL. [Online]. http://www.hsqldb.org

[37] George Reese, *Database Programming with JDBC and Java*, 2nd ed.: O'Reilly, 2000.

[38] Ceki Gulcu, *The Complete Log4j Manual: The Reliable, Fast and Flexible Logging Framework for Java*.: QOS.ch, 2003.

[39] Bernd Östereich und Christian Weiss, *APM - Agiles Projektmanagement: Erfolgreiches Timeboxing für IT-Projekte*, 1st ed.: Dpunkt Verlag, 2007.

[40] Mike Cohn, *Agile Softwareentwicklung: Mit Scrum zum Erfolg!*, 1st ed. München: Addison-Wesley, 2010.

[41] Won Kim, *Introduction to Object-Oriented Databases*.: Mit Pr, 1990.

[42] Chong-Mok Park, Michael J. Carey und Stefan Deßloch, "MAJOR: A Java Language Binding for Object-Relational Databases.," in *Advances in Persistent Object Systems, Proceedings of the 8th International Workshop on Persistent Object Systems and Proceedings of the 3rd International Workshop on Persistence and Java*, Tiburon, California, 1998, pp. 112-122.

[43] Oracle Corp., *Oracle Call Interface Programmer’s Guide Release 8.0*., 1997.

[44] de.Wikipedia.org. (2011, Januar) Schichtenarchitektur. [Online]. http://de.wikipedia.org/wiki/Schichtenarchitektur

[45] Wolfgang Keller und Jens Coldewey, "Relational Database Access Layers: A Pattern

Language," in *Collected Papers from the PLoP'96 and EuroPLoP'96 Conferences*, Washingtun University, Department of Computer Science, Februar 1997.

[46] Wolfgang Keller, "Mapping Object to Tables. A Pattern Language. – Project ARCUS," 1997.

[47] Wolfgang Keller, "Persistence Options for Object-Oriented Programs," 2004.

[48] Aaron Fontaine, Anh-Thu Truong und Thomas Manley, "A Survey of Strategies for Object Persistence," 2006.

[49] Muralidhar Subramanian und Vishu Krishnamurthy, "Performance Challenges in Object-Relational DBMSs," 1999.

[50] K.M. Curewitz, P. Krishnan und J.S. Vitter, "Practical Prefetching via data compression," in *Proceedings of International Conference on Management of Data, ACM SIGMOD*, Washington, DC, 1993, pp. 257-266.

10 Eigenständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig, ohne unzulässiger Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Magdeburg, den 04.03.2011

Michael Lipaczewski