Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Bachelorarbeit

**Diana Topko**

## Object-Mapper für NoSQL-Datenbanksysteme

*Fakultät Technik und Informatik*
*Studiendepartment Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

Diana Topko

# Object-Mapper für NoSQL-Datenbanksysteme

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 23. August 2018

**Diana Topko**

**Thema der Arbeit**
Object-Mapper für NoSQL-Datenbanksysteme

**Stichworte**
Objekt-Mapper, Persistenz-Framework, NoSQL-Datenbanksystem, Annotationen, Datenmodellierung, Caching-Strategien, Hibernate, DataNucleus, Morphia

**Kurzzusammenfassung**
NoSQL-Datenbanksysteme bieten neue Möglichkeiten für die Datenspeicherung und für die Datenverwaltung. Dabei erfassen NoSQL ein weites Spektrum von Datenbanken. Es besteht jedoch keine standardisierte Schnittstelle für die NoSQL-Datenbankzugriffe. Wie kann ein Objektmodell einer Anwendung auf ein Datenbankmodell einer NoSQL-Datenbank abgebildet werden? Diese Frage liegt im Fokus dieser Arbeit. Als eine Lösung werden Object-Mapper für NoSQL-Datenbanksysteme erörtert. Um einen Überblick über derzeit populären Object-Mapper zu verschaffen, werden die Basisfunktionalitäten drei ausgewählter Object-Mapper Frameworks beschrieben. Weiterhin wird ein praktischer Performanz-Vergleich der beschriebenen Mapper durchgeführt, indem die Speichergeschwindigkeit sowie die Lesegeschwindigkeit von denen gemessen und verglichen wird.

**Title of the paper**
Object mapper for NoSQL database systems

**Keywords**
Object-Mapper, persistence framework, NoSQL database system, annotations, data modelling, caching strategies, Hibernate, DataNucleus, Morphia

**Abstract**
NoSQL datastores provide new data storage and data management possibilities. As a result, NoSQL covers a broad range of datastores. However, NoSQL datastores do not provide a uniform access interface. How to map an application object model to a database model of NoSQL database? That is the question which this bachelor thesis is focused on. Object mappers for NoSQL database systems are discussed as a solution. The basis functionality of three chosen object mappers is described with a goal to provide an overview of currently popular object mapper frameworks. Furthermore, an empirical performance comparison of the described object mappers is carried out by measuring and comparing the data reading and data storing velocity.

# Danksagung

Ich danke Herrn Prof. Dr. Olaf Zukunft und Herrn Prof. Dr. Stefan Sarstedt für die Bereitschaft, die vorliegende Arbeit zu betreuen. Außerdem danke ich Ihnen für die fachliche und menschliche Unterstützung.

Ein weiterer Dank geht an Ksenia Kovalenko, Mariam Guseva, Amrei Kleeberg und Edris Taieb, die mir während dieser Zeit mit viel Geduld und Verständnis zur Seite standen und die Aufgabe auf sich genommen haben meine Arbeit auf Grammatikfehlern zu untersuchen.

# Inhaltsverzeichnis

# 1. Einleitung

NoSQL-Datenbanksysteme sind ein relativ neuer, schnell an Bedeutung gewinnender Ansatz in der Datenspeicherung und Datenverwaltung. Object-Mapper für NoSQL-Datenbanksysteme bilden eine Datenbankabstraktionsschicht für NoSQL-Datenbanken. Diese ermöglicht den Entwicklern das Abbilden der Objektmodelle auf die Datenbankmodellen durch eine gemeinsame Schnittstelle, ohne explizit auf die Details der Datenspeicherung in der Datenbank eingehen zu müssen.

## 1.1. Motivation

Die NoSQL-Datenbanken sind sehr vielfältig und bieten im Gegensatz zu den SQL-Datenbanken keine standardisierte Schnittstelle für die Datenbankzugriffe. Selbst die grundlegende Operationen wie das Lesen oder das Schreiben unterscheiden sich sowohl syntaktisch als auch in dem Funktionsumfang. Dadurch entsteht eine Herausforderung für das Object-NoSQL Data Mapping.

Object-Mapper für NoSQL-Datenbanken sind genauso vielseitig wie auch die NoSQL-Datenbanksysteme. In erster Linie unterscheiden sich die Object-Mapper Frameworks durch die von denen implementierte Spezifikationen. Die bekanntesten Spezifikationen sind Java Persistence API (JPA) und Java Data Objects (JDO). Darüber hinaus existieren sowohl solche Object-Mapper Frameworks, die eine einheitliche Schnittstelle für einen Zugriff auf mehrere NoSQL-Datenbanksysteme bereitstellen (zB. Hibernate und DataNucleus Frameworks), als auch solche Object-Mapper Frameworks, welche auf ein bestimmtes Datenbanksystem ausgerichtet sind (zB. Morphia und Objectify Frameworks).

## 1.2. Zielsetzung

Das Ziel dieser Bachelorarbeit ist es einen Überblick über Object-NoSQL Data Mapping zu verschaffen. Um dieses Ziel zu erreichen, sollen die von Object-Mappers implementierte Persistenz-Spezifikationen betrachtet werden. Zudem sollen die Notationen des Object-

Mapping erörtert werden. Darüber hinaus soll es auf die Datenmodellierung für verschiedene NoSQL-Datenbankkategorien eingegangen werden.

Ein weiteres Ziel der Arbeit ist es drei ausgewählte Object-Mapper Frameworks sowohl im Bezug auf deren Funktionalitäten als auch im Bezug auf deren Performanz miteinander zu vergleichen. Dabei soll die Performanz durch die Geschwindigkeit bestimmt werden, mit der die Daten in der Datenbank abgelegt als auch aus der Datenbank geladen werden.

Die Auswahl der Object-Mapper Repräsentanten für den Vergleich soll durch die Persistenz-Spezifikationen sowie durch die unterstützten Datenbanksysteme bestimmt werden.

## 1.3. Gliederung

In Kapitel 2 werden die Grundlagen des NoSQL-Konzeptes sowie die Gründe für den Einsatz von NoSQL-DBMS erläutert. Außerdem wird ein Einblick in die Namensbildung von NoSQL-Datenbanken genommen. Zudem werden zwei gängigen Persistenz-Spezifikationen vorgestellt-die JPA- und die JDO-Spezifikation.

In Kapitel 3 wird der theoretische Hintergrund des Object-Mapping beleuchtet mit dem Fokus auf NoSQL-DBMS . Dieser umfasst die Gründe für den Einsatz von Object-Mappers, die möglichen Notationen des Object-Mapping sowie Object-Mapper Architektur. Ferner wird die Datenmodellierung für vier verschiedene NoSQL-Datenbankkategorien betrachtet. Anschließend werden die einzelnen Object-Mapper Frameworks beschrieben mit dem Ziel, die Gemeinsamkeiten und die Unterschiede herauszufinden.

In Kapitel 4 wird die vergleichende Untersuchung der drei ausgewählten Object-Mappers beschrieben. Zudem wird die Auswahl der Object-Mappers erklärt, sowie das Szenario für die Untersuchung erläutert. Darüber hinaus wird die Testanwendung vorgestellt.

In Kapitel 5 werden die Messdaten aus in Kapitel 4 beschriebener Untersuchung tabellarisch dargestellt. Auf der Grundlage des tabellarischen Vergleiches werden die Ergebnisse der Untersuchung zusammengefasst und analysiert.

Abschließend fasst Kapitel 6 die wesentlichen Aspekte dieser Arbeit zusammen.

# 2. NoSQL

Neben dem bereits weitgehend etablierten, relationalen Datenbankmodell gibt es in der Datenspeicherung und Datenverwaltung heutzutage neue Ansätze, die den mit der Zeit gewachsenen und stetig wachsenden Anforderungen besser gerecht werden. Hier sei an dieser Stelle insbesondere auf die NoSQL-Datenbanken und deren starke Popularitätszunahme in den letzten Jahren hingewiesen. In diesem Kapitel soll ein grundlegendes Verständnis über NoSQL-Datenbanken geschaffen werden. Darüber hinaus werden die Schnittstellen erläutert, welche den Zugriff auf die Daten in NoSQL-Datenbanken gewährleisten.

## 2.1. NoSQL-Datenbanken

Der Begriff “NoSQL” wurde erstmals im Jahre 1998 für eine von Carlo Strozzi entwickelte Datenbank ohne Structured Query Language (SQL)-Schnittstelle verwendet. Der Entwickler selbst weist jedoch auf seiner Homepage daraufhin, dass die von ihm verwendete Bezeichnung keinen Bezug zu der später entstandenen NoSQL-Bewegung hat, in der NoSQL als Konzept, welches sich vom relationalen Modell abspaltet, in Verbindung steht. Diese Bewegung hätte sich laut ihm logischerweise “NoREL” nennen sollen, da aus der Tatsache heraus, dass das Konzept nicht relational ist, implizit hervorgeht, dass keine SQL-Schnittstelle vorhanden sein kann[28].

Einige Jahre später im Jahre 2009 bekam der Begriff NoSQL eine zum Akronym passende Bedeutungserweiterung hinzu: nicht nur SQL (vom englischen „not only SQL“). Diese wurde vom Webentwickler Johan Oskarsson eingeführt. [10]. NoSQL-Datenbanken schließen nämlich die Benutzung von SQL-Datenbanken nicht aus, sondern bieten zusätzliche Möglichkeiten und können auch parallel zu SQL-Datenbanken eingesetzt werden. Aus diesem Grund heraus hat sich die Bedeutungserweiterung in der Fachwelt durchgesetzt.

Nichtsdestotrotz gibt es in der Fachwelt keine einheitliche Definition des NoSQL-Begriffs. Grundsätzlich werden als NoSQL die Datenbanksysteme angesehen, welche folgende Eigenschaften aufweisen [20, S. 222] :

- das Datenbankmodell ist nicht relational
- Ausrichtung auf vertikale und horizontale Skalierbarkeit [1]
- schwache bis gar keine Schema-Restriktionen
- einfache Datenreplikation
- einfacher Zugriff über Application Programming Interface (API)
- anderes Konsistenzmodell als ACID [2] (z.B. BASE[3])

Die Schemafreiheit ist ein großer Vorteil, welcher erlaubt der Applikation das Datenmodell zu bestimmen. Das Thema Datenmodellierung in NoSQL-Datenbanken wird im Abschnitt 3.4 weiter betrachtet. Solche Schemaverwaltung bietet den agilen Entwickler zahlreiche Möglichkeiten das Datenmodell während der Entwicklung zu erzeugen und anzupassen.

Analysiert man die NoSQL-Datenbanken nach dem CAP-Theorem [4] , so sind in den meisten Fällen die Verfügbarkeit und die Partitionstoleranz gewährleistet. Konsistenz gehört hingegen zu den Stärken von relationalen Datenbankmanagementsystemen.

Generell lassen sich NoSQL-Datenbanken je nach der Art, wie die Daten gehalten werden, in vier Kategorien unterteilen:

- Schlüssel-Wert Datenbanken (Key-Value Stores),
- Spalten- oder Spaltenfamilien-Datenbanken (Column Based Stores),

[1] Unter vertikaler Skalierung (Scale-up) versteht man das Aufrüsten vorhandener Server durch bessere Hardwarekomponenten (CPU, Hauptspeicher, Netz, Platten usw.), wohingegen die horizontale Skalierung(Scale-out) die Erhöhung der Anzahl der verfügbaren Server, die in einer verteilten Datenbank miteinander vernetzt werden, bezeichnet. [9]

[2] ACID (Atomicity, Consistency, Isolation, Durability) bedeutet, dass die Datenbank folgende Eigenschaften aufweist: Atomarität (eine Folge von Datenbankoperationen (Transaktion) wird ganz oder gar nicht ausgeführt), Konsistenz (zu dem Zeitpunkt, wenn eine Transaktion beendet ist, muss die Datenbank sich im konsistenten Zustand befinden auch wenn die Konsistenz während der Transaktion verletzt war), Isolation (gleichzeitig ablaufende Transaktionen müssen voneinander isoliert sein und nicht einander beeinflussen), Dauerhaftigkeit (nach dem Beenden einer Transaktion müssen Daten in der Datenbank dauerhaft gespeichert sein, was auch im Fall eines Systemfehlers garantiert werden muss).

[3] BASE (Basically Available, Soft State, Eventually Consistent): Es wird erlaubt, dass replizierte Knoten zwischenzeitlich unterschiedliche Datenversionen halten und erst zeitlich verzögert aktualisiert werden. [20, S. 148]

[4] CAP-Theorem: Sagt aus, dass in einem massiv verteilten Datenhaltungssystem jeweils nur zwei Eigenschaften aus den drei der Konsistenz (C), Verfügbarkeit (A) und Ausfalltoleranz (P) garantiert werden können. [20, S. 149]

- dokumentenorientierte Datenbanken (Document Stores),
- Graphdatenbanken (Graph Stores).

In den Schlüssel-Wert DBMS sind die Daten, wie der Name es erahnen lässt, unter eindeutigen Schlüsseln abgelegt. Eine Datenbank mit den folgenden Eigenschaften wird Schlüssel-Wert Datenbank genannt [20, S. 224]:

- es gibt eine Menge von Identifikatoren: die Schlüssel.
- zu jedem Schlüssel gibt es genau ein assoziiert-deskriptives Datenobjekt, welches den Wert zum zugehörigen Schlüssel darstellt.
- mit der Angabe des Schlüssels kann der zugehörige Wert aus der Datenbank abgefragt werden.

Zu dieser Kategorie gehören Database Management System (DBMS) wie Redis, Riak und viele andere.

Spaltenfamilien-DBMS erweitern das Konzept des Schlüssel-Wert DBMS . Diese sind strukturell besser konzipiert. Es werden für eine Zeile selten alle Spalten benötigt, aber es gibt Gruppen von Spalten, die häufig zusammen gelesen werden. Aus diesem Grund ist es sinnvoll, für die Optimierung des Zugriffs, Daten in solchen Gruppen von Spalten - sogenannte Spaltenfamilien - als Speichereinheit zu strukturieren. Bei Spaltenfamilien-DBMS werden die Daten nicht in relationalen Tabellen, sondern in erweiterten und strukturierten mehrdimensionalen Schlüsselräumen gespeichert. Das am meisten eingesetzte Datenmodell ist Googles entwickelte Big Table, ein Datenbankmodell für die verteilte Speicherung von strukturierten Daten. Als Beispiel kann man hier Apache Cassandra DBMS nennen.

Dokumentenorientierte DBMS sind DBMS mit folgenden Eigenschaften [20, S. 230]:

- sie ist eine Schlüssel-Wert Datenbank
- die gespeicherten Datenobjekte als Werte zu den Schlüsseln werden Dokumente genannt; die Schlüssel dienen der Identifikation.
- die Dokumente enthalten Datenstrukturen in der Form von rekursiv verschachtelten Attribut-Wert-Paaren ohne referenzieller Integrität.

Diese Datenstrukturen ähnlich zu den anderen NoSQL-Datenbanken sind schemafrei, d.h. in jedem Dokument können beliebige Attribute verwendet werden, ohne diese zuerst innerhalb eines Schemas zu definieren.

Beispiele zu dieser Datenbank-Kategorie sind MongoDB und CouchDB.

Die Graphdatenbanken unterscheiden sich von drei anderen Kategorien indem sie ein strukturierendes Schema des Eigenschaftsgraphes haben. Der Eigenschaftsgraph besteht aus Knoten und aus gerichteten Kanten.

Die Knoten haben bestimmte Eigenschaften und die Beziehungen zwischen den Knoten werden durch die Kanten beschrieben. Die Kanten selbst können ebenfalls Eigenschaften aufweisen. Eine Graphdatenbank ist ein DBMS mit folgenden Eigenschaften:

- die Daten und/oder das Schema werden als Graphen oder graphähnlichen Strukturen abgebildet, welche das Konzept von Graphen generalisieren (z. B. Hypergraphen).
- Datenmanipulationen werden als Graph-Transformationen ausgedrückt, oder als Operationen, welche direkt typische Eigenschaften von Graphen ansprechen (z. B. Pfade, Nachbarschaften, Subgraphen, Zusammenhänge, etc.).
- die Datenbank unterstützt die Prüfung von Integritätsbedingungen, welche die Datenkonsistenz sicherstellt. Die Definition von Konsistenz bezieht sich direkt auf Graphstrukturen (z. B. auf Knoten- und Kantentypen, Attribut-Wertebereiche und referenzielle Integrität der Kanten).

Graphdatenbanken kommen beispielsweise überall dort zum Einsatz, wo Daten in Netzwerken organisiert sind. [20, S. 238]

Eine Graphdatenbank Beispiel ist Neo4J Datenbank.

## 2.2. Popularität von NoSQL-Datenbanken

Wir leben im “digitalen Zeitalter”. Es wird angenommen, dass es der Menschheit im Jahre 2002 das erste Mal möglich war, mehr Informationen digital als im Analogformat zu speichern [13, S. 60-65]. Heute werden in zwei Minuten mehr Fotos gemacht als im ganzen 19. Jahrhundert [15]. Den Trend stetig wachsender Datenmengen und steigender Anzahl an Benutzern kann man anhand von Twitter gut erkennen: Waren es 2010 noch etwa 65 Millionen Tweets pro Tag, so hat sich die Anzahl innerhalb eines Jahres mehr als verdreifacht. Im Jahr 2011 wurden etwa 200 Millionen Tweets pro Tag ermittelt. [14, S. 220] Die Menge an Daten wächst exponentiell weiter. Aktuellen Schätzungen zufolge verdoppelt sich das weltweit erzeugte Datenvolumen alle zwei Jahre (Mooresches Gesetz). [15]

Die entstehenden großen Datenmengen und deren Verarbeitung werden mit den Begriffen Big Data und Data Analytics gekennzeichnet.

Typisch für Big Data sind drei "V" Eigenschaften (Eigenschaften die auf Englisch mit "V" beginnen) [20, S. 11-13]: Volume - umfangreicher Datenbestand (liegt im Tera- bis Zettabytebereich), Variety - Vielfalt der Datenformate (strukturierte, semi-strukturierte, unstrukturierte Multimedia Daten wie Text, Grafik, Bilder, Audio, Video usw.), Velocity - hohe Verarbeitungsgeschwindigkeit und Echtzeitverarbeitung.

Data Analytics ist eine Form der Datenanalyse, bei der man nicht nur große Datenmengen zu bewältigen hat, sondern man versucht noch Korrelationen und Muster zu erkennen. [14, S. 220]

Big Data und Data Analytics verursachen den Bedarf nach Skalierung und Möglichkeiten, verschiedene Datenformate flexibel zu bearbeiten.

Ein relationales Datenmodell ist auf die Konsistenz und auf die Sicherheit der Daten ausgerichtet, d.h. bei der Datenbearbeitung sind aufwendige Aktionen wie das Sperren von Datensätzen erforderlich. Dadurch wird die Möglichkeit zur Parallelverarbeitung eingeschränkt und zudem die Abarbeitungsgeschwindigkeit reduziert. [14, S. 220] Falls es zu größeren Datenmengen kommt, wird traditionell die verteilte Skalierung angewendet. Solch eine Skalierung kann sehr teuer werden. Der Nachteil dabei ist auch das nicht unerschöpfliche Potenzial dieser Skalierungsart. Darüber hinaus sind relationale Datenbanken an einem festen Schema gebunden. Hieraus wird ersichtlich, dass für die flexible Verarbeitung großer Datenmengen, die eventuell die horizontale Skalierung voraussetzen, relationale Datenbanken nicht geeignet sind. Gerade bei Anwendungen, bei denen Performanz wichtiger als Konsistenz ist (z. B. soziale Medien), besteht der Bedarf nach nicht-relationalen Lösungen. Auch bei webbasierten Dienstleistungen mit heterogenen Datenbeständen die in Echtzeit bewältigt werden müssen, ist das relationale Modell nicht der optimale Ansatz.

Für die Verarbeitung großer Datenmengen sind die flexiblen, nicht an Struktur gebundenen NoSQL-Datenbanken meist besonders gut geeignet. Es ist nachvollziehbar, dass mit der Verbreitung der Webanwendungen und mit der Herausforderung der Bewältigung exponentiell wachsender Datenmengen (in 2000er-Jahren) die NoSQL-Datenbanken sehr schnell an Bedeutung gewonnen haben.

## 2.3. Bewertung von NoSQL-Datenbanken

Es steht außer Zweifel, dass NoSQL-DBMS viele Vorteile aufweisen können. Sie lassen die Daten horizontal und vertikal skalieren sowie replizieren und sind durch diese Eigenschaft für die Verarbeitung umfangreicher Datenbestände vorteilhaft, sie sind an keinem festen Schema gebunden und können mit verschiedenen Datenformaten umgehen. Darüber hinaus

ermöglichen diese einen schnellen Datenzugriff und eine erleichterte Datenmanipulation. Viele NoSQL-Systeme sind open source.

Dennoch bringen auch NoSQL-Datenbanken gewisse Nachteile mit sich. Ein wesentlicher Nachteil von NoSQL-Datenbanken entsteht dadurch, dass diese wegen proprietären APIs und Abfragesprachen keinen datenbankunabhängigen, standardisierten Zugriff aufweisen können. Dieses macht die Abfragen von Persistenz-Frameworks bei Objekt-zu-Datenbank-Mapping komplizierter als bei relationalen Datenbanken, die den SQL-Standard unterstützen.

Da der Markt der nicht-relationalen Datenbanken relativ neu und unerforscht ist, gibt es auch einen weiteren Nachteil, nämlich den Mangel an Spezialisten für dieses Gebiet. Eine Umschulung der vorhandenen Fachkräfte ist für ein Unternehmen eine kostspielige Angelegenheit. Dieser Nachteil kann sich jedoch mit der Zeit auflösen, sobald die NoSQL-DBMS sich in der Praxis ausgeweitet und fest etabliert haben.

## 2.4. Persistenz-APIs

Persistenz-Application Programming Interfaces (APIs) sind die Interfaces, welche den in Java geschriebenen Anwendungen die Zuordnung und die Übertragung von Objektdaten in Form von Plain Old Java Objects (POJOs) in eine Datenbank ermöglichen. Solche Schnittstellen definieren nur den Zugriff auf persistenten Daten, wobei die Art der physikalischen Speicherung beliebig sein kann. Großer Vorteil der Persistenz-Frameworks, welche die APIs implementieren, ist die Möglichkeit des Zugriffs auf verschiedenen Datenbanken durch eine gemeinsame Schnittstelle. Dieses ermöglicht die Realisierung von polyglot persistance, beispielsweise, den gemischten Einsatz von SQL- und NoSQL-Datenbanken innerhalb einer Anwendung. Mehr Informationen zu dem Nutzen der polyglot persistance kann z.B. in [12] gefunden werden.

JDO- und JPA-Spezifikationen sind industrielle Standards zum Persistieren von Java-Objekten in Java Standard (J2SE) sowie Java Enterprise (J2EE) Edition.

Jede Persistenz-API-basierte Anwendung hat mindestens eine datastore-controlling Factory. Bei JDO handelt es sich um die *PersistenceManagerFactory*[5] . JPA hingegen erzeugt eine *EntityManagerFactory*[6] . In der Praxis wird häufig eine datastore-controlling Factory pro Datenbank erstellt. Solche Factory Klasse ermöglicht den Zugriff auf entsprechend *Entity*- oder *PersistenceManager* [7] , welche die Objekte in die Datenbank ablegen / aus der Datenbank holen

[5] javax.jdo.PersistenceManagerFactory
[6] javax.persistence.EntityManagerFactory
[7] javax.persistence.EntityManager oder javax.jdo.PersistenceManager

[22] . Die Erzeugung von datastore-controlling Factories und die Lebenszyklusverwaltung von persistenten Objekten übernehmen *PersistenceUnit* [8] und *JDOHelper* [9] Interfaces.

In der Praxis wird JPA heutzutage häufiger als JDO eingesetzt. Ein Grund dafür ist die Ausrichtung dieser Spezifikation auf relationale Datenbanken. Anderseits ist es möglich, die JPA-Annotationen auch zum Persistieren von Objekten in nicht relationalen Datenbanken zu verwenden. Durch ihre Popularität gewährleistet JPA mehr Portabilität für die Anwendung als JDO, das hingegen häufiger für NoSQL-Datenbanken verwendet wird.

Zu der in JPA verfügbaren Funktionalitäten bietet JDO noch zusätzliche Erweiterungen, wie z. B. Enhancement von Klassen oder Extent Interface. Darüber hinaus, wie ersichtlich aus der Tabelle 2.1, erweitert JDO die Wertgenerator-Strategien von JPA. Die folgenden Unterkapitel befassen sich mit den zusätzlichen Funktionalitäten von JDO.

### 2.4.1. Enhancement von Klassen

Transparente Persistenz ist eine Art der Datenspeicherung/Datenverwaltung, bei der die Änderungen an den Objekten außerhalb des Anwendungscodes umgesetzt werden.

Eines der JDO-basierten Object-Mapper Merkmale ist das Streben nach transparenter Persistenz.

Im Gegensatz zu den anderen Object-Mapper Frameworks, wie Hibernate benötigt DataNucleus weder Reflection noch Proxies, um die Änderungen an den Objekten festzustellen. Hiermit gewinnt das Framework an der Transparenz. Wird ein Objekt geändert, bekommt Object-Mapper eine Meldung und übernimmt die Speicherung der Änderungen („transparent change tracking“). Damit transparent change tracking in der Praxis realisiert wird, müssen alle persistente Klassen der Anwendung das Persistable [10] Interface implementieren.

Enhancement der Klassen ist performanter als andere Persistenzverfahren. Dies wird deutlich, wenn beispielsweise eine große Anzahl von Objekten gelesen wird, jedoch weisen nur einige dieser Objekte Änderungen auf. Vergleichen von Objekt-Zuständen während eines Commits einer Transaction wird grundsätzlich zu Verzögerungen führen.

Um die Implementierung des Persistable Interface technisch durchzuführen, wird bytecode Enhancement bereitgestellt, welches die compilierte Klassen modifiziert, damit diese alle benötigte Methoden implementieren.

Grundsätzlich können die Klassen einer Anwendung in drei Kategorien unterteilt werden:

[8] javax.persistence.Persistence
[9] javax.jdo.JDOHelper
[10] org.datanucleus.enhancement.Persistable

- PersistenceCapable. Zu der Kategorie gehören die Klassen, welche in der Datenbank gespeichert werden.
- PersistenceAware Klassen verändern die Felder anderer Klassen, werden jedoch nicht in der Datenbank gespeichert.
- die restlichen Klassen, die nicht in dem Speicherungsprozess beteiligt sind.

Die Kategorien werden in Metadaten den Klassen zugeordnet. Alles Weitere übernimmt der Enhancer-Prozess. [19]

### 2.4.2. Extent Interface

Extent[11] ist ein JDO Interface. Instanz dieses Interfaces ist eine Collection aller persistenten Objekte einer bestimmten Klasse (welche optional auch die Objekte der Unterklassen enthalten kann). Die mögliche Anwendungsfälle, welche das Nutzen des Interfaces ersichtlich machen, sind:

- das Iterieren durch die Objekte,
- das Ausführen der Anfragen auf alle Objekte einer Klasse gleichzeitig [19] .

### 2.4.3. Wertegenerator

Sowohl JDO- als auch JPA-basierten Object-Mapper können Generierung der Werten übernehmen. Diese Eigenschaft wird beispielsweise zum Generieren einer eindeutigen ID für Entitäten verwendet. Dazu gibt es diverse Strategien [1]. Diese sind in der Tabelle 2.1 aufgelistet.

| Strategie | JPA | JDO |
|---|---|---|
| NATIVE (AUTO) | x | x |
| SEQUENCE | x | x |
| IDENTITY | x | x |
| INCREMENT (TABLE) | x | x |
| UUID-string | - | x |
| UUID-hex | - | x |

Tabelle 2.1.: Strategien der Wertgenerierung

[11] javax.jdo.Extent

- NATIVE - der default Wert. Es wird automatisch eine am besten zu der Datenbank passende Strategie ausgewählt (SEQUENCE, IDENTITY oder INCREMENT) und angewendet (auch AUTO generation strategy genannt),
- SEQUENCE - ein Sequenz-Verfahren, wird nur dann angewendet, wenn die Datenbank die Datenbanksequenz unterstützt,
- IDENTITY - verwendet autoincrement/identity/serial Eigenschaften der Datenbank unter der Voraussetzung, dass diese von der Datenbank zur Verfügung gestellt werden,
- INCREMENT - eine datenbankunabhängige Strategie, welche die Werte inkrementiert, und dabei eine Tabelle zur Verwaltung der inkrementierten Werte verwendet (auch TABLE generation strategy genannt),
- UUID-string - Universally Unique Identifier (UUID) in Form eines Strings,
- UUID-hex - UUID in hexadezimaler Form.

Dazu kommen noch Object-Mapper-spezifische Strategien. Diese werden in dem Abschnitt 3.5 beschrieben.

# 3. Object Mapping

Eine Anwendung muss die Daten der Objekte nicht nur verarbeiten und verwalten, sondern auch nach Beendigung der Bearbeitung die Objektdaten und die Objektzustände dauerhaft speichern können. In diesem Kapitel wird eine der möglichen Persistenzlösung - Object Mapping vorgestellt. Im Fokus dieser Arbeit liegt das Mapping für NoSQL-Datenbanken, deshalb wird es grundsätzlich auf nicht relationalen Abbildungskonzepte eingegangen. Darüber hinaus wird Datenmodellierung für verschiedene NoSQL-Datenbankkategorien betrachtet. Abschließend werden einige derzeit populären Object-Mapper Frameworks beschrieben. Dabei ist das Ziel, Gemeinsamkeiten und Unterschiede dieser Mappers herauszufinden.

## 3.1. Einsatz von Object-Mappers

Während der Programmausführung befinden sich die Objekte im Hauptspeicher. Die Objektvariable enthält eine Speicheradresse, die auf den Inhalt der Variable verweist. Außerhalb des Hauptspeichers ist diese Adresse ohne Bedeutung und nach dem Beenden des Programms sind die Daten verloren. [21, S. 89] Persistenz bedeutet wiederum, dass die Daten, durch die Speicherung auf Speichermedien, wie Festplatten etc. gesichert werden. Als Folge sind die Objektdaten für die Zukunft wiederverwendbar.

In der Entwicklung werden für Anwendungen, die in einer objektorientierten Programmiersprache geschrieben sind, häufig Persistenzlösungen wie Serialisierung und Objekt-Mapping eingesetzt.

Bei Serialisierung wird der Zustand eines Objektes in Bytefolge umgewandelt und hiermit zum Ablegen in eine Datenbank vorbereitet. Solch ein Persistenzverfahren kann bei großen Datensätzen bei der Such- oder Sortier-Operationen sehr aufwendig werden, da man in der Bytefolge nicht auf bestimmte Daten zugreifen kann. Zuerst muss die ganze Bytefolge deserialisiert [1] werden, erst dann gelangt man an die gewünschten Daten.

Beim Objekt-Mapping werden Objekte mithilfe deren Metadaten in eine Datenbank abgelegt. Aus den in der Datenbank gespeicherten Daten können die Objekte wieder erzeugt werden.

[1] Bei Deserialisierung wird die Bytefolge in ein programmierbares Objekt umgewandelt, in dem Zustand, in welchem das Objekt sich vor der Serialisierung befand.

Zum Objekt-Mapping zwischen dem Programm und der Datenbank existiert heutzutage eine Vielzahl von open source sowie kommerziellen Persistenz-Frameworks. Wie im Abschnitt 2.4 erwähnt, zu den Aufgaben eines Persistenz-Frameworks gehört unter anderem das Abbilden der Objektmodelle auf die Datenbankmodelle. Dabei müssen die Frameworks Referenzen zwischen Objekten, Assoziations- und Aggregationsbeziehungen, sowie Vererbungshierarchien in der Datenbank deutlich darstellen. Somit verwaltet ein Persistenz-Framework die Datenbank und steuert das Mapping zwischen der Datenbank und den Objekten.

## 3.2. Notation des Mappings

Grundsätzlich gibt es zwei Möglichkeiten das Mapping zu definieren, entweder durch traditionelle XML-Files oder durch einen neueren Ansatz - durch Annotationen. In den nachfolgenden Unterabschnitten wird es auf diese genauer eingegangen und deren Funktionsweise wird anhand eines Beispiels veranschaulicht.

### 3.2.1. XML-Files

Bei dem auf die XML-Files basierten Verfahren handelt es sich um die Mapping Dokumenten, in welchen die Objektmetadaten im XML-Format gespeichert werden. Solche Mapping Dokumenten legen fest, wie die Klassen auf die Datenbankspeichereinheiten (z.B. auf Datenbanktabellen in relationalen Datenbanken oder auf Dokumenten in dokumentenorientierten Datenbanken) abgebildet werden. Dabei können die Mapping Dokumente sowohl manuell geschrieben als auch von Hilfsprogrammen wie z.B. XDoclet oder AndroMDA generiert werden.

Die XML-Files von Hibernate werden wie folgt erzeugt:

- Root-Element vom Mapping XML-File ist **<hibernate mapping>**, dieses enthält alle weiteren **<class>** Elemente.
- der Klassenname wird in einem **name** Attributen gespeichert.
- die entsprechende Speichereinheit in der Datenbank, zu der die Klasse gemappt wird (z.B. Tabelle oder Dokument) wird in einem Attributen **table** gespeichert.
- Klassenbeschreibung kann in einem optionalen Elementen **<meta>** gespeichert werden.
- zu der eindeutigen Identifizierung eines Klassenobjektes in der Datenbank wird den Objekten ein Identifier (ID) zugewiesen. Im XML-File wird dieser in einem **<id>** Elementen gespeichert. Das **<id>** Element hat die folgenden Attribute: **name, column, type**

und das untergeordnete Element - **<generator>**. Dabei entspricht **name** dem Namen der Objektvariable, **column** dem Feldnamen in der Datenbankspeichereinheit und **type** dem Datentypen der ID. Das **<generator>** Element generiert IDs automatisch mit einer der in Tabelle 2.1 beschriebenen Wertgenerierung-Strategien.

- das **<property>** Element bewirkt das Mapping von Objektattributen auf die Datenbankfelder, dabei werden ähnlich wie bei dem **<id>** Elementen die Attribute **name, column** und **type** verwendet.

Es gibt noch weitere Elemente und Attribute, die beim Mapping zusätzliche Funktionalitäten leisten. Das XML-File wird nach dem Klassennamen benannt und im **.hbm.xml** Format gespeichert.

### 3.2.2. Annotationen

Annotationen stellen ein zu den XML-Files alternatives Mapping Verfahren dar und wurden mit Java 5.0 eingefügt. Der Java Specification Request (JSR) 175 nennt Annotationen „Metadata Facility for the Java Programming Language“. Durch Annotationen werden die Daten direkt im Anwendungscode generell beschrieben und können weiterhin auf diverse Datenquellentypen gemappt werden.

Jede Persistenz-API bietet ihre eigene Annotationen (dabei können die Namen gleich sein). Darüber hinaus stehen den Benutzern die Persistenz-Framework-spezifische Annotationen zur Verfügung, die zu dem vorhandenen Standard extra Funktionalitäten gewährleisten.

Bei der Verwendung von Annotationen werden die Metadaten eines Objektes in dem Java File mit dem Klassencode gespeichert.

Dieses Verfahren wird in dem praktischen Teil dieser Arbeit im Kapitel 4 verdeutlicht.

## 3.3. Object-Mapper Architektur

Die Abbildung 3.1 stellt eine generische Object-Mapper Architektur dar. Eine Anwendung verwendet ein einheitliches Datenmodell, z. B. bei JPA sind es Annotationen. Für das Modell besteht eine einheitliche Schnittstelle, deren Operationen vom Middleware Engine implementiert werden. Die Operationen werden weiterhin zu einen für die Datenbank passenden Datenbank-Mapper delegiert [2]. [24]

[2] Datenbank-Mapper ist ein Modul, welches die API von Datenbanktreiber implementiert.

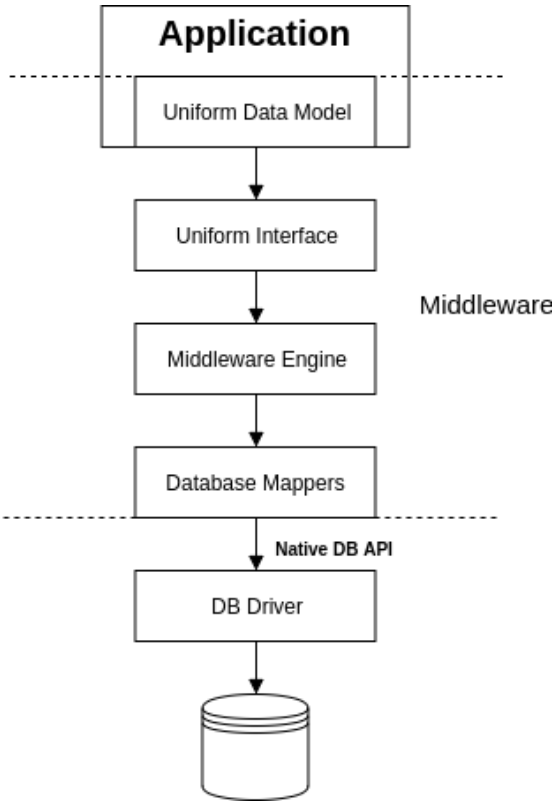


Abbildung 3.1.: Object-Mapper Architektur

## 3.4. Datenmodellierung in NoSQL-Datenbanken

Bei der Modellierung einer Persistenzschicht unterscheidet sich die Vorgehensweise bei den NoSQL-Datenbanken enorm von den SQL-Datenbanken. Die Daten werden bei der Speicherung nicht wie in relationalen Datenbanken auf mehrere Tabellen in Form der Tupel verteilt, und somit keine Normalisierung von Daten notwendig ist. Dadurch werden sowohl Flexibilität als auch bessere Leistung erreicht, weiterhin sind Änderungen an dem Datenmodell leichter durchzuführen.

Da die Daten nicht über mehrere Tabellen verteilt gespeichert werden, benötigen NoSQL-Datenbanken im Gegensatz zu SQL-Datenbanken keine aufwendigen JOINS bei den Datenabfragen.

Wie in dem Abschnitt 2.1 schon erwähnt, werden NoSQL-Datenbanken in vier Gruppen unterteilt (Schlüssel-Wert Datenbanken, Spaltenfamilien-Datenbanken, dokumentenorientierte

Datenbanken sowie Graphdatenbanken). Für jede dieser Gruppen wird die Datenmodellierung unterschiedlich durchgeführt.

Im Grunde der Modellierung liegen solche Fragen wie - "Wofür werden die Daten benötigt?". Und dementsprechend - "Wie wird es auf die Daten zugegriffen?" Sowie - "Welche Möglichkeiten bietet die Datenbank um Daten zu strukturieren und abzufragen?"[29] .

### 3.4.1. Datenmodellierung in Schlüssel-Wert Datenbanken

Die Schlüssel-Wert Datenbanken verfolgen einen relativ einfachen Ansatz. Es werden eindeutige Schlüssel auf den intransparenten Werten in Form der binären Daten abgebildet. Dabei können die als Werte gespeicherten Daten eine beliebige Struktur haben.

Da die Datenbank in vielen Fällen (ohne sekundären Indizes) die Werte nicht interpretieren kann, sind nur die Schlüssel-basierten Abfragen möglich. Dieser Aspekt zeigt die Bedeutung der Vorgabe von Schlüsselnamen für die Anwendungsdaten. Einige Schlüssel-Wert Datenbanken, wie z.B. Redis, unterstützen die Aufteilung des globalen Namensraums in verschiedene Segmente, die bei der Datenabfragen separat behandelt werden. Bei solcher zweistufiger Hierarchie sind die Schlüsselnamen nur innerhalb dessen Schlüsselraums eindeutig. Außerhalb des Schlüsselraums werden die Namen der Schlüssel in Kombination mit jeweiligem Namen des Schlüsselraums verwendet [27].

Sekundäre Indizes sind aus relationalen DBMS bekannt. Immer mehr Schlüssel-Wert Datenbanken bieten sekundäre Indizes um die Daten, welche unter einem Schlüssel gespeichert sind, für die Anfragen strukturiert zu gestalten - gesuchte Werte können über sekundäre Indizes schneller abgefragt werden. Es gibt Datenbanken, wie z.B. Riak, welche sortierte Indizes verwenden, um Bereichsabfragen auf Werte bereit zu stellen (der in den Index aufzunehmende Wert wird in der Anwendung geschrieben, die Datenbank kennt den Inhalt der Werte nicht).

Mehr Informationen zu dem Einsatz von sekundären Indizes in Schlüssel-Wert Datenbanken sind in [30] zu finden.

### 3.4.2. Datenmodellierung in Spaltenfamilien-Datenbanken

Spaltenfamilien-Datenbanken sind ähnlich zu den Schlüssel-Wert Datenbanken mit Schlüsseln und dazugehörigen Werten, allerdings transparent, in Form von so genannten Columns, aufgebaut. Dabei können solche Columns entweder gleich die Endwerte, oder die auf hierarchisch unten liegende Columns verweisende Schlüssel erhalten. Somit ähnelt sich die Struktur der in der Datenbank persistierten Daten einem verschachtelten assoziativen Array.

Die Datenabfragen erfolgen über Schlüsselbereiche, wie bei Schlüssel-Wert Datenbanken. Da die Spaltenfamilien-Datenbanken in meisten Fällen horizontal skalierbar sind, wird dieser Ansatz besonders effizient, wenn zusammen abgefragte Daten auf dem gleichen Server liegen (erreichbar durch Anwender konfigurierbare Partitionierung der Datenbank) [27].

### 3.4.3. Datenmodellierung in dokumentenorientierte Datenbanken

Bei der Datenmodellierung in dokumentenorientierten Datenbanken behandelt man Entitäten als eigenständige Dokumente. Dokumente aus derselben Domäne bilden eine Collection. Die unterliegenden Entitäten können auch als eingebettete Objekte in einem anderen Dokument als ein Unterdokument persistiert sein. Die Dokumente werden am häufigsten im JavaScript Object Notation (JSON)-Format dargestellt.

Auch dokumentenorientierten Datenbanken sind an kein festes Schema gebunden. Das heißt, die Daten in dem JSON-Dokument können in verschiedenen Formaten gespeichert werden. Darüber hinaus sind die Entitäten leicht veränderbar, ohne die anderen eigenständigen (nicht eingebetteten) Entitäten durch die Änderungen zu beeinflussen.

In dokumentenorientierten Datenbanken, ähnlich zu den Schlüssel-Wert Datenbanken, ist jeder Dokument zu einem eindeutigen Schlüssel gebunden und kann durch diesen Schlüssel abgefragt werden. Darüberhinaus sind die Attribute in den Dokumenten typisiert, und können gesucht werden. Dieses ermöglicht komplexere Datenabfragen als bei den oben beschriebenen Datenbankgruppen.

Das Lesen eines vollständigen Datensatzes aus einer Datenbank wird durch die Speicherung aller Attributen in einem Dokument auf einen Lesevorgang reduziert. Auch das Aktualisieren mehrerer Attribute eines Datensatzes erfolgt in nur einem Schreibvorgang.

Die Daten werden in folgenden Fällen in einem einzelnen Dokument eingebettet:

- zwischen Entitäten gibt es contains -Beziehungen,
- zwischen Entitäten gibt es eins-zu-viele -Beziehungen,
- es gibt eingebettete Daten, die sich selten ändern,
- es gibt eingebettete Daten, die nicht grenzenlos wachsen,
- es gibt eingebettete Daten, die von integraler Bedeutung für die Daten in einem Dokument sind. [23]

### 3.4.4. Datenmodellierung in Graphdatenbanken

Das Datenmodell der Graphdatenbanken ist auf der Graphentheorie basiert. Ein Graph mit Knoten, Kanten und Eigenschaften wird unter Berücksichtigung der Beziehungen zwischen Objekten erstellt. Dabei muss kein Schema im Vorhinein festliegen. Außerdem kann der Graph jederzeit dynamisch um neue Kanten und Knoten erweitert oder abgebaut werden, unter der Voraussetzung, dass durch die Erweiterung oder das Entfernen von Elementen keine offene Beziehung (Kante ohne Start- oder Endknoten) entsteht.

Da die Graphdatenbanken schemafrei sind, und Darstellung der Beziehungen zwischen Objekten in Form der Kanten eines Graphen erfolgt, wird im Gegensatz zu der Datenmodellierung für dokumentenorientierte Datenbanken oder Schlüssel-Wert Datenbanken kein konventionelles 1 zu 1, 1 zu n oder n zu m Mapping benötigt.

Um die persistierten Daten später abzufragen, wird mit Cypher ein Graph erstellt. Die Daten, welche auf dem erstellten Graphen matchen, werden als Ergebnis ausgegeben. Hierfür ist es nicht nötig, den gesamten Graphen mit allen Beziehungen anzugeben. Dadurch werden die Abfragen auf Hierarchien sehr vereinfacht.

Schwierigkeiten beginnen bei den Abfragen, wo auf bestimmte Eigenschaften gefiltert wird. Die in Cypher vorhandene Where-Clauses verlieren an der Performance gegenüber anderen Datenbanken, die Stärken einer Graphdatenbank werden somit nicht ausgenutzt. Aus diesem Grund wird es empfohlen auf Where-Clauses zu verzichten. Als alternative Lösung wird die Übertragung der Daten in einen Graphen erwähnt, z.B. das Erstellen eines Timetrees für das Abbilden von Daten. [18]

## 3.5. Objekt Mapper Überblick

In diesem Kapitel werden einige der aktuell verfügbaren open source Object-Mapper Frameworks geschildert. Zunächst wird ein tabellarischer Vergleich allgemeiner Parameter durchgeführt. Danach werden die Object-Mapper einzeln betrachtet.

### 3.5.1. Tabellarischer Vergleich

Alle Object-Mapper Daten in dieser Arbeit gelten für die in der Tabelle 3.1 aufgelistete Releases.

Die Daten aus der Tabellen: Tabelle 3.2, Tabelle 3.3, Tabelle 3.4 sollen einen Überblick verschaffen, welche Spezifikationen, Datenbanken sowie die Abfragesprachen von weiter betrachteten Object-Mapper Frameworks unterstützt werden.

| | Release |
|---|---|
| Hibernate | 5.3.1 |
| DataNucleus | 5.1 |
| Morphia | 1.3.2 |

Tabelle 3.1.: Release Informationen

| | JPA | JDO | Gora API |
|---|---|---|---|
| Hibernate | x | - | - |
| DataNucleus | x | x | - |
| Morphia | - | - | - |

Tabelle 3.2.: Spezifikationen

Wie bekannt, erfassen NoSQL ein weites Spektrum von Datenbanken, eine allgemeingültige Lösung für alle nicht relationale Datenbanken und für alle Anwendungsfälle ist daher kaum vorstellbar. Aktuell ermöglichen die betrachteten Object-Mapper folgende Funktionalitäten für alle unterstützte NoSQL-DBMS:

- CRUD (create, read, update, delete) Operationen auf Entitäten,
- polimorphische Entitäten (Unterstützung für Vererbungshierarchien),
- embeddable objects (Klassenkomponenten),
- basis Datentypen, wie Nummern, Strings, URLs, Dates, enums, etc.,
- Assoziationen,
- Collections (Set, List, Map, etc.).

### 3.5.2. Hibernate

Hibernate Framework wurde von Gavin King und seinen Kollegen bei Cirrus Technologies im Jahr 2001 als Alternative zu Enterprise JavaBeans (EJB) 2-style Entity Beans entwickelt. Das Ziel war die Verbesserung der Persistenzeigenschaften des existierenden EJB 2 Standards. 2003 wurde das erste Release von Hibernate 2.0 veröffentlicht. Kurz danach wurden die Entwickler des Hibernate Frameworks von JBoss Inc. eingestellt um dessen Entwicklung fortzusetzen. Im Jahre 2005 kam Hibernate 3.0 heraus mit einer neuen Interceptor/Callback Architektur,

| | dokumentenorientierte DB | Graph DB | Key-Value DB | Spalten DB |
|---|---|---|---|---|
| Hibernate | MongoDB, CouchDB | Neo4J | Infinispan, Ehcache | - |
| DataNucleus | MongoDB | Neo4J | - | Cassandra, HBase |
| Morphia | MongoDB | - | - | - |

Tabelle 3.3.: Datenbanken

| | JPQL | Native Queries | JDOQL | HQL |
|---|---|---|---|---|
| Hibernate | x | x | - | x |
| DataNucleus | x | x | x | - |
| Morphia | - | - | - | - |

Tabelle 3.4.: Abfragesprachen

sowie benutzerdefinierten Filtern und JDK 5.0 Annotationen. Alle Hibernate Releases ab 3.5.0 implementieren die JPA Spezifikation und sind konform mit dem JSR 317 Standard [8].

Neben den Standard JPA-Annotationen werden für das Mapping zusätzlich Hibernate-spezifische Annotationen bereitgestellt, da Hibernate mehr Möglichkeiten als die JPA-API zu bieten hat.

Das Framework wurde ursprünglich auf SQL-Datenbanken ausgerichtet und unterstützte ausschließlich Object-Relational Mapping (ORM) . Seit Ende 2013 unterstützt Hibernate 4.0 und alle späteren Releases auch Object-NoSQL Data Mapping (ONDM).

Wie ersichtlich aus der Tabelle 3.3 , schränkt sich das Framework nicht auf eine bestimmte NoSQL-Datenbankkategorie ein. Nichtsdestotrotz wird für Hibernate ONDM der Name - Hibernate OGM genutzt.

Hibernate OGM verwendet viele Hibernate ORM Konzepte wieder, wie beispielsweise Primär-schlüssel- und Fremdschlüssel-Notation. Darüber hinaus wird die Verwaltung des Objektlebenszyklus von Hibernate ORM übernommen.

Als Datenbankabstraktionsschicht stellt Hibernate, wie viele andere Object-Mapper einen Cache Mechanismus zum Eliminieren der aufwendigen Datenbankzugriffe bereit. Das Framework verfügt sowohl über den 1st level als auch über den 2nd level Cache. Dabei gibt es einen zusätzlichen query Cache, welcher in manchen Quellen 3rd level Cache genannt wird [3] .

Der 1st level Cache ist Session-scoped, das heißt, jede Entität muss während der Session (in persistent context) nur einmal aus der Datenbank geladen werden. Danach liegt diese im

[3]Query Cache in Hibernate ist eine Erweiterung des 2nd level Cache (process level Cache)

Cache und wird beim nächsten Aufruf daraus geholt. Der 1st level Cache wird gleichzeitig mit dem *EntityManager* erzeugt und mit der Session, zu welcher der gehört, terminiert.

Der 2nd level Cache hingegen ist SessionFactory-scoped. Der wird von allen Entitäten aller Sessions einer *SessionFactory* geteilt. Dieser Cache ist in Hibernate optional. Um eine Entität sichtbar für den 2nd level Cache zu machen, muss diese als @Cachable annotiert werden.

Der 2nd level Cache besteht aus folgenden Partitionen:

- entity,
- collection,
- query,
- timestamp.

Entity und collection cachen Entitäts- und Beziehungsdaten (nicht die gesamte Entitäten).

Query und timestamp sind verbunden. Query speichert die Datenbankanfragen und die Ergebnisse als ein Schlüssel-Wert Paar in einer Lookup-Tabelle. Dabei wird der String-Wert der Anfrage mit dazugehörigen Parametern zum Schlüssel. Der zu diesem Schlüssel gehörende Wert wird aus den Primärschlüsseln der Entitäten geformt, welche als Ergebnis der Anfrage erhalten wurden.

Die Datenbankanfragen werden zu dem query-Cache gereicht und mit den in der Lookup-Table liegenden Queries verglichen. Wenn eine passende Query gefunden wird, können die Primärschlüssel der Entitäten (Ergebnisse der Anfrage) durch den 1st oder durch den 2nd level Cache aufgelöst werden.

Timestamp-Partition des 2nd level Cache verfolgt die Modifikationen der Speichereinheiten (Tabellen, Dokumente, etc.). Wenn eine Anfrage auf die Speichereinheiten aufgerufen wird, welche in der Zeit, nachdem die letzte Anfrage im Cache abgelegt wurde, geändert ist, wird das Schlüssel-Wert Paar (Anfrage und die dazugehörende Primärschlüssel) aus dem query-Cache entfernt.

Der Nutzen des query-Cache bleibt zweifelhaft. Durch die aufwendigen criteria queries, welche in der Lookup-Tabellen liegen, wird der Speicherplatz verbraucht. Darüber hinaus, wenn eine Änderung einer Entität aus der Speichereinheit durchgeführt wird, gelten alle Entitäten dieser Speichereinheit, welche in dem query-Cache liegen, als veraltet. Die Wahrscheinlichkeit, dass eine gesamte Speichereinheit in der Zeit zwischen den Anfragen unverändert bleibt, ist sehr gering. Dazu kommen noch Synchronisationsprobleme zwischen den timestamp- und den query-Einträgen.

Dennoch gibt es Fälle, wo der query-Cache effizient benutzt wird. Als Beispiel können die Abfragen mit immutable [4] natural IDs genannt werden.

In Hibernate kann eine Entität zusätzlich zu einem Primärschlüssel eine natural ID haben. Diese wird durch die Annotation @NaturalId generiert [5] .

Zuerst wird der Primärschlüssel mittels der natural ID abgefragt. Diese Abfrage wird im query-Cache abgelegt. Bei einer immutable natural ID wird die timestamp-Prüfung ausgelassen (weil diese nicht geändert werden kann). Als Ergebnis wird der Primärschlüssel erhalten, welcher für die nächste Abfrage benutzt wird. Dabei wird es nach dem Ergebnis in den 1st und in den 2nd level Cache gesucht (weil diese Primärschlüssel zum Caching nutzen).

Hibernate ist zu mehreren 2nd level Cache Providers kompatibel, welche in der Tabelle 3.5 aufgelistet sind. Mehr Informationen zu diesen kann in [8] gefunden werden.

Dabei können verschiedene catching Strategien verwendet werden:

- read-only - die Strategie ist geeignet für die Daten, welche oft gelesen werden müssen, dafür selten modifiziert werden,
- nonstrict read-write - die Strategie eignet sich für die Daten, die selten modifiziert werden, und bei denen es unwahrscheinlich ist, dass mehrere Transaktionen auf eine Entität gleichzeitig zugreifen,
- read-write - die Strategie ist anwendbar, wenn die Daten oft aktualisiert werden,
- transactional - diese Strategie ist für transactional Cache Providers wie JBoss TreeCache verfügbar.

| | read-only | nonstrict read-write | read-write | transactional |
|---|---|---|---|---|
| HashTable (in der Testphase) | x | x | x | |
| EHCache | x | x | x | |
| OSCache | x | x | x | |
| SwarmCache | x | x | | |
| JBoss Cache 1.x | x | | | x |
| JBoss Cache 2.x | x | | | x |

Tabelle 3.5.: 2nd level Cache Optionen

[4] Standardmäßig sind die natural IDs immutable; um eine natural ID mutable zu machen, wird das *mutable* Attribut der @NaturalId Annotation explizit gesetzt

[5] org.hibernate.annotations.NaturalId

Das Framework hat einen eigenen ID /Sequence generator, dabei gibt es drei Möglichkeiten IDs zu generieren:

- assigned generator – das Generieren der ID wird der Applikationslogik überlassen,
- UUIDHexGenerator – die ID wird von Hibernate generiert. Dabei ist es möglich, 32 hexadecimal UUID String Werte zu generieren,
- UUIDGenerator – ein ID Generator von Hibernate, der mehr Flexibilität bietet (Auswahl zwischen java.lang.UUID, 16 Byte Array und hexadezimalem String Wert) und RFC 4122 kompatibel ist.

Darüber hinaus stellt das Framework eine Möglichkeit bereit, eine Entität oder eine Collection unveränderlich zu machen. Dafür wird die Entität mit @Immutable annotiert. Danach werden die Änderungen an der Entität oder an der Collection nicht an die Datenbank gereicht, sondern durch HibernateException gemeldet [16].

### 3.5.3. DataNucleus

Data Nucleus ist ein Persistenz-Framework, das im Jahr 2003 unter dem Namen Java Persistent Objects (JPOX) erschien und im Jahr 2008 unter dem Namen Data Nucleus wiedereingeführt wurde. Das Framework wird vom DataNucleus Team entwickelt.

DataNucleus implementiert sowohl JPA- als auch JDO-Spezifikationen. Beide APIs können mithilfe DataNucleus-spezifischen Persistenzeigenschaften konfiguriert werden.

Für ONDM empfehlen die Entwickler von DataNucleus die JDO-Spezifikation. Die JPA-Spezifikation ist hingegen auf relationalen Datenbanken ausgerichtet, diese kann aber auch für nicht relationale Datenbanken verwendet werden.

Ähnlich wie Hibernate, verfügt DataNucleus-Framework über zwei level Cache. Der 2nd level Cache ist optional und speichert alle Objekte aller *PersistenceManager* . JDO bestimmt nicht darüber, ob der 2nd level Cache aktiviert ist, dieses wird dem Object-Mapper überlassen.

In DataNucleus ist der 2nd level Cache von dem Typen *soft* per default aktiv. Es besteht jedoch eine Möglichkeit, per property *datanucleus.cache.level2.type* diesen Typen umzustellen oder den Cache zu deaktivieren. Außerdem ist es möglich, mit der property *datanucleus.cache.level2.mode* das Cache mode umzustellen. In DataNucleus default mode (UNSPECIFIED) werden alle Objekte aller Entitäten gecached, wenn die Entität nicht als non-cacheable markiert ist.

Die in DataNucleus aktuell verfügbaren Caching Optionen sind in der Tabelle 3.6 aufgelistet. Mehr Information zu diesen kann in [1] gefunden werden.

| | 1st level Cache | 2nd level Cache | provider |
|---|---|---|---|
| none | x | x | JDO |
| weak | x | x | JDO |
| soft | x | x | JDO |
| strong | x | | JDO |
| EHCache | | x | datanucleus-cache plugin |
| EHCacheClassBased | | x | datanucleus-cache plugin |
| OSCache | | x | datanucleus-cache plugin |
| SwarmCache | | x | datanucleus-cache plugin |
| Oracle Coherence | | x | datanucleus-cache plugin |
| javax.cache | | x | datanucleus-core plugin |
| JCache | | x | datanucleus-cache plugin |
| spymemcached | | x | datanucleus-cache plugin |
| xmemcached | | x | datanucleus-cache plugin |
| cacheonix | | x | datanucleus-cache plugin |

Tabelle 3.6.: Caching Optionen für den 1st und für den 2nd level Cache

Nach jedem commit() von *PersistenceManager* Transaktion werden alle persistenten cacheable Objekte von Entitäten im 2nd level Cache plaziert, und die persistente Objekte, die gelöscht wurden, aus dem 2nd level Cache entfernt.

Das DataNucleus Cache-Plugin weitet die Features von dem 2nd level Cache aus. Es erlaubt die Objektspeicherung in dem Cache zu verwalten mithilfe der Methoden:

- evict(),
- pin(),
- unpin().

Die **evict()** Methode entfernt das Objekt aus dem Cache. Die **pin()** Methode speichert das Objekt in dem Cache und sichert, dass dieses in dem Cache solange bleibt, bis es entfernt wird, und dass es nicht von *GarbageCollector* gelöscht wird. Wenn das Objekt nicht ungepinnt wird, bleibt es in dem Cache über die gesamte Applikationslebenszeit, dieses beschleunigt wesentlich die Zugriffe auf das Objekt. Nach dem Aufruf von **unpin()** kann das Objekt von dem *GarbageCollector* gelöscht werden, wenn es über bestimmte Zeit nicht genutzt wurde.

DataNucleus stellt standard JPA- und JDO-Persistenzeigenschaften bereit, darüber hinaus hat das Framework seine eigenen, Framework-spezifischen Persistenzeigenschaften, z. B. zur Konfiguration der Datenbank (Datastore Definition) oder zur Steuerung der Transaktionen

(Transactions and Locking). Es gibt einige Persistenzeigenschaften, welche auf bestimmte Datenbanken oder Datenbanktypen ausgerichtet und nur auf entsprechende Datenbanken verwendbar sind.

Durch die JDO-Implementierung stellt DataNucleus die Enhancement von Klassen Funktionalität bereit [1]. Diese wurde im Unterabschnitt 2.4.1 dieser Arbeit erläutert.

Ähnlich zu Hibernate hat DataNucleus einen eigenen Wertgenerator und stellt dadurch einige ObjectMapper-spezifische Wertgenerierung-Strategien zur Verfügung [1]:

- UUID - ein reiner UUID -Generator, welcher auf JDK 1.5 UUID Klasse zugreift,
- AUID - ein reiner UUID -Generator, welcher dem OpenGroup Standard folgt,
- timestamp - erstellt *java.sql. Timestamp* für die aktuelle Zeit,
- timestamp-value - erstellt long (Millisekunden) aus der aktuellen Zeit,
- max - ist nur für relationale Datenbanken verwendbar, verwendet max(column)+1 Verfahren,
- datastore-uuid-hex - ist nur für relationale Datenbanken verwendbar, erstellt eine UUID in hexadezimaler Form,
- user-supplied value generator - unterstützt das Einbinden eines benutzerdefiniertes Generators.

### 3.5.4. Morphia

Morphia ist ein Object-Document Mapper. Erste Framework Releases sind in 2010 erschienen. Es wurde entwickelt von MongoDB Team ausschließlich für Object Mapping auf die Mongo-Datenbank. Scott Hernandez, einer der Framework Entwickler, beschreibt das Framework als eine Typ-sichere Bibliothek zum Mappen der Objekte auf MongoDB.

Morphia stellt ein speziell auf dem MongoDB-Java-Treiber basierendes, nicht portierbares API dar. Es wird ein Teil der JPA bekannten Techniken bereitgestellt ohne JPA -Spezifikation zu implementieren.

Das Framework stellt zwei Mechanismen bereit, welche die Interaktion mit der Datenbank ermöglichen:

- eine Datastore Schnittstelle [6] ,

[6] org.mongodb.morphia.Datastore

- ein DAO - API , welches auf dem Datastore aufsetzt.

Im Gegensatz zu JPA-basierten *EntityManager* und zu JDO-basierten *PersistenceManager*, wird ein definierter Lebenszyklus des Datastore nicht benötigt, weil mit dieser keiner first level Cache verbunden ist. Außerdem stellt das Framework keinen 2nd level Cache bereit.

Morphia erlaubt Default-Mapping. Das heißt, es werden auch solche Objekte persistiert, die nichts von Morphia wissen, bzw. nicht mit Morphia-Annotationen annotiert sind, unter der Voraussetzung, dass die Objektklasse einen Default-Konstruktor besitzt. So können die Objekte gespeichert werden, deren Quellcode nicht veränderbar ist, z.B. die Objekte aus fremden Bibliotheken. Damit ein Feld beim Speichern und beim Laden eines Objekts ignoriert wird, muss es mit @Transient annotiert sein.

Beim Persistieren wird in dem Dokument zusätzlich zu den Objektattributen der Klassenname sowie eine ID gespeichert. Wenn die ID nicht in der Objektklasse definiert ist, wird diese von dem Datenbankserver generiert. Allerdings können die Objekte mit solcher ID nicht später durch Morphia wieder identifiziert werden, weil das Klassenobjekt nicht die von der Datenbank vergebene ID kennt. [26]

Eine Besonderheit von dem Framework ist evolution annotations. Morphia (ähnlich wie Objectify Framework für Google Cloud Datastore) stellt eigene Annotationen zur Verfügung, um bei schema-flexiblen NoSQL-Datenbanksystemen eine lazy Daten-Migration durchzuführen.

**a**

```
1  @Entity
2  class Person {
3    @Id
4    Integer id;
5
6
7    String lastName;
8    String firstName;
9    Char middleInitial;
10
11
12   ContactInfo contactInfo;
13 }
```

**b**

```
1  @Entity
2  class Person {
3    @Id
4    Integer id;
5
6    @AlsoLoad("lastName") // Umbenennen
7    String surname;
8    String firstName;
9                          // Entfernen
10   String nickName;      // Hinzufuegen
11
12   ContactInfo contactInfo;
13 }
```

Abbildung 3.2.: **a** Eine Java Klasse mit Annotationen für den Object-Mapper sowie **b** evolutionäre Änderungen an der Klasse im Lauf der Anwendungsentwicklung

Die Abbildung 3.2 veranschaulicht, wie die Änderungen an dem Datenbankschema durchgeführt werden. Codeabschnitt a persistiert ein Person-Objekt mit folgenden Attributen: id, lastName, firstName, middleInitial, contactInfo. Codeabschnitt b zeigt 3 Basisoperationen für eine Migration, das Umbenennen, das Hinzufügen sowie das Entfernen von Attributen.

Nachdem Ausführen vom veränderten Programmcode entsprechen die bisher persistierten JSON-Dokumente nicht dem neuen Schema. Beim Laden werden die bestehende Person-Daten lazy migriert. lastName ist nun mit @AlsoLoad annotiert und wird dadurch in surname umbenannt. Das Attribut middleInitial wird nicht geladen, dieses passiert weil das Feld aus der Klassendeklaration entfernt wurde. Ein neues Attribut nickName wurde hinzugefügt, dieses wird beim ersten Laden mit null-Wert initialisiert.

Mit dem nächsten Persistieren des Person-Objekts werden die Daten passend zum aktuellen Schema festgeschrieben [25] .

Darüber hinaus besteht durch life-cycle Annotationen eine Möglichkeit, beliebigen Migrationscode in Methoden der Entität-Klasse zusammenzufassen und beim Speichern eines Objektes auszuführen. [17]

# 4. Vergleichende Untersuchung

Wie in dem Abschnitt 3.5 beschrieben, gibt es diverse Frameworks für das Abbilden von Objekten in NoSQL-Datenbanken. Um das Verhalten von ausgewählten Frameworks in der Praxis zu beobachten sowie deren Performanz [1] zu evaluieren, werden in mehreren Versuchen Speicher- und Lese-Operationen mit einigen derzeit populären Object-Mapper durchgeführt. Die Versuche werden mit einer nur für diese Arbeit entwickelter Anwendung realisiert. In diesem Kapitel werden die Anwendung, das Szenario, nach welchem die Speicher- und die Lese-Operationen ablaufen, sowie der Versuchsablauf mit jedem der drei für diese Untersuchung ausgewählten Object-Mapper beschrieben.

## 4.1. Begründung der Auswahl von Object-Mappers

Für die vergleichende Untersuchung wurden zwei heutzutage weit verbreitete Implementierungen der im Abschnitt 2.4 beschriebenen Persistenz-APIs ausgewählt. Als Repräsentant für JPA -API ist das Hibernate Framework vorgestellt. Das DataNucleus Framework repräsentiert hingegen die Implementierung von JDO -API . Beide Object-Mapper Frameworks unterstützen mehrere NoSQL-Produkte. Als drittes Framework wurde für die Untersuchung ein Object-Mapper ausgewählt, welches auf ein bestimmtes Produkt ausgerichtet ist - Morphia Framework.

Die genannten Object-Mapper wurden zur Datenspeicherung und danach zu m Lesen von gespeicherten Daten auf zwei Datenbanken untersucht. Durch den Einsatz des auf MongoDB ausgerichteten Morphia Framework wurde als eine der Datenbanken MongoDB ausgewählt. Als zweite Datenbank wurde Neo4j verwendet. Dadurch sind in der Untersuchung zwei NoSQL-Datenbankgruppen vertreten - dokumentenorientierte Datenbanken und Graphdatenbanken.

In mehreren Versuchen wird die Geschwindigkeit gemessen, in der die Daten persistiert wurden. Danach wird es gemessen, wie schnell die persistierten Daten wieder gelesen werden. Im Kapitel 5 findet die Auswertung der Messergebnisse statt.

[1] im Hinsicht auf die Schreibgeschwindigkeit und auf die Lesegeschwindigkeit

## 4.2. Anforderungen an die Beispielanwendung

Bei der Speicherung von Anwendungsdaten in eine Datenbank ist es wichtig sowohl die Entitäten mit dazugehörigen Attributen zu persistieren als auch die Relationen zwischen den Entitäten. Um das Verhalten der Object-Mappers bei der Speicherung verschiedener Relationstypen zu sehen, müssen in der Anwendung die Basis-Relationstypen vorhanden sein:

- eine Aggregation,
- eine Komposition,
- eine 1 zu 1 Beziehung,
- eine n zu m Beziehung,
- eine 1 zu n Beziehung,
- eine n zu 1 Beziehung,
- eine Vererbungsbeziehung.

Damit die Relationen zwischen den in der Datenbank gespeicherten Entitäten nachvollziehbar bleiben, müssen alle Beziehungstypen bidirektional sein (jede Entität enthält eine Referenz auf die in einer Relation zu deren stehende andere Entität). Eine Ausnahme ist das UserData Objekt. Da UserData ein eingebettetes Objekt von User ist, müssen die Instanzen dieses Objektes nicht den entsprechenden User direkt kennen.

Ein Vorteil der nicht-relationalen Datenbanksysteme ist die Möglichkeit, Daten in Form einer Collection in die Datenbank zu speichern. Um Collection-Mapping zu beobachten, müssen in der Anwendung Collections als Objektattribute vorhanden sein.

## 4.3. Beispielanwendung

Die Struktur der Beispielanwendung ist in Form eines Klassendiagramms in der Abbildung 4.1 dargestellt.

Die Anforderungen nach verschiedenen Beziehungstypen wurden wie folgt umgesetzt:

- Ein Benutzer kann sich mit dem Passwort aus dem UserData Objekt in ein Online-Forum einloggen. Jeder Benutzer hat genau ein UserData Objekt, das heißt, die Entitäten stehen in einer **1 zu 1 Beziehung** zueinander.

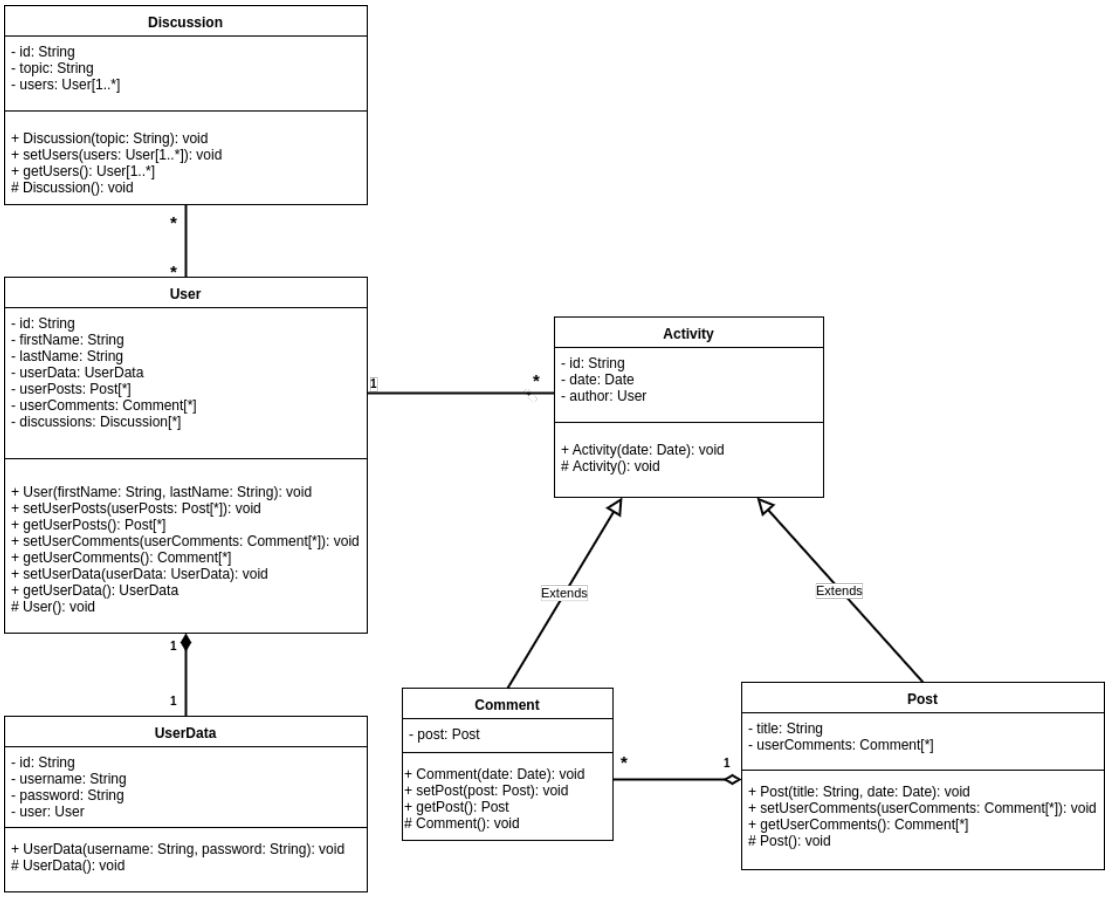


Abbildung 4.1.: Datenmodell der Anwendung

- Der Lebenszyklus jeder UserData Instanz wird durch die entsprechende User Instanz bestimmt. Wird ein User Objekt gelöscht, verschwindet auch das dazugehörige UserData Objekt. Hierdurch ist eine **Komposition** vertreten.
- Jeder Benutzer kann an beliebig vielen Diskussionen teilnehmen, gleichzeitig nehmen beliebig viele Benutzer an einer Diskussion teil. Hierdurch ist eine **bidirektionale m zu n Beziehung** entstanden. Objekte kennen sich gegenseitig durch die Attribute *users* und *discussions* .
- Weiterhin kann ein Benutzer beliebig viele Aktivitäten durchführen, die Beziehung zwischen dem Benutzer und der Aktivität ist also eine **1 zu n Beziehung** (oder eine **n**

**zu 1 Beziehung** aus der Aktivitätsseite betrachtet). Auch diese Beziehung ist durch die Attribute *userPosts*, *userComments* und *author* **bidirektional**.

- Eine Aktivität ist entweder ein neuer Post oder ein neuer Kommentar, beide Entitäten stehen in einer **Vererbungsbeziehung** zu Aktivität.
- Es besteht noch eine **1 zu n Beziehung** zwischen den Entitäten Post und Comment, jeder Post kann beliebig viele Kommentare haben. Die Attribute *post* und *userComments* zeichnen eine **bidirektionale** Beziehung aus.
- Das Kommentar Objekt steht in einer **Aggregationsbeziehung** zu dem Post Objekt. Jeder neue Kommentar wird zu einem Teil eines Posts, jedoch ist dieser eine eigenständige Instanz.

Bei einer 1 zu n Beziehung hat jede "1-Seite" Entität ein Attribut in Form einer **Collection**, welches die Referenzen auf entsprechenden "n-Seite" Entitäten enthält. Ähnlich haben bei jeder m zu n Beziehung alle Entitäten solche Attribute. Damit ist auch die Anforderung nach Collections erfüllt.

## 4.4. Szenario

Um höhere Genauigkeit der Ergebnisse zu erreichen, werden 3 Speicherversuche gemacht, bei jedem Versuch wird 500 Mal ein gleiches Szenario durchgeführt.

Das Szenario wurde so erfunden, dass die Entitäten aus der Beispielanwendung und die angeforderten Beziehungstypen wenigstens einmal vorhanden sind.

Zuerst werden zwei Benutzer erzeugt. Für jeden der Benutzer wird ein UserData Objekt erstellt. Dies wird mittels eines Sequenzdiagramms in der Abbildung 4.2 veranschaulicht.

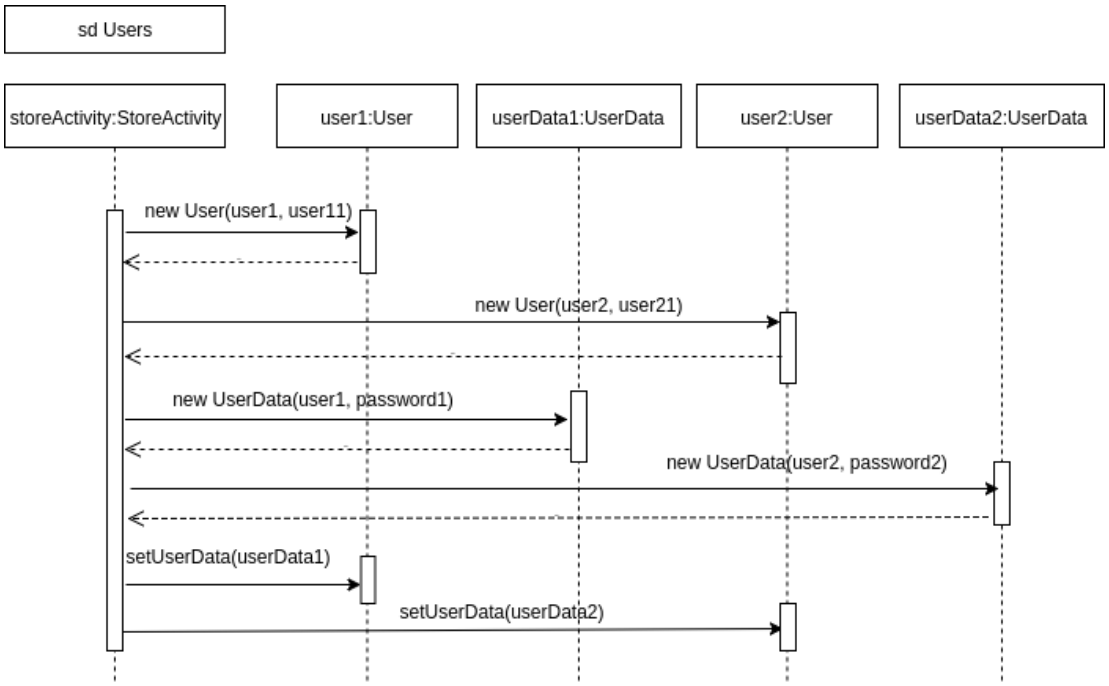


Abbildung 4.2.: Users

Weiterhin werden zwei Discussion Instanzen kreiert. Jeder der Instanzen werden beide Benutzer zugewiesen, das heißt, beide Benutzer nehmen an beiden Diskussionen teil: Abbildung 4.3.

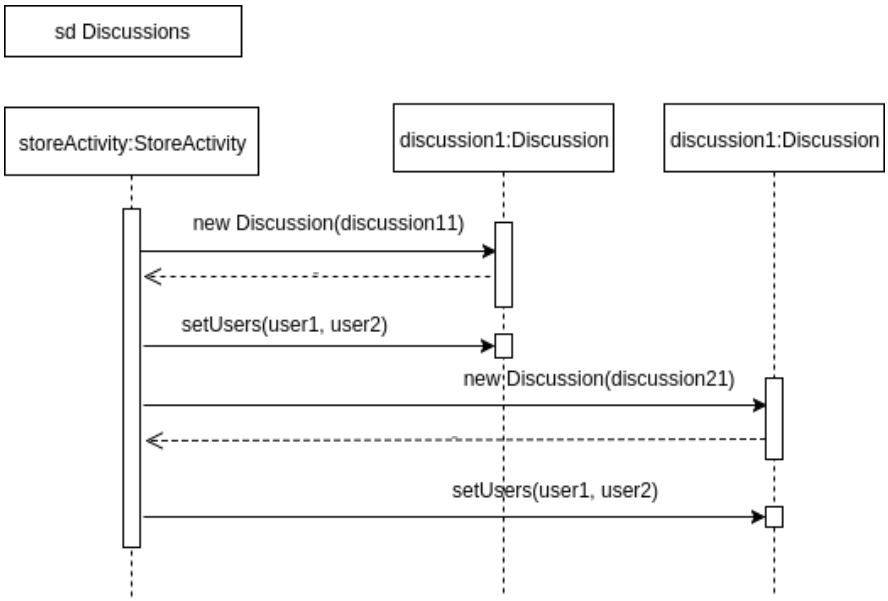


Abbildung 4.3.: Discussions

Danach werden drei Post Instanzen erstellt, welche von der Activity Klasse erben. Zwei der Post Instanzen werden dem ersten Benutzer zugewiesen, die dritte Instanz wird dem zweiten Benutzer zugewiesen: Abbildung 4.4 und Abbildung 4.5 .

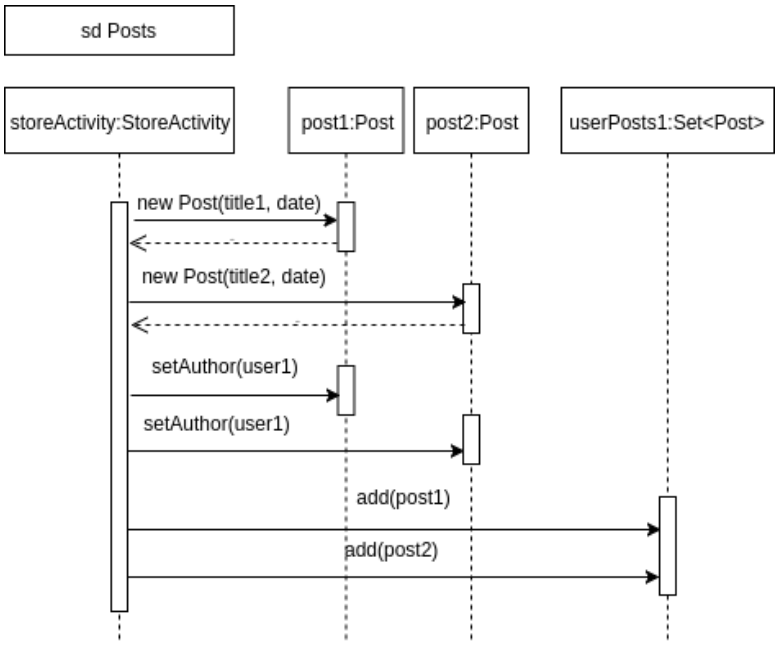


Abbildung 4.4.: Posts

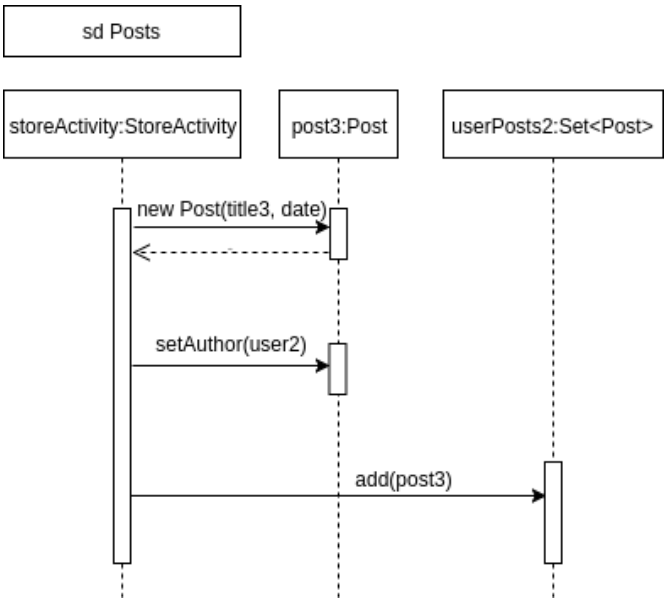


Abbildung 4.5.: Posts

Weiterhin werden zwei Comments erstellt, welche, ähnlich wie die Posts, von der Activity Klasse erben. Die Comments werden dem ersten Benutzer zugewiesen: Abbildung 4.6.

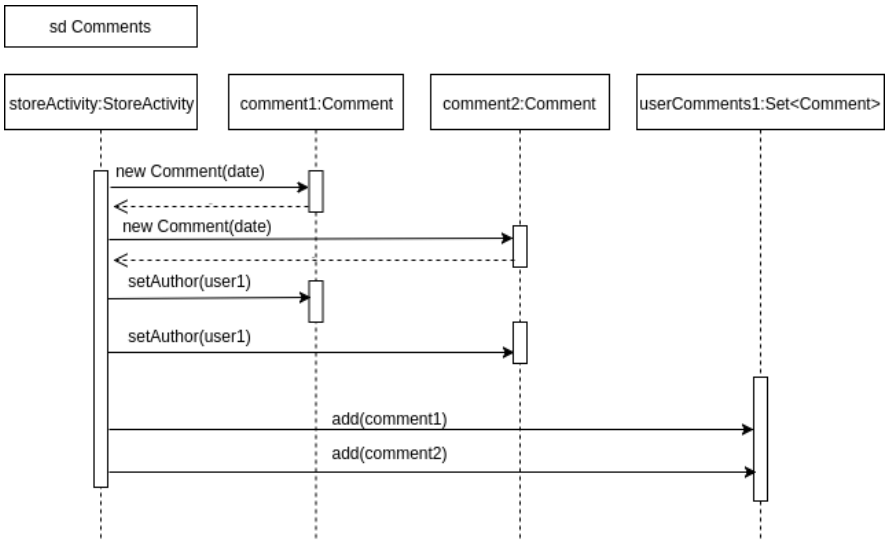


Abbildung 4.6.: Comments

Der erste Comment wird dem ersten Post zugewiesen. Der zweite Comment wird dem zweiten Post zugewiesen: Abbildung 4.7.

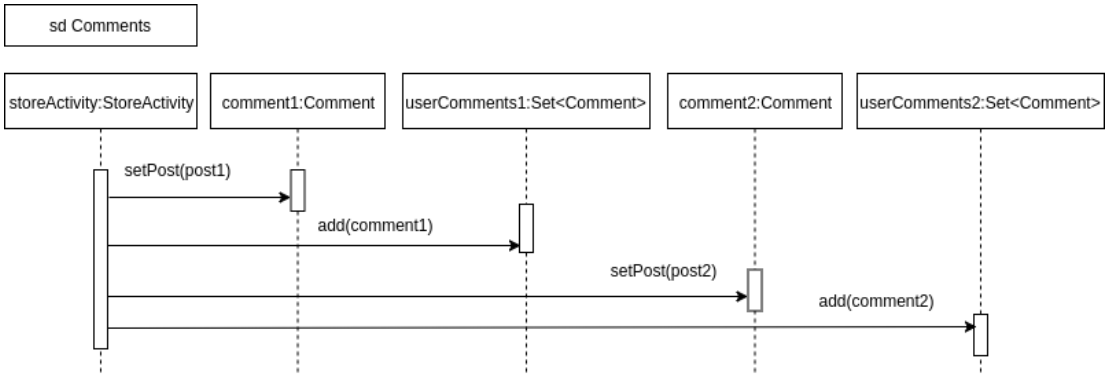


Abbildung 4.7.: Comments

Schließlich werden die Objekte persistiert. Dieser Vorgang ist Object-Mapper-spezifisch und wird im Unterabschnitt 4.6.6 für jedes Framework explizit beschrieben.

Im Anhang A sind die entstandene Datenbankeinträge im JSON-Format für MongoDB sowie als Graphen für Neo4j dargestellt.

Nach dem Speichern aller Entitäten werden zwei Leseversuche gemacht. Dabei befinden sich in der Datenbank zu der Zeit des Lesens folgende Daten:

- User: (3000 Einträge),
- Post (4500 Einträge),
- Comment (3000 Einträge),
- Discussion (3000 Einträge).

Im ersten Versuch werden in einer Datenbanktransaktion vier Lesevorgänge durchgeführt, um alle gespeicherten Entitäten aus der Datenbank zu holen (fünfte Entität - UserData ist ein eingebettetes Objekt und wird deswegen zusammen mit User Entität gelesen).

Im zweiten Versuch werden in einer neuen Session (damit der 2nd level Cache nicht betroffen wird) nur die User Entitäten aus der Datenbank gelesen, dabei wird es auf *firstName* gefiltert. Somit werden 1500 Entitäten gefunden und aus der Datenbank geladen.

Das Lesen in Hibernate geschieht durch JPQL-Queries, in DataNucleus durch JDOQL-Queries und in Morphia, entsprechend, durch Morphia spezifischen Queries. Die Abfragen sind im Anhang B aufgelistet.

## 4.5. Technische Einrichtungen

Alle Versuche auf der MongoDB Datenbank laufen gegen einen remote MongoDB Server v3.6.4.

Laut der aktuellen Dokumentation von DataNucleus Framework (Release 5.2) ist das Framework nur für eine embedded Neo4j Datenbank einsetzbar.

Die DataNucleus offizielle Website berichtet, in der Zeit der Entwicklung von Object-Mapper Modulen für Neo4j existierte noch kein binärer Protokoll zum Verbinden von Java Client mit einem remote Neo4j Server. Die Unterstützung von remote Neo4j ist in der Zukunft geplant [1].

Seit Neo4j v3.0.0 ist es möglich, den Java Client mithilfe des bolt connector mit einer remote Datenbank zu verbinden. Es wurde durch ein Java Programm zur Visualisierung der Ergebnisse aus der Neo4j Datenbank (welche im Anhang A gefunden werden können) geprüft [4]. Dieses lässt hoffen, dass eine neue Version des DataNucleus Object-Mapper mit remote Neo4j support den Benutzern bald zur Verfügung steht.

Damit die Messergebnisse von Hibernate und von DataNucleus Frameworks vergleichbar sind, wurde für die Versuche mit Hibernate Object-Mapper auch eine embedded Neo4j Datenbank verwendet. Dabei unterscheiden sich die Versionen von Neo4j, welche jeweils mit letzten Versionen von Hibernate und DataNucleus kompatibel sind. Da in dieser Arbeit die letzten zurzeit verfügbaren Object-Mapper Releases verglichen werden, kommt für Hibernate Neo4j v.3.3.3 zur Einsatz. Für DataNucleus wird hingegen die ältere Neo4j v.3.0.0 genutzt.

Die Object-Mapper werden in einer nur für die vorliegende Arbeit entwickelten Anwendung zur Datenspeicherung eingesetzt. Anwendungscode ist unter folgenden Links zu finden: [5] , [2] , [6] , [3] , [7] .

Object-Mapper Bibliotheken wurden durch maven dependencies in dem Projekt eingebunden, welche im Anhang C aufgelistet sind.

Als Laufumgebung wurde Eclipse Oxygen.1a verwendet. Zum Enhancement von Klassen mit DataNucleus Object-Mapper wurde DataNucleus Eclipse plugin verwendet.

Alle Tests wurden auf einem Rechner mit folgender Ausstattung ausgeführt:

- Hauptspeicher: 4GB,
- Prozessor: 4 x 2.50GHz,
- Festplatte: 500GB,
- Betriebssystem: Ubuntu 16.04.4 LTS.

## 4.6. Versuchsbeschreibung

Für die Untersuchung wurden in der Tabelle 4.1 aufgelistete Object-Mapper Releases verwendet.

| | Release |
|---|---|
| Hibernate | 5.3.1.Final |
| DataNucleus | 5.1.9 |
| Morphia | 1.3.2 |

Tabelle 4.1.: In der Untersuchung verwendete Releases

### 4.6.1. Persistente Klassen

Jede Klasse, deren Instanzen in der Datenbank abgelegt werden, muss für die Object-Mapper als persistent erkennbar sein. In **Hibernate** sowie in **DataNucleus** werden die persistente

Klassen entsprechend markiert. So wird jede solche Klasse in Hibernate mit *@Entity* annotiert. In DataNucleus hingegen wird *@PersistenceCapable* Annotation gesetzt.

Im Gegensatz zum **Hibernate** und zum **DataNucleus** Frameworks ist in **Morphia** die *@Entity* Annotation optional, das heißt, es werden auch die Instanzen von solchen Klassen persistiert, welche über keine Persistenzannotationen verfügen. Um bestimmte Felder als nicht persistent zu markieren, wird *@Transient* Annotation verwendet.

**Morphia** speichert per default den Klassennamen als zusätzlichen Attributen. Um dadurch keine Verletzung der Integrität zu verursachen, ist es möglich, dieses über *@Entity* Annotationsparameter *noClassnameStored* abzuschalten.

In der Anwendung werden folgenden Klassen als persistent markiert:

- User,
- Discussion,
- Activity,
- Post,
- Comment.

Im Versuch mit dem Morphia Framework wird zusätzlich der *noClassnameStored* Parameter auf *true* gesetzt.

### 4.6.2. Schlüsselerzeugung

Zum Generieren einer UUID wird im Versuch mit dem **Hibernate** Mapper der Wertgenerator von **Hibernate** verwendet mit der UUID-Strategie.

Unabhängig von dem Attributen Namen von der ID, speichert **Hibernate OGM** jede ID in einem *_id* Feld von MongoDB Dokumenten, somit erkennt die Datenbank das *_id* Feld und kann es entsprechend weiterverwalten.

Dabei kann ID ein eingebauter Datentyp sein (wie es im Abschnitt 3.5 erwähnt wurde, unterstützt **Hibernate** die basis Datentypen wie Nummern, Strings, etc.), oder es kann auch ein komplexer Datentyp sein, der durch eine eingebettete Klasse mit entsprechenden Attributen erstellt wird.

Um die Integrität der Daten zwischen den Anwendungen zu gewährleisten, wird es generell empfohlen, den Datentypen von MongoDB *objectId* zu verwenden. Dazu gibt es zwei Möglichkeiten:

- id in Form *org.bson.types.ObjectId* zu definieren,
- id in Form eines Strings zu definieren mit der Annotation *@Type(type="objectid")* .

Das Ergebnis sieht dann wie folgt aus:

"_id" : ObjectId("5b59dedfbcf53b3ebe4198f4")

Für die Anwendung wurde der basis Datentyp String verwendet. Die Verwendung von dem empfohlenen *objectId* Datentypen war nicht möglich, weil die Beziehung zwischen den Klassen im Fall einer Vererbung mit anderen ID-Typen nicht anerkannt wird.

Daraus wurde folgendes ID-Format erzeugt:

"_id" : "cd5ea98a-67ee-4cc7-8cf4-f5a43712f562"

Im Versuch mit **DataNucleus** Framework werden UUIDs mit dem JDO -Wertgenerator erzeugt. Auch hier, ähnlich wie bei **Hibernate**, wird UUID-Strategie verwendet.

Zusätzlich zu einer von MongoDB generierter ID - *objectId*:

"_id" : ObjectId("5b00a84cbcf53b1b45b831fd")

generiert der JDO -Wertgenerator eine weitere ID , die andere Entitäten als Referenz verwenden:

"id" : "5ba61ebd-2440-48ef-a2b6-f170fe360f3b"

Somit verfügt jede persistierte Entität über zwei ID-Werte. Solche Effekte lassen sich beispielsweise durch identity-Strategie für Wertgenerierung vermeiden.

Im **Morphia** Framework ist *@Id* eine optionale Annotation, die die von MongoDB erzeugte UUID zu der ID-Wert der Entität mappt.

### 4.6.3. Attribute

Wenn Attribute einer Entität im Objektmodell als eigene Klasse modelliert sind, können diese ebenfalls persistiert werden. Es handelt sich um eine 1 zu 1 Beziehung oder um eine 1 zu n Beziehung im Fall einer ElementCollection [16] .

Eine 1 zu 1 Beziehung besteht zwischen den User und UserData Instanzen. Dabei stehen die Entitäten in einer Kompositionsbeziehung zu einander, somit wird UserData zu einer

eingebetteten Klasse. Dieses wird entsprechend annotiert. In **Hibernate** wird dafür *@Embeddable* Annotation verwendet. In **DataNucleus** wird der Parameter von *@PersistenceCapable* Annotation *embeddedOnly* auf *true* gesetzt. In **Morphia** wird *@Embedded* gesetzt. Auf der "User-Seite" ist das UserData Attribut bei allen drei Frameworks als *@Embedded* annotiert.

### 4.6.4. Assoziationen

Die Entitäten User und Discussion stehen in einer m zu n Beziehung zu einander, und die Entitäten User und Activity (entweder Post oder Comment) in einer 1 zu n Beziehung. Es besteht noch eine 1 zu n Beziehung zwischen den Post und Comment Entitäten (da diese Beziehung eine Aggregationsbeziehung ist, können die Comment Entitäten nicht eingebettet werden).

Das Attribut *author*, welches eine Referenz auf User Entität darstellt, ist sowohl für Comment als auch für Post Entitäten relevant, und wird aus diesem Grund in der abstrakten Klasse Activity deklariert, von welcher beide genannten Unterklassen erben.

Dieses muss im **Hibernate** sowie im **Morphia** Framework entsprechend annotiert werden. **DataNucleus** hingegen, als eine JDO -basierte Framework, bietet keine Beziehungstyp-Annotationen (diese werden von der JDO -Spezifikation nicht unterstützt). Generell bietet DataNucleus zwei Möglichkeiten die Assoziationsbeziehungen zu kennzeichnen:

- Join Table,
- Foreign Key.

Diese sind jedoch die Konzepte von relationalen Datenbanken und werden ausschließlich auf SQL-Datenbanken angewendet. Beziehungen zwischen den Entitäten für die Speicherung in NoSQL-Datenbanken werden in **DataNucleus** ohne Annotationen deklariert.

In **Hibernate** werden die Beziehungstypen mit folgenden Annotationen gekennzeichnet:

- *@OneToOne*,
- *@OneToMany*,
- *@ManyToMany*,
- *@ManyToOne*.

Bei *@OneToMany* und *@ManyToMany* Beziehungen werden die Collections mit einem Parameter *mappedBy = "author"* versehen, welcher eine Referenz für die andere Beziehungsseite definiert.

In **Morphia** werden die Beziehungen zwischen den Objekten mit *@Reference* annotiert, ohne auf den Beziehungstypen explizit einzugehen (wie in Hibernate). Fehlt *@Reference* Annotation, so wird die Entität eingebettet und nicht referenziert. Die Datenbankeinträge unterscheiden sich von den **Hibernate** und **DataNucleus** Object-Mapper indem Objektreferenzen in Form *DBRef* gespeichert sind.

### 4.6.5. Vererbung

Die Vererbung im **Hibernate** und im **DataNucleus** Framework wird durch *@Inheritance* Annotation der JPA - API oder der JDO - API realisiert. Dabei wird ein passender Inheritance-Typ ausgewählt, welcher besagt, wie die Vererbung gemappt werden soll.

Im Versuch mit **Hibernate** wurde der Typ TABLE_PER_CLASS verwendet. Dies bedeutet, dass die Entitäten von Unterklassen in eigenen Collections gespeichert werden.

Werden die Entitäten von Unterklassen in einer Collection gespeichert (SINGLE_TABLE Strategie), so wird eine zusätzliche Annotation auf beiden Seiten benötigt - *@Discriminator-Column*, ein nicht zu der Entität gehöriges Attribut, welches den Unterklassentyp enthält. Der Name solches Attributes wird auf der Seite der Oberklasse deklariert, TYPE ist dabei der default Wert.

**DataNucleus** stellt folgende Vererbungsstrategien bereit:

- New Table,
- Subclass Table,
- Superclass Table,
- Complete Table.

Für NoSQL-Datenbanken eignet sich nur eine Vererbungsstrategie - Complete Table. Dabei besagt COMPLETE_TABLE-Strategie auf der Oberklassen Seite, dass sowohl die Attributen aus Oberklassen als auch die Attributen von Unterklassen sowie auch die gesamte Unterklassen in der eigenen Tabellen persistiert werden.

In **Morphia** wird eine Vererbung ohne Annotationen implementiert.

### 4.6.6. Speichern der Entitäten

In **Hibernate** wird die CASCADE-Funktionalität von JPA -API zur Speicherung der Entitäten eingesetzt. Der verwendete CascadeTyp PERSIST gewährleistet automatische Mitspeichern von Entitäten, deren Attribute mit dem Typen markiert wurden.

In der Anwendung werden folgende Attribute mit dem CascadeTyp PERSIST versehen:

- *users* in der Klasse *Discussion*,
- *userPosts* in der Klasse *User*,
- *userComments* in der Klasse *User*.

Somit reicht es die Discussion Entitäten explizit zu speichern. Die dazugehörige User, Post und Comment Entitäten werden mitgespeichert.

JDO unterstützt persistence-by-reachability Konzept, dies bedeutet, wird ein neues Objekt einer Klasse in der Datenbank persistiert, so werden auch die neuen verwandten Objekte anderer Klassen automatisch gespeichert. Es werden also in **DataNucleus**, ähnlich wie in **Hibernate**, nur Discussion Entitäten explizit persistiert.

In **Morphia** werden alle Entitäten explizit gespeichert. Eine Besonderheit beim Speichern ist, dass die Objekte erst dann den anderen Objekten zugewiesen werden können, wenn diese persistiert wurden. Darüber hinaus müssen die Objekte nach dem Zuweisen anderer Objekte wiederholt gespeichert werden.

Nach dem Speichern der Objekte:

- erhält jede User Entität in MongoDB ein eingebettetes Objekt UserData, sowie die Referenzen in Form von deren IDs auf dazugehörigen Post, Comment und Discussion Entitäten,
- jede Discussion Entität erhält Referenzen auf dazugehörige User Entitäten (eine bidirektionale m zu n Beziehung),
- jede Comment Entität referenziert den Author des Kommentaren sowie den dazugehörigen Post,
- die Post Entität referenziert den Author des Posten, wie es die Oberklasse Activity besagt, erhält darüber hinaus Referenzen auf Post bezogene Kommentare,
- die Comment Entität referenziert den Author des Kommentars, wie es die Oberklasse Activity besagt, erhält darüber hinaus Referenz auf den Post, zu welchem diese gehört.

In Neo4j wird für jede persistierte Entität ein Node erstellt. Es werden auch für eingebettete Entitäten Nodes erzeugt.

### 4.6.7. Laden von Entitäten

Alle Leseversuche werden zweimal durchgeführt, mittels lazy loading[2] und mittels eager loading[3]. Die Ladestrategien werden in Object-Mapper Frameworks wie folgt umgesetzt:

- in **Hibernate** wird der *fetch* Parameter jeder Beziehungstyp-Annotation (*@OneToMany, @ManyToOne* etc.) entweder auf LAZY oder auf EAGER *FetchType* gesetzt,
- in **DataNucleus** wird der *defaultFetchGroup* Parameter jeder *@Persistent* Annotation entweder auf *true* oder auf *false* Wert gesetzt,
- in **Morphia** wird der *lazy* Parameter jeder *@Reference* Annotation entweder auf *true* oder auf *false* Wert gesetzt.

[2] Ein Entwurfsmuster, bei dem Datenobjekte grundsätzlich Werte oder andere, abhängige Objekte bereitstellen, diese aber erst bei einer konkreten Anfrage aus der Datenquelle holen

[3] Ein Entwurfsmuster, bei dem möglichst effizient sofort alle absehbar benötigten Daten geholt werden [11, S. 200-214]

# 5. Performanzvergleich

In diesem Kapitel werden die Ergebnisse der im Kapitel 4 beschriebener Untersuchung geschildert. Gemessene Werte werden in Tabellenform gegenübergestellt, verglichen und analysiert. Auf der Analyse basierend, werden die beobachteten Object-Mapper evaluiert.

## 5.1. Erläuterung der Messdaten

Wie im Kapitel 4 erwähnt, wurde als Erstes gleicher Speichervorgang mehrerer Entitäten dreimal durchgeführt. Für jeden Durchlauf wurde die Gesamtdauer gemessen, daraus wurde die Anzahl der in der Datenbank gespeicherten Einträgen pro Sekunde und die Durchschnittsdauer eines Speichervorgangs für einen Eintrag in Millisekunden berechnet. Danach wurde das arithmetische Mittel von der Daten der drei Versuchsdurchläufen bestimmt.

Als Zweites wurden alle in drei Vorgängen gespeicherte Entitäten aus der Datenbank geladen, dabei wurde zuerst lazy loading, dann eager loading eingesetzt. Es wurde jeweils die Gesamtdauer des Lesens gemessen, daraus wurde die Anzahl der aus der Datenbank gelesener Einträge pro Sekunde und die Durchschnittsdauer eines Lesevorgangs für einen Eintrag in Millisekunden berechnet.

Als Letztes wurden die User Entitäten mit einem bestimmten Namen aus der Datenbank geladen, dabei, ähnlich wie bei dem vorherigen Versuch, wurde zuerst lazy loading, danach eager loading angewendet. Es wurde jeweils die Gesamtdauer des Lesens gemessen, daraus wurde die Anzahl der aus der Datenbank gelesener Einträge pro Sekunde und die Durchschnittsdauer eines Lesevorgangs für einen Eintrag in Millisekunden berechnet.

## 5.2. Vergleich der Speicherperformanz

Die Speichergeschwindigkeit der ausgewählten Object-Mappers ist in der Tabelle 5.1 für MongoDB dargestellt. Ferner, in der Tabelle 5.2, ist deren Speichergeschwindigkeit in der Neo4j Datenbank abgebildet.

Es sind jeweils die **gespeicherten Einträge pro Sekunde** (Eintr/s) und **die Dauer des Speichervorgangs für einen Eintrag in Milisekunden** (ms/Eintr) geschildert.

Die Mittelwerte sind unterhalb der Tabellen in Form der Balkendiagramme abgebildet (Abbildung 5.1 und Abbildung 5.2 ).

Im Abschnitt 5.4 werden die Messergebnisse analysiert.

### 5.2.1. Speichern in MongoDB

| Mongo | Hibernate | | DataNucleus | | Morphia | |
|---|---|---|---|---|---|---|
| Schreiben | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr |
| Lauf 0 | 37.9 | 26.38 | 80.61 | 12.41 | 114.71 | 8.72 |
| Lauf 1 | 30.49 | 32.8 | 169.33 | 5.91 | 260.77 | 3.83 |
| Lauf 2 | 20.72 | 48.27 | 134.71 | 7.42 | 239.45 | 4.18 |
| Mittelwert | 29.7 | 35.82 | 128.22 | 8.58 | 204.98 | 5.58 |

Tabelle 5.1.: Speichervorgänge in MongoDB

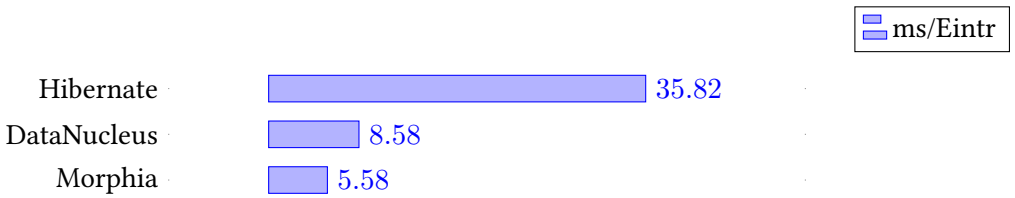


Abbildung 5.1.: Speichervorgänge in MongoDB

### 5.2.2. Speichern in Neo4j

| Neo4j | Hibernate | | DataNucleus | |
|---|---|---|---|---|
| Schreiben | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr |
| Lauf 0 | 7.48 | 133.75 | 7.25 | 138.01 |
| Lauf 1 | 11.08 | 90.29 | 6.19 | 161.6 |
| Lauf 2 | 9.68 | 103.35 | 5.78 | 172.93 |
| Mittelwert | 9.41 | 109.13 | 6.41 | 157.52 |

Tabelle 5.2.: Speichervorgänge in Neo4j


Abbildung 5.2.: Speichervorgänge in Neo4j

## 5.3. Vergleich der Leseperformanz

Das Laden der Entitäten aus der Datenbank wird mittels lazy loading und mittels eager loading durchgeführt. Dabei wird es zwischen dem Lesen mit einem Filter und ohne einem Filter unterschieden. Ähnlich wie beim Schreiben, liefen alle Versuche gegen die MongoDB und gegen die Neo4j Datenbank.

- Für **lazy loading** wird die Lesegeschwindigkeit bei der Anfragen **ohne einem Filter** tabellarisch in der Tabelle 5.3 für MongoDB sowie in der Tabelle 5.7 für Neo4j geschildert.
- Für **eager loading** wird die Lesegeschwindigkeit bei der Anfragen **ohne einem Filter** tabellarisch in der Tabelle 5.4 für MongoDB sowie in der Tabelle 5.8 für Neo4j geschildert.
- Für **lazy loading** wird die Lesegeschwindigkeit bei der Anfragen **mit einem Filter** tabellarisch in der Tabelle 5.5 für MongoDB sowie in der Tabelle 5.9 für Neo4j geschildert.
- Für **eager loading** wird die Lesegeschwindigkeit bei der Anfragen **mit einem Filter** tabellarisch in der Tabelle 5.6 für MongoDB sowie in der Tabelle 5.10 für Neo4j geschildert.

Ähnlich wie bei dem Speichern, sind in der Tabellen jeweils die **geladenen Einträge pro Sekunde** (Eintr/s) und **die Dauer des Lesevorgangs für einen Eintrag in Milisekunden** (ms/Eintr) geschildert.

Die Mittelwerte sind unterhalb der Tabellen in Form der Balkendiagramme abgebildet (Abbildung 5.3 , Abbildung 5.7 , Abbildung 5.4 , Abbildung 5.8 , Abbildung 5.5 , Abbildung 5.9 , Abbildung 5.6 , Abbildung 5.10 ).

Im Abschnitt 5.4 werden die Messergebnisse analysiert.

### 5.3.1. Lesen ohne Filter aus MongoDB

| Mongo | Hibernate | | DataNucleus | | Morphia | |
|---|---|---|---|---|---|---|
| Lesen aller Einträge (lazy loading) | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr |
| | 1119.15 | 0.89 | 1232.56 | 0.81 | 381.54 | 2.62 |

Tabelle 5.3.: Lazy loading aller Einträge aus MongoDB


Abbildung 5.3.: Lazy loading aller Einträge aus MongoDB

| Mongo | Hibernate | | DataNucleus | | Morphia | |
|---|---|---|---|---|---|---|
| Lesen aller Einträge (eager loading) | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr |
| | 207.46 | 4.82 | 1105.09 | 0.9 | 105.84 | 9.45 |

Tabelle 5.4.: Eager loading aller Einträge aus MongoDB

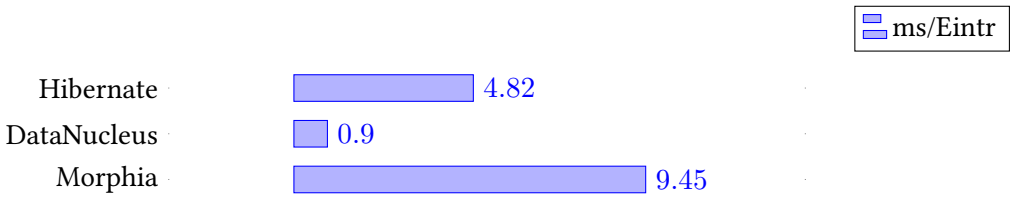


Abbildung 5.4.: Eager loading aller Einträge aus MongoDB

### 5.3.2. Lesen mit Filter aus MongoDB

| Mongo | Hibernate | | DataNucleus | | Morphia | |
|---|---|---|---|---|---|---|
| Lesen mit Filter (lazy loading) | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr |
| | 2628.71 | 0.38 | 21974.23 | 0.05 | 1553.92 | 0.64 |

Tabelle 5.5.: Lazy loading der Einträge aus MongoDB mit einem Filter

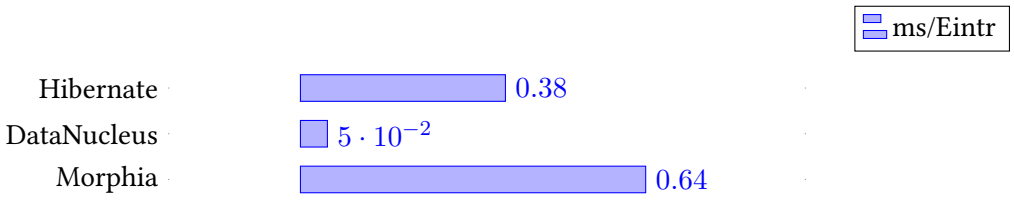


Abbildung 5.5.: Lazy loading der Einträge aus MongoDB mit einem Filter

| Mongo | Hibernate | | DataNucleus | | Morphia | |
|---|---|---|---|---|---|---|
| Lesen mit Filter (eager loading) | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr |
| | 250.06 | 4.0 | 1436.9 | 0.7 | 346.52 | 2.89 |

Tabelle 5.6.: Eager loading der Einträge aus MongoDB mit einem Filter

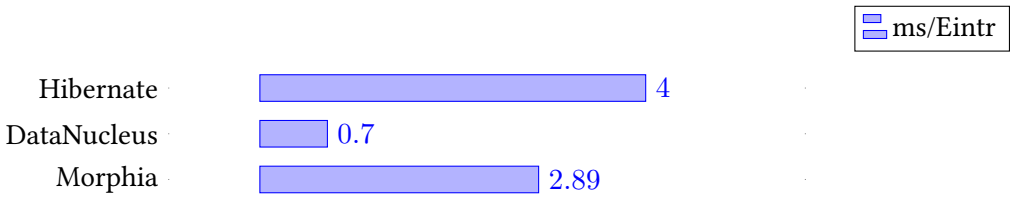


Abbildung 5.6.: Eager loading der Einträge aus MongoDB mit einem Filter

### 5.3.3. Lesen ohne Filter aus Neo4j

| Neo4j | Hibernate | | DataNucleus | |
|---|---|---|---|---|
| Lesen aller Einträge (lazy loading) | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr |
| | 823.54 | 1.21 | 892.86 | 1.12 |

Tabelle 5.7.: Lazy loading aller Einträge aus Neo4j

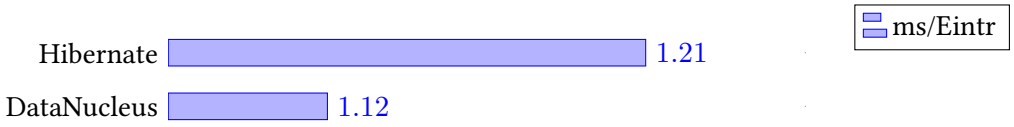


Abbildung 5.7.: Lazy loading aller Einträge aus Neo4j

| Neo4j | Hibernate | | DataNucleus | |
|---|---|---|---|---|
| Lesen aller Einträge (eager loading) | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr |
| | 161.8 | 6.18 | 271.11 | 3.69 |

Tabelle 5.8.: Eager loading aller Einträge aus Neo4j

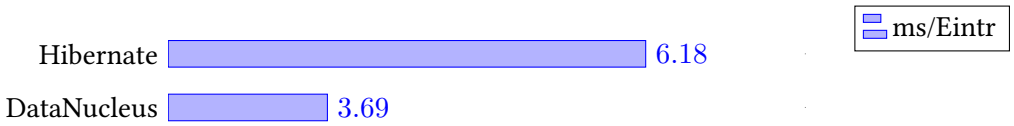


Abbildung 5.8.: Eager loading aller Einträge aus Neo4j

### 5.3.4. Lesen mit Filter aus Neo4j

| Neo4j | Hibernate | | DataNucleus | |
|---|---|---|---|---|
| Lesen mit Filter (lazy loading) | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr |
| | 1051.58 | 0.95 | 11571.8 | 0.09 |

Tabelle 5.9.: Lazy loading der Einträge aus Neo4j mit einem Filter

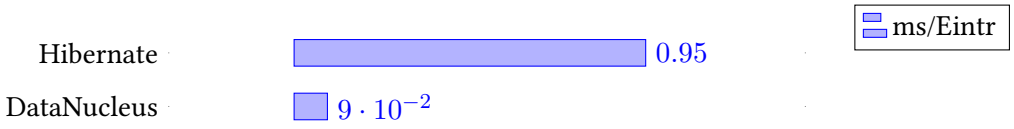


Abbildung 5.9.: Lazy loading der Einträge aus Neo4j mit einem Filter

| Neo4j | Hibernate | | DataNucleus | |
|---|---|---|---|---|
| Lesen mit Filter (eager loading) | Eintr/s | ms/Eintr | Eintr/s | ms/Eintr |
| | 175.5 | 5.7 | 10303.81 | 0.1 |

Tabelle 5.10.: Eager loading der Einträge aus Neo4j mit einem Filter

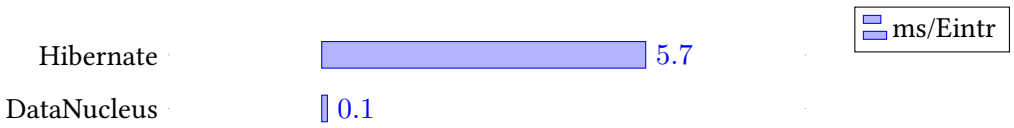


Abbildung 5.10.: Eager loading der Einträge aus Neo4j mit einem Filter

## 5.4. Diskussion

Dargestellte Diagramme machen ersichtlich, dass die Datenspeicherung in einer dokumentenorientierten Datenbank MongoDB mittels auf MongoDB ausgerichtetem Framework - Morphia am schnellsten gelingt.

Von den zwei anderen ObjectMapper Frameworks, welche auf mehreren Datenbanken einsetzbar sind, ist DataNucleus beim Speichern ca. 4 Mal schneller als Hibernate.

Bei dem Speichern der Objekte in einer Graphdatenbank Neo4j ist jedoch das Hibernate Framework ca. 1,5 Mal schneller als das DataNucleus Framework.

Beim Lesen aus der MongoDB hat sich DataNucleus Object-Mapper, als das schnellste bewiesen, sowohl mit als auch ohne einem Filter. Das explizit auf MongoDB ausgerichtete Morphia Object-Mapper hingegen weist beim Lesen wesentlich längere Antwortzeiten als beide andere Object-Mapper Frameworks ca. zweimal langsamer als Hibernate Framework.

Auch beim Lesen aus der Neo4j Datenbank ist DataNucleus schneller als Hibernate, wobei der Geschwindigkeitsvorteil in den Suchanfragen mit einem Filter wesentlich höher ist. Beim eager loading ist das Lesen mit DataNucleus ca. 500 Mal schneller als mit Hibernate. Beim lazy loading ist DataNucleus ca. 10 Mal schneller.

Die Ergebnisanalyse führt zu folgenden Erkenntnissen:

1. Für die Anwendungen, welche die große Datenmengen speichern und dabei selten auf die Daten zugreifen, ist Morphia Framework eine aus der Performanzsicht gute Lösung. Darüber hinaus, wie im Unterabschnitt 3.5.4 beschrieben, eignet sich Morphia durch Evolution-Annotationen für die Schema Evolution innerhalb einer Anwendung. Allerdings ist das Object-Mapper nur für MongoDB geeignet. Wenn die Daten in der Zukunft in einer anderen Datenbank migriert werden, stellen Hibernate oder DataNucleus Frameworks eine bessere Lösung dar, weil diese mehrere Datenbanksysteme unterstützen.

2. Für die Anwendungen, welche sowohl große Datenmengen speichern als auch auf die persistierten Daten oft zugreifen müssen, eignet sich DataNucleus Framework aus der Sicht der Performanz am besten. Wobei, kann Hibernate Framework einen Leseperformanz-Overhead durch eine effiziente Benutzung des 3rd level Cache erreichen, wie es im Unterabschnitt 3.5.4 beschrieben wurde.

3. Lazy loading gewährleistet einen großen Lesegeschwindigkeitsgewinn sowohl bei dem Hibernate als auch bei dem Morphia Framework. Es fällt jedoch auf, dass bei dem DataNucleus Object-Mapper der Lesegeschwindigkeitsunterschied zwischen dem lazy und dem eager loading am geringsten ist. Aus der Unterabschnitt 5.3.4 ersichtlich, es wurde keinen Geschwindigkeitsunterschied beim Lesen mit einem Filter aus der Neo4j Datenbank festgestellt. Auch beim Lesen ohne einem Filter aus MongoDB (Unterabschnitt 5.3.1) ist der Unterschied gering. Dieses lässt bezweifeln, dass DataNucleus Framework das Fetch Groups Konzept von JDO-API korrekt umsetzt. Um dieses sicher festzustellen, sind jedoch weitere gezielte Untersuchungen im data fetching Bereich notwendig.

# 6. Zusammenfassung

Im Rahmen dieser Arbeit wurde Object-NoSQL Data Mapping Konzept sowohl in der Theorie als auch in der Praxis erforscht.

Der theoretische Hintergrund des Object-Mapping wurde erörtert, indem es in Kapitel 2 zuerst auf die Relevanz des Themas eingegangen wurde. Danach wurde in Kapitel 3 die Funktionsweise von Object-Mappers erklärt. Da im Fokus dieser Arbeit das Mapping für NoSQL-Datenbanksysteme liegt, wurde die Datenmodellierung in unterschiedlichen NoSQL-Datenbank-kategorien betrachtet. Um die theoretische Grundlage für die in Kapitel 4 folgende Untersuchung zu verschaffen, wurden die Basisfunktionalitäten der für die Untersuchung ausgesuchten Object-NoSQL Mappers beschrieben.

Um sich dem Thema Object-Mapper für NoSQL-Datenbanksysteme aus der praktischen Seite zu nähern, wurde in Kapitel 4 eine vergleichende Untersuchung von drei derzeit verbreiteten Object-Mapper Frameworks durchgeführt. Zwei dieser Frameworks repräsentieren JPA- und JDO-Spezifikation und sind für verschiedene NoSQL-Datenbankkategorien einsetzbar, das dritte Framework hingegen hat eine eigene Spezifikation und ist auf ein bestimmtes NoSQL-Produkt auf MongoDB ausgerichtet. Somit wurde ein Überblick über unterschiedliche Object-Mapping Ansätze verschaffen.

In der vergleichenden Untersuchung wurde die Geschwindigkeit, mit welcher die Daten in der Datenbank gespeichert werden, sowie die Geschwindigkeit, mit welcher die Daten aus der Datenbank gelesen werden, für die ausgewählten Object-Mapper Frameworks bestimmt und verglichen. Beim Lesen der Daten wurden Anfragen mit und Anfragen ohne einen Filter an die Datenbanken gestellt. Für das Laden der Daten wurden lazy loading als auch eager loading Techniken eingesetzt. Alle Versuche liefen gegen zwei NoSQL-Datenbanksysteme, Repräsentanten der dokumentenorientierten und Graph-Datenbanken, gegen MongoDB und gegen Neo4j.

Um die Untersuchung zu realisieren, wurde eine nur für diese Bachelorarbeit erfundene Anwendung entwickelt. Die Entwicklung erfolgte auf Basis der für den Vergleich benötigten Kriterien, welche im Abschnitt 4.2 als Anforderungen an die Beispielanwendung aufgelistet sind.

# A. Datenbankeinträge

## A.1. Hibernate

```
1
2 > db.getCollection("User").find()
3 { "_id" : "cd5ea98a-67ee-4cc7-8cf4-f5a43712f562", "userData" :
    { "password" : "password1", "username" : "user1" }, "
   firstName" : "user1", "lastName" : "user110", "userComments
   " : [ "5de03078-7477-470a-916f-68875e682576", "d2502ab2-
   dc57-4b48-86d8-e22d9c73a8a1" ], "userPosts" : [ "81c28f99-
   fe90-4ab5-8efe-1b1e2b2bb037", "fa0e59ac-4d7c-4875-9e63-81
   ee801e6dca" ], "discussions" : [ "0b4d32da-be0f-4148-a8ec-2
   d548ad4d8b6", "dcd3bc85-2f24-49da-8c8a-560766f793ee" ] }
```

Listing A.1: Hibernate: User Eintrag in MongoDB

```
1 > db.getCollection("Discussion").find()
2 { "_id" : "dcd3bc85-2f24-49da-8c8a-560766f793ee", "topic" : "
   discussion210", "users" : [ "cd5ea98a-67ee-4cc7-8cf4-
   f5a43712f562", "95e42c63-0abd-4719-bf1b-647eff9fb8f9" ] }
```

Listing A.2: Hibernate: Discussion Eintrag in MongoDB

```
1 > db.getCollection("Comment").find()
2 { "_id" : "5de03078-7477-470a-916f-68875e682576", "date" :
   ISODate("2018-05-18T16:39:36.641Z"), "author_id" : "
   cd5ea98a-67ee-4cc7-8cf4-f5a43712f562", "post_id" : "81
   c28f99-fe90-4ab5-8efe-1b1e2b2bb037" }
```

Listing A.3: Hibernate: Comment Eintrag in MongoDB

```
1 > db.getCollection("Post").find()
2 { "_id" : "81c28f99-fe90-4ab5-8efe-1b1e2b2bb037", "date" :
    ISODate("2018-05-18T16:39:36.641Z"), "author_id" : "
    cd5ea98a-67ee-4cc7-8cf4-f5a43712f562", "userComments" : [
    "5de03078-7477-470a-916f-68875e682576" ] }
```

Listing A.4: Hibernate: Post Eintrag in MongoDB

```
1 Node: 0 Property: topic - discussion110, Property: id -
    79453099-7910-43e2-8d2c-1a7f24784644,
2 Node: 1 Property: firstName - user1, Property: lastName -
    user110, Property: id - 83c0b593-517e-4c67-9da5-18
    accb717518,
3 Node: 2 Property: password - password1, Property: username -
    user1,
4 Node: 3 Property: date - 2018/07/03 13:04:14:502 +0200,
    Property: id - 25baa131-6499-4d95-ba17-88053256fb65,
5 Node: 4 Property: date - 2018/07/03 13:04:14:502 +0200,
    Property: id - 182a7807-f3ea-47e5-b445-c29b0990bbe8,
6 Node: 5 Property: date - 2018/07/03 13:04:14:502 +0200,
    Property: id - d886b053-d45e-430a-84ce-21deaaa3cf3e,
7 Node: 6 Property: date - 2018/07/03 13:04:14:502 +0200,
    Property: id - 3e628b95-5e42-4246-9af8-fddc206acea4,
8 Node: 7 Property: firstName - user2, Property: lastName -
    user210, Property: id - 51d70ce3-28d2-4e0c-9e5c-79
    da85f973e9,
9 Node: 8 Property: password - password2, Property: username -
    user2,
10 Node: 9 Property: date - 2018/07/03 13:04:14:502 +0200,
    Property: id - deee578b-ed63-404c-84bb-14984d23e234,
11 Node: 10 Property: topic - discussion210, Property: id - 2280
    a84d-07e4-45ea-832d-41f8d7bd2446
```

Listing A.5: Hibernate: Einträge in Neo4j

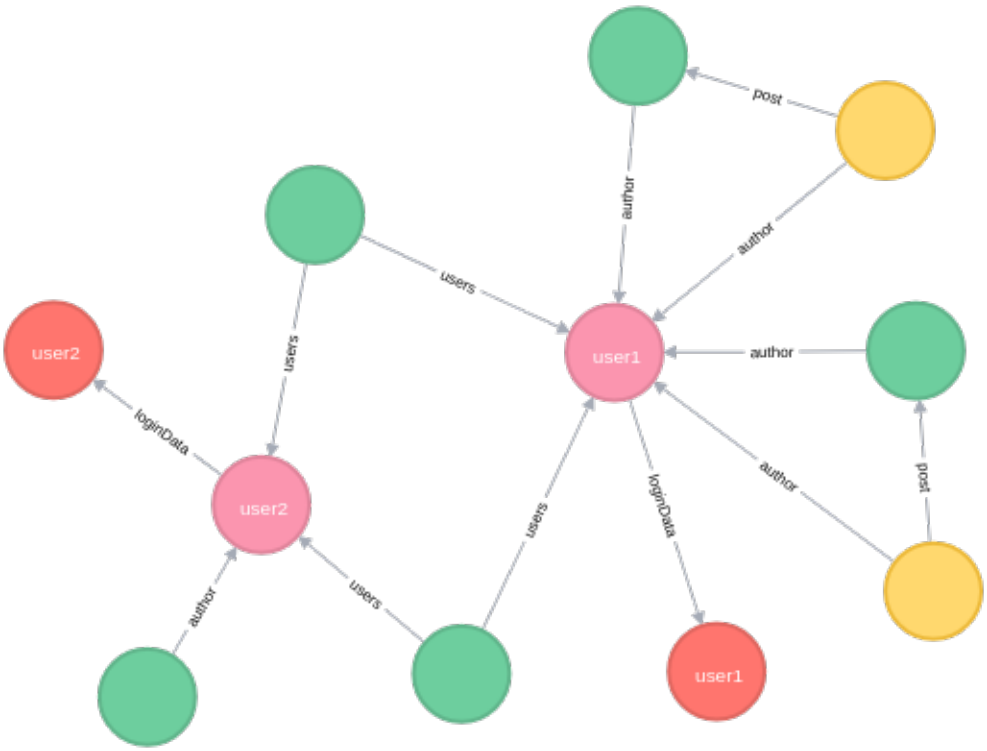


Abbildung A.1.: Hibernate: Graph

Eingebetete Entitäten werden als Nodes in der Datenbank gespeichert.

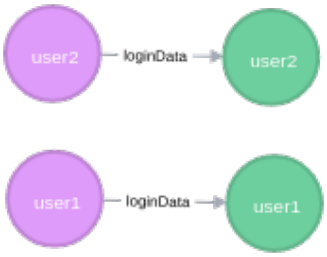


Abbildung A.2.: Hibernate: eingebettete Entitäten

## A.2. DataNucleus

```
1
2 > db.getCollection("User").find()
3 { "_id" : ObjectId("5b00a84dbcf53b1b45b83200"), "userPosts" :
    [ "com.haw_hamburg.de.objectMapping.dataNucleus.entities.
    Post:6b6cf249-ec24-40b7-9309-130f6472d805", "com.
    haw_hamburg.de.objectMapping.dataNucleus.entities.Post:
    ba906118-00af-47c6-8c51-a03ac72b1ce2" ], "userComments" : [
     "com.haw_hamburg.de.objectMapping.dataNucleus.entities.
    Comment:7cee03ac-bcea-4e2d-8a2c-d851a875de27", "com.
    haw_hamburg.de.objectMapping.dataNucleus.entities.Comment
    :7858dcd5-1daa-49e9-a149-faa522a864a4" ], "userData" : { "
    userData_username" : "user1", "userData_password" : "
    password1" }, "lastName" : "user110", "id" : "5ba61ebd
    -2440-48ef-a2b6-f170fe360f3b", "firstName" : "user1", "
    discussions" : [ "com.haw_hamburg.de.objectMapping.
    dataNucleus.entities.Discussion:35d9e199-3213-4dbb-adf8-39
    b1adbe8597", "com.haw_hamburg.de.objectMapping.dataNucleus.
    entities.Discussion:5210e71f-9430-4435-b1b5-d41a9242024a" ]
     }
```

Listing A.6: DataNucleus: User Eintrag in MongoDB

```
1 > db.getCollection("Discussion").find()
2 { "_id" : ObjectId("5b00a84cbcf53b1b45b831fd"), "users" : [ "
    com.haw_hamburg.de.objectMapping.dataNucleus.entities.User
    :5ba61ebd-2440-48ef-a2b6-f170fe360f3b", "com.haw_hamburg.de
    .objectMapping.dataNucleus.entities.User:fae3699b-2b9c-483d
    -9711-8529170636ba" ], "topic" : "discussion210", "id" :
    "5210e71f-9430-4435-b1b5-d41a9242024a" }
```

Listing A.7: DataNucleus: Discussion Eintrag in MongoDB

```
1 > db.getCollection("Post").find()
2 { "_id" : ObjectId("5b00a84cbcf53b1b45b831fb"), "userComments"
     : [ "com.haw_hamburg.de.objectMapping.dataNucleus.entities
```

```
   .Comment:7858dcd5-1daa-49e9-a149-faa522a864a4" ], "id" : "
   ba906118-00af-47c6-8c51-a03ac72b1ce2", "date" : ISODate
   ("2018-05-19T22:42:20.033Z"), "author" : "com.haw_hamburg.
   de.objectMapping.dataNucleus.entities.User:5ba61ebd-2440-48
   ef-a2b6-f170fe360f3b" }
```

Listing A.8: DataNucleus: Post Eintrag in MongoDB

```
1 > db.getCollection("Comment").find()
2 { "_id" : ObjectId("5b00a84cbcf53b1b45b831fa"), "post" : "com.
   haw_hamburg.de.objectMapping.dataNucleus.entities.Post:
   ba906118-00af-47c6-8c51-a03ac72b1ce2", "id" : "7858dcd5-1
   daa-49e9-a149-faa522a864a4", "date" : ISODate("2018-05-19
   T22:42:20.034Z"), "author" : "com.haw_hamburg.de.
   objectMapping.dataNucleus.entities.User:5ba61ebd-2440-48ef-
   a2b6-f170fe360f3b" }
```

Listing A.9: DataNucleus: Comment Eintrag in MongoDB

```
1 Node: 0 Property: userData_password - password1, Property:
   firstName - user1, Property: userData_username - user1,
   Property: lastName - user110, Property: id - 0,
2 Node: 1 Property: date - Di Jul 03 13:29:36 MESZ 2018,
   Property: id - 2ac4a9af-b135-4aa3-a224-e13d3e6f778b,
3 Node: 2 Property: date - Di Jul 03 13:29:36 MESZ 2018,
   Property: id - 8e7409d8-0d86-4a89-af55-719789209ea1,
4 Node: 3 Property: date - Di Jul 03 13:29:36 MESZ 2018,
   Property: id - 34c68092-9535-44db-a049-84de685e0aff,
5 Node: 4 Property: date - Di Jul 03 13:29:36 MESZ 2018,
   Property: id - 99ffeb54-0395-4fb4-aace-5697610bd1f5,
6 Node: 5 Property: id - 5, Property: topic - discussion210,
7 Node: 6 Property: userData_password - password2, Property:
   firstName - user2, Property: userData_username - user2,
   Property: lastName - user210, Property: id - 6,
8 Node: 7 Property: date - Di Jul 03 13:29:36 MESZ 2018,
   Property: id - 4a730f9e-a325-41ad-8c0c-22f20b30539c,
9 Node: 8 Property: id - 8, Property: topic - discussion110
```

Listing A.10: DataNucleus: Einträge in Neo4j


Abbildung A.3.: DataNucleus: Graph

## A.3. Morphia

```
1 > db.getCollection("User").find()
2 { "_id" : ObjectId("5b0065a5bcf53b46c01c3d84"), "firstName" :
    "user1", "lastName" : "user110", "userData" : { "username"
    : "user1", "password" : "password1" }, "userPosts" : [
    DBRef("Post", ObjectId("5b0065a5bcf53b46c01c3d7f")), DBRef
    ("Post", ObjectId("5b0065a5bcf53b46c01c3d80")) ], "
    userComments" : [ DBRef("Comment", ObjectId("5
    b0065a5bcf53b46c01c3d83")), DBRef("Comment", ObjectId("5
    b0065a5bcf53b46c01c3d82")) ], "discussions" : [ DBRef("
    Discussion", ObjectId("5b0065a5bcf53b46c01c3d7e")), DBRef("
    Discussion", ObjectId("5b0065a5bcf53b46c01c3d7d")) ] }
```

Listing A.11: Morphia: User Eintrag in MongoDB

```
1 > db.getCollection("Post").find()
2 { "_id" : ObjectId("5b0065a5bcf53b46c01c3d7f"), "userComments"
     : [ DBRef("Comment", ObjectId("5b0065a5bcf53b46c01c3d82"))
     ], "date" : ISODate("2018-05-19T17:57:57.406Z"), "author"
    : DBRef("User", ObjectId("5b0065a5bcf53b46c01c3d84")) }
```

Listing A.12: Morphia: Post Eintrag in MongoDB

```
1 > db.getCollection("Comment").find()
2 { "_id" : ObjectId("5b0065a5bcf53b46c01c3d82"), "post" : DBRef
    ("Post", ObjectId("5b0065a5bcf53b46c01c3d7f")), "date" :
    ISODate("2018-05-19T17:57:57.641Z"), "author" : DBRef("User
    ", ObjectId("5b0065a5bcf53b46c01c3d84")) }
```

Listing A.13: Morphia: Comment Eintrag in MongoDB

```
1 > db.getCollection("Discussion").find()
2 { "_id" : ObjectId("5b0065a5bcf53b46c01c3d7d"), "topic" : "
    discussion110", "users" : [ DBRef("User", ObjectId("5
    b0065a5bcf53b46c01c3d84")), DBRef("User", ObjectId("5
    b0065a6bcf53b46c01c3d85")) ] }
```

Listing A.14: Morphia: Discussion Eintrag in MongoDB

# B. Queries

## B.1. JPQL-Queries

```
1  entityManager.getTransaction().begin();
2
3  users = entityManager.createQuery("SELECT u FROM User u ",
       User.class).getResultList();
4
5  // Query mit einem Filter
6  users = entityManager.createQuery("SELECT u FROM User u WHERE
       firstName = 'user1' ", User.class).getResultList();
7
8  posts = entityManager.createQuery("SELECT p FROM Post p ",
       Post.class).getResultList();
9  comments = entityManager.createQuery("SELECT c FROM Comment c
       ", Comment.class).getResultList();
10 discussions = entityManager.createQuery("SELECT d FROM
       Discussion d ", Discussion.class).getResultList();
11 entityManager.getTransaction().commit();
```

Listing B.1: Hibernate: Queries

## B.2. JDOQL-Queries

```
1  persistenceManager.currentTransaction().begin();
2
3  Query<User> queryUsers = persistenceManager
4  .newQuery("SELECT FROM com.haw_hamburg.de.objectMapping.
       dataNucleus.entities.User ");
```

```
5 users = queryUsers.executeList();
6
7 // Query mit einem Filter
8 Query<User> queryUsers = persistenceManager.newQuery(
9 "SELECT FROM com.haw_hamburg.de.objectMapping.dataNucleus.
     entities.User WHERE firstName == 'user1' ");
10 users = queryUsers.executeList();
11
12 Query<Post> queryPosts = persistenceManager
13 .newQuery("SELECT FROM com.haw_hamburg.de.objectMapping.
     dataNucleus.entities.Post ");
14 posts = queryPosts.executeList();
15 Query<Comment> queryComments = persistenceManager
16 .newQuery("SELECT FROM com.haw_hamburg.de.objectMapping.
     dataNucleus.entities.Comment ");
17 comments = queryComments.executeList();
18 Query<Discussion> queryDiscussions = persistenceManager
19 .newQuery("SELECT FROM com.haw_hamburg.de.objectMapping.
     dataNucleus.entities.Discussion ");
20 discussions = queryDiscussions.executeList();
21 persistenceManager.currentTransaction().commit();
```

Listing B.2: DataNucleus: Queries

## B.3. Queries in Morphia

```
1 Datastore datastore = frTest.getDatastore();
2
3 Query<User> queryUsers = datastore.createQuery(User.class);
4 users = queryUsers.asList();
5
6 // Query mit einem Filter
7 Query<User> queryUsers = datastore.createQuery(User.class);
8 queryUsers.filter("firstName = ", "user1");
9 users = queryUsers.asList();
```

```
10
11 Query<Post> queryPosts = datastore.createQuery(Post.class);
12 posts = queryPosts.asList();
13 Query<Comment> queryComments = datastore.createQuery(Comment.
      class);
14 comments = queryComments.asList();
15 Query<Discussion> queryDiscussions = datastore.createQuery(
      Discussion.class);
16 discussions = queryDiscussions.asList();
```

Listing B.3: Morphia: Queries

# C. Maven Dependencies

## C.1. Hibernate

```
<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.hibernate.ogm</groupId>
<artifactId>hibernate-ogm-bom</artifactId>
<type>pom</type>
<version>5.3.1.Final</version>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

Listing C.1: Hibernate: Maven Dependencies

```

<dependency>
<groupId>org.hibernate.ogm</groupId>
<artifactId>hibernate-ogm-mongodb</artifactId>
</dependency>
```

Listing C.2: Hibernate: Mongo Dependency Version v3.6.4

```
1
2 <dependency>
3 <groupId>org.hibernate.ogm</groupId>
4 <artifactId>hibernate-ogm-neo4j</artifactId>
5 </dependency>
```

Listing C.3: Hibernate: Neo4j Dependency Version 3.3.3

## C.2. DataNucleus

```
1 <dependency>
2 <groupId>org.datanucleus</groupId>
3 <artifactId>datanucleus-core</artifactId>
4 <version>5.1.9</version>
5 </dependency>
```

Listing C.4: DataNucleus: Maven Dependencies

```
1 <dependency>
2 <groupId>org.datanucleus</groupId>
3 <artifactId>datanucleus-api-jdo</artifactId>
4 <version>5.1.6</version>
5 </dependency>
```

Listing C.5: DataNucleus: JDO API Dependency

```
1 <dependency>
2 <groupId>org.datanucleus</groupId>
3 <artifactId>javax.jdo</artifactId>
4 <version>3.2.0-m5</version>
5 </dependency>
```

Listing C.6: DataNucleus: JDO Dependency

```
1
2 <dependency>
3 <groupId>org.datanucleus</groupId>
4 <artifactId>datanucleus-mongodb</artifactId>
5 <version>5.1.0-release</version>
6 </dependency>
```

Listing C.7: DataNucleus: Mongo Dependency Version v3.6.4

```
1 <dependency>
2 <groupId>org.datanucleus</groupId>
3 <artifactId>datanucleus-neo4j</artifactId>
4 <version>5.1.3</version>
5 </dependency>
```

Listing C.8: DataNucleus: Neo4j Dependency Version v3.0.0

## C.3. Morphia

```
1 <dependency>
2 <groupId>org.mongodb.morphia</groupId>
3 <artifactId>morphia</artifactId>
4 <version>1.3.2</version>
5 </dependency>
```

Listing C.9: Morphia: Maven Dependencies

Ähnlich wie bei den anderen Object-Mappers, wurde für den Versuch remote MongoDB in Version v3.6.4 eingesetzt.

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Listings

# Abkürzungsverzeichnis

**DBMS** Database Management System

**DB** Database

**SQL** Structured Query Language

**API** Application Programming Interface

**POJO** Plain Old Java Object

**JPA** Java Persistence API

**JDO** Java Data Objects

**JSON** JavaScript Object Notation

**JPQL** Java Persistence Query Language

**JDOQL** Java Data Objects Query Language

**HQL** Hibernate Query Language

**ONDM** Object-NoSQL Data Mapping

**ORM** Object-Relational Mapping

**OGM** Object-Grid Mapping

**ID** Identifier

**UUID** Universally Unique Identifier

**JSR** Java Specification Request

**EJB** Enterprise JavaBeans

**JDK** Java Development Kit

**JPOX** Java Persistent Objects

**RFC** Request for Comments

**DAO** Data Access Object

# Literaturverzeichnis


[1] Datanucleus. http://www.datanucleus.org (Letzter Zugriff am 20.08.2018).

[2] Testprojekt datanucleus mongodb. https://github.com/liksita/objectMappingDataNucleusMongoDB.

[3] Testprojekt datanucleus neo4j. https://github.com/liksita/objectMappingDataNucleusNeo4j.

[4] Testprojekt graph visualiser. https://github.com/liksita/Neo4jGraphVisualiser.git.

[5] Testprojekt hibernate mongodb. https://github.com/liksita/objectMappingHibernateMongoDB.

[6] Testprojekt hibernate neo4j. https://github.com/liksita/objectMappingHibernateNeo4j.

[7] Testprojekt morphia mongodb. https://github.com/liksita/ObjectMappingMorphia.

[8] E. Bernard, S. Grinovero, G. Morling, D. D'Alto, G. Scheibel, Paluch M., and Smet G. Hibernate ogm 5.3.1.final: Reference guide. März 2018. https://docs.jboss.org/hibernate/stable/ogm/reference/en-US/html_single (Letzter Zugriff am 22.08.2018).

[9] J. Chris Anderson, J. Lehnardt, and N. Slater. *CouchDB - The Definitive Guide.* O'REILLY-Verlag, 2012.

[10] E Evans. Nosql: What's in a name? Oktober 2009. http://blog.sym-link.com/2009/10/30/nosql_whats_in_a_name.html (Letzter Zugriff am 22.08.2018).

[11] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2003.

[12] M. Fowler. Polyglotpersistence. November 2011. `https://martinfowler.com/bliki/PolyglotPersistence.html` (Letzter Zugriff am 20.08.2018).

[13] M. Hilbert and P López. The world's technological capacity to store, communicate, and compute information. *Science*, 332, Issue 6025(10.1126/science.1200970), April 2011.

[14] M. Inden. *Der Java-Profi: Persistenzlösungen und REST-SERVICES.* dpunkt.verlag, 2016.

[15] T. Jüngling. Datenvolumen verdoppelt sich alle zwei jahre. *WELT*, Juli 2013.

[16] Th. Kestler. *Hibernate Kochbuch.* elevato GmbH. `http://www.elevato.de/media/HibernateKochbuch.pdf` (Letzter Zugriff am 20.08.2018).

[17] Störl U.-Scherzinger St. Klettke, M. Herausforderungen bei der anwendungsentwicklung mit schema-flexiblen nosql-datenbanken. *HMD Praxis der Wirtschaftsinformatik*, 4/2016.

[18] K. Landeck. Graphdatenbanken in der praxis – anwendungsbereiche für neo4j. *Java-SPEKTRUM*, 06/2016.

[19] St. Marr. The java data objects persistence model. Seminar System Modeling, 2005. `http://stefan-marr.de/pages/jdo-persistence-model` (Letzter Zugriff am 20.08.2018).

[20] A. Meier and M. Kaufmann. *SQL- & NoSQL-Datenbanken.* Springer-Verlag Berlin Heidelberg, 2016.

[21] E. Merker. *Grundkurs Java-Technologien.* Friedr. Vieweg & Sohn Verlag / GWV Fachverlage GmbH, 2004.

[22] M. Merz. The management of users, roles, and permissions in jdosecure. *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, August 2006.

[23] A. Ramachandran. Modellieren von dokumentdaten für nosql-datenbanken, Mai 2016. `https://docs.microsoft.com/de-de/azure/cosmos-db/modeling-data` (Letzter Zugriff am 20.08.2018).

[24] Rafique A.-Landuyt D. Joosen W. Reniers, V. Object-nosql database mappers: a benchmark study on the performance overhead. *Journal of Internet Services and Applications*, Januar 2017.

[25] Scherzinger St.-Tegawendé F. Bissyandé Ringlstetter, A. Data model evolution using object-nosql mappers: Folklore or state-of-the-art? *2016 2nd International Workshop on BIG Data Software Engineering*, 2016.

[26] Uwe Schäfer. Morphia – pojo-persistenz mit mongodb, September 2010. `https://jaxenter.de/morphia-pojo-persistenz-mit-mongodb-2-7624` (Letzter Zugriff am 22.08.2018).

[27] J. Steemann. Datenmodellierung in nicht relationalen datenbanken. `https://entwickler.de/online/datenbanken/datenmodellierung-in-nicht-relationalen-datenbanken-137872.html` (Letzter Zugriff am 20.08.2018).

[28] C. Strozzi. *Strozzi Nosql (RDBMS).* Soph Press, 2012.

[29] Hauf Th.-Klettke M. Scherzinger St. Störl, U. Schemaless nosql data stores – object-nosql mappers to the rescue? *BTW'15*, 2015.

[30] Ruiz-Carrillo R. Yu C. Yousuf Ahmad M. Kemme B. Vinish D'silva, J. Secondary indexing techniques for key-value stores: Two rings to rule them all. `http://ceur-ws.org/Vol-1810/DOLAP_paper_10.pdf` (Letzter Zugriff am 20.08.2018).

*Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.*

Hamburg, 23. August 2018 Diana Topko